Abstract

The RAISE Development Method is a formal software development method with an associated formal specification language, RSL, and a set of tools supporting the method and RSL.

A typical RAISE development of imperative software starts with an abstract applicative specification which is developed into a concrete applicative specification. This concrete applicative specification is then transformed into an imperative specification, which can be further developed using the RAISE Development Method.

While the development of applicative and imperative specifications is defined by a refinement relation, the transformation from applicative into imperative specification is only described informally.

This project defines a set of transformation rules, which can be used to transform an applicative RSL specification into an imperative RSL specification for a subset of RSL.

A notion of correctness of the transformation from applicative into imperative specification is defined and a verification of the correctness of the transformation rules is outlined. This means that when developing from applicative into imperative specification using the verified transformation rules, the correctness of this development step need not to be verified.

A transformation tool that implements the transformation rules has been developed.

Keywords RAISE, RSL, tool, transformation, applicative, imperative.

Resumé

RAISE udviklingsmetoden er en formel metode til udvikling af software med tilhørende formelt specifikationssprog, RSL, og et sæt værktøjer, der understøtter metoden og RSL.

En typisk RAISE udvikling af imperativt software starter med en abstrakt applikativ specifikation, som bliver udviklet til en konkret applikativ specifikation. Denne konkrete applikative specifikation transformeres til en imperativ specifikation, som derefter kan videreudvikles ved hjælp af RAISE udviklingsmetoden.

Mens udviklingen af applikative og imperative specifikationer er defineret ved hjælp af en forfiningsrelation, er transformationen fra applikativ til imperativ specifikation kun beskrevet uformelt.

I dette projekt defineres en række transformationsregler, som kan bruges til at transformere en applikativ RSL specifikation til en imperativ RSL specifikation for en delmængde af RSL.

Et korrekthedsbegreb for transformationen bliver defineret, og et bevis for korrektheden af transformationsreglerne bliver skitseret. Det betyder, at korrektheden af udviklingen fra applikativ til imperativ specifikation ikke behøver at blive verificeret, hvis transformation foretages ved hjælp af transformationsreglerne.

Et værktøj, der implementerer transformationsreglerne, er blevet udviklet.

Nøgleord RAISE, RSL, værktøj, transformation, applikativ, imperativ.

Preface

This report documents the M.Sc. project of Tine Damkjær Jørgensen. The project has been carried out in the period from September 27th, 2004, to April 25th, 2005, at the Technical University of Denmark, Department of Informatics and Mathematical Modelling, the Computer Science and Engineering division. The project was supervised by Associate Professor, Ph.D. Anne E. Haxthausen and Associate Professor Hans Bruun.

I would like to thank my supervisors for their great interest in my work and their constructive criticism and inspiration during the project period.

Kgs. Lyngby, April 25, 2005

Tine Damkjær Jørgensen, s991160

Contents

List of Tables

List of Figures

Chapter 1

Introduction

There are several approaches to the development of software systems. Some of these approaches are rather informal while others are very formal involving mathematical techniques. The formal methods are especially useful when developing safety critical systems. One formal method used today is RAISE (Rigorous Approach to Industrial Software Engineering) which consists of the following parts:

- The *RAISE Development Method*, which is a method for developing software in a formal way.
- *RSL* (RAISE Specification Language), which is a mathematically based notation useful for formal specification of software.
- A set of tools supporting the two items above.

1.1 The RAISE Development Method

When developing software using the RAISE Method the following steps must be conducted:

Initial Specification: The requirements of the system are specified in RSL based on a description of these in natural language.

Specification and Design: More and more detailed RSL specifications are produced. All the development steps are verified by validating that the specifications developed fulfill the requirements in the previous specification.

Translation: A program or a collection of programs written in a suitable programming language are produced based on the final RSL specification.

This stepwise development uses the *invent-and-verify* approach, meaning that in each step a new specification is invented. It is verified that this new specification is a correct development of the previous specification in the sense that the new specification conforms to the previous specification.

The RAISE Method suggest a certain order of development, which is illustrated in Figure 1.1.

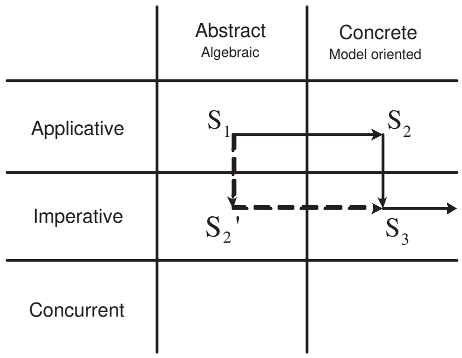

Figure 1.1: Development process using the RAISE Method.

When the initial specification containing the requirements of the system is made, an *abstract applicative* specification S_1 is developed, possibly using several steps. An abstract applicative specification contains abstract types such as sorts. Signatures and axioms are preferred over explicit function definitions.

When the abstract applicative specification S_1 is made, a *concrete applicative* specification S_2 is developed. In a concrete applicative specification concrete types such as maps are used and explicit function definitions are preferred. In order to verify that the development step from S_1 to S_2 is correct, a development relation is formulated asserting that the concrete applicative specification implements the abstract applicative specification: $S_2 \preceq S_1$. Justification of this relation shows that the development step is correct.

Then a *concrete imperative* specification S_3 is developed. In a concrete imperative specification variables are defined and explicit function definitions are used. The verification of this development step is done either informally or formally. If the informally verification method is used, it is checked whether the method for the transition described in [Hax99, p. 17] is followed correctly. If the formal verification method is used, the imperative axioms corresponding to the applicative axioms for the abstract applicative specification are formulated and justified for the concrete imperative specification. A development relation cannot be formulated for this step.

Instead of developing first a concrete applicative specification S_2 and then a concrete imperative specification S_3, another route of development can be followed. After the abstract applicative specification S_1 is developed an *abstract imperative* specification S_2' can be developed. In an abstract imperative specification no variables are defined but the RSL keyword **any** is used in the function access descriptions in order to specify that some variables are accessed in the function bodies, but that the names of these variables are not specified yet. Furthermore, axioms are preferred over explicit function definitions.

Based on S_1 a concrete imperative specification can be developed. Verification is needed in both steps. The verification of the step from S_1 to S_2' is done as for the step from the concrete applicative specification S_2 to the concrete imperative specification S_3. For the development step from abstract imperative specification to concrete imperative specification a development relation $S_3 \preceq S_2'$ can be formulated and justified. This development route is not very common compared to the first one described.

The concrete imperative specification can be further developed either to a final concrete imperative specification or to a concurrent specification, but these development steps are not of interest to this project.

The boundaries between the different kinds of specifications are not fixed. A specification can have characteristics from all of the above types of specifications. That is a specification can consist of both sorts, concrete types, variables, axioms and explicit function definitions.

1.2 Motivation

When looking at concrete applicative specifications and their concrete imperative counterparts many patterns between the two kinds can be found. This leads to the idea that the step from a concrete applicative specification to a concrete imperative specification can be done by applying a set of transformation rules to the concrete applicative specification which would create an imperative specification.

By verifying the correctness of this set of transformation rules the correctness of the transformation from a concrete applicative specification to a concrete imperative specification would automatically be verified, if the transformation was done using the verified transformation rules.

If it is possible to formulate precise systematic transformation rules it may be possible to develop a tool that can carry out this step in the RAISE development process.

1.3 Thesis Objectives

The objectives of this project are the following:

1. Formulation of a set transformation rules that can be applied to a concrete applicative RSL specification returning a concrete imperative RSL specification.
2. Verification of the correctness of this set of transformation rules.
3. Development of a tool, the *transformer*, that can carry out the transformation from concrete applicative specification into concrete imperative specification using the formulated set of transformation rules.

1.4 Requirements of the Project

The objectives listed above can be reached in many ways, but in order for the final transformer to be useful some requirements have to be fulfilled:

- The transformation rules should be constructed such that the output is readable and recognizable.
  Reason: If the resulting imperative RSL specification is to be further developed it is necessary that the specification is readable. Furthermore it should be possible to recognize the applicative specification in the corresponding imperative specification in order to ease an iterative development process.
- The resulting imperative specifications should be a syntactically correct RSL specifications without any references to the original applicative specifications.
  Reason: When using the tool it should not be necessary to correct the output. The idea behind a tool is to save time and to reduce the number of errors.
- The transformer has to be constructed such that it can be maintained and extended easily.
  Reason: As the first version of the tool should only cover a subset of RSL it should be easy to extend, such that it, in time, covers all parts of RSL. Furthermore, it should be easy to adapt the tool to possible future changes in RSL.

1.5 Prerequisites for Reading the Report

There is a number of prerequisites needed in order to get the full understanding of this report:

- A good knowledge of RSL and the different styles of RSL specification.
- A knowledge of the mathematical foundations of RAISE.

Furthermore, a basic knowledge of Java, language processors and the tool translating a subset of RSL into Java, named the *RSL2Java tool*, developed in the project described in [Hja04] would be an advantage.

1.6 Structure of the Report

The report is structured as follows:

Chapter 2 gives a general idea of what it means to transform an applicative RSL specification into an imperative RSL specification. This is illustrated by an example. Furthermore, the concepts of the transformer are presented.

Chapter 3 presents the main syntax and terms used throughout the report. This chapter can be used to get an overview of the syntax and terms and for easy reference during the reading of the report.

Chapter 4 presents the subset of RSL the transformer can handle.

Chapter 5 offers a definition of the conditions an applicative specification within the subset of RSL has to fulfill such that it can be transformed into a corresponding imperative specification.

Chapter 6 informally defines the transformation rules for the subset of RSL considered.

Chapter 7 defines a notion of correctness of the transformation from applicative into imperative specification and gives an outline of a verification of the transformation rules.

Chapter 8 describes the RSL specifications of the transformer.

Chapter 9 provides a description of the structure and implementation of the transformer program. Furthermore, an overview of the implemented functionality is given.

Chapter 10 demonstrates the use of the transformer on some simple examples.

Chapter 11 describes the tests performed on the transformer.

Chapter 12 provides transformation rules of some RSL constructs, which are not part of the RSL subset considered. Furthermore, ideas for extensions of the transformer are given.

Chapter 13 contains the conclusion on the project as a whole.

1.7 Contents of the Appendices

The appendices are structured as follows:

Appendix A describes how to use the transformer and how to extend it.

Appendix B describes the contents of the enclosed CD-ROM.

Appendix C contains the RSL specification of the transformer.

Appendix D contains RSL specifications of the abstract syntax tree of RSL and the transformer, both written within a subset of RSL, named RSL_1.

Appendix E contains the ANTLR grammar file which is used to generate the lexer and parser.

Appendix F contains the handwritten Java source code of the transformer.

Appendix G contains the an overview of all tests performed on the transformer.

Chapter 2

General Idea

In order to give an understanding of what the development step from a concrete applicative specification to a concrete imperative specification comprises, this step will be discussed in the following based on a simple example, a stack. Furthermore, the general concept of the transformer will be discussed.

2.1 The STACK Example

A stack must provide the following functions:

- *empty*: Empties the stack.
- *push*: Puts an element on the top of the stack.
- *pop*: Removes the top element of the stack, if the stack is not empty.
- *top*: Returns the top element of the stack, if the stack is not empty.

The type of the elements on the stack is left unspecified. This leads to the concrete applicative specification given in Specification 2.1. The function *is_empty* is an auxiliary function.

Specification 2.1 – Concrete applicative specification of a stack

---

scheme A_STACK =
  class
    type
      Stack = Element*,
      Element

value
empty : Stack = $\langle\rangle$,

is_empty : Stack $\rightarrow$ **Bool**
is_empty(stack) $\equiv$ stack = $\langle\rangle$,

push : Element $\times$ Stack $\rightarrow$ Stack
push(elem, stack) $\equiv$ $\langle$elem$\rangle$ $^\frown$ stack,

pop : Stack $\stackrel{\sim}{\rightarrow}$ Stack
pop(stack) $\equiv$ **tl** stack
pre $\sim$ is_empty(stack),

top : Stack $\stackrel{\sim}{\rightarrow}$ Element
top(stack) $\equiv$ **hd** stack
pre $\sim$ is_empty(stack)
end

This applicative specification can be developed into a corresponding imperative specification. Before developing the imperative specification it has to be decided, which types that should be represented by variables, the so called *types of interest*. Despite the term "type of interest", the type of interest refers to the name of a type, not the type itself.

In the case of the stack it is obvious that the stack should be represented by a variable of type *Stack*. This variable is given the name *stack*. Then instead of referring to parameters of the *Stack* type in the functions, references to the *stack* variable are made.

When a value of type *Stack* is an argument of a function in the applicative specification, the variable *stack* is *read from* in the imperative specification in order to get the value of the *stack* variable. Functions reading from the types of interest are called *observers*. The *Stack* type name is removed from the left hand side of the arrow of the single typing, and so is the corresponding parameter of the formal function application. A **read** stack access descriptor is inserted at the right hand side of the function arrow in the single typing.

When a value of type *Stack* is returned by a function in the applicative specification, the *stack* variable is *written to* in the imperative specification in order to change the value of the *stack* variable. Functions writing to the types of interest are called *generators*. The *Stack* type name is removed from the right hand side of the arrow of the single typing and a **write** stack access descriptor is inserted at the right hand side of the function arrow in the single typing.

If a function is both an observer and a generator these methods are combined. If the above changes leave either the left or right hand side or both sides without any type expressions **Unit** is inserted.

Explicit value definitions of the *Stack* type is transformed into explicit function definitions, in which the *stack* variable is assigned the value expression of the explicit value definition.

This leads to the concrete imperative specification given in Specification 2.2.

Specification 2.2 – Concrete imperative specification of a stack

---

scheme I_STACK =
  class
    type
      Stack = Element*,
      Element

    variable
      stack : Stack

    value
      empty : **Unit** → **write** stack **Unit**
      empty() ≡ stack := ⟨⟩,

      is_empty : **Unit** → **read** stack **Bool**
      is_empty() ≡ stack = ⟨⟩,

      push : Element → **write** stack **Unit**
      push(elem) ≡ stack := ⟨elem⟩ ⌒ stack,

      pop : **Unit** $\stackrel{\sim}{\to}$ **write** stack **Unit**
      pop() ≡ stack := **tl** stack
      pre ∼ is_empty(),

      top : **Unit** $\stackrel{\sim}{\to}$ **read** stack Element
      top() ≡ **hd** stack
      pre ∼ is_empty()
  end

The following changes have been made:

- The constant *empty* is transformed into an explicit function definition in which the *stack* variable is assigned the empty list.

- In the observer function *is_ empty* the value of the *stack* variable is tested to see if it equals the empty list.
- In the combined observer and generator function *push* the *stack* variable is assigned its old value to which the new element is concatenated.
- In the combined observer and generator function *pop* the *stack* variable is assigned the tail of the *stack*. The constant *is_ empty* of the type *Stack* is replaced with a function application *is_ empty()* in order to adjust to the transformation of the explicit value definition *is_ empty*.
- In the observer function *top* the *stack* is read from in order to return the top element of the *stack*. The constant *is_ empty* of the type *Stack* is replaced with a function application *is_ empty()* in order to adjust to the transformation of the explicit value definition *is_ empty*.

As can be seen there are many similarities between the two specifications. The idea is that instead of developing the imperative specification by hand the similarities and patterns can be exploited in order to develop a tool that can do the development step automatically.

2.2 The Transformer Concept

The general idea behind the transformer is that it should be a function that can be applied to an applicative specification and then return a corresponding imperative specification. The input given to the transformer is an applicative specification, the name of the new imperative specification and a list of types of interest and corresponding variable names. This leads to a simple program for the user to use, but it has some drawbacks concerning the possibilities of the transformer as described below.

As the transformation should be automatic, it is necessary to make the decision, that every occurrences of type names of the types of interest are to be conceived as types of interest and transformed accordingly. This leads to situations where two semantically equivalent specifications are transformed differently, see Example 2.1 and Example 2.2.

Example 2.1 – First transformation, types of interest: T

---

scheme A_INCREMENT =
    class
        type
            T = **Int**

```
    value
      increment : T × Int → T
      increment(t, i) ≡ t + i
  end
```

▷

```
scheme I_INCREMENT =
  class
    type
      T = Int
    variable
      t : T
    value
      increment : Int → read t write t Unit
      increment(i) ≡ t := t + i
  end
```

Example 2.2 – Second transformation, types of interest: T, S

```
scheme A_INCREMENT =
  class
    type
      T = Int,
      S = Int
    value
      increment : T × S → T
      increment(t, s) ≡ t + s
  end
```

▷

```
scheme I_INCREMENT =
  class
    type
      T = Int,
      S = Int
    variable
      t : T,
      s : S
```

value
increment : **Unit** → **read** t, s **write** t **Unit**
increment() ≡ t := t + s
end

The problem occurs because it is necessary to distinguish between the type names of the types of interest and the actual types of the types of interest. This means that the type inference of the value expressions changes accordingly. A value expressions with type of interest $T = type\_expr$ is treated as a value expression of type T and not of type $type\_expr$ as the static inference of RSL normally prescribes. The type of a value expression is solely determined based on the single typing of the function definition in which the value expression is incorporated.

This makes new demands on the specification style. It is very important that the types of interest are used with care in the applicative specification. They shall only be used in contexts where they are actually wanted. When it is necessary that only some occurrences of a type of interest should be transformed as types of interest, it is necessary to establish other type names with the same type as the type of interest. Then these new type names should be used whenever an expression should not be transformed as a type of interest.

The problem could be avoided by making an interactive transformer in which the user interactively could decide which occurrences of the types of interest that should be transformed as such and which should be transformed as ordinary types. This solution is not implemented in this project.

Chapter 3

Terminology

During the report certain syntax and terminology are used. These are defined in this chapter in order to give an overview and for quick reference during the reading.

3.1 Syntax

The syntax

$$\mathsf{applicative\_specification} \triangleright \mathsf{imperative\_specification}$$

is used to denote that the applicative_specification is transformed into the imperative_specification. The development step from applicative_specification to imperative_specification is correct in the sense that their exists a certain relation between the semantics of the two specifications as explained in Chapter 7.

3.2 Terms and Definitions

A list of the terms used in the report together with their definitions are given below.

Type of Interest

When transforming an applicative specification into an imperative specification it has to be decided which types should be represented by variables. These types are called *types of interest*. The term, type of interest, is actually misleading as the types of interest refers to type names, not to the type expressions of these type names. The term originate from the literature, e.g. [Gro95].

A type of interest is during the report most often named T, and the corresponding variable is named t.

Expected Type

The *expected type* of a value expression is the type the value expression is expected to have considering the context in which the value expression occurs. For example, the expected type of a function application expression is the type of the right hand side of the single typing of the function. Normally, the expected type is the unfolded type expression of this type, but in this project type names of the types of interest are not unfolded.

Observer

Functions reading a value of a type of interest, i.e. functions taking a value of the type of interest as parameter in an applicative specification, are called *observers*. Generally throughout the report *obs* is used for the id of an observer function. An example of an observer function can be seen in Example 3.1.

Example 3.1 – Observer

---

type
  T = **Bool**
value
  obs : T → **Int**
  obs(t) ≡ **if** t **then** 1 **else** 0 **end**

Generator

Functions returning values of the types of interest in an applicative specification are called *generators*. Generally throughout the report *gen* is used for the id of a generator function. An example of a generator function can be seen in Example 3.2.

Example 3.2 – Generator

---

type
  T = **Int**
value
  gen : **Int** → T
  gen(i) ≡ i

Hidden Observer

A function observing the type T is named a *hidden observer* if T is not part of the left hand side of the function signature, see Example 3.3.

Example 3.3 – Hidden observer

type
  T = **Int**
value
  hidden_obs : **Int** → **Int**
  hidden_obs(x) ≡ obs(x),

  obs : T → **Int**
  obs(t) ≡ t

Hidden Generator

A generator altering values of the type T is named a *hidden generator* if T is not part of the right hand side of the function signature, see Example 3.4.

Example 3.4 – Hidden generator

type
  T = **Int**
value
  hidden_gen : **Int** → **Int**
  hidden_gen(x) ≡ gen(x),

  gen : **Int** → T
  gen(i) ≡ i

Implicit Observer

An *implicit observer* is a function which implicitly converts a value of a type of interest into one of the equivalent type expressions during the evaluation of the function body, see Example 3.5.

Example 3.5 – Implicit observer

type
  T = **Int**
value
  f : **Int** → **Int**
  f(x) ≡ x + 1,

  implicit_obs : T → **Int**
  implicit_obs(t) ≡ f(t)

Implicit Generator

An *implicit generator* is a function which implicitly converts a value of a certain type into an equivalent type of interest during the evaluation of the function body, see Example 3.6.

Example 3.6 – Implicit generator

type
  T = **Int**
value
  f : T → **Int**
  f(t) ≡ t + 1,

  implicit_gen : **Int** → **Int**
  implicit_gen(x) ≡ f(x + 1)

T-generator

A *T-generator* is a generator returning a value of the type T, that is a generator with the following single typing in the applicative case:

$$\text{gen} : ... \rightarrow ... \times \text{T} \times ...$$

Proper T-generator

A *proper T-generator* is a generator which returns only a value of the type of interest T, that is a generator with the following single typing in the applicative case:

$$\text{gen} : ... \rightarrow \text{T}$$

T-free value expression

A *T-free value expression* is a value expression not containing any values of the types of interest.

RSL$_A$

RSL$_A$ is the applicative subset of RSL, which the applicative specifications must be written within.

RSL$_I$

RSL$_I$ is the imperative subset of RSL, in which the resulting imperative specification of a transformation is written within.

Chapter 4

Constraints

In this project only a subset of RSL, named RSL_A for an applicative subset of RSL, is considered. This chapter gives an overview of the RSL expressions and forms not considered. Explanations for the omissions are given when possible.

It is assumed that the specifications given to the transformer are static correct specifications according to the static semantics of RSL. This means that a specification should be type checked using the RSL type checker before it is transformed using the transformer.

4.1 The RSL Constructions Not Considered

In order to be able to transform applicative specifications into imperative specifications and in order to avoid that the work gets to extensive compared to the benefits only a subset of RSL is considered.

The RSL constructs not considered are either of no relevance or are not considered in order to simplify matters. Furthermore, most of the constraints introduced do not limit the expressive power, as these constructs can be rewritten into constructs which comply with the constraints.

In the following a list of the RSL constructs not within RSL_A is given as well as reasons as to why these are not considered. The list is divided into syntactical and non syntactical constraints, according to how the checks for them are implemented in the transformer.

Syntactical Constraints

- Abstract specifications (e.g. sort definitions and axiomatic value definitions).
  Reason: To simplify matters. Furthermore, the most common way of applying the RAISE Development Method, is to start out with an abstract applicative specification, which is made concrete before it is

further developed into an imperative specification. This makes it superfluous to be able to transform abstract applicative specifications. There is one exception to this constraint. Sort definitions that are not of a type of interest are transformed, as this transformation does not cause any trouble.

- Concurrent specifications
  Reason: To simplify matters. Concurrent specifications are not considered in order to limit the size of the project.

- Union definitions.
  Reason: To simplify matters. Due to major revisions needed in one of the tools used in order to be able to do the transformation of union definitions, union definitions are left out. Furthermore, union definitions can be rewritten into variant definitions.

- Post conditions.
  Reason: To simplify matters. Post conditions are usually used in abstract specifications which are not considered.

- Parameterized scheme definitions.
  Reason: To simplify matters. Furthermore, a parameterized scheme definition can be rewritten into a non-parameterized scheme definition by incorporating the classes over which the scheme is parameterized.

- Extending class expressions.
  Reason: To simplify matters. Furthermore, an extending class expression can be rewritten into a basic class expression by incorporating the extended class in the extending class.

- Hiding class expressions.
  Reason: To simplify matters.

- Renaming class expressions.
  Reason: To simplify matters. Furthermore, a renaming class expression can be rewritten into a basic class expression by rewriting the original class using the new names.

- Scheme instantiations.
  Reason: To simplify matters. Furthermore, scheme instantiations can be rewritten by using a whole scheme definition for each instantiation.

- Object declarations.
  Reason: To simplify matters.

- Quantified expressions.
  Reason: To simplify matters.

- Function expressions.
  Reason: To simplify matters. In some cases function expressions can be rewritten by introducing a new function at outermost level.

- Initialise expressions.
  Reason: Initialise expressions are not considered as the initial values of the established variables are unknown. This means that an initialse expression would have no effect.

- Infinite map expressions.
  Reason: To simplify matters. Due to major revisions needed in one of the tools used in order to be able to do the transformation of infinite map expressions, infinite map expressions are left out. Furthermore, infinite maps are not widely used.

- Test cases.
  Reason: To simplify matters.

Non-Syntactical Constraints

- Subtype expressions in which a type of interest is part of the single typing.
  Reasons: The transformation of subtype expressions in which a type of interest is part of the single typing would lead to a syntactically wrong imperative specification.

- Explicit value definitions of the form $product\_binding : type\_expr = value\_expr$.
  Reason: To simplify matters. Furthermore, most value definitions of the form $product\_binding : type\_expr = value\_expr$ can be rewritten by establishing more explicit value definitions, one for each $id\_binding$ in the $product\_binding$.

- The type expression of a single typing of an explicit function definition shall be a function type expression.
  Reason: To simplify matters. Furthermore, this can easily be accomplished by inserting the actual type of the function instead of using a corresponding type name.

- Higher order functions.
  Reason: To simplify matters. In some cases higher order functions can be rewritten into normal functions, e.g. a function of the type U $\rightarrow$ T $\rightarrow$ V can be rewritten into a function of the type U $\times$ T $\rightarrow$ V.

- Collections of values of the types of interest such as sets and lists of values of the types of interest and maps containing values of the types

of interest.
Reason: Only collections of a fixed or bounded size would be possible to transform, but very few specifications need such collections, wherefore this is not considered in the transformer.

- Recursive type definitions of the types of interest.
Reason: As only one variable is established per type of interest, recursive type definitions of the types of interest are not possible to transform as this would require more than one variable per type of interest.

- In application expressions of the form *value_ expr(value_ expr*$_1$*, ..., value_ expr*$_n$*)*, *value_ expr* has to be a value name.
Reason: To simplify matters.

- Overloading.
Reason: To simplify matters.

- Value expressions in pre-conditions containing generators, either explicit, implicit or hidden.
Reason: The requirement that the transformed specification shall be syntactically correct is impossible to fulfill if pre-conditions consist of any kinds of generators. This is due to the fact that pre-conditions are read-only. The only exception to this rule is that function applications taking only value names of the types of interest are allowed as these value names are just removed during the transformation.

- Patterns in case expressions which are constants of a type of interest.
Reason: The requirement that the transformed specifications shall be syntactically correct is impossible to fulfill if patterns of case expressions are of a type of interest. This is due to the fact that constants of a type of interest are transformed into explicit function definitions and function applications are not allowed as patterns according to the syntax of RSL.

- Patterns in case expressions containing value names of the types of interest.
Reason: The requirement that the transformed specifications shall be syntactically correct is impossible to fulfill if patterns of case expressions consist of value names of the types of interest. This is due to the fact that value names of the types of interest are transformed into variables and variables are not allowed as patterns according to the syntax of RSL.

- The number of parameters of a function has to be at least as big as the number of components of the type expression on the left hand side

of the function signature - both formal and actual parameters.
Reason: To simplify matters. Problems arise especially when type names of the types of interest are involved.

- In let bindings only a product binding having at least as many components as the product expression is allowed.
  Reason: To simplify matters. Problems can arise especially when type names of the type of interest are involved.

Ideas for transformations of some of the constructs given in the lists above are provided in Chapter 12.

Chapter 5

Transformability

Not every specification in RSL_A can be transformed using the transformation rules given in Chapter 6. The reason for this is discussed in this chapter. First an example of a concrete problem is offered and then the conditions a specification has to fulfill such that it can be transformed are defined.

5.1 Problems

Problems can arise when transforming concrete applicative specifications within RSL_A. One of the problems is illustrated in Example 5.1 in which an applicative specification is transformed according to the transformation rules which will be described in Chapter 6.

Example 5.1 – Problematic transformation

---

scheme A_SPEC =
  class
    type
      T = **Int**
    value
      f : T → **Int**
      f(x) ≡ g(x) + x,

      g : T → T
      g(x) ≡ x + 1
  end

▷

```
scheme I_SPEC =
   class
      type
         T = Int
      variable
         t : T
      value
         f : Unit → read t write t Int
         f() ≡ (g() ; t) + t,

         g : Unit → read t write t Unit
         g() ≡ t := t + 1
   end
```

It is obvious that the the imperative specification in Example 5.1 is not a correct development of the applicative specification. This means that the transformation is invalid. The problem is that the variable *t* changes its value during execution of the function *g* in the imperative specification, while in the applicative specification it is the original value of the *x* parameter that is used during the whole execution of the function body of *f*.

This means that the transformation from applicative into imperative specification not always is a correct development when using the transformation rules defined in Chapter 6. This means that only some applicative specifications written within RSL_A can be transformed using the transformation rules. The problem is that there is only one placeholder for each variable. This means that only one value per variable can be stored at a time.

The applicative specifications that can be transformed are in the following called *transformable*. Before an applicative specification is transformed, it has to be checked if the transformation can be done. That is, if the specification is transformable.

5.2 Conditions for Transformability

The following conditions for transformability are based on the transformation rules proposed in this project. Other transformation rules would probably lead to other conditions for transformability.

In order for an applicative specification to be transformable, it has to fulfill stronger scope rules than the static semantics of RSL allows. The general idea is that at any point of execution references to at most one value name of a certain type of interest are allowed.

The scope rules for a transformable applicative specification are as follows, where "after" corresponds to the order of execution, left to right:

Referencing a value name id_1 of the type of interest T is not allowed after:

- declaration of a new value name id_2 of which type T is a part.
- application of an explicit, implicit or hidden generator of T.

The scope rules are illustrated in Figure 5.1.

type
  T

value
  gen : ... $\rightarrow$ ... $\times$ T $\times$ ...

  f : T $\rightarrow$ ...
  $f(t_1) \equiv t_1$ + **let** $t_2 = t_1$ + gen(...) **in** t_2 + gen(...) + ... **end**

```
scope t1:      |________________|

scope t2:                                          |_______|
```

Figure 5.1: The scope

This means that the scope of a value name of the type T is constrained by generators of T and let expressions. The value name can only be referenced inside its scope. If it is referenced outside its scope the specification is not transformable.

Chapter 6

Transformations

In this chapter the transformations of different RSL constructs of within RSL_A are considered one by one. Only specifications with one type of interest is considered. In Section 6.6 the extension into specifications with more than one type of interest is discussed.

The structure of this chapter follows the syntax summary of RSL given in [Gro92, pp. 371-380] to the extend possible.

A complete RSL specification of the transformations can be found in Appendix C.

In the following it is assumed that the constructs are part of a transformable applicative RSL specification.

6.1 Transforming Scheme Definitions

A scheme definition has the following form:

$$\textbf{scheme}\ \text{id} = \text{class\_expr}$$

When transforming a scheme definition the *id* of the scheme need not to be changed, but it is good practice to rename the scheme as the specification has to be saved under the name of the scheme. If the applicative name is kept, the applicative specification will be overwritten. After that the class expression, *class_ expr*, must be transformed as in Section 6.2.

6.2 Transforming Class Expressions

There are several kinds of class expressions but only basic class expressions are a part of RSL_A. Ideas for transformation of other kinds of class expressions can be found in Chapter 12.

6.2.1 Transforming Basic Class Expressions

A basic class expression consists of a list of declarations which must be transformed one at a time according to the transformations given in this chapter.

6.3 Transforming Type Definitions

There are 5 different kinds of type definitions: sort definitions, variant definitions, union definitions, short record definitions and abbreviation definitions. Union definitions are not part of RSL_A, but a suggestion of how to transform union definitions is given in Chapter 12. The transformation of type definitions can be done in two steps:

1. Transformation of the constituent type expressions
2. Introduction of variable definitions.

These steps are described below.

6.3.1 Transforming Type Expressions in Type Definitions

In the following descriptions of the transformation of type expressions are given.

Type Literals
The transformation of type literals is straight forward as they have to be transformed within the same language, namely RSL. This means that a type literal is transformed into itself.

Type Names
Type names are transformed into themselves.

Product Type Expressions
Product type expressions are of the form:

$$(\text{type\_expr}_1, \text{type\_expr}_2, ..., \text{type\_expr}_n)$$

A product type expression is transformed by transforming the constituent type expressions one by one.

Set Type Expressions
Set type expressions are of one of the forms:

$$\text{type\_expr-}\textbf{set}$$
$$\text{type\_expr-}\textbf{infset}$$

A set type expression is transformed by transforming the constituent type expression.

List Type Expressions

List type expressions are of one of the forms:

$$\text{type\_expr}^{*}$$
$$\text{type\_expr}^{\omega}$$

A list type expression is transformed by transforming the constituent type expression.

Finite Map Type Expressions

Finite map type expressions are of the form:

$$\text{type\_expr}_1 \underset{m}{\rightarrow} \text{type\_expr}_2$$

A finite map type expression is transformed by transforming the constituent type expressions.

Subtype Expressions

Subtype expressions are of the form:

$$\{|\ \text{binding} : \text{type\_expr} \bullet \text{value\_expr}\ |\}$$

A subtype expression is transformed by transforming the constituent type expression. The value expression has to be transformed using the transformation rules given in Section 6.4.3.

Bracketed Type Expressions

Bracketed type expressions are of the form:

$$(\ \text{type\_expr}\ )$$

A bracketed type expression is transformed by transforming the constituent type expression.

6.3.2 Introducing Variable Definitions

The first thing to do when transforming an applicative specification into an imperative specification is to decide which type that has to be the type of interest and give name to the corresponding variable.

If a type name is declared as a type of interest a variable declaration has to be established. The variable can be defined as in Example 6.1.

Example 6.1 – Introducing a variable

type T = ...

▷

type T = ...
variable t : T

If the type of interest is a product type it can be an advantage to define a variable for each component of the product in order to be able to refer to these in the function declarations, see Example 6.2.

Example 6.2 – Introducing variables from product type

type
  T = U × V,
  U = ...,
  V = ...

▷

type
  T = U × V,
  U = ...,
  V = ...
variable
  u : U,
  v : V

6.4 Transforming Explicit Function Definitions

After deciding on the type of interest and introducing the corresponding variable the function definitions can be transformed. An explicit function definition has the form:

single_typing
formal_function_application ≡ value_expr
optional_pre_condition

The transformation of an explicit function definition can be done in four steps:

1. Transformation of the single typing, also known as the function signature.
2. Transformation of the formal function application.
3. Transformation of the value expression.
4. Transformation of the possible pre-condition.

These steps are described below.

6.4.1 Transforming Single Typings

When transforming a single typing the single typing keeps its id. Only single typings in which the type expression is a function type expression are allowed. The function type expression is transformed in the following way:

- The type of the arrow, either total or partial, of the type expression is preserved.
- If the applicative function takes the type of interest T as argument, this is removed from the left hand side of the arrow and replaced by an access descriptor **read** t at the right hand side of the arrow, where t is the variable name of the type of interest.
- If the applicative function returns the type of interest, this is replaced by an access descriptor **write** t.
- Any gaps in the single typing after the above transformation are filled with **Unit**, that is if the whole left or right hand side of the arrow are removed.

Examples of transformations of single typings are given in Table 6.1 on the following page.

Furthermore, if the value expression of the function contains hidden observers or generators or implicit observers or generators, **read**/**write** access descriptors matching those of the observers/generators have to be included in the single typing. This has to be done recursively, such that if a function contains an application of a function containing observers and/or generators the appropriate **read** and **write** access descriptors matching these observers and generators have to be included in the single typing of the original function, and so on.

The inclusion of both **read** and **write** access descriptors of the same variable is not necessary due to the fact that **write** implies **read**. The

Applicative single typing	$\triangleright$	Imperative single typing
U → V	$\triangleright$	U → V
T → V	$\triangleright$	**Unit** → **read** t V
U → T	$\triangleright$	U → **write** t **Unit**
T → T	$\triangleright$	**Unit** → **read** t **write** t **Unit**
T × U → V	$\triangleright$	U → **read** t V
U → T × V	$\triangleright$	U → **write** t V
T × U → T × V	$\triangleright$	U → **read** t **write** t V

Table 6.1: Transformation of single typings. T is a type of interest with associated variable t, U and V are not of the type of interest.

superfluous **read** access descriptor is included in order to ease the verification of the transformation rules, see Chapter 7.

The reason for these transformations is that instead of referring to the type of interest using parameters, you refer to the corresponding variable of the type of interest, and instead of returning a type of interest, you assign the results to the variable of the type of interest.

6.4.2 Transforming Formal Function Applications

When transforming formal function applications all occurrences of parameters of the type of interest are removed as in the following example where *t* is a parameter of the type of interest:

$$\begin{gathered}\mathrm{f(u,\ t,\ v)} \equiv ... \\ \triangleright \\ \mathrm{f(u,\ v)} \equiv ...\end{gathered}$$

If the type of interest is an abbreviation type and this is exploited in the formal function application, all parameters forming the type of interest are removed as in the following example where *(u, v)* is of the type of interest:

$$\begin{gathered}\mathrm{f(p,\ (u,\ v),\ q)} \equiv ... \\ \triangleright \\ \mathrm{f(p,\ q)} \equiv ...\end{gathered}$$

When this is done it is necessary to establish a let expression in the transformation of the value expression at the right hand side of the equivalence operator ≡:

$$\mathrm{f(p,\ q)} \equiv \textbf{let}\ \mathrm{(u,\ v) = t}\ \textbf{in}\ ...\ \textbf{end}$$

This is done in order to be able to refer to the parameters removed as these are not variables that can be referred to as *t* can.

6.4.3 Transforming Value Expressions in Function Definitions

This section concerns the transformation of value expressions in function definitions. First a general introduction to the transformations is given and then the different kinds of value expressions are discussed one by one.

Introduction to Transforming Value Expressions

A function can always be transformed by making the value expression a function application of the corresponding applicative function in which all parameters of the types of interest are replaced by the corresponding variables, see Example 6.3.

However, since the goal of the transformation is to remove all occurrences of objects of the applicative specification this is not satisfactory and can be improved in most cases by exploiting the following idea:

Example 6.3 – General transformation

```
scheme A_SPEC =
  class
    type
      T = ...,
      U = ...,
      V = ...
    value
      obs1 : T → U
      obs1(t) ≡ ...,
      obs2 : T × U → V
      obs2(t, u) ≡ ...,
      gen1 : T → T
      gen1(t) ≡ ...,
      gen2 : T × U → T
      gen2(t, u) ≡ ...,
      gen3 : T → T × U
      gen3(t) ≡ ...
      gen4 : T × U → T × V
      gen4(t, u) ≡ ...
  end

▷

scheme I_SPEC =
  class
```

```
object
  A : A_SPEC
type
  T = ...,
  U = ...,
  V = ...
variable
  t : T
value
  obs1 : Unit → read t U
  obs1() ≡ A.obs1(t),
  obs2 : U → read t V
  obs2(u) ≡ A.obs2(t, u),
  gen1 : Unit → read t write t Unit
  gen1() ≡ t := A.gen(t),
  gen2 : U → read t write t Unit
  gen2(u) ≡ t := A.gen(t, u),
  gen3 : Unit → read t write t U
  gen3() ≡ let (t′, u) = A.gen(t) in t := t′ ; u end
  gen4 : U → read t write t V
  gen4(u) ≡ let (t′, v) = A.gen(t, u) in t := t′ ; v end
end
```

When the function is an observer the corresponding imperative function is formed by replacing occurrences of the parameter referring to the type of interest with the variable referring to the type of interest. This is illustrated in Example 6.4.

Example 6.4 – Observer

```
type
  T = Bool
value
  obs : T → Int
  obs(x) ≡ if x then 1 else 0 end

▷

type
  T = Bool
variable
  t : T
```

value
obs : **Unit** → **read** t **Int**
obs() ≡ **if** t **then** 1 **else** 0 **end**

When the function is a generator the variable of the type of interest must be assigned the transformed value expression of the function in the imperative specification, as in Example 6.5.

Example 6.5 – Generator

type
T = **Int**
value
gen : **Int** → T
gen(i) ≡ i

▷

type
T = **Int**
variable
t : T
value
gen : **Int** → **write** t **Unit**
gen(i) ≡ t := i

When the function is both an observer and a generator the two methods mentioned above are combined. An example of a combined observer and generator function can be seen in Example 6.6.

Example 6.6 – Combined observer and generator

type
T = **Int**
value
obsgen : T → T
obsgen(x) ≡ x + 1

▷

type
  T = **Int**
variable
  t : T
value
  obsgen : **Unit** → **read** t **write** t **Unit**
  obsgen() ≡ t := t + 1

Transformation of the Different Value Expressions

In the following the transformations of the different value expressions of RSL_A are given.

Value Literals

The transformation of value literals is straight forward as they have to be transformed within the same language, namely RSL. This means that a value literal is transformed into itself.

Value Names

A value name, i.e. either the id of a value declaration, the id of a let binding or a formal parameter of a function definition, is transformed into itself except when it is of the type of interest. In that case it is either transformed into the variable name of the type of interest or, if it is a constant of the type of interest, it is transformed as an application expression taking no parameters.

Basic Expressions

When dealing with basic expressions it is only necessary to deal with **chaos** as the other basic expressions not are part of RSL_A. **chaos** is transformed into itself.

Product Expressions

A product expression has the following form:

$$(\text{value\_expr}_1, \text{value\_expr}_2, ..., \text{value\_expr}_n)$$

A product expression is transformed by transforming the constituent value expressions one by one, as in Example 6.7.

Example 6.7 – Transformation of a simple product expression

scheme A_SPEC =
  class
    type
      T = **Int**

    value
      f : T → **Int** × **Bool** × **Bool**
      f(x) ≡ (1 + x − 5, ∼false, obs(x)),

      obs : T → **Bool**
      obs(x) ≡ x > 10
  end

▷

scheme I_SPEC =
  class
    type
      T = **Int**

    variable
      t : T

    value
      f : **Unit** → **read** t **write** t **Int** × **Bool** × **Bool**
      f() ≡ (1 + t − 5, ∼ **false**, obs()),

      obs : **Unit** → **read** t **Bool**
      obs() ≡ t > 10
  end

There is one exception to this rule. If the product expression contains components which have expected type T, it is necessary to establish let expressions, for example:

(1, gen(2), 3)
▷
let (pa_0, pa_1, pa_2) = (1, gen(2), 3) **in** (pa_0, pa_2) **end**

where *gen* is a proper T-generator. The reason for this is that if the product expression is a return value of a function it has to follow the transformed single typing in which all occurrences of the type of interest are removed.

An example of a correct transformation of an applicative specification containing product expressions can be found in Example 6.8. In order to illustrate the rules, 3 types of interest, *T*, *R* and *S*, are introduced in the example. In the example the function *g1* is a proper T-generator and therefore *pa_0* is not part of the result in the imperative specification. The function *g2* is an S-generator but not a proper S-generator and is therefore part of the resulting product in the imperative specification. The last component of the product expression is an implicit proper R-generator, which is not part of the result in the imperative specification.

Example 6.8 – Transformation of a product expression

scheme A_SPEC =
  class
    type
      T = **Int**,
      S = **Int**,
      R = **Int**

    value
      f : T → T × **Bool** × (S × **Bool**) × R
      f(x) ≡ (g1(x), **false**, g2(2), 4),

      g1 : T → T
      g1(x) ≡ x + 1,

      g2 : **Int** → S × **Bool**
      g2(x) ≡ (x + 10, **true**)
  end

▷

scheme I_SPEC =
  class
    type
      T = **Int**,
      S = **Int**,
      R = **Int**

    variable

```
      t : T,
      s : S,
      r : R

   value
      f : Unit → read t write r, t, s Bool × (Bool)
      f() ≡
         let
            (pa_0, pa_1, pa_2, pa_3) =
               (g1(), false, g2(2), r := 4)
         in
            (pa_1, pa_2)
         end,

      g1 : Unit → read t write t Unit
      g1() ≡ t := t + 1,

      g2 : Int → write s Bool
      g2(x) ≡
         let (pa_0, pa_1) = (s := x + 10, true) in (pa_1) end
end
```

Set Expressions

A set expression takes one of the following forms:

- **Ranged set expression**:

  {value_expr$_1$ .. value_expr$_2$}

- **Enumerated set expression**:

  {value_expr$_1$, ..., value_expr$_n$}

- **Comprehended set expression**:

  {value_expr$_1$ | typing$_1$, ..., typing$_n$ • value_expr$_2$}

Ranged set expressions, enumerated set expressions and comprehended set expressions are transformed by transforming the constituent value expressions.

List Expressions

A list expression takes one of the following forms:

- **Ranged list expression**:

  $\langle$value_expr$_1$ .. value_expr$_2\rangle$

- **Enumerated list expression**:

  $\langle$value_expr$_1$, ..., value_expr$_n\rangle$

- **Comprehended list expression**:

  $\langle$value_expr$_1$ | binding **in** value_expr$_2$ • value_expr$_3\rangle$

Ranged list expressions, enumerated list expressions and comprehended list expressions are transformed by transforming the constituent value expressions.

Map Expressions

A map expression takes one of the following forms:

- **Enumerated map expression**:

  [value_expr$_1$ $\mapsto$ value_expr$_1$, ...,value_expr$_n$ $\mapsto$ value_expr$_n$]

- **Comprehended map expression**:

  [value_expr$_1$ $\mapsto$ value_expr$_2$ | typing$_1$, ..., typing$_n$ • value_expr$_3$]

Enumerated map expressions and comprehended map expressions are transformed by transforming the constituent value expressions.

Application Expressions

An application expression is of the following form:

value_expr(value_expr$_1$, value_expr$_2$, ..., value_expr$_n$)

and is evaluated by first evaluating *value_ expr* to give an applicable value, then *value_ expr*$_1$, then *value_ expr*$_2$, ..., then *value_ expr*$_n$ to give the actual parameters, and then finally applying the applicable value to the actual parameters.

There are three kinds of application expressions; function application expressions, list application expressions and map application expressions.

When transforming function application expressions there are three cases to consider:

1. The parameters are only value names of the type of interest.
   Transformation: The parameters are removed, see function *f1* in Example 6.9.

2. One parameter.
   Transformation: The constituent value expressions are transformed as described in this section, see function *f2* in Example 6.9.

3. All other cases.
   Transformation: The parameters are transformed as one product expression, which means that let expressions may be established if the function is an observer, see function *f3* in Example 6.9.

If *value_ expr* has no side effects the let binding of a possible let expression can be moved outside the actual function application expression. This solution is not used here in order for the transformer to be easy to extend such that specifications containing application expressions where *value_ expr* has side effects can be transformed.

Example 6.9 – Transformation of a function application expressions

scheme A_SPEC =
  class
    type
      T = **Int**

    value
      f1 : T → T
      f1(x) ≡ gen1(x),

      f2 : **Int** → T
      f2(i) ≡ gen1(i+1),

```
        f3 : Int × T → T
        f3(i, x) ≡ gen2(i, x),

        gen1 : T → T
        gen1(x) ≡ x + 1,

        gen2 : Int × T → T
        gen2(i, x) ≡ i + x
  end
```

▷

```
scheme I_SPEC =
  class
    type
      T = Int

    variable
      t : T

    value
      f1 : Unit → read t write t Unit
      f1() ≡ gen1(),

      f2 : Int → read t write t Unit
      f2(i) ≡ gen1(t := i + 1),

      f3 : Int → read t write t Unit
      f3(i) ≡
        gen2(let (pa_0, pa_1) = (i, t := t) in (pa_0) end),

      gen1 : Unit → read t write t Unit
      gen1() ≡ t := t + 1,

      gen2 : Int → read t write t Unit
      gen2(i) ≡ t := i + t
end
```

List application expressions and map application expressions are tranformed by transforming the constituent value expressions. An example of the transformation of a list application can be found in Example 6.10.

Example 6.10 – Transformation of a list application

scheme A_SPEC =
  class
    type T = **Int**,
        LIST = **Int***

    value
      f : T × LIST → **Int**
      f(x, l) ≡ l(g1(x)),

      g1 : T → T
      g1(x) ≡ x + 1
  end

▷

scheme I_SPEC =
  class
    type
      T = **Int**,
      LIST = **Int***

    variable
      t : T

    value
      f : LIST → **read** t **write** t **Int**
      f(l) ≡ l((g1() ; t)),

      g1 : **Unit** → **read** t **write** t **Unit**
      g1() ≡ t := t + 1
  end

Bracketed Expressions

A bracketed expression has the following form:

(value_expr)

A bracketed expression is transformed by transforming the constituent value expression.

Value Infix Expressions

A value infix expression has the following form:

$$\text{value\_expr}_1 \text{ infix\_op value\_expr}_2$$

A value infix expression is transformed by transforming the constituent value expressions. The infix operator stays the same.

Value Prefix Expressions

A value prefix expression has the following form:

$$\text{prefix\_op value\_expr}$$

A value prefix expression is transformed by transforming the constituent value expression. The prefix operator stays the same.

Let Expressions

A let expression takes the following form:

$$
\begin{array}{l}
\textbf{let} \\
\quad \text{letbinding}_1 = \text{value\_expr}_1, \\
\quad \vdots \\
\quad \text{letbinding}_n = \text{value\_expr}_n \\
\textbf{in} \\
\quad \text{value\_expr}_{n+1} \\
\textbf{end}
\end{array}
$$

A let expression is transformed by transforming the constituent value expressions. Furthermore, the let bindings have to be transformed in order to comply with the return values of the corresponding value expressions. The reason for this is that in the imperative specification generators do not return values of the type of interest. There are three cases to consider when dealing with a let expression of the form:

$$
\begin{array}{l}
\textbf{let} \\
\quad \text{letbinding} = \text{value\_expr} \\
\textbf{in} \\
\quad ... \\
\textbf{end}
\end{array}
$$

The type of *value_ expr*:

1. does not contain T.
   Transformation: The *letbinding* is transformed into itself, see function *f1* in Example 6.11.

2. is T.
   Transformation: A so called "dummy" binding is inserted in order to keep the let expression and the order of the let definitions, see function *f2* in Example 6.11.

3. is a product in which T occurs.
   Transformation: The bindings corresponding to the values of the type of interest are removed from the *letbinding*, see function *f3* in Example 6.11

Example 6.11 – Transformation of let expressions

---

scheme A_SPEC =
  class
    type
      T = **Int**

    value
      f1 : T → **Int**
      f1(x) ≡ **let** z = obs(x) **in** z **end**,

      f2 : **Int** → **Int**
      f2(x) ≡ **let** x1 = gen1(x) **in** x1 + 1 **end**,

      f3 : **Int** → **Int**
      f3(x) ≡ **let** (x1, z) = gen2(x) **in** z **end**,

      obs : T → **Int**
      obs(x) ≡ x,

      gen1 : **Int** → T
      gen1(x) ≡ x + 1,

      gen2 : **Int** → T × **Int**
      gen2(x) ≡ (x + 1, x − 1)
  end

▷

```
scheme I_SPEC =
  class
    type
      T = Int

    variable
      t : T

    value
      f1 : Unit → read t write t Int
      f1() ≡ let z = obs() in z end,

      f2 : Int → read t write t Int
      f2(x) ≡ let dummy = gen1(x) in t + 1 end,

      f3 : Int → write t Int
      f3(x) ≡ let z = gen2(x) in z end,

      obs : Unit → read t Int
      obs() ≡ t,

      gen1 : Int → write t Unit
      gen1(x) ≡ t := x + 1,

      gen2 : Int → write t Int
      gen2(x) ≡
        let (pa_0, pa_1) = (t := x + 1, x − 1) in (pa_1) end
  end
```

If the let binding is a record pattern or list pattern these are transformed according to the rules given on page 50.

If Expressions

A canonical if expression takes the following form:

if value_expr$_1$
then value_expr$_2$
else value_expr$_3$
end

A canonical if expression is transformed by transforming the constituent value expressions.

If expressions of the form:

if value_expr$_1$
then value_expr$_1$
elsif value_expr$_2$
then value_expr$_2$
⋮
elsif value_expr$_n$
then value_expr$_n$
else value_expr$_{n+1}$
end

are shorthand syntax for:

if value_expr$_1$
then value_expr$_1$
else
    if value_expr$_2$
    then value_expr$_2$
    else
        ⋮
        if value_expr$_n$
        then value_expr$_n$
        else value_expr$_{n+1}$
        end
        ⋮
    end
end

and are transformed by transforming the constituent value expressions.

Case Expressions

A case expression takes the following form:

case value_expr **of**
    pattern$_1$ $\rightarrow$ value_expr$_1$
    ⋮
    pattern$_n$ $\rightarrow$ value_expr$_n$
end

A case expression is transformed by transforming the constituent value expressions and patterns.

The patterns are transformed as follows:

Value Literal Pattern

A value literal pattern is transformed into itself.

Name Pattern

A name pattern is transformed into itself unless if it is of the type of interest. In that case the specification cannot be transformed as a pattern cannot consist of a variable.

Wildcard Pattern

A wildcard pattern is transformed into itself.

Product Pattern

A product pattern is of the form:

$$(\text{pattern}_1, \text{pattern}_2, ..., \text{pattern}_n)$$

A product pattern is transformed by transforming the constituent patterns.

Record Pattern

A record pattern is of the form:

$$\text{id}(\text{pattern}_1, \text{pattern}_2, ..., \text{pattern}_n)$$

A record pattern is transformed by transforming the constituent patterns.

List Pattern

A list pattern is of one of the forms:

$$\langle\text{pattern}_1, \text{pattern}_2, ..., \text{pattern}_n\rangle$$
$$\langle\text{pattern}_1, \text{pattern}_2, ..., \text{pattern}_n\rangle \frown \text{pattern}_{n+1}$$

A list pattern is transformed by transforming the constituent patterns.

Introducing Assignment Expressions

When transforming an applicative specification into the imperative counterpart it is necessary to establish assignment expressions when the value of the variable of the type of interest must be changed. This can happen in two situations.

When dealing with a generator of the form:

id : type_expr$_{arg}$ → type_expr$_{res}$
id(id$_1$, ..., id$_n$) ≡ value_expr

the variable has to be assigned the proper value. There are two cases to consider. The type expression *type_ expr$_{res}$* of the result of the function is:

1. T.
   Transformation: The variable must be assigned the transformed *value_ expr*, see function *gen1* in Example 6.12.

2. a product in which T occurs.
   Transformation: The variable must be assigned the proper part of the product expression of *value_ expr*, see function *gen2* in Example 6.12.

Example 6.12 – Assignment expressions in generators

```
type
  T = Int
value
  gen1 : Int → T
  gen1(i) ≡ i,

  gen2 : Int × Bool → T × Bool
  gen2(i, b) ≡ (i, b)
```

▷

```
type
  T = Int
variable
  t : T
value
  gen1 : Int → write t Unit
  gen1(i) ≡ t := i,

  gen2 : Int × Bool → write t Bool
  gen2(i, b) ≡
    let (pa_0, pa_1) = (t := i, b) in (pa_1) end
```

When dealing with implicit generators, that is when a value of the type of interest is expected, assignment expressions have to be established. This is due to the fact that when dealing with values of the type of interest, these are transformed into variables. In order for these variables to hold the proper value, assignment expressions must be established, see Example 6.13.

Example 6.13 – Assignment expressions when dealing with implicit generators

---

type
  T = **Int**
value
  f : T → **Int**
  f(t) ≡ t + 1,

  implicit_gen : **Int** → **Int**
  implicit_gen(x) ≡ f(x)

▷

type
  T = **Int**
variable
  t : T
value
  f : **Unit** → **read** t **Int**
  f() ≡ t + 1,

  implicit_gen : **Int** → **read** t **write** t **Int**
  implicit_gen(x) ≡ f(t := x)

Introducing Sequencing Expressions

When transforming an applicative specification it is necessary to introduce sequencing expressions when dealing with a T-generator in connection with an implicit observer. The reason is that a T-generator does not return any value when transformed, but the presence of the implicit observer shows that a value is needed. In that case a sequencing expression is introduced, consisting of the transformed T-generator and the variable of the type of interest. An example of this can be seen in Example 6.14.

In the *implicit_obs* function of the applicative specification in the example an **Int** is expected as the first argument of the plus infix expression. But because the function application expression *gen(x)* returns a value of the type T, the value of type T has to be converted into an **Int**. This is done by an implicit observer.

In the imperative specification the T-generator *gen* does not return any value, but a value is needed as first argument of the plus infix expression. This is accomplished by establishing a sequencing expression *(gen(x) ; t)* in order to extract the proper value to be used in the plus infix expression.

Example 6.14 – Establishing sequencing expressions

scheme A_SPEC =
  class
    type
      T = **Int**

    value
      implicit_obs : **Int** → **Int**
      implicit_obs(x) ≡ gen(x) + x,

      gen : **Int** → T
      gen(x) ≡ x + 1
  end

▷

scheme I_SPEC =
  class
    type
      T = **Int**

    variable
      t : T

    value
      implicit_obs : **Int** → **read** t **write** t **Int**
      implicit_obs(x) ≡ (gen(x) ; t) + x,

      gen : **Int** → **write** t **Unit**
      gen(x) ≡ t := x + 1
  end

6.4.4 Transforming Pre-Conditions

A pre-condition in an explicit function definition is optional. If present it is of the form:

pre value_expr

The pre-condition is transformed by transforming the constituent value expression according to the transformation rules given in Section 6.4.3. If the pre-condition contains explicit, implicit or hidden generators it cannot be transformed. The only exception to this rule is that function applications taking only value names of the types of interest as arguments can be transformed as the arguments are removed during the transformation.

6.5 Transforming Explicit Value Definitions

An explicit value definition is of the form:

value id : type_expr = value_expr

If the *type_expr* is the type of interest, the explicit value definition is turned into an explicit function definition in which the body of the function is the transformed version of *value_expr*. The function id is the *id* of the explicit value definition. The single typing is constructed such that it agrees with the resulting function. This means that an explicit value definition of the above form is transformed as an explicit function definition of the form:

id : **Unit** → type_expr
id() ≡ value_expr

An example of a transformation of an explicit value definition can be found in Example 6.15.

Example 6.15 – Transforming explicit value definitions

---

scheme A_SPEC =
  class
    type
      T = **Int***

    value

```
        empty : T = ⟨⟩,

        f : Unit → T
        f() ≡ empty
  end
```

▷

```
scheme I_SPEC =
  class
    type
      T = Int*

    variable
      t : T

    value
      empty : Unit → write t Unit
      empty() ≡ t := ⟨⟩,

      f : Unit → write t Unit
      f() ≡ empty()
  end
```

6.6 More Than One Type of Interest

In some cases it is necessary to introduce more than one type of interest. The transformation when dealing with more than one type of interest resembles that of a transformation with one type of interest. The only difference is that there are more types of interest to take care of. Each type of interest is treated as described above without affecting the other types of interest.

Chapter 7

Correctness of Transformation Rules

This chapter contains some ideas [Hax05] for how a notion of correctness of the transformation from applicative into imperative RSL specification can be defined and verified. It should be emphasized that this is an initial investigation an hence not complete and polished.

7.1 General Introduction

In order to be able to verify the correctness of the transformation rules it is necessary to define what a correct transformation is.

An imperative specification SP' is a correct transformation of an applicative specification SP if there is an appropriate relation between the semantics of SP and SP'.

This means that it is necessary to define the semantics of RSL_I specifications. The original semantics of the full RSL, [Mil90], is too complicated for the purpose. Instead, the much more simple semantics defined in [Lin04] for an applicative subset *mRSL* of RSL can be extended to cover the imperative subset RSL_I of RSL. The definition of the semantics is based on institutions. An *institution* is a category theoretic definition of the notion of a logical system.

Section 7.2 defines the notion of an institution and the mathematical background. Section 7.3 gives a definition of what it means for a transformation to be correct and explains how this can be achieved, while Section 7.4 gives an outline of how an institution of RSL_I could be defined. Finally, Section 7.5 offers a small example in order to illustrate the idea of the verification.

7.2 Institutions and their Mathematical Background

In this section a category theoretic definition of the notion of a logical system is given.

The following definitions can be found in the literature, e.g. [ST], [ST02], [Tar02] and [Lin04], but are included in order for the chapter to be self contained.

In order to be able to define the notion of a logical system it is necessary to define the mathematical notion of a *category* and of a *functor*.

Definition 7.1. A *category* $\mathbf{K}$ consists of:

- a collection $Obj(\mathbf{K})$ of $\mathbf{K}$-*objects*
- for each $A, B \in Obj(\mathbf{K})$, a collection $\mathbf{K}(A, B)$ of $\mathbf{K}$-*morphisms* from A to B
- for each $A, B, C \in Obj(\mathbf{K})$, a *composition operation* $\circ : \mathbf{K}(A, B) \times \mathbf{K}(B, C) \to \mathbf{K}(A, C)$

such that

1. for all $A, B, A', B' \in Obj(\mathbf{K})$, if $\langle A, B\rangle \neq \langle A', B'\rangle$ then $\mathbf{K}(A, B) \cap \mathbf{K}(A', B') = \emptyset$
2. for each $A \in Obj(\mathbf{K})$, there is a morphism $id_A \in \mathbf{K}(A, A)$ such that $id_A \circ g = g$ for all morphisms $g \in \mathbf{K}(A, B)$ and $f \circ id_A = f$ for all morphisms $f \in \mathbf{K}(A, B)$
3. for any $f \in \mathbf{K}(A, B)$, $g \in \mathbf{K}(B, C)$ and $g \in \mathbf{K}(C, D)$, $f \circ (g \circ h) = (f \circ g) \circ h$

Definition 7.2. A *functor* $\mathbf{F} : \mathbf{K1} \to \mathbf{K2}$ from a category $\mathbf{K1}$ to a category $\mathbf{K2}$ consists of:

- a function $\mathbf{F}_{Obj} : |\mathbf{K1}| \to |\mathbf{K2}|$
- for each $A, B \in |\mathbf{K1}|$, a function $\mathbf{F}_{A,B} : \mathbf{K1}(A, B) \to \mathbf{K2}(\mathbf{F}_{Obj}(A), \mathbf{F}_{Obj}(B))$

such that

- $\mathbf{F}$ preserves identities: $\mathbf{F}_{A,A}(id_A) = id_{\mathbf{F}_{Obj(A)}}$ for all objects $A \in |\mathbf{K}|$
- $\mathbf{F}$ preserves composition: for all morphisms $f : A \to B$ and $g : B \to C$ in $\mathbf{K1}$, $\mathbf{F}_{A,C}(f \circ g) = \mathbf{F}_{A,B}(f) \circ \mathbf{F}_{B,C}(g)$

The concept of a logical system is formalized using the concept of an *institution*. An institution consists of signatures, sentences, models and a satisfaction relation such that certain conditions are met.

A *signature* provides the vocabulary for writing sentences, that is the type names and operation names. A *sentence* describes properties. A *model* describes the meaning.

Definition 7.3. An *institution* consists of:

- a category $Sign$ of signatures. That is a set of signatures and a set of signature morphism $\sigma : \Sigma \to \Sigma'$, mapping from one signature $\Sigma \in |Sign|$ to another signature $\Sigma' \in |Sign|$.
- a functor $Sen : Sign \to Set$ that maps each signature $\Sigma \in |Sign|$ to a set of sentences over that signature $Sen(\Sigma)$ and each signature morphism σ to a sentence morphism $Sen(\sigma) : Sen(\Sigma) \to Sen(\Sigma')$.
- a functor $Mod : Sign \to Cat$ that maps each signature $\Sigma \in |Sign|$ to the category of models over that signature $Mod(\Sigma)$ and each signature morphism $\sigma : \Sigma \to \Sigma'$ to a model morphism $Mod(\sigma) : Mod(\Sigma') \to Mod(\Sigma)$.
- for each signature $\Sigma \in |Sign|$, a satisfaction relation $\models_\Sigma \subseteq |Mod(\Sigma)| \times Sen(\Sigma)$

such that for each $m' \in |Mod(\Sigma')|$, $\sigma : \Sigma \to \Sigma'$ in $Sign$ and $e \in Sen(\Sigma)$

$$m' \models_\Sigma Sen(\sigma)(e) \quad \text{iff} \quad Mod(\sigma)(m') \models_\Sigma e$$

This requirement is known as the *satisfaction condition*.

Sentences over a signature Σ, that is sentences in $Sen(\Sigma)$, are called Σ-*sentences* and models over the signature Σ, i.e. models in $|Mod(\Sigma)|$, are referred to as Σ-*models*.

The notation $m \models_\Sigma e$ means that the Σ-model m satisfies the Σ-sentence e. For a set of Σ-models M and a Σ-sentence e, $M \models_\Sigma e$ if and only if $m \models_\Sigma e$ for all $m \in M$. For a Σ-model m and a set of Σ-sentences E, $m \models_\Sigma E$ if and only if $m \models_\Sigma e$ for all $e \in E$.

Definition 7.4. The *theory* $Th(M)$ of a set of models $M \subseteq |Mod(\Sigma)|$ is the set of all Σ-sentences satisfied by each model in M:

$$Th(M) = \{\varphi \in Sen(\Sigma) \mid M \models_\Sigma \varphi\}$$

Definition 7.5. A Σ-*presentation* is a pair $\langle \Sigma, \Phi \rangle$ where $\Sigma \in |Sign|$ and $\Phi \subseteq Sen(\Sigma)$. $Mod(\Sigma, \Phi) \subseteq |Mod(\Sigma)|$ is a *model* of a Σ-presentation $\langle \Sigma, \Phi \rangle$ iff $Mod(\Sigma, \Phi) \models_\Sigma \varphi$ for all $\varphi \in \Phi$.

The notation $\langle \Sigma, \Phi \rangle \models_\Sigma \varphi$ means that $Mod(\Sigma, \Phi) \models_\Sigma \varphi$, i.e. $\varphi \in Th(Mod(\Sigma, \Phi))$.

Definition 7.6. A *presentation morphism* $\sigma : \langle \Sigma, \Phi \rangle \to \langle \Sigma', \Phi' \rangle$ is a signature morphism $\sigma : \Sigma \to \Sigma'$ such that $Sen(\sigma)(\varphi) \in \Phi'$ for every $\varphi \in \Phi$.

Lemma 7.1. It follows from the satisfaction condition of an institution that the following holds:

1. $Mod(\sigma)(Mod(\Sigma', \Phi')) \subseteq Mod(\Sigma, \Phi)$
2. $Sen(\sigma)(Th(Mod(\Sigma, \Phi))) \subseteq Th(Mod(\Sigma', \Phi'))$

for $\sigma : \langle \Sigma, \Phi \rangle \to \langle \Sigma', \Phi' \rangle$ being a presentation morphism [Tar86].

7.3 Definition of Correctness

In this section it is explained how correctness of the transformation rules can be achieved by defining an appropriate institution for RSL_I. This section only provides the requirements to such an institution. An outline of an actual institution of RSL_I is given in Section 7.4.

An RSL specification can be interpreted as a presentation, that is a specification of the form $SP = \langle \Sigma, \Phi \rangle$, as any specification is shorthand for a signature and some axioms, cf. [Gro92].

The *semantics* of a specification $SP = \langle \Sigma, \Phi \rangle$ is the class of models $Mod(\Sigma, \Phi)$ satisfying all the sentences of Φ.

The transformation of a specification as defined in Chapter 6 consists implicitly of a signature morphism and a corresponding sentence morphism as the signature of an RSL specification can be extracted by removing all sentences from the specification. This means that a transformation can be interpreted as a presentation morphism. This is illustrated in Figure 7.1.

$$\begin{tikzcd} SP \arrow[r, "\sigma"] \arrow[d, equal] & SP' \arrow[d, equal] \\ \langle \Sigma, \Phi \rangle \arrow[r, "\sigma"] & \langle \Sigma', \Phi' \rangle \end{tikzcd}$$

Figure 7.1: Transformation of a specification considered as a presentation morphism

This means that defining a model morphism $Mod(\sigma)$ such that the satisfaction condition holds a definition of an institution for RSL_I is obtained.

If this can be achieved the two properties given in Lemma 7.1 applies to any presentation morphism within RSL_I that is for any transformation within RSL_I.

The first property of Lemma 7.1 means that the semantics of the transformed models of the transformed specification is a subset of the models of the original specification. This means that the desired relation between the semantics of the specifications of a transformation is achieved. This is illustrated in Figure 7.2, where $\langle\Sigma, \Phi\rangle$ corresponds to an applicative specification, $\langle\Sigma', \Phi'\rangle$ corresponds to an imperative specification and σ to the transformation from applicative into imperative specification.

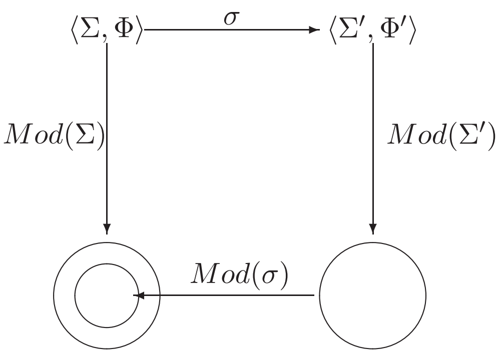

Figure 7.2: The relation between the semantics of an applicative specification and the corresponding imperative specification

The second property of Lemma 7.1 means that if it is proven that a sentence e holds in the specification $\langle\Sigma, \Phi\rangle$, then the transformed version of e, named e', will hold in the transformed specification $\langle\Sigma', \Phi'\rangle$ as illustrated in Figure 7.3.

$$\begin{tikzcd}
\langle\Sigma, \Phi\rangle \arrow[d, "\sigma"'] & \models_\Sigma & e \arrow[d, "Sen(\sigma)"] \\
\langle\Sigma', \Phi'\rangle & \models_\Sigma & e'
\end{tikzcd}$$

Figure 7.3: The condition for a transformation to be semantically correct

The result is that if a property is proven for an applicative specification it is not necessary to prove that the same property holds in an imperative specification obtained by a transformation of the applicative specification. This complies with the RAISE Development Method and is exactly what should be obtained.

This means that the transformation rules defined in Chapter 6 are correct if an institution of the RSL_I subset can be defined such that the transformation rules constitute signature and sentence morphisms such that the satisfaction condition holds, cf. Lemma 7.1.

7.4 An Institution for RSL$_I$

In [Lin04] and [LH04] an institution for mRSL is defined. As mentioned above mRSL is an applicative subset of RSL. It is in the following outlined how an institution for the imperative subset RSL$_I$ can be defined and in that way an outline of a verification of the transformation rules is given. To simplify matters sort definitions, variant definitions and short record definitions are not considered, nor are hidden and implicit observers or hidden and implicit generators. In a proper definition they must be included. Furthermore, only maximal type expressions are considered.

Signatures

An *RSL$_I$ signature* is a triple $\Sigma = \langle A, OP, V\rangle$ where

- $A = Id \xrightarrow[m]{} T(A, V)$ is a map, which represents a set of non-cyclic abbreviation definitions. It maps type identifiers to the type expressions they abbreviate.
- $OP = Id \xrightarrow[m]{} T(A, V)$ is a map, which represents a set of value declarations. It maps value identifiers to their declared type expressions.
- $V = Id \xrightarrow[m]{} T(A, V)$ is a map, which represents a set of variable definitions. It maps variable identifiers to the corresponding type expressions.

such that **dom** A, **dom** OP and **dom** V are disjoint.

$T(A, V)$ is a set of RSL type expressions referring to type identifiers in **dom** A and variable identifers in **dom** V.

A signature is said to be applicative if $V = [\,]$, since there are no variable declarations in the applicative case.

Signature Morphisms

An *RSL$_I$ signature morphism* $\sigma : \Sigma \to \Sigma'$ is defined as the transformation from the transformable specification SP to the specification SP' such that $Sign(SP) = \Sigma$ and $Sign(SP') = \Sigma'$. Composition is defined in the obvious way and the identities are defined such that a signature is mapped to itself.

Recalling the definition of the transformation rules, note that for a signature morphism $\sigma : \Sigma \to \Sigma'$, where $\Sigma = \langle A, OP, V\rangle$ and $\Sigma' = \langle A', OP', V'\rangle$:

- $A = A'$, since the abbreviation definitions are the same
- **dom** OP = **dom** OP', since the value declarations are transformed into value declarations of the same names
- **dom** $V \subseteq$ **dom** V', since variables cannot be removed, only added

Sentences

An *RSL$_I$ Σ-sentence* is a boolean value expression, which is well-formed according to the static semantics of RSL and is within the subset of RSL$_I$.

Sentence Morphisms

As a specification represents a signature and a collection of sentences, the transformation rules do not only define a transformation between the signatures of the specifications but also a transformation between sentences. An *RSL$_I$ sentence morphism* $Sen(\sigma) : Sen(\Sigma) \rightarrow Sen(\Sigma')$ should be defined such that it respects the defined transformation rules for sentences.

Sentence Functor

The functor $Sen : Sign \rightarrow Set$ is a functor that maps signatures Σ in $Sign$ to the set $Sen(\Sigma)$ of Σ-sentences and maps each signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ in $Sign$ to the sentence morphism $Sen(\sigma) : Sen(\Sigma) \rightarrow Sen(\Sigma')$ as described above.

Models

Let $\Sigma = \langle A, OP, V \rangle$ be a signature. An *RSL$_I$ Σ-model* is a triple $m = \langle m_A, m_{OP}, s_{init} \rangle$ where

- $m_A : Id \xrightarrow[m]{} Types$, such that **dom** $m_A =$ **dom** A and $m_A(a) = M(A^*(a))$. That is m_A maps type identifiers into the types their abbreviations denote.
- $m_{OP} : Id \xrightarrow[m]{} Values$, such that **dom** $m_{OP} =$ **dom** OP and $m_{OP}(op) \in M(OP(op))$. That is m_{OP} maps value identifiers into the values they denote. These values must be in the types denoted by their type expressions.
- $s_{init} \in Store$. That is s_{init} denotes the initial store that maps variable identifiers to their initial values.

where

- A^* is a function that takes a type expression to its canonical form, i.e. a type expression not referring to any type identifiers, by recursively expanding type identifiers according to their abbreviations.
- $M : T(A, V) \rightarrow Types$ is a meaning function mapping type expressions into the types they denote: $M(t) = Value_{A^*(t)}$.

  A type is a set of values, also known as a *value domain*. $Value_t$ denotes the value domain for a type expression $t \in T(\emptyset, V)$, e.g.:

- $Value_{\mathbf{Bool}} = \mathbb{B} = \{tt, ff\}$
- $Value_{\mathbf{Int}} = \mathbb{Z}$
- $Value_{\mathbf{Real}} = \mathbb{R}$
- $Value_{\mathbf{Unit}} = \{d\_unit\}$
- $Value_{t_1 \times t_2} = Value_{t_1} \times Value_{t_2}$
- $Value_{t_1 \overset{\sim}{\rightarrow} \mathbf{read}\ rs\ \mathbf{write}\ ws\ t_2} =$

  $\{$

  $f \in Value_{t_1} \rightarrow Effect_{t_2} \mid$
  $read\_conforms(f, rs) \ \wedge \ write\_conforms(f, ws)$

  $\}$

  if $t_1, t_2 \in T(\emptyset, V)$ are maximal.
  That is a value of type $t_1 \overset{\sim}{\rightarrow} \mathbf{read}\ rs\ \mathbf{write}\ ws\ t_2$ is a function f from values of type t_1 to effects of type t_2 such that f only reads variables in rs and writes variables in ws.

$read\_conforms : (Value_{t1} \rightarrow Effect_{t2}) \times V\text{-}\mathbf{set} \rightarrow \mathbf{Bool}$ returns true for a function f and a set of **read** variables, if f returns the same result for any value and any two stores that are equivalent with respect to the **read** variables. Note that compared to the RSL language definition, $read\_conforms$ does not take $rs \cup ws$ as an argument, only rs. This solution is chosen in order to ease the definition of the model morphisms.

$write\_conforms : (Value_{t1} \rightarrow Effect_{t2}) \times V\text{-}\mathbf{set} \rightarrow \mathbf{Bool}$ returns true for a function f and a set of **write** variables, if f does not alter the variables that are not in the set of **write** variables.

- $Types = \{Value_t \mid t \in T(\emptyset, V)\}$. That is a set of all types.

- $Values = \bigcup_{t \in T(\emptyset,V)} Value_t$. That is a set of all values.

- $Store = \{\varrho : \mathbf{dom}\, V \rightarrow Values \mid \forall v \in \mathbf{dom}\, V \bullet \varrho(v) \in M(V(v))\}$. That is a store ϱ maps each variable identifier v of V to a value in the type denoted by the type expression $V(v)$ of v.

- $Procs_t = (Store \times Value_t) \cup \{\perp\}$ for $t \in T(\emptyset, V)$. That is, a process is either a pair consisting of a store and a value or a non diverging process represented by $\perp$.

- $Effect_t = Store \rightarrow Procs_t$ for $t \in T(\emptyset, V)$. An effect is a function that returns a process for a given store. Effects are used as denotations for value expressions (including function applications) with respect to a given model: when evaluating it in a store, it either diverges or it updates the store and returns a value.

Model Morphisms

Let $\Sigma = \langle A, OP, V\rangle$ and $\Sigma' = \langle A', OP', V'\rangle$ be signatures, $\sigma : \Sigma \rightarrow \Sigma'$ a signature morphism and $m' = \langle m'_A, m'_{OP}, s'_{init}\rangle$ a Σ'-model. The model morphism $Mod(\sigma) : Mod(\Sigma') \rightarrow Mod(\Sigma)$ of m' is the Σ-model $m = \langle m_A, m_{OP}, s_{init}\rangle$ where

- $m_A = m'_A$
- for m_{OP}:
  - $\mathbf{dom}\, m_{OP} = \mathbf{dom}\, OP = \mathbf{dom}\, OP' = \mathbf{dom}\, m'_{OP}$
  - $m_{OP}(c)$ is defined in terms of $m'_{OP}(c)$ for $c \in \mathbf{dom}\, m_{OP}$. Some examples of this is given below.
- $s_{init} = s'_{init} \backslash (\mathbf{dom}\ V' \backslash \mathbf{dom}\ V)$. Note that $s_{init} = [\,]$ in the applicative case.

In the following examples of how $m_{OP}(f)$ is defined in terms of $m'_{OP}(f)$ are given. In the examples it is assumed that the signature Σ is applicative and the σ-transformation only has one type of interest t_v with corresponding variable v. Three cases are considered.

1. **Observer**

   $$\text{f} : \text{te}_1 \times \text{t}_v \times \text{te}_2 \xrightarrow{\sim} \text{t}_{res} \quad \overset{\sigma}{\curvearrowright} \quad \text{f} : \text{te}_1 \times \text{te}_2 \xrightarrow{\sim} \mathbf{read}\ \text{v}\ \text{t}_{res}$$

   for all $(ve_1, x, ve_2) \in Value_{A^*(te_1 \times t_v \times te_2)}$:

   $\text{m}_{OP}(\text{f})(\text{ve}_1, \text{x}, \text{ve}_2)([\,]) \equiv$
   $\quad$ **case** $\text{m}'_{OP}(\text{f})(\text{ve}_1, \text{ve}_2)([\text{v} \mapsto \text{x}])$ **of**
   $\quad\quad \perp \rightarrow \perp,$
   $\quad\quad (\text{v}_{res}, [\text{v} \mapsto \text{x}']) \rightarrow (\text{v}_{res}, [\,])$
   $\quad$ **end**

   Note that $[v \mapsto x] = [v \mapsto x']$ as $m'_{OP}(f)$ must write conform with the empty set of **write** variables.
   Explanation: $m_{OP}(f)$ returns **chaos** if $m'_{OP}(f)$ returns **chaos**. Otherwise $m_{OP}(f)$ returns the same result as $m'_{OP}(f)$. When evaluating $m'_{OP}(f)$ a store $s = [v \mapsto x]$ is kept reflecting the actual value x of v. This value is not altered during the evaluation of f, since f is an observer.

2. **Generator**

   $$\text{f} : \text{t}_{arg} \xrightarrow{\sim} \text{te}_1 \times \text{t}_v \times \text{te}_2 \quad \overset{\sigma}{\curvearrowright} \quad \text{f} : \text{t}_{arg} \xrightarrow{\sim} \mathbf{write}\ \text{v}\ \text{te}_1 \times \text{te}_2$$

for all $(ve_1, x, ve_2) \in Value_{A^*(te_1 \times t_v \times te_2)}$:

$\mathrm{m}_{OP}(\mathrm{f})(\mathrm{ve}_{arg})([\,]) \equiv$
case $\mathrm{m'}_{OP}(\mathrm{f})(\mathrm{ve}_{arg})([\,\mathrm{v} \mapsto \mathrm{dv}\,])$ **of**
$\perp \to \perp$,
$((\mathrm{ve}_1,\ \mathrm{ve}_2),\ [\,\mathrm{v} \mapsto \mathrm{dv'}\,]) \to ((\mathrm{ve}_1,\ \mathrm{dv'},\ \mathrm{ve}_2),\ [\,])$
end

Note that **dom** $[v \mapsto dv] =$ **dom** $[v \mapsto dv'] = \{v\}$.
Explanation: $m_{OP}(f)$ returns **chaos** if $m'_{OP}(f)$ returns **chaos**. Otherwise $m_{OP}(f)$ returns the same result as $m'_{OP}(f)$, but with the actual value dv' of v included. When evaluating $m'_{OP}(f)$ a store $s = [v \mapsto dv]$ is kept reflecting the actual value dv of v. This value is altered during the evaluation of f, since f is a generator.

3. **Combined observer and generator**

$\mathrm{f} : \mathrm{te}_1 \times \mathrm{t}_v \times \mathrm{te}_2 \xrightarrow{\sim} \mathrm{te}_3 \times \mathrm{t}_v \times \mathrm{te}_4 \quad \overset{\sigma}{\curvearrowright}$
$\mathrm{f} : \mathrm{te}_1 \times \mathrm{te}_2 \xrightarrow{\sim}$ **read** v **write** v $\mathrm{te}_3 \times \mathrm{te}_4$

for all $(ve_1, x, ve_2) \in Value_{A^*(te_1 \times t_v \times te_2)}$, $(ve_3, x, ve_4) \in Value_{A^*(te_3 \times t_v \times te_4)}$:

$\mathrm{m}_{OP}(\mathrm{f})(\mathrm{ve}_1,\ \mathrm{x},\ \mathrm{ve}_2)([\,]) \equiv$
case $\mathrm{m'}_{OP}(\mathrm{f})(\mathrm{ve}_1,\ \mathrm{ve}_2)([\,\mathrm{v} \mapsto \mathrm{x}\,])$ **of**
$\perp \to \perp$,
$((\mathrm{ve}_3,\ \mathrm{ve}_4),\ [\,\mathrm{v} \mapsto \mathrm{x'}\,]) \to ((\mathrm{ve}_3,\ \mathrm{x'},\ \mathrm{ve}_4),\ [\,])$
end

Explanation: $m_{OP}(f)$ returns **chaos** if $m'_{OP}(f)$ returns **chaos**. Otherwise $m_{OP}(f)$ returns the same result as $m'_{OP}(f)$, but with the actual value of x' of v included. When evaluating $m'_{OP}(f)$ a store $s = [v \mapsto x]$ is kept reflecting the actual value x of v. This value is altered during the evaluation of f, since f is a generator.

Model Category

Composition is defined the obvious way. Identities are defined such that a model is mapped to itself.

Model Functor

The functor $Mod : Sign \to Cat$ is a functor that maps $Sign$ to the category $Mod(\Sigma)$ of Σ-models and maps each signature morphism $\sigma : \Sigma \to \Sigma'$ in $Sign$ to the model morphism $Mod(\sigma) : Mod(\Sigma') \to Mod(\Sigma)$ which is described above.

Satisfaction Relation

Then the satisfaction relation $\models_\Sigma$ must be defined, such that the satisfaction condition can be formulated. The dynamic semantics of mRSL value expressions was defined in [Lin04]. This should be extended to cover the whole subset of RSL_I.

Proof of Satisfaction Condition

The last step is to prove that the satisfaction condition holds. The satisfaction condition is proved in [Lin04] for the institution of mRSL. Most likely this can be carried over to the institution of RSL_I.

If this is the case the transformation rules are correct. This means that the transformation of an applicative RSL specification into an imperative RSL specification using the transformation rules will be a correct development step.

7.5 Example

The following simple example illustrates the ideas of this chapter.

First an applicative specification *A_SPEC* is defined:

Specification 7.1 – Applicative specification *A_SPEC*

---

scheme A_SPEC =
  class
    type T = **Bool**

    value
      f : T → T
      f(x) ≡ ∼ x
  end

A_SPEC can be regarded as a signature Σ and a sentence e.

The signature $\Sigma = \langle A, OP, V \rangle$ of *A_SPEC* is defined as follows:

- $A = [T \mapsto \textbf{Bool}]$, reflecting the type of T
- $OP = [f \mapsto T \rightarrow T]$, reflecting the type of the function f
- $V = [\,]$, as there are no variable definitions in *A_SPEC*

The sentence is defined as $e = \forall\ \mathrm{x} : \mathrm{T} \bullet \mathrm{f(x)} \equiv \sim \mathrm{x}$

Then the model $m = \langle m_A, m_{OP}, s_{init} \rangle$ of *A_SPEC* can be defined:

- $m_A = [T \mapsto Value_{\mathbf{Bool}}] = [T \mapsto \mathbb{B}]$
- $m_{OP} = [f \mapsto \lambda x : \mathbb{B} \bullet\ \sim x]$
- $s_{init} = [\,]$, as there are no variable definitions in A_SPEC

The semantics of the satisfaction relation is defined such that $m \models_\Sigma A\_SPEC$.

Then the imperative specification I_SPEC is obtained by transforming A_SPEC using the transformation rules defined in Chapter 6 such that $A\_SPEC \rhd I\_SPEC$. The type of interest is T and the corresponding variable is named t.

Specification 7.2 – Imperative specification I_SPEC

---

scheme I_SPEC =
  class
    type T = **Bool**

    variable t : T

    value
      f : **Unit** → **read** t **write** t **Unit**
      f() ≡ t := ∼ t
  end

I_SPEC can be regarded as a signature $\Sigma' = \sigma(\Sigma)$ and a sentence $e' = Sen(\sigma)(e)$.

The signature $\Sigma' = \langle A', OP', V' \rangle$ of I_SPEC is defined as follows:

- $A' = [T \mapsto \mathbf{Bool}]$, reflecting the type of T
- $OP' = [f \mapsto \mathbf{Unit} \to \mathbf{read}\ t\ \mathbf{write}\ t\ \mathbf{Unit}]$, reflecting the type of the function f
- $V' = [t \mapsto T]$, reflecting the type of the variable t

The sentence is defined as $e' = \Box$ f() ≡ t := ∼ t

Then the model $m' = \langle m'_A, m'_{OP}, s'_{init} \rangle$ of I_SPEC can be defined:

- $m'_A = [T \mapsto Value_{\mathbf{Bool}}] = [T \mapsto \mathbb{B}]$
- $m'_{OP} = [f \mapsto \lambda u : \mathbf{Unit} \bullet s' : Store \bullet ([t \mapsto \sim s'(t)], d\_unit)]$

- $s'_{init} = [t \mapsto t_{init}]$ where $t_{init} \in \mathbb{B}$ that is t can be mapped to either **true** or **false**. This means that there are two models of I_SPEC only distinguishable by the initial value of the store.

The semantics of the satisfaction relation is defined such that $m' \models_\Sigma I\_SPEC$.

Then $m'' = \langle m''_A, m''_{OP}, s''_{init} \rangle = Mod(\sigma)(m')$ is calculated:

- $m''_A = Mod(\sigma)(m'_A) = m'_A = [T \mapsto Value_{\mathbf{Bool}}] = [T \mapsto \mathbb{B}]$
- for m''_{OP} the following holds:
  - $\mathbf{dom}\, m''_{OP} = \mathbf{dom}\, m'_{OP} = \{f\}$
  - $\mathrm{m''}_{OP}(\mathrm{f})(\mathrm{x})([\,]) \equiv$
    case $\mathrm{m'}_{OP}(\mathrm{f})(\mathbf{Unit})([\,\mathrm{t} \mapsto \mathrm{x}\,])$ **of**
    $(\mathbf{Unit}, [\,\mathrm{t} \mapsto \sim \mathrm{x}\,]) \to (\sim \mathrm{x}, [\,])$
    end

    where the case $\perp$ is left out because f is total. This means that $m''_{OP} = [f \mapsto \lambda x : \mathbb{B} \cdot \sim x]$
- $s''_{init} = Mod(\sigma)(s'_{init}) = s'_{init} \backslash (\mathbf{dom}\, V' \backslash \mathbf{dom}\, V) = [t \mapsto t_{init}] \backslash (\{t\} \backslash \{\}) = [t \mapsto t_{init}] \backslash \{t\} = [\,]$

As can be seen

$$m = m'' = Mod(\sigma)(m')$$

This implies that

$$Mod(\sigma)(m') \models_\Sigma e \quad \text{and} \quad m' \models_\Sigma e'$$

where $e' = Sen(\sigma)(e)$. This means that the satisfaction condition holds for the instance of a sentence and a model considered.

In order to prove that it holds for the entire RSL_I subset, similar considerations should be made for the entire RSL_I subset, e.g. all sentences and all models. This can be obtained by basing the proofs on the structure of the sentences as done in [Lin04].

Chapter 8

Specifications

This chapter describes the different RSL specifications made during the project. Not every specification is fully described. The purpose of the chapter is to give a good understanding of the specifications by describing the overall structures of the specifications and by showing representative snippets of the specifications.

The transformer is developed using the RSL2Java tool, which can translate a subset RSL_1 of RSL into Java. This RSL2Java tool is further described in Chapter 9.

8.1 Overview of the Different Specifications

The main functionality of the transformer is the transformation from abstract syntax tree representation, AST, of an applicative specification into abstract syntax tree representation of a corresponding imperative specification, if the transformation is possible.

In order to formally specify this transformation two specifications are necessary:

1. A specification of the RSL AST, which is described in Section 8.2.
2. A specification of the transformation, which is described in Section 8.3.

In order to develop the actual transformer the RSL2Java tool is used to generate an executable Java program from the specifications. In order to be able to translate the specifications into Java using the RSL2Java tool, the specifications have to be written within RSL_1. The specification of the RSL AST is written within RSL_1, but the specification of the transformation is written within RSL. This means that the specification of the transformation has to be rewritten such that it is within the RSL_1 subset. A description of how the transformer specification is translated into RSL_1 can be found in Section 8.4.

The rewriting of the specification from RSL to RSL_1 requires some changes. The correctness of these changes is discussed in Section 8.5.

8.2 Specification of the RSL AST

The specification of the RSL AST RSLAst_Module2.rsl, which can be found in Appendix D, is written in RSL_1, such that the specification can be transformed into Java using the RSL2Java tool and thereby be used as part of the transformer.

The specification of the RSL AST consists purely of type definitions that are either variant definitions or short record definitions. These type definitions describe the structure of an RSL AST. The specification of the RSL AST is based on the concrete syntax summary of RSL offered in [Gro92, pp. 371-380] and the changes to RSL which can be found in [Gro95]. The construction of the RSL AST is straight forward when following these syntax summaries.

In RSL graphical symbols like parenthesizes and commas are used to distinguish between the different kinds of RSL constructs. When the type of an RSL construct is determined these symbols are superfluous and can be discarded. This is exploited in the specification of the RSL AST where only the necessary information is kept.

Otherwise the syntax of RSL consists of terminals T and nonterminals NT. Terminals are indivisible entities, whereas nonterminals are symbols that can be expressed in terms of other symbols, terminals or nonterminals.

The relationship between the RSL syntax and the RSL AST specification is described in the following.

Syntax of the form

$$\mathsf{NT} ::= \mathsf{NT}_1 \ldots \mathsf{NT}_n$$

becomes

$$\text{NT} :: \text{nt}_1 : \text{NT}_1 \ldots \text{nt}_n : \text{NT}_n$$

whereas

$$\begin{aligned} \mathsf{NT} &::= \mathsf{NT}_1 | \ldots | \mathsf{NT}_n \\ \mathsf{NT}_1 &::= \mathsf{NT}_{11} \ldots \mathsf{NT}_{1n} \\ &\vdots \\ \mathsf{NT}_n &::= \mathsf{NT}_{n1} \ldots \mathsf{NT}_{nn} \end{aligned}$$

becomes

$$
\begin{aligned}
&\mathrm{NT} == \\
&\quad \mathrm{NT}_1(\mathrm{nt}_{11} : \mathrm{NT}_{11} \ldots \mathrm{nt}_{1n} : \mathrm{NT}_{1n}) \mid \\
&\quad \vdots \\
&\quad \mathrm{NT}_n(\mathrm{nt}_{n1} : \mathrm{NT}_{n1} \ldots \mathrm{nt}_{nn} : \mathrm{NT}_{nn}), \\
\\
&\mathrm{NT}_{11} :: \ldots, \\
&\vdots \\
&\mathrm{NT}_{nn} :: \ldots
\end{aligned}
$$

When the alternatives are terminals as in

$$\mathsf{NT} \quad ::= \quad \mathsf{T}_1 | \ldots | \mathsf{T}_n$$

it is specified as

$$\mathrm{NT} == \mathrm{T}_1 \mid \ldots \mid \mathrm{T}_n$$

An example of some RSL syntax and the corresponding RSL AST specification can be found in Example 8.1.

Example 8.1 – Specification of the RSL AST based on the syntax of RSL

RSL syntax:

$$
\begin{aligned}
\mathsf{value\_expr} \quad &::= \quad \mathit{value\_or\_variable}-\mathsf{name} \quad | \\
&\qquad \mathsf{product\_expr} \quad | \\
&\qquad \mathsf{infix\_expr} \\
\\
\mathsf{name} \quad &::= \quad \mathsf{id} \\
\\
\mathsf{product\_expr} \quad &::= \quad (\mathsf{value\_expr}-\mathrm{list}) \\
\\
\mathsf{infix\_expr} \quad &::= \quad \mathsf{value\_expr} \quad \mathsf{infix\_op} \quad \mathsf{value\_expr} \\
\\
\mathsf{infix\_op} \quad &::= \quad + \quad | \\
&\qquad - \quad | \\
&\qquad =
\end{aligned}
$$

RSL specification:

```
ValueExpr ==
    Make_ValueOrVariableName(
        value_or_variable_name : ValueOrVariableName) |
    ProductExpr(value_expr_list : ValueExpr*) |
    ValueInfixExpr(
        left : ValueExpr,
        op : InfixOperator,
        right : ValueExpr),

ValueOrVariableName :: id : Id,

InfixOperator ==
    PLUS |
    MINUS |
    EQUAL,

Id :: getText : Text
```

As can be seen in Example 8.1 the parenthesizes and commas of the product expression are removed in the RSL AST specification.

8.3 Specification of the Transformer

The specification of the transformer Transformer.rsl described in this section is written in RSL. It specifies how an applicative version of an RSL AST can be transformed into the corresponding imperative version of the RSL AST. The full specification of the transformer in RSL can be found in Appendix C.

In the following a description of four maps central to the transformer specification will be given and then the structure of the specification will be described.

8.3.1 The Maps

In order to do the transformation some extra types are necessary. These extra types are described in the following.

The TRANS Map

The map *TRANS* is defined as follows:

$$\text{TRANS} = \text{Id} \underset{m}{\to} \text{Id}$$

The TRANS map maps each type of interest to the corresponding variable name. This map is established from the user input and is used both during the check for transformability and during the transformation phase in order to figure out what the type names of the types of interest are and in order to do proper variable replacements.

The TYPINGS Map

The map *TYPINGS* is defined as follows:

$$\text{TYPINGS} = \text{Id} \underset{m}{\overrightarrow{}} \text{Id}^*$$

The TYPINGS map maps type name identifiers to the list of type identifiers obtained by recursively expanding the right hand side of the type definitions. This means that the type definition:

$$\begin{aligned} &\text{A} = \text{B} \times \text{C},\\ &\text{B} = \text{D},\\ &\text{C} = \text{A},\\ &\text{D} = \textbf{Int} \end{aligned}$$

results in the TYPINGS map:

$$\begin{aligned} &\text{TYPINGS} =\\ &\quad [\ \text{A} \mapsto \langle \text{B, C, D, A} \rangle,\\ &\quad\quad \text{B} \mapsto \langle \text{D} \rangle,\\ &\quad\quad \text{C} \mapsto \langle \text{A, B, C, D} \rangle,\\ &\quad\quad \text{D} \mapsto \langle\rangle\ \ ] \end{aligned}$$

The TYPINGS map is used to check that types of interests are not recursively defined and that they are not part of sets, lists and maps. The RSL type checker only checks that types are not recursively defined through abbreviation type definitions, while recursion through union, short record and variant type definitions is allowed. For example, a type definition of the form:

Elem,
Collection == empty | add(Elem, Collection)

is allowed in RSL but not allowed when using the transformer.

The value of the TYPINGS map is computed in the beginning of the transformation in the function *TRClassExpr* and this value is not altered during the transformation.

The FUNC Map

The map, *FUNC*, is defined as follows:

$$
\begin{array}{l}
\text{FUNC} = \text{Id} \underset{m}{\overrightarrow{\ \ }} \text{FuncSpec}, \\
\text{FuncSpec} :: \\
\quad \text{type\_expr} : \text{TypeExpr} \\
\quad \text{value\_expr} : \text{ValueExpr} \\
\quad \text{read\_list} : \text{Access}^* \\
\quad \text{write\_list} : \text{Access}^*
\end{array}
$$

Access is defined in the RSL AST as follows:

$$
\begin{array}{l}
\text{Access} == \\
\quad \text{AccessValueOrVariableName(} \\
\qquad \text{variable\_name} : \text{ValueOrVariableName),} \\
\text{ValueOrVariableName} :: \text{id} : \text{Id}, \\
\text{Id} :: \text{getText} : \textbf{Text}
\end{array}
$$

The FUNC map maps function identifiers to the declared type of the function, the body value expression of the function declaration, a list of the types of interest that are read from during the evaluation of the function and a list of the types of interest that are written to during the evaluation of the function. The two access lists contain all types of interest accessed during execution of the function, even if these types of interest are accessed through function applications. These lists are established using the functions *getAccessObs* and *getAccessGen*, which recursively investigate the function and any other functions which may be applied in the function itself. The recursion stops when a function has already been evaluated.

The FUNC map is used both during the check for transformability and during the transformation phase. It is used to decide whether application expressions are function application expression or map or list application expressions, it is used to determine the expected type of the arguments and results of function applications and it is used to establish access descriptors in the single typings of the functions.

The FUNC map is established in the beginning of the transformation in the function *TRClassExpr* and does not change its value during the transformation.

The ENV Map

The map *ENV* is defined as follows:

$$
\text{ENV} = \text{Id} \underset{m}{\overrightarrow{\ \ }} \text{Binding}
$$

The ENV map maps types of interest to the value names under which they are known at a certain point of execution. The ENV map is related to a function and not to the entire specification . The initial value of the ENV map is computed both in *CheckValueDef* and *TRValueDef* when dealing with explicit function definitions,

The ENV map is used to check if the special scope rules defined in Chapter 5 are fulfilled. Furthermore, it is used to decide which value names are of the type of interest and should be replaced with variables.

When deciding if a value expression is transformable the following idea is pursued: A type identifier is mapped to a corresponding value name in ENV map, when the value name is in the scope. When the value name gets out of scope the entry is removed from the map. A value name can only be read from when it is in the scope, that is when it is in the range of the ENV map.

When starting the check for transformability of the value expression of a function, the ENV map maps any type of interest occurring in the argument type of the function to the corresponding parameter name.

During the check for transformability of let expressions the ENV map has to be updated according to the bindings in the let expressions to reflect the scopes of the value names. The content of the ENV map during a check for transformability is illustrated in Figure 8.1. By comparing this figure with Figure 5.1 on page 27 the connection between the scope rules and the ENV map should be obvious.

type
  T

value
  gen : ... → ... × T × ...

  f : T → ...

$$f(t_1) \equiv \underset{[\,T \mapsto t_1\,]}{\underset{\uparrow}{t_1}} + \textbf{let}\ t_2 = t_1 + gen(...)\ \textbf{in}\ \underset{[\,T \mapsto t_2\,]}{\underset{\uparrow}{t_2}} + gen(...) + \underset{[\,]}{\underset{\uparrow}{...}}\ \textbf{end}$$

Figure 8.1: The ENV during transformability check

The ENV map is a finite map, which means that it can hold at most one entry per type of interest. This conforms to the fact that there is only one placeholder for a variable, and that only one value name of a type of interest can be in scope at a certain point of execution.

The most natural way to represent the ENV map is the reversed way, that is such that value names are mapped to type identifiers. But it turned out, that using this approach made updating of the map more difficult than the selected solution.

The TYPES Map

The *TYPES* map is defined as follows:

$$\text{TYPES} = \text{Binding} \underset{m}{\rightarrow} \text{ExpType},$$
$$\text{ExpType} == \text{Known}(\text{type\_expr} : \text{TypeExpr}) \mid \text{Unknown}$$

The TYPES map maps value names to their corresponding expected type if known. It is used both during the check for transformability and during the transformation phase to decide if value names are of the type of interest. As for the ENV map, the TYPES map is related to a function and not to the entire specification. The initial value of the TYPES map is computed in the functions *CheckValueDef* and *TRValueDef* when dealing with explicit function definitions.

8.3.2 The Structure of a Transformation

The main function of the transformer is *TRRSLAst* which as arguments takes an applicative RSL AST, a new scheme id and a TRANS map and returns the corresponding imperative RSL AST, if the applicative RSL AST is transformable. In some cases the applicative specification cannot be transformed either because it is not transformable or because it is not within RSL_A. This requires the result of the transformation to be specified as a variant definition:

$$\begin{aligned}&\text{TRResult} ==\\&\quad \text{RSLast\_transformable}(\text{result} : \text{RSLAst}) \mid\\&\quad \text{RSLast\_not\_transformable}\end{aligned}$$

The transformation of an applicative RSL AST is divided into three parts:

1. The TYPINGS map and FUNC map are established.
2. A check for transformability is performed.
3. If the specification is transformable the actual transformation is done.

First the two maps TYPINGS and FUNC are established. Then the transformer starts from the top of the RSL AST and then moves down the RSL AST until the whole RSL AST has been processed checking for transformability and transforming the RSL AST on its way.

In the following the structure of the check for transformability and the transformation will be described.

The Check for Transformability

The check for transformability is simply a check to see whether the special scope rules defined in Chapter 5 are fulfilled. This check is done using the ENV map, which is updated such that it at all times shows which value names of the type of interest that are in the scope at a certain point of execution, cf. Section 8.3.1.

During the check for transformability it is also checked whether the specification fulfills the non-syntactically constraints listed in Chapter 4. To get an idea of how this is done, the main function *CheckTypeDef* for checking if a type definition is transformable is shown in Specification 8.1.

Specification 8.1 – Check for transformability of a type definition

CheckTypeDef :
  TypeDef × TYPINGS × FUNC × TRANS → **Bool**
CheckTypeDef(td, typings, func, trans) ≡
  case td **of**
    SortDef(id) → id ∉ **dom** trans,
    VariantDef(id, vl) →
      CheckVariantList(id, vl, typings, func, trans) ∧
      id ∉ **elems** typings(id),
    ShortRecordDef(id, cl) →
      CheckComponentKindList(
        id, cl, typings, func, trans) ∧
      id ∉ **elems** typings(id),
    AbbreviationDef(id, te) →
      CheckTypeExpr(te, typings, func, trans) ∧
      id ∉ **elems** typings(id)
  end

As can be seen in Specification 8.1 the check of a type definition is made by casing over the kind of type definitions. For sort type definitions it is checked whether the type is not of the type of interest using the TRANS map. For the other kinds of type definitions it is checked whether they are not recursively defined using the TYPINGS map and it is checked if the contents of the type definitions fulfill the constraints given in Chapter 4 using the functions *CheckVariantList*, *CheckComponentKindList* and *CheckTypeExpr*.

The Actual Transformation

The actual transformation of the RSL AST is done walking through the constructs of the RSL AST and whenever needed changing the applicative constructs into corresponding imperative constructs according to the transformation rules given in Chapter 6. An interesting example of this can be seen in Specification 8.2, which shows some parts of the main function *TR-ValueExpr* for transforming value expressions.

Specification 8.2 – Transformation of a value expression

```
TRValueExpr :
   ValueExpr × ExpType × FUNC × TRANS × TYPES →
      ValueExpr × TYPES
TRValueExpr(ve, et, func, trans, types) ≡
   case ve of
      Make_ValueLiteral(vl) →
         (makeAssignExpr(ve, et, trans), types),
      Make_ValueOrVariableName(vn) →
         if ∼ isinBinding(IdBinding(id(vn)), dom types)
         then
            if id(vn) ∈ dom func
            then /*Constant of the type of interest.*/
               TRValueExpr(
                     ApplicationExpr(ve, ⟨⟩), et, func,
                     trans, types)
            else (ve, types)
            end
         else
            if
               checkTRANS(
                     types(IdBinding(id(vn))), dom trans)
            then
               /*The value expression is of a type of interest.
               */
               (makeAssignExpr(
                     getTRANS(types(IdBinding(id(vn))), trans),
                     et, trans), types)
            else
               /*The value expression is not of a type of interest.
               */
               (makeAssignExpr(ve, et, trans), types)
            end
         end,
      ⋮
      ValueInfixExpr(first, op, second) →
```

```
let
  (first′, types′) =
    TRValueExpr(
      first, Unknown, func, trans, types),
  (second′, types″) =
    TRValueExpr(
      second, Unknown, func, trans, types′)
in
  (makeAssignExpr(
      ValueInfixExpr(first′, op, second′), et,
      trans), types″)
end,
  ⋮
end
```

As can be seen in Specification 8.2 the transformation of a value expression is done by casing over the kind of the value expression. A value literal is not transformed. A value or variable name is either a constant of a type of interest and is transformed as an application expression with no parameters, it is a value name of a type of interest and is transformed into the corresponding variable, or it is a value name not of the type of interest in which case it is transformed into itself. An infix expression is transformed by first transforming the left hand side of the infix operator, then transforming the right hand side of the infix operator and finally establishing a new infix expression consisting of the two transformed value expressions and the original infix operator. In all cases the function *makeAssignExpr* establishes assignment expressions when necessary.

The value of the TYPES map is altered during the transformation to reflect the point of execution. It is clear from this specification that it follows the transformation rules given in Section 6.4.3

The rest of the transformation follows the same pattern.

8.4 Specification of the Transformer in RSL_1

Due to the use of the RSL2Java tool it is necessary to specify the transformer in RSL_1. This is done based on the specification of the transformer in RSL, which is translated into RSL_1 by hand . The specification of the transformer in RSL_1 TransformerRSL1.rsl can be found in Appendix D.

In the following the main changes made in order to translate from RSL into RSL_1 are described.

8.4.1 More Than One Return Value

As the RSL2Java tool can only handle functions with one return value, it is necessary to change functions with more than one return value. This is achieved by introducing short record definitions consisting of fields representing all the types returned by the function. An example of such a change is shown in Example 8.2.

Example 8.2 – From many return values to one return value

RSL specification:

value
  CheckOptValueExprPairList :
    OptionalValueExprPairList × MapType × TYPINGS ×
    FUNC × TRANS × ENV × TYPES →
      Bool × ENV × TYPES
  CheckOptValueExprPairList(
    ovel, et, typings, func, trans, env, types) ≡
    case ovel **of**
      ValueExprPairList(vel) →
        CheckValueExprPairList(
          vel, et, typings, func, trans, env, types),
      NoValueExprPairList → (**true**, env, types)
    end

RSL$_1$ specification:

type
  BOOL_ENV_TYPES ::
    bool : **Bool** envMap : ENV typesMap : TYPES
value
  CheckOptValueExprPairList :
    OptionalValueExprPairList × MapType × TYPINGS ×
    FUNC × TRANS × ENV × TYPES →
      BOOL_ENV_TYPES
  CheckOptValueExprPairList(
    ovel, et, typings, func, trans, env, types) ≡
    case ovel **of**
      ValueExprPairList(vel) →
        CheckValueExprPairList(

```
            vel, et, typings, func, trans, env, types),
      NoValueExprPairList →
         mk_BOOL_ENV_TYPES(true, env, types)
   end
```

8.4.2 Let Expressions

As let expressions are not a part of RSL_1, all let expressions must be removed. This is done by inserting the actual value of the bindings in the value expression. In cases where the let bindings are products, the destructors of the short record definitions which were introduced in Section 8.4.1, are exploited in order to obtain the desired values. An example of such a change can be found in Example 8.3.

Example 8.3 – Removing let expressions

RSL specification:

```
value
   TROptPreCondition :
      OptionalPreCondition × ExpType × FUNC × TRANS ×
      TYPES →
         OptionalPreCondition × TYPES
   TROptPreCondition(precond, et, func, trans, types) ≡
      case precond of
         PreCondition(ve) →
            let
               (ve′, types′) =
                  TRValueExpr(ve, et, func, trans, types)
            in
               (PreCondition(ve′), types′)
            end,
         NoPreCondition → (NoPreCondition, types)
      end
```

RSL_1 specification:

```
type
```

```
  PRECOND_TYPES ::
    preCond : OptionalPreCondition   typesMap : TYPES
value
  TROptPreCondition :
    OptionalPreCondition × ExpType × FUNC × TRANS ×
    TYPES →
      PRECOND_TYPES
  TROptPreCondition(precond, et, func, trans, types) ≡
    case precond of
      PreCondition(ve) →
        mk_PRECOND_TYPES(
          PreCondition(
              valueExpr(
                TRValueExpr(ve, et, func, trans, types))
          ),
          typesMap(
              TRValueExpr(ve, et, func, trans, types))),
      NoPreCondition →
        mk_PRECOND_TYPES(NoPreCondition, types)
    end
```

8.4.3 Lack of Operators

Only a small subset of operators are included in the RSL_1 subset. The lack of these operators makes it necessary to implement their functionality using functions. Some of the operators are straight forward to implement, whereas some limit the functionality of the transformer.

An example of the first kind is the inequality operator $\neq$ and the negation operator $\sim$. A solution to the lack of the these operators can be seen in Example 8.4.

Example 8.4 – Lack of operators – inequality and negation

RSL specification:

```
⋮
    if tel ≠ ⟨⟩
    then
      (false, mk_FormalFunctionParameter(⟨⟩), env,
```

```
            types, prlet)
      else
          (true, mk_FormalFunctionParameter(⟨⟩), env,
            types, prlet)
      end
⋮
```

RSL_1 specification:

```
⋮
      if not(tel = ⟨⟩)
      then
          mk_BOOL_FFP_ENV_TYPES_LDL(
              false, mk_FormalFunctionParameter(⟨⟩), env,
              types, prlet)
      else
          mk_BOOL_FFP_ENV_TYPES_LDL(
              true, mk_FormalFunctionParameter(⟨⟩), env,
              types, prlet)
      end
⋮

not : Bool → Bool
not(b) ≡ if b then false else true end
```

An example of the second kind is the lack of the minus operator. As the minus operator cannot be implemented directly using the limited range of operators in RSL_1, it is necessary to implement this in a way that limits the functionality of transformer. This can be seen in Example 8.5. The limitation is that the *minusOne* function can only handle numbers from 1 through 10 and that it only is capable of subtracting by one.

Example 8.5 – Lack of operators – minus

RSL specification:

```
toExpTypeList : ExpType × Nat → ExpType*
toExpTypeList(et, length) ≡
```

if length = 0 **then** $\langle\rangle$
else $\langle$et$\rangle$ $\frown$ toExpTypeList(et, length $-$ 1)
end

RSL$_1$ specification:

toExpTypeList : ExpType $\times$ **Int** $\rightarrow$ ExpType*
toExpTypeList(et, length) $\equiv$
if equals(length, 0) **then** $\langle\rangle$
else $\langle$et$\rangle$ $\frown$ toExpTypeList(et, minusOne(length))
end,

minusOne : **Int** $\rightarrow$ **Int**
minusOne(i) $\equiv$
case i **of**
1 $\rightarrow$ 0,
2 $\rightarrow$ 1,
3 $\rightarrow$ 2,
4 $\rightarrow$ 3,
5 $\rightarrow$ 4,
6 $\rightarrow$ 5,
7 $\rightarrow$ 6,
8 $\rightarrow$ 7,
9 $\rightarrow$ 8,
10 $\rightarrow$ 9
end

8.4.4 Errors in the RSL2Java Tool

Due to some type decoration problems in the RSL2Java tool, which result in errors when trying to compile the generated Java code, it is necessary to make small functions, which help the RSL2Java tool to determine the type of certain value expressions. An example of such a function, namely *emptyId*, can be found in Example 8.6.

Example 8.6 – Helping the RSL2Java tool type decorator

RSL specification:

getTYPINGSVariantList : Variant* $\rightarrow$ Id*
getTYPINGSVariantList(vl) $\equiv$
  if vl = $\langle\rangle$ **then** $\langle\rangle$
  else
    removeDupletsId(
      getTYPINGSVariant(**hd** vl) $^\frown$
      getTYPINGSVariantList(**tl** vl))
  end

RSL$_1$ specification:

getTYPINGSVariantList : Variant* $\rightarrow$ Id*
getTYPINGSVariantList(vl) $\equiv$
  if vl = $\langle\rangle$ **then** emptyId()
  else
    removeDupletsId(
      getTYPINGSVariant(**hd** vl) $^\frown$
      getTYPINGSVariantList(**tl** vl))
  end,

emptyId : **Unit** $\rightarrow$ Id*
emptyId() $\equiv$ **tl** $\langle$mk_Id("DUMMY")$\rangle$

8.4.5 Missing Types

Only a limited number of types is included in the RSL$_1$ subset. When translating from RSL to RSL$_1$ it is necessary to find proper replacements for the missing types, e.g. maps and sets. In some cases this is straight forward, in other cases the substitution leads to the introduction of many new functions.

The replacements are chosen such that they offer the same flexibility as the original types. In some cases it is more correct to base the new types on subtypes in order for the new types to reflect the old types more precisely. Unfortunately, this is not possible as the RSL2Java tool does not support subtypes. Instead the functions using the new types are constructed such that they bear as close a resemblance to the old types as possible.

The different substitutions are discussed in the following.

Nat

The built-in type **Nat**, which represents the natural numbers, is not a part of the RSL_1 subset. The substitution of **Nat** is straight forward as it can easily be replaced by the built-in type **Int**, which is a part of RSL_1.

Sets

Set types are not part of RSL_1. The replacement for sets is lists and some functions simulating set operations on lists. Arguments for the correctness of this substitution can be found in Section 8.5.

Maps

Map types are not part of RSL_1. The replacement for maps is lists of short record definitions. The short record definitions consist of two fields: one for the domain value and one for the associated range value. Furthermore, some functions simulating map operations on these lists are specified. An example of a replacement of a map type can be seen in Example 8.7.

Example 8.7 – Replacements of a map type

RSL specification:

TRANS = Id $\underset{m}{\rightarrow}$ Id

RSL_1 specification:

TRANSMapEntrance :: first : Id second : Id,
TRANS :: map : TRANSMapEntrance*

The obvious thing was to make *TRANS* an abbreviation type, but as abbreviation type definitions are not part of RSL_1 this was not possible. Arguments for the correctness of this substitution can be found in Section 8.5.

8.5 Development Relation

During the translation from RSL into RSL_1 several changes were necessary as described above. The correctness of some of these changes is obvious. This holds for the removal of let expressions, the definition of functions replacing missing operators and the auxiliary functions introduced to solve the errors in the tool. Introduction of these changes in a specification results in a semantically equivalent specification.

But the correctness of the other changes is not obvious. In some cases it was necessary to change from one concrete type definition to another non-equivalent one. Because the types are non-equivalent it is impossible to obtain semantically equivalence between the original specification and the new one, but it is possible to obtain something that is close to implementation. The idea behind this is described in this section and an example is given based on maps.

8.5.1 The Method for Changing Concrete Types

The idea behind the change is as follows. The first thing to do when changing concrete types is to obtain an abstract type ABS from the original concrete one $CONC_1$ such that $CONC_1$ implements ABS. This can be done easily as described in [Gro95, pp. 207-208]. Then ABS is developed into the new concrete type $CONC_2$ such that $CONC_2$ implements ABS. The idea is outlined in Figure 8.2.

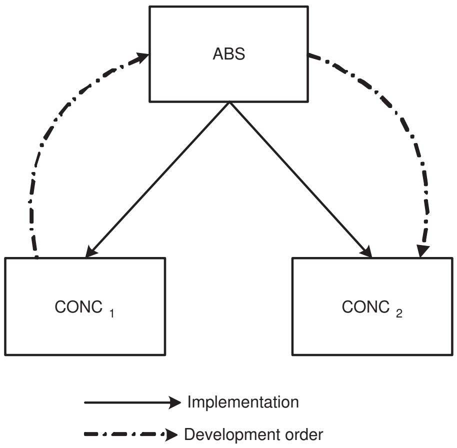

Figure 8.2: Method for changing concrete types

8.5.2 Changing from Maps to Lists

The idea is illustrated in the case of CONC_1 being a map. The map type has to be changed into a list type as done several times in the actual translation of the transformer specification from RSL into RSL_1. The operators on maps are turned into functions. Only the operators used in the specifications of the transformer are considered here.

As the type of the elements in the map is not important at this stage the elements of the map are defined as sorts, see Specification 8.3.

Specification 8.3 – Specification of an element

---

scheme ELEMENT = **class type** Elem **end**

The concrete specification of the map *CONC_MAP* can be found in Specification 8.4. The functions are defined using map operators.

Specification 8.4 – Concrete specification of a map

---

scheme CONC_MAP(T1 : ELEMENT, T2 : ELEMENT) =
  class
    type Map = T1.Elem $\xrightarrow[m]{\sim}$ T2.Elem

    value
      /∗ generators ∗/
      empty : Map = [ ],

      override : Map $\times$ Map $\rightarrow$ Map
      override(m1, m2) $\equiv$ m1 † m2,

      restrict_by : Map $\times$ T1.Elem-**infset** $\rightarrow$ Map
      restrict_by(m, t1) $\equiv$ m $\backslash$ t1,

      /∗ observers ∗/
      domain : Map $\rightarrow$ T1.Elem-**infset**
      domain(m) $\equiv$ **dom** m,

      range : Map $\rightarrow$ T2.Elem-**infset**
      range(m) $\equiv$ **rng** m,

      get_map_value : Map $\times$ T1.Elem $\xrightarrow{\sim}$ T2.Elem

```
        get_map_value(m, t1) ≡ m(t1)
        pre t1 ∈ domain(m)
  end
```

The abstract specification *ABS_MAP* of a map type is then constructed as described in [Gro95]. The map type is defined as an abstract type *Map*. The signature of a hidden observer *map_of* which when applied to the *Map* type returns a value of the type of the original map, is defined. Every occurrence of the old map type is replaced by *Map*. Observers of the old type are defined in terms of *map_of*. The generators, which can be defined in terms of others, become derived functions. For all other generators *map_of_generator* axioms are defined. The result can be seen in Specification 8.5

Specification 8.5 – Abstract specification of a map

```
scheme ABS_MAP(T1 : ELEMENT, T2 : ELEMENT) =
  hide map_of in
  class
    type Map

    value
      /* generators */
      empty : Map,
      override : Map × Map → Map,
      restrict_by : Map × T1.Elem-infset → Map,

      /* observer */
      map_of : Map → (T1.Elem ~m→ T2.Elem),

      /* derived observers */
      domain : Map → T1.Elem-infset
      domain(m) ≡ dom map_of(m),

      range : Map → T2.Elem-infset
      range(m) ≡ rng map_of(m),

      get_map_value : Map × T1.Elem ~→ T2.Elem
      get_map_value(m, t1) ≡ map_of(m)(t1)
      pre t1 ∈ domain(m)
```

```
    axiom
      [ map_of_empty ]
        map_of(empty) ≡ [ ],
      [ map_of_override ]
        ∀ m1, m2 : Map •
          map_of(override(m1, m2)) ≡
            map_of(m1) † map_of(m2),
      [ map_of_restrict_by ]
        ∀ m1 : Map, t1 : T1.Elem-infset •
          map_of(restrict_by(m1, t1)) ≡ map_of(m1) \ t1
  end
```

Then a concrete specification of maps using lists and short record definitions is defined. The new concrete type *LIST_MAP* is a development of *ABS_MAP* and can be seen in Specification 8.6.

Specification 8.6 – Concrete specification of a map using a list

```
scheme LIST_MAP(T1 : ELEMENT, T2 : ELEMENT) =
  class
    type
      MapEntrance :: first : T1.Elem   second : T2.Elem,
      Map = MapEntrance*

    value
      /* generators */
      empty : Map = ⟨⟩,

      override : Map × Map → Map
      override(m1, m2) ≡
        case m1 of
          ⟨⟩ → m2,
          ⟨me⟩ ^ m →
            if first(me) ∈ domain(m2)
            then override(m, m2)
            else ⟨me⟩ ^ override(m, m2)
            end
        end,

      restrict_by : Map × T1.Elem-infset → Map
      restrict_by(m, t1) ≡
```

$$
\begin{array}{l}
\quad\quad \textbf{case}\ \mathrm{m}\ \textbf{of} \\
\quad\quad\quad \langle\rangle \rightarrow \langle\rangle, \\
\quad\quad\quad \langle \mathrm{me}\rangle \,\widehat{}\; \mathrm{m1} \rightarrow \\
\quad\quad\quad\quad \textbf{if}\ \mathrm{first(me)} \in \mathrm{t1}\ \textbf{then}\ \mathrm{restrict\_by(m1,\ t1)} \\
\quad\quad\quad\quad \textbf{else}\ \langle \mathrm{me}\rangle \,\widehat{}\; \mathrm{restrict\_by(m1,\ t1)} \\
\quad\quad\quad\quad \textbf{end} \\
\quad\quad \textbf{end}, \\
\\
\quad /\!*\ \mathrm{observers}\ */ \\
\quad \mathrm{domain} : \mathrm{Map} \rightarrow \mathrm{T1.Elem}\textbf{-infset} \\
\quad \mathrm{domain(m)} \equiv \\
\quad\quad \textbf{case}\ \mathrm{m}\ \textbf{of} \\
\quad\quad\quad \langle\rangle \rightarrow \{\}, \\
\quad\quad\quad \langle \mathrm{me}\rangle \,\widehat{}\; \mathrm{m1} \rightarrow \{\mathrm{first(me)}\} \cup \mathrm{domain(m1)} \\
\quad\quad \textbf{end}, \\
\\
\quad \mathrm{range} : \mathrm{Map} \rightarrow \mathrm{T2.Elem}\textbf{-infset} \\
\quad \mathrm{range(m)} \equiv \\
\quad\quad \textbf{case}\ \mathrm{m}\ \textbf{of} \\
\quad\quad\quad \langle\rangle \rightarrow \{\}, \\
\quad\quad\quad \langle \mathrm{me}\rangle \,\widehat{}\; \mathrm{m1} \rightarrow \{\mathrm{second(me)}\} \cup \mathrm{range(m1)} \\
\quad\quad \textbf{end}, \\
\\
\quad \mathrm{get\_map\_value} : \mathrm{Map} \times \mathrm{T1.Elem} \stackrel{\sim}{\rightarrow} \mathrm{T2.Elem} \\
\quad \mathrm{get\_map\_value(m,\ t1)} \equiv \\
\quad\quad \textbf{case}\ \mathrm{m}\ \textbf{of} \\
\quad\quad\quad \langle \mathrm{me}\rangle \,\widehat{}\; \mathrm{m1} \rightarrow \\
\quad\quad\quad\quad \textbf{if}\ \mathrm{t1} = \mathrm{first(me)}\ \textbf{then}\ \mathrm{second(me)} \\
\quad\quad\quad\quad \textbf{else}\ \mathrm{get\_map\_value(m1,\ t1)} \\
\quad\quad\quad\quad \textbf{end} \\
\quad\quad \textbf{end} \\
\quad \textbf{pre}\ \mathrm{t1} \in \mathrm{domain(m)} \\
\textbf{end}
\end{array}
$$

In order to show that *LIST_MAP* implements *ABS_MAP* a conservative extension *LIST_MAP1* of *LIST_MAP* is defined, see Specification 8.7. This extension defines *map_of*. The extension is conservative as it does not add new properties to the entities from *LIST_MAP*. The only change is that the functionality of the *map_of* function is now defined.

Specification 8.7 – Conservative extension of *LIST_MAP*

```
scheme LIST_MAP1(T1 : ELEMENT, T2 : ELEMENT) =
   extend LIST_MAP(T1, T2) with
   class
      value
         map_of : Map → (T1.Elem →m T2.Elem)
         map_of(m) ≡
            case m of
               ⟨⟩ → [ ],
               ⟨x⟩ ⁀ m1 →
                  [ first(x) ↦ second(x) ] ∪ map_of(m1)
            end
   end
```

It can be shown that *LIST_MAP1* implements *ABS_MAP* because:

1. *LIST_MAP1* statically implements *ABS_MAP* as it is obvious that the visible part of the maximal signature of *ABS_MAP* is included in the visible part of the maximal signature of *LIST_MAP1*.

2. Every theorem of *ABS_MAP* is a theorem of *LIST_MAP1*. This step requires justification of the implementation conditions, but this part is left out in order to focus on the interesting things.

Because *CONC_MAP* that resembles the map type of RSL implements *ABS_MAP* and because the conservative extension of *LIST_MAP* implements *ABS_MAP* the closest form of implementation between the map type of RSL and *LIST_MAP* is proven. The translation between RSL and RSL_1 using lists instead of maps can be claimed to be legal.

The one thing missing in the above specifications is to specify that the new *Map* type is deterministic. This part is left out in order to focus on the idea itself. Furthermore, this restriction on the *Map* type cannot be done in the specification of the transformer in RSL_1 as subtypes are not part of RSL_1. This problem is solved by making sure that the new functions implementing the map operators respect this restriction. An example of this can be seen in the *override* function in Specification 8.6. This is the reason why the *Map* type is defined as an infinite map in the *CONC_MAP* specification. It would cause problems in the above specifications if it was defined as a finite map.

8.5.3 Changing Other Types

Similar arguments can be applied to the other changes of concrete types which have been made during the translation of the transformer specification from RSL into RSL_1, e.g. the change from sets to lists.

In fact the specification of the transformer in RSL corresponds to $CONC_1$ in Figure 8.2 on page 89 and the specification of the transformer in RSL_1 corresponds to $CONC_2$. An abstract specification can be developed in order to prove the closest form of implementation between these two specifications. In this way it can be proved that the translation of the transformer from RSL into RSL_1 is correct.

8.5.4 Comparison to VDM

VDM is another formal specification language, [Jon86]. The *map_of* function is equivalent to the *retrieve* function in VDM. In VDM it is required that the retrieve function is total and surjective.

The property of being total can be achieved through invariants.

The reason why the retrieve function has to be surjective is because all values of the abstract type must have corresponding values when using the concrete type. This idea is illustrated in Figure 8.3.

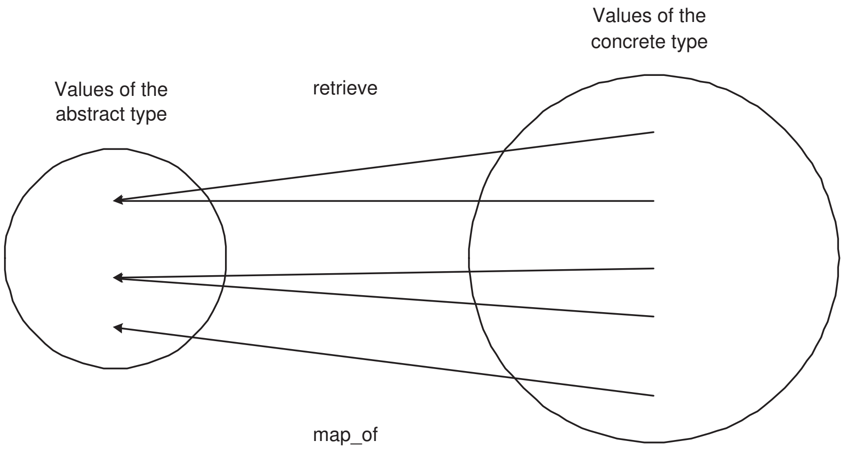

Figure 8.3: Illustration of the retrieve function

The retrieve function need not be bijective as one value of the abstract type can have more representations using the concrete type. An example of this is that the map value $[\,0 \mapsto \textbf{false},\ 1 \mapsto \textbf{true}\,]$ can have the list representations $\langle$mk_MapEntrance(0, **false**), mk_MapEntrance(1, **true**)$\rangle$ or $\langle$mk_MapEntrance(1, **true**), mk_MapEntrance(0, **false**)$\rangle$.

Using RAISE, it is not necessary to show that *map_of* is surjective as the lack of this property would be revealed during the justification of the implementation relation.

Chapter 9

Implementation of the Transformer

The implementation of the transformer is rather special as different tools have been used and exploited. This process and the resulting structure of the transformer are described in this chapter. Furthermore, a discussion of one of the tools used are given. Finally, an overview of the implemented functionality is offered.

9.1 The RSL2Java Tool

The implementation of the transformer is based on the RSL2Java tool, which is build in another master thesis project, [Hja04]. The RSL2Java tool can translate the subset RSL_1 of RSL into Java. It consists of a front end, a translation module, a back end and a control module.

- The front end consists of a lexer, which performs lexical analysis of the given RSL specification, and a parser, which parses the specification into an RSL AST.
- The translation module does the actual translation from the RSL AST of an RSL specification into a corresponding Java AST. This is done by decorating the RSL AST with types and then performing the translation on the type decorated RSL AST.
- The back end translates the Java AST into actual Java code.
- The control module binds the three other modules together.

In order not to confuse the parts of the RSL2Java tool with the parts of the transformer all parts belonging to the RSL2Java tool are prefixed with the word RSL2Java tool, e.g. *RSL2Java tool front end*.

9.2 Exploiting the RSL2Java Tool

The idea behind basing the implementation of the transformer on the RSL2Java tool is to be able to reuse some of the more or less tedious work done in [Hja04] in order to focus on the more interesting parts of the transformer. The way to exploit the RSL2Java tool is described in the following using tombstone diagrams as described in [WB00].

First an RSL specification of the transformation from an RSL AST of an applicative specification, named RSL_A AST, into an RSL AST of an imperative RSL specification, named RSL_I AST, is written within RSL_1. Then this specification is translated into Java using the RSL2Java tool, see Figure 9.1. The resulting Java program is combined with an extended version of the RSL2Java tool front end and a revised version of the RSL AST to RSL unparser provided by the RSL2Java tool, Figure 9.2. The result is a Java program, which can do the desired transformation from applicative RSL specifications into imperative RSL specifications, see Figure 9.3 on the next page.

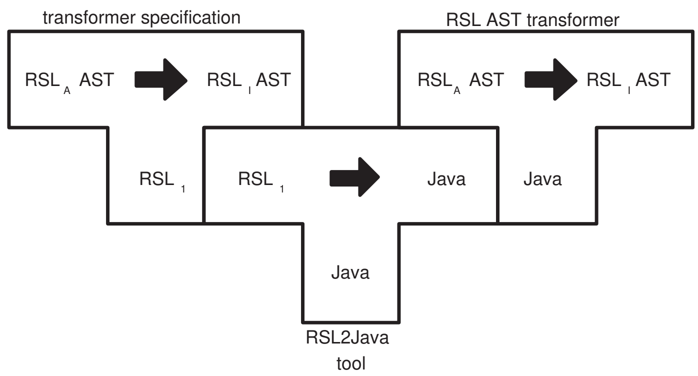

Figure 9.1: Translating the specification of the transformer

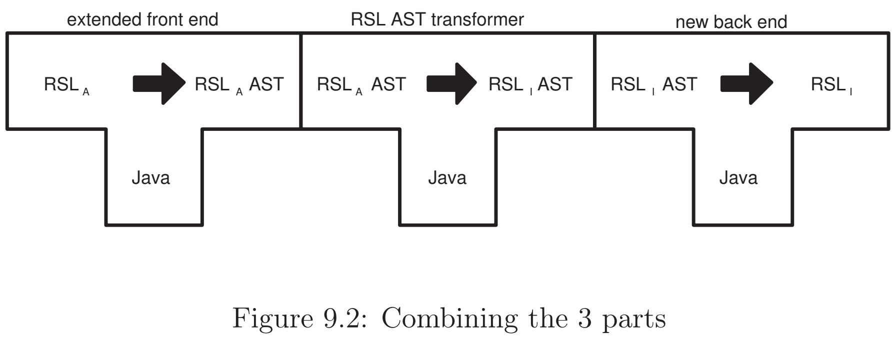

Figure 9.2: Combining the 3 parts

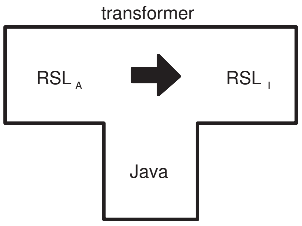

Figure 9.3: The resulting transformer

9.3 Development Process

Besides having to write the RSL specification of the transformation, some changes must be applied to the RSL2Java tool, since it only works for RSL_1. The transformer has to be able to translate a different subset RSL_A of RSL into yet another RSL subset RSL_I. The relationships between the three subsets are illustrated in Figure 9.4.

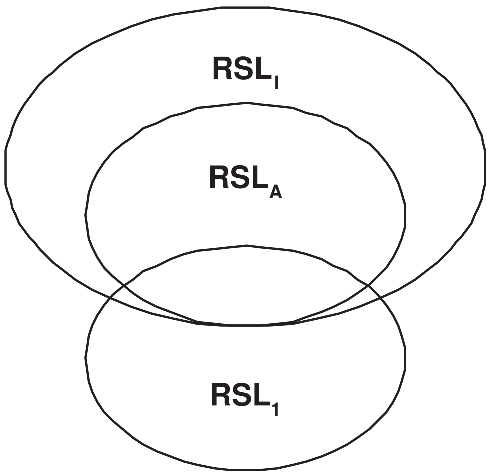

Figure 9.4: The different subsets of RSL

In the following I will describe the different steps in the process of developing the transformer by changing the RSL2Java tool.

9.3.1 Front End

The front end, which is developed using the ANTLR tool, [Par], must be extended and in some cases altered in order to be able to parse RSL_A. This means extending and altering the grammar file rsltorslast.g from which the lexer and parser are generated and then generate the actual lexer and parser.

Furthermore, the specification of the RSL AST, named RSLAst_Module.rsl, which is translated into Java using the RSL2Java tool, has to be extended to deal with RSL_I in order to conform to the output from the lexer, the parser and the transformer module.

The structure and workings of the lexer and parser are described in Section 9.5 and a description of the specification of the RSL AST can be found in Chapter 8.

9.3.2 Transformer Module

The specification of the transformer, named TransformerRSL1.rsl, which can transform an applicative RSL specification into an imperative one has to be written and translated using the RSL2Java tool.

Unlike the RSL2Java tool, no type decoration is done explicitly before the transformation. This means that the type decorator visitor, the parent visitor and the wrapper modules of the RSL2Java tool are no longer needed.

The specification of the transformer is described in Chapter 8.

9.3.3 Back End

The back end consists of a revised version of the super class of the visitor modules, named RSLAstVisitor.java, and the RSL AST to RSL visitor, named StringRSLAstVisitor.java, provided by the RSL2Java tool. These visitors have to be extended in order to deal with all the constructs of RSL_I.

The abstract class RSLElement.java, which is the super class of all classes generated by the RSL2Java tool when translating the RSL AST specification, is adapted to the changes done. It does not need any fields used by the type decorator and the parent visitor in the RSL2Java tool. The only thing it has to offer is the abstract method accept which is used to visit the RSL AST nodes.

Further description of the visitors can be found in Section 9.6.

9.3.4 Control Module

The control module, RSLRunner.java, is completely altered in order to reflect the above changes. The module takes care of reading the user input, which consists of the name of the applicative specification to transform, the name of the new imperative specification and a list of type names and corresponding variable names, which is used to determine the types of interest and corresponding variable names. It then uses the above parts to make the actual transformation. The result of the transformation is written to a file. The source code of the control module can be found in Appendix F.

See Appendix A for more details on the use of the transformer.

The classes in the rsllib package containing functionality for sets, lists and maps are not altered. The list functionality is used when implementing the transformer.

9.4 The Transformer

The result of the process described above is a transformer, which can transform a transformable applicative RSL specification into an imperative RSL specification according to the transformation rules given in Chapter 6. If the applicative specification not is transformable or not within RSL_A this is reported to the user through error messages.

The transformation is done in 3 steps as shown in Figure 9.5. First of all the applicative RSL specification is transformed into the corresponding RSL AST using the lexer and parser. The RSL AST is transformed into an imperative version of the RSL AST. Finally, the RSL AST is turned into an imperative RSL specification using the string visitor StringRSLAstVisitor.java.

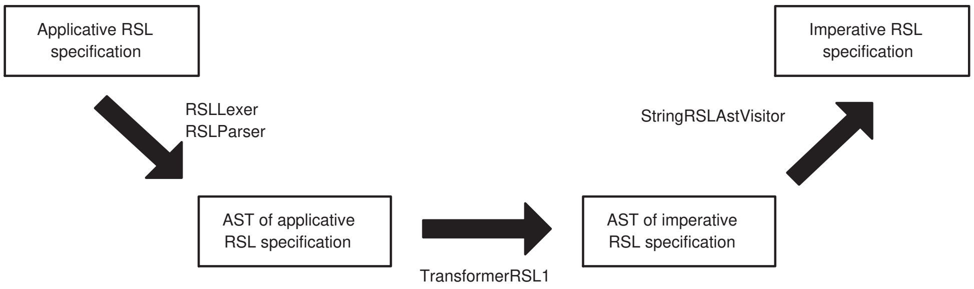

Figure 9.5: The flow of a transformation

9.5 The Lexer and Parser

The lexer and parser which transforms an RSL ASCII file into an RSL AST is made using the ANTLR tool, [Par]. This tool generates a lexer and a parser in Java based on a grammar file. The grammar file can be found in Appendix E. The grammar file contains both the grammar of the lexer and the parser.

The lexer performs a lexical analysis of the specification, which means that it reads the specification and turns it into tokens. The parser then determines the syntactic structure of these tokens and turns them into an RSL AST if the parsing is successful.

The specification of the parser very much resembles that of a BNF grammar. The grammar is built according to the syntax of RSL, which is illustrated in Example 9.1. The example shows the part of the parser grammar which recognizes patterns. The actual grammar is placed between the outermost : and ;. The pattern can be a value literal pattern which consists of a value literal, it can be a name pattern which consists of a value name, it can be a record pattern which is of the form *name(pattern, ..., pattern)*, it can be a wild card pattern which consists of an underscore and so on.

Example 9.1 – Example of the ANTLR grammar

RSL Syntax:

```
        pattern ::= value_literal |
                    pure_value_name |
                    record_pattern |
                    wildcard_name |

 record_pattern ::= pure_value_name(pattern-list)
```

ANTLR grammar:

```
pattern returns[Pattern p] {
        p = null;
        Make_ValueOrVariableName vovn = null;
        Make_ValueOrVariableName vovn2 = null;
        Make_ValueOrVariableName vovn3 = null;
        Pattern ip = null;
        ValueLiteral vl = null;
        RSLListDefault<Pattern> innerPatternList =
                                new RSLListDefault<Pattern>();
        ListPattern lp = null;
}
:               vl = value_literal
                {p = new ValueLiteralPattern(vl);}
        |
                vovn = name
                {p = new NamePattern(
                    vovn.value_or_variable_name().id(),
                    new NoOptionalId());}
        |
                vovn = name LPAREN ip = pattern
                {innerPatternList.getList().add(ip);}
                (
                        COMMA ip = pattern
                        {innerPatternList.getList().add(ip);}
```

```
            )*
            RPAREN
            {p = new RecordPattern(
                vovn.value_or_variable_name(),
                innerPatternList);}
        |
            UNDERSCORE {p = new WildcardPattern();}
        |

        ...

;
```

9.6 The Visitor Modules

The visitors are built using the *visitor design pattern*. The purpose of the visitor design pattern is to separate a hierarchical structure of classes from tasks which must be performed on the structure itself. When following the visitor design pattern, tasks that would normally be placed in the functionality of the individual class are put inside one single class, called a visitor.

When using the visitor design pattern an abstract visitor class is created. This abstract class contains one method, *visitClassName*, for each class, *ClassName*, in the hierarchical structure. To each class in the structure a method often called *accept* or *visit* is added. This method takes a visitor as argument and calls the *visitClassName* method.

In order to fulfill a task a concrete visitor, which can perform the actual task, is implemented as an implementation of the abstract visitor. Each of the *visit* methods in the concrete visitor are implemented such that each of them implements a part of the whole task.

The advantage of the visitor design pattern is that when a new task is needed, the only thing to do is to make a new concrete visitor. No changes are needed in the classes of the structure. If the functionality of a visitor must be changed, the only class that needs to be changed is the visitor class itself.

In the transformer program the `RSLAstVisitor.rsl` is an abstract visitor and the `StringRSLAstVisitor.rsl` is a concrete visitor, which can perform the task of turning an RSL AST into an RSL ASCII specification. Part of a class diagram showing the use of the visitor design pattern in the transformer program can be found in Figure 9.6 on the following page. Code snippets of `StringRSLAstVisitor.java` can be found in Example 9.2. The entire source code of the visitor modules can be found in Appendix F.

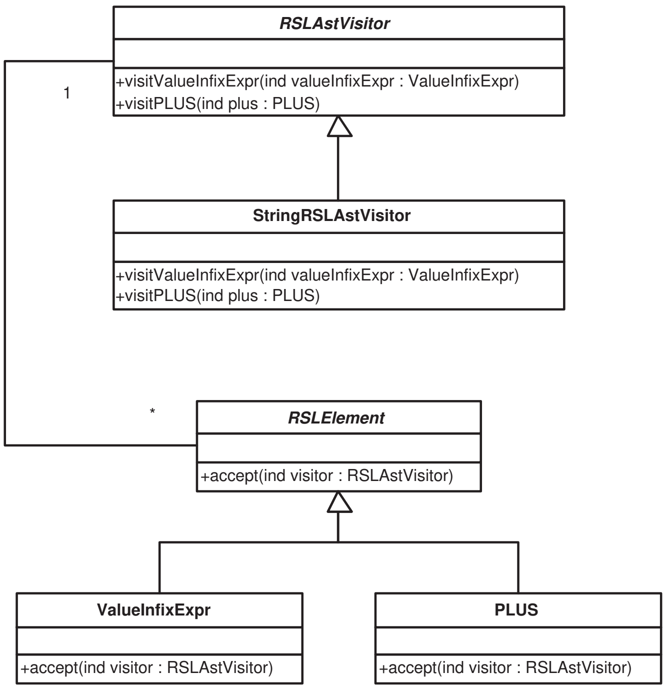

Figure 9.6: The use of the visitor design pattern in the transformer

Example 9.2 – Code snippet of the `StringRSLAstVisitor.java`

```
public void visitValueInfixExpr(ValueInfixExpr valueInfixExpr) {
    valueInfixExpr.left().accept(this);
    result.append(" ");
    valueInfixExpr.op().accept(this);
    result.append(" ");
    valueInfixExpr.right().accept(this);
}

public void visitLetExpr(LetExpr letExpr) {
    result.append("let ");
    for (LetDef letDef : letExpr.let_def_list().getList()) {
        letDef.accept(this);
        result.append(", ");
    }
    if (letExpr.let_def_list().len() > 0) {
        result.delete(result.length() - 2, result.length());
    }
    result.append(" in ");
    letExpr.let_value_expr().accept(this);
```

```
    result.append(" end");
}

public void visitPLUS(PLUS rsl_plus) {
    result.append("+");
}

public void visitMINUS(MINUS rsl_minus) {
    result.append("-");
}
```

9.7 Disadvantages In Using the RSL2Java Tool

As mentioned in this chapter there have been some advantages involved by choosing to use the RSL2Java tool in the implementation of the transformer. However, it turned out during the development process that there were also some disadvantages. In this section some of the disadvantages will be discussed.

The RSL2Java tool can only translate a subset RSL_1 of RSL into Java. This made the specification written in RSL_1 very long and messy. Especially the lack of let expressions, maps, some operators and the fact that functions are only allowed to have one return value complicated the matter. Furthermore, some errors connected to the type decoration of the RSL2Java tool required the introduction of extra functions.

These problems led to the idea, that a specification of the transformation should be written in both RSL and RSL_1. This was necessary in order to offer a readable specification for the reader and at the same time to be able to use the RSL2Java tool. This choice led to a lot of effort in keeping the two specifications up to date, both during the implementation phase and during the process of correcting errors. The method for obtaining this was first to correct the RSL specification and then transfer the corrections to the RSL_1 specification. The approach may have complicated the RSL_1 specification more than necessary but were necessary in order to keep both of the two specifications up to date.

The resulting transformer works as expected, but in some cases the transformation process is rather slow, especially in cases of specifications containing nested product expressions. To give an example, it takes a few seconds to transform a specification without nested product expressions, whereas it can take up to several minutes to transform a specification of the same size containing one nested product expression. This is due to the fact that no code optimization is done in the RSL2Java tool and at the same time it is very difficult to optimize the code by optimizing the specification in RSL_1. Due to these difficulties no effort is put into optimizing the transformer.

All this could have been avoided either by implementing the transformer directly in Java or by extending and improving the RSL2Java tool.

9.8 Implemented Functionality

An overview of the functionality of the transformer is given below.

9.8.1 Check for Syntactical Constraints

The transformer does not check the syntax of the applicative RSL specification. This means that the applicative specification should be syntax checked using the RSL syntax checker before applying the transformer to the specification.

Furthermore, the lexer and parser do only parse constructs not mentioned in the list of syntactical constraints in Chapter 4. If the user tries to transform a specification containing e.g. extending class expressions the user will be notified through error messages.

9.8.2 Check for Non-Syntactically Constraints

The presence of constraints mentioned in the list of non-syntactical constraints in Chapter 4 are checked during the check for transformability. The occurrence of such a construct in a specification given to the transformer will result in an error message.

It is not checked whether overloading is used in the specification. This means that it is possible to transform specifications containing overloading but the result is probably not correct, which means that overloading should be avoided.

9.8.3 Check for Transformability

Before the actual transformation of a given applicative specification it is checked whether the specification is transformable according to the rules given in Chapter 5. If the specification is not transformable it is reported to the user and the specification is not transformed.

9.8.4 Transformable RSL Expressions

The following is a list of the RSL expressions that can be transformed if they do not violate any of the constraints given in the lists in Chapter 4 and if they are transformable in the sense explained in Chapter 5.

General Expressions

- Non-parameterized scheme definitions.
- Basic class expressions.
- Sort definitions not of the type of interest.
- Variant definitions.
- Short record definitions.
- Abbreviation definitions.
- Explicit value definitions.
- Explicit function definitions.

Type Expressions

- Type literals.
- Type names.
- Product type expressions.
- Set type expressions (finite, infinite).
- List type expressions (finite, infinite).
- Finite map type expressions.
- Subtype expressions.
- Bracketed type expressions.

The transformation of infinite map type expressions is implemented, but infinite map type expressions cannot be parsed due to the structure of the parser of the RSL2Java tool. It would require major changes to the grammar file if the parser should be able to parse infinite map type expressions.

Value Expressions

- Value literals.
- Value names.
- Basic expression (chaos).
- Product expression.

- Set expressions (ranged, enumerated, comprehended).
- List expressions (ranged, enumerated, comprehended).
- Map expressions (enumerated, comprehended).
- Application expressions (function, list, map).
- Bracketed expressions.
- Value infix expressions.
- Value prefix expressions.
- Let expressions.
- If expressions.
- Case expressions.
- Value literal patterns.
- Name patterns.
- Wildcard patterns.
- Product patterns.
- Record patterns.
- List patterns.

9.8.5 Pretty Printing

The resulting specification of a transformation is not pretty printed. No focus is put on that aspect as the RSL pretty printer can be used for that purpose.

Chapter 10

Examples of Transformations

In order to give an idea of how the transformer works a couple of examples of transformations are given in this chapter. The first example just shows an example of applying the transformer to an applicative specification. The second example shows the development process from initial specification to imperative specification emphasizing the use of the transformer in the development process.

10.1 The Stack

The stack example given in Chapter 2 is transformed by hand. In order to show that the automatic transformation done by the transformer follows the principles given in that chapter, the stack specification has been transformed using the transformer. The result can be seen in Example 10.1. For easy reference the applicative version of the stack specification is repeated here.

Example 10.1 – Automatic transformation of the stack example

---

scheme A_STACK =
  class
    type
      Stack = Element*,
      Element

    value
      empty : Stack = $\langle\rangle$,

      is_empty : Stack → **Bool**
      is_empty(stack) ≡ stack = $\langle\rangle$,

```
            push : Element × Stack → Stack
            push(elem, stack) ≡ ⟨elem⟩ ⌢ stack,

            pop : Stack ~→ Stack
            pop(stack) ≡ tl stack
            pre ∼ is_empty(stack),

            top : Stack ~→ Element
            top(stack) ≡ hd stack
            pre ∼ is_empty(stack)
    end
```

▷

```
scheme I_STACK =
    class
        type
            Stack = Element*,
            Element

        variable
            stack : Stack

        value
            empty : Unit → write stack Unit
            empty() ≡ stack := ⟨⟩,

            is_empty : Unit → read stack Bool
            is_empty() ≡ stack = ⟨⟩,

            push : Element → read stack write stack Unit
            push(elem) ≡ stack := ⟨elem⟩ ⌢ stack,

            pop : Unit ~→ read stack write stack Unit
            pop() ≡ stack := tl stack
            pre ∼ is_empty(),

            top : Unit ~→ read stack Element
            top() ≡ hd stack
            pre ∼ is_empty()
    end
```

When comparing the above imperative specification made using the transformer with the corresponding imperative specification in Chapter 2 made by hand only two differences are found. Both in the *push* and *pop* functions **read** stack access descriptors are added in the automatically generated imperative specification. These **read** access descriptors are not necessary but are a consequence of considerations concerning the verification of the transformation rules as explained in Section 6.4.1.

10.2 The Database

The following example is taken from [Gro92, p. 49-50 and 78-79]. It shows the specification and development of a database system. The example is used to demonstrate how the transformer can be used in the development process.

10.2.1 Requirements of the Database

The first step in the development process is to set up the requirements of the database.

The database associates unique keys with data. That is, one key is associated with at most one data element in the database. The database should provide the following functions:

- *insert*, which associates a key with a data element in the database. If the key is already associated with a data element the new association overrides the old one.
- *remove*, which removes an association between a key and a data element.
- *defined*, which checks whether a key is represented in the database.
- *lookup*, which returns the data element associated with a particular key.

10.2.2 Abstract Applicative Specification of the Database

Then an abstract applicative specification can be defined. An abstract applicative specification of the database can be found in Specification 10.1.

Specification 10.1 – Abstract applicative specification of the database

```
scheme A_DATABASE1 =
  class
    type Database, Key, Data

    value
      empty : Database,
      insert : Key × Data × Database → Database,
      remove : Key × Database → Database,
      defined : Key × Database → Bool,
      lookup : Key × Database ~→ Data

    axiom
      [ remove_empty ]
        ∀ k : Key • remove(k, empty) ≡ empty,
      [ remove_insert ]
        ∀ k, k1 : Key, d : Data, db : Database •
          remove(k, insert(k1, d, db)) ≡
            if k = k1 then remove(k, db)
            else insert(k1, d, remove(k, db))
            end,
      [ defined_empty ]
        ∀ k : Key • defined(k, empty) ≡ false,
      [ defined_insert ]
        ∀ k, k1 : Key, d : Data, db : Database •
          defined(k, insert(k1, d, db)) ≡
            k = k1 ∨ defined(k, db),
      [ lookup_insert ]
        ∀ k, k1 : Key, d : Data, db : Database •
          lookup(k, insert(k1, d, db)) ≡
            if k = k1 then d else lookup(k, db) end
          pre k = k1 ∨ defined(k, db)
  end
```

Further description of the specification can be found in [Gro92, p. 49-50].

Validation

It can be verified informally that *A_DATABASE1* meets the requirements.

10.2.3 Concrete Applicative Specification of the Database

The next step in the development process is to develop a concrete specification of the database. This includes deciding on a type for representing the database. The choice of type is a map mapping keys to data.

Specification 10.2 – Concrete applicative specification of the database

---

scheme A_DATABASE2 =
  class
    type
      Database = Key $\underset{m}{\rightarrow}$ Data,
      Key,
      Data

    value
      empty : Database = [ ],

      insert : Key × Data × Database → Database
      insert(k, d, db) ≡ db † [k ↦ d],

      remove : Key × Database → Database
      remove(k, db) ≡ db \ {k},

      defined : Key × Database → **Bool**
      defined(k, db) ≡ k ∈ **dom** db,

      lookup : Key × Database $\xrightarrow{\sim}$ Data
      lookup(k, db) ≡ db(k)
      pre defined(k, db)
  end

Further description of the specification can be found in [Gro92, p. 78-79].

Validation

It can be validated that *A_ DATABASE2* implements *A_ DATABASE1*, that is A_DATABASE2 $\preceq$ A_DATABASE1.

10.3 Using the Transformer

The next step in the development process is to develop a concrete imperative version of the database. This step can be done using the transformer. *A_DATABASE2* is not within RSL_A because the pre-condition in the function *lookup* will be transformed into a value expression that is not read-only. The specification can be rewritten by removing the pre-condition. This can then be added later in the development process. The resulting specification, *A_DATABASE3*, can be found in Specification 10.3.

Specification 10.3 – Concrete applicative specification of the database written within RSL_A

---

scheme A_DATABASE3 =
  class
    type
      Database = Key $\overrightarrow{m}$ Data,
      Key,
      Data

    value
      empty : Database = [ ],

      insert : Key × Data × Database → Database
      insert(k, d, db) ≡ db † [k ↦ d],

      remove : Key × Database → Database
      remove(k, db) ≡ db \ {k},

      defined : Key × Database → **Bool**
      defined(k, db) ≡ k ∈ **dom** db,

      lookup : Key × Database $\xrightarrow{\sim}$ Data
      lookup(k, db) ≡ db(k)
  end

10.3.1 Concrete Imperative Specification of the Database

A_DATABASE3 is transformed by the transformer. The natural choice of the type of interest is the *Database* type. The corresponding variable is named *database*. The transformation is done using the following command:

```
java translator.RSLRunner A_DATABASE3 I_DATABASE1 database:Database
```

The result can be found in Specification 10.4.

Specification 10.4 – Concrete imperative specification of the database

scheme I_DATABASE1 =
  class
    type
      Database = Key $\overrightarrow{m}$ Data,
      Key,
      Data

    variable
      database : Database

    value
      empty : **Unit** → **write** database **Unit**
      empty() ≡ database := [ ],

      insert :
        Key × Data → **read** database **write** database **Unit**
      insert(k, d) ≡ database := database † [k ↦ d],

      remove : Key → **read** database **write** database **Unit**
      remove(k) ≡ database := database \ {k},

      defined : Key → **read** database **Bool**
      defined(k) ≡ k ∈ **dom** database,

      lookup : Key $\xrightarrow{\sim}$ **read** database Data
      lookup(k) ≡ database(k)
  end

Validation

Normally the development step from applicative into imperative specification cannot be formally verified, but by using the transformation rules, which are believed to be correct according to Chapter 7, *I_DATABASE1* is a correct development of *A_DATABASE3*.

In order for the specification to be a correct development of *A_DATABASE2*, a transformed version of the pre-condition in the *lookup* function has to be added to *I_DATABASE1*. The resulting specification *I_DATABASE2* can be found in Specification 10.5.

Specification 10.5 – Concrete imperative specification of the database including pre-condition

---

scheme I_DATABASE2 =
  class
    type
      Database = Key $\underset{m}{\rightarrow}$ Data,
      Key,
      Data

    variable
      database : Database

    value
      empty : **Unit** → **write** database **Unit**
      empty() ≡ database := [ ],

      insert :
        Key × Data → **read** database **write** database **Unit**
      insert(k, d) ≡ database := database † [k ↦ d],

      remove : Key → **read** database **write** database **Unit**
      remove(k) ≡ database := database \ {k},

      defined : Key → **read** database **Bool**
      defined(k) ≡ k ∈ **dom** database,

      lookup : Key $\stackrel{\sim}{\rightarrow}$ **read** database Data
      lookup(k) ≡ database(k)
      pre defined(k)
  end

As can be seen the specification resembles that of [Gro92, p. 147-148]. The only differences are the inclusion of some extra **read** database access descriptors in *I_DATABASE2* and the fact that the specification in [Gro92, p. 147-148] is written using axioms.

10.4 Further Development of the Database Specification

From this point the imperative specification of the database can be further developed into a final specification that can be translated into a program. As the specifications further on are within the imperative subset of RSL the development steps can be validated and in this way the whole development process from abstract applicative specification into final concrete imperative specification can be validated. Normally, the step from concrete applicative into concrete imperative is only informally verified, but using the transformer this step is automatically verified.

Chapter 11

Test

This chapter contains descriptions of the different kinds of tests performed on the transformer. Furthermore, discussions on the choice of test methods are given.

11.1 Test Methods

The purpose of testing a software system is to render probable that the system fulfills the requirements and works as expected. It is impossible to show that software systems of even limited size have no errors, but testing a system thoroughly can minimize the likelihood of errors.

The different parts of the transformer are developed differently, some directly and some using code generating tools. This calls for different kinds of test methods of the different parts. Furthermore, some of the parts are reused from the RSL2Java tool as described in Chapter 9, and these parts have been tested in advance.

There are two kinds of tests to consider, black box tests and white box tests. The purpose of a black box test is to test that the functionality of the program works as expected and that the program fulfills the requirements. This is without considering the structure and inner workings of the program. In a white box test the program is tested by basing the test data on the logic of the program.

When dealing with automatically generated code, white box testing is very complicated and time-consuming due to the often complicated structure of automatically generated code. On the other hand a black box test can be too limited. A solution to this problem could be to perform a black box test using test data constructed based on the knowledge of the code and the problems that can arise. This could be called a *grey box test*.

In the following the different parts of the program are considered in order to find the most appropriate way to test the transformer.

11.1.1 Lexer and Parser

The code of the lexer and parser is generated automatically from the grammar file using the ANTLR tool. The grammar file used originates from the RSL2Java tool as described in Chapter 9, but it is extended in order to comply with RSL_A. Furthermore, a few changes have been necessary.

The automatically generated code from the original grammar file is tested in the project described in [Hja04]. Furthermore, as the code is automatically generated a black box test or a grey box test is the most obvious choice.

11.1.2 RSL AST

The many classes forming the RSL AST are generated automatically from the specification of the RSL AST using the RSL2Java tool. The only exception is the abstract class RSLElement.java. The RSL AST specification used originates from the RSL2Java tool as described in Chapter 9, but it is extended in order to comply with RSL_I and furthermore, a few necessary changes have been implemented.

As the code is automatically generated and as the most of the functionality has been tested in the RSL2Java tool project, a black box test would be appropriate.

11.1.3 Visitors

The visitors are implemented by hand, but this implementation is straight forward when the RSL AST is specified. The visitors used originate from the RSL2Java tool as described in Chapter 9, but they are extended in order to comply with the new RSL AST.

As the visitors are implemented by hand a white box test can be performed, but as the implementation of the visitors follows a straight forward pattern, a white box test should not be necessary.

11.1.4 Transformer

The classes forming the transformer are generated automatically from the specification of the transformer in RSL_1 using the RSL2Java tool.

As the transformer is automatically generated a white box test is very complicated, especially considering the amount of code generated. A black box test would not suffice considering the complexity of the transformer. The obvious choice is a grey box test.

11.1.5 Control Module

The control module RSLRunner.java is implemented by hand and the functionality is crucial to the transformer program.

This means that a black box test should be performed to test the functionality, both for correct and incorrect input. A white box test could be performed in order to test the program structure but as the logic of the program is very straight forward and because many of the branches in the program structure can only be reached through system errors a black box test of the control module is to prefer.

11.2 Choice of Tests

The considerations discussed above leads to the following tests of the transformer:

1. A grey box test of the program as a whole. In this way all parts of the program will be tested.
2. A black box test of the control module.

The decision that the system should be tested as a whole is made for several reasons. Parts of the lexer, parser, RSL AST and the visitors are already tested in the RSL2Java tool project. By making a thorough test of the whole system, all parts of these components will be used. Furthermore, the possibility that e.g. an error in the parser would be balanced out later in the process of the transformer is negligible.

11.2.1 Grey Box Test of the Program

The grey box test of the program was performed by transforming different applicative RSL specifications, both transformable and not transformable specifications, and by introducing one or more types of interest. The results of the transformation were then compared with the transformation rules of Chapter 6 if transformable, and with the conditions for transformability if not transformable. If transformable, the result of the transformation was also type checked using the RSL type checker.

The test specifications are chosen such that all corners of RSL_A are reached. That way all parts of the lexer and parser, the visitor modules, the RSL AST code and the transformer module itself are tested.

Furthermore, specifications that could lead to problems are tested. These are chosen based on knowledge of the inner working of the program. An example is that the expected type of a value expression has to be carried on the right way through the transformation. For example, when transforming the product expression *(x, (y, z))* of expected type **Int** × (**Bool** × **Text**), then x should have the expected type **Int** and *(y, z)* should have the expected type **Bool** × **Text**. But then again it is not necessary to test whether y has expected type **Bool** and z has expected type **Text** during the transformation due to the recursive structure of the transformer module.

An overview of the tests and the results can be found in Appendix G.

11.2.2 Black Box Test of the Control Module

The black box test of the control module was performed by initializing the transformer with different inputs. These inputs were constructed such that both the handling of correct and incorrect input were tested.

An overview of the test and the results can be found in Appendix G

11.3 Test Results

All tests turned out as expected. In the grey box test of the program as a whole the applicative specifications were transformed as expected. The black box test of the control module did not reveal any errors.

The conclusion is that the transformer works as expected.

Chapter 12

Possible Extensions of the Transformer

This chapter offers ideas of how to transform some of the constructs not in RSL$_A$. Furthermore, a discussion of further work on the transformer is offered.

12.1 Transformations of Constructs Outside RSL$_A$

The following section offers ideas of transformation rules not implemented in the transformer. The rules are a result of considerations that have been made during the project before the actual RSL$_A$ subset was determined.

12.1.1 Object Definitions

When transforming an object definition the id of the object need not be changed, but it is good practice to rename the object. Any parameters are transformed into themselves. After that the class expression must be transformed.

12.1.2 Extending Class Expressions

When transforming an extending class expression both constituent class expressions must be transformed.

12.1.3 Hiding Class Expressions

When transforming a hiding class expression the names of the defined items in the list are transformed into their counterparts in the imperative specification and then the class expression is transformed.

12.1.4 Renaming Class Expressions

When transforming a renaming class expression the names of the defined items in the list are transformed into their counterparts in the imperative specification and then the class expression is transformed.

12.1.5 Infinite Map Type Expressions

Infinite map type expressions are of the form:

$$\text{type\_expr}_1 \ \underset{m}{\overset{\sim}{\rightarrow}} \ \text{type\_expr}_2$$

An infinite map type expression is transformed by transforming the constituent type expressions.

12.1.6 Single Typings of Higher Order Functions

The idea behind transforming single typings of higher order functions is similar to the idea behind transforming single typings of normal functions, which is explained in Section 6.4.1. Examples of transformations of single typings of higher order functions are given in Table 12.1.

Applicative single typing	▷	Imperative single typing
T → (U → V)	▷	**Unit** → **read** t (U → V)
(U → V) → T	▷	(U → V) → **write** t **Unit**
(T → U) → V	▷	(**Unit** → **read** t U) → V
(U → T) → V	▷	(U → **write** t **Unit**) → V
U → (T → V)	▷	U → (**Unit** → **read** t V)
U → (V → T)	▷	U → (V → **write** t **Unit**)

Table 12.1: Transformation of single typings of higher order functions. T is a type of interest with associated variable t, U and V are not types of interest.

12.1.7 Disambiguation Expressions

A disambiguation expression has the following form:

$$\text{value\_expr} : \text{type\_expr}$$

A disambiguation expression can be transformed by transforming the constituent value and type expressions.

12.1.8 Axioms

Axioms can be transformed almost as value definitions since value definitions are a short way of writing axioms. The extension should therefore be almost straight forward. It is necessary to consider quantified expressions though.

12.1.9 Quantified Expressions

Quantified expressions are of the form:

$$\text{quantifier typing}_1, \ldots, \text{typing}_n \bullet \text{value\_expr}$$

A quantified expression is transformed by transforming the constituent value expression. If the typings contain types of interest the corresponding variables must be assigned the values of the value names in the typing. Due to side effects an equivalence expression must be established, for example:

$$\exists\, x : T \bullet \text{obs}(x)$$
$$\rhd$$
$$\exists\, x : T \bullet (\, t := x \,;\, \text{obs}() \equiv t := x \,;\, \textbf{true}\,)$$

where T is a type of interest and t is the corresponding variable.

If the quantified expression contains universal quantification over a type of interest this is replaced by the always combinator $\square$, see Example 12.1.

Example 12.1 – Transformation of a quantified expression

---

type
  T = **Int**
value
  f : T → **Int**

axiom
  ∀ x : T • f(x) > x

▷

type
  T = **Int**
variable
  t : T
value
  f : **Unit** → **read** t **Int**
axiom
  □ f() > t

12.1.10 Recursive functions

Recursive functions can be part of an imperative specification, but are sometimes not wanted. Using the transformer the recursive functions from the applicative specification stay recursive after the transformation. This is done to simplify the transformer.

Recursions can usually be turned into either for loops or while loops. Another option is to transform recursive functions into non-recursive functions using continuations, as described in [Wan80]. The idea behind continuations is that the future course of the computation is represented while executing the computation. The use of continuations may have the unwanted effect of reducing the readability of the specification compared to using loops.

12.1.11 Collections of a Fixed or Bounded Size

Collections of a fixed or bounded size can be implemented by introducing the proper amount of objects or variables. This requires that the maximum sizes of the collections are known in advance.

12.2 Further Work

As this project was limited in time not all the desired functionality could be implemented. During the process several ideas for further development of the transformer have come to mind. In the following some of these ideas will be discussed.

12.2.1 A Greater Subset of RSL

One obvious extension is to extend the transformer such that it covers a greater subset of RSL. Ideas for some of the transformation rules are given above. This would make the transformer even more useful. Furthermore, it could be useful if the transformer could deal with specifications that were partly applicative and partly imperative. This would be an even better way to support the RAISE Development Method.

12.2.2 Integrating the Transformer with Other RAISE Tools

Another idea is to integrate the transformer with Emacs such that it only requires one Emacs command to start the transformation. Furthermore, it would be pleasant if the transformer process were coupled with the RSL type checker and pretty printer such that type checking and pretty printing were a part of the transformation process.

12.2.3 Optimization of the Transformer

As described in Chapter 9 the transformation process can in some cases be very slow. It would be nice if the transformer could be optimized such that the transformation process for some specifications was a bit faster.

12.2.4 Interactive Transformer

Another extension could be to make the transformer interactive such that the user can choose which of the occurrences of values and type names of the types of interest that have to be transformed as such. This would ease the requirements on the specification style, but it would complicate the use of the transformer.

12.2.5 Full Verification of the Transformation Rules

In Chapter 7 a proof of the correctness of the transformation rules is outlined. If this proof was completed, the step from applicative into imperative specification would be verified automatically if it was done applying the transformation rules, e.g. by using the transformer. This would mean that no further work would have to be done in order to verify the development step, which would ease the development step even further.

12.2.6 Other Extensions

When a specification cannot be transformed it is not specified what and where the problem is unless the problem is caught by the lexer and parser. An extension to the transformer could be to improve the error handling. This would ease the use of the transformer.

Chapter 13

Conclusion

In this chapter the results of the project will be summarized and compared to the requirements, which were set up in the beginning of the project and have been pursued throughout the project. Furthermore, a discussion of the actual result will be given and some related work will be discussed. Finally, some concluding remarks are given.

13.1 Achieved Results

The aims of the project are fulfilled. Transformation rules which applied to an applicative RSL specification return an imperative RSL specification are constructed. It has been a goal to make the resulting imperative specification as readable and recognizable as possible. It is shown through many examples and test cases that this is achieved.

A notion of correctness of the transformation from applicative into imperative specification is given together with an outline of a how the correctness of the transformation rules could be verified. Nothing has indicated that the transformation rules are wrong and I am convinced that they actually can be verified.

A tool, the transformer, has been developed to support the transformation rules such that the actual transformation from applicative into imperative specification can be done automatically. The transformer has been thoroughly tested and the tests revealed no errors. All transformations followed the transformation rules and the resulting imperative specifications could all be type checked using the RSL type checker.

The transformer is built such that it can be easily extended. The transformation rules are constructed such that an extension does not require changing the transformation rules. The only thing to do is to put up transformation rules that covers the extension and add the transformation rules to the transformer. This requires extension of the ANTLR grammar file, extension of the visitor modules and of course extensions of the specification of the

RSL AST and the transformer.

The conclusion is that the aims of the project have been fulfilled.

13.2 Discussion of the Result

The transformer can perform a single step in a software development process as demonstrated in Chapter 10. This step is somewhat intuitive and easy to do. The problem is that the correctness of this step cannot be verified as all the other steps in the development process can by using the refinement relation.

The contribution of this project is that some concrete and correct transformation rules are put up that applied to an applicative RSL specification gives an imperative RSL specification. By using the transformation rules the verification of this development step from applicative to imperative specification is superfluous, on the assumption that the transformation rules or correct and can be verified. Furthermore, a tool implementing the transformation rules has been constructed such that the development from applicative to imperative specification can be automated. This means that the work required in the development step from applicative to imperative specification is minimized and that it is assured that the transformation rules are applied correctly.

The conclusion is that the contribution of the transformer is:

1. that the correctness of the entire development process from initial to final specification can be verified.

2. that the development step from applicative to imperative specification is automated.

The result of the transformation using the transformer is readable and recognizable. This means that the further development of the imperative specification is not made more difficult compared to the further development of a hand made imperative specification.

13.3 Related Work

This project is not the only work in which transformations from applicative specifications or programs into imperative counterparts are considered.

Transformation from applicative RSL specifications into imperative RSL specifications is considered in [Gro95]. This work served as inspiration during the initial phase of this project, but it became obvious that the conditions for transformability, in [Gro95] named linearity, were far too strict, resulting in a much smaller subset of RSL to be transformable.

Translation from applicative code into imperative code has been considered before. The reason is that assembly code is imperative. This means that when compiling applicative programming languages such as Scheme, SML and some parts of Lisp transformation from applicative program into imperative assembly code is necessary. Many of the techniques used when compiling applicative programs cannot be exploited when developing transformation rules for this project as assembly code and RSL specifications are very different. But a few techniques can. This includes techniques for removing recursions, techniques that can be applied if extending the transformer to cover a greater part of the transformation from applicative RSL specification into imperative RSL specification. One of these techniques is the use continuations as described in [Wan80]. The drawback of using continuations compared to e.g. loops is a lack of clearness in most cases.

The work described in [Lin04] and [LH04] considers the possibility of translating specifications written within a the mRSL subset of RSL and proof obligations into *higher-order logic* (HOL) specifications and proof obligations in order to be able to use the generic proof assistant *Isabelle*. In this way the the proof support for the RAISE Development Method could be improved. The project is primarily theoretical and defines among other things an institution for mRSL as mentioned in Chapter 7.

The results of the related work are used to the extend possible, but as much of the work of this project never has been done before, it has been done without the possibility of support from related work.

13.4 Concluding Remarks

This project has been an instructive experience for me. First of all I have had the opportunity to try my strength against theoretical work, which have been both challenging and fun. Furthermore, the project has given me a better understanding of language processors and translation. Last but not least, by working a lot with RSL and the RAISE method I have gained a lot of experience in these – an experience I hope to be able to utilize when entering the labour markets.

It is my belief that the use of formal methods will increase in the years to come due to two facts. First of all the amount of safety critical systems will increase and at the same time the formal methods will be improved as more and more tools will be developed. My hope is, that this project and the resulting transformer will have a positive influence on this development.

Bibliography

[DOF] *Data & Object Factory.* www.dofactory.com.

[Gro92] The RAISE Language Group. *The RAISE Specification Language.* The BCS Practitioners Series. Prentice-Hall, 1992.

[Gro95] The RAISE Method Group. *The RAISE Development Method.* The BCS Practitioners Series. Prentice-Hall International, 1995.

[Hax99] Anne E. Haxthausen. Lecture Notes on The RAISE Development Method, April 1999. Lecture notes.

[Hax05] Anne E. Haxthausen. Towards a Formal Development Relation between Applicative and Imperative RSL Specifications. Handwritten notes, June 2004 - January 2005.

[Hja04] Ulrik Hjarnaa. Translation of a Subset of RSL into Java. Master's thesis, DTU, 2004.

[Jon86] Cliff B. Jones. *Systematic Software Development Using VDM.* Series in Computer Science. Prentice-Hall, 1986.

[LH04] M. P. Lindegaard and A. E. Haxthausen. Proof Support for RAISE – by a Reuse Approach based on Institutions. In *Proceedings of AMAST'04*, number 3116 in Lecture Notes in Computer Science, pages 319–333. Springer-Verlag, 2004.

[Lin04] Morten P. Lindegaard. *Proof Support For RAISE.* PhD thesis, Technical University of Denmark, Informatics and Mathematical Modelling, 2004.

[Mil90] R. Milne. Semantic Foundation of RSL. Technical report, CRI, March 1990.

[Par] Terence Parr. *ANTLR Reference Manual.* www.antlr.org.

[ST] Donald Sannello and Andrzej Tarlecki. Foundations of Algebraic Specifications and Formal Program Development. To appear.

[ST02] Donald Sannello and Andrzej Tarlecki. *Algebraic Foundations of Systems Specification*, chapter 2. Springer-Verlag, 2002.

[Tar86] Andrzej Tarlecki, 1986. Lecture notes.

[Tar02] Andrzej Tarlecki. *Algebraic Foundations of Systems Specification*, chapter 4. Springer-Verlag, 2002.

[Wan80] Mitchell Wand. Continuation-Based Program Transformation Strategies. *Journal of the Association for Computing Machinery*, 27(1):164–180, January 1980.

[WB00] David A. Watt and Deryck F. Brown. *Programming Language Processors in Java*. Prentice-Hall, 2000.

Appendix A

Using and Extending the Transformer

In this appendix descriptions of how to set up the transformer, how to use the transformer and how to extend the transformer can be found.

A.1 Setting Up the Transformer

Before the transformer can be used a few things have to be set up. The executable version of the transformer has been compiled with J2SE 5.0. This means that Java 1.5.0 or a later edition must be installed on the computer. Then the following steps have to be taken:

- Copy the executable code, which can be found on the enclosed CD-ROM in the `executable_code` directory, to a directory on the local computer.
- Add the directory containing the transformer to the class path of the system.

The transformer can then be tested as follows:

- Copy the `A_TEST.rsl` file from the `example` directory on the enclosed CD-ROM to a directory on the local computer.
- Run the transformer with the command

  `java translator.RSLRunner A_TEST I_TEST t:T`.
- The result of the transformation can be found in the file `I_TEST.rsl` in the same directory.

A.2 Using the Transformer

In order to transform a specification using the transformer the following command must be executed:

```
java translator.RSLRunner RSL-file new-RSL-file [variable-list]
```

where `RSL-file` is the name of an RSL file without the .rsl extension. `new-RSL-file` is the the file name without the .rsl extension to which the transformed specification is written. `variable-list` is a list of the form `var1:type1 var2:type2...` where `var1` is the name of the variable that `type1` should be represented by and so on.

A.3 Extending the Transformer

To extend the transformer the following programs need to be installed:

- ANTLR, which can be found in `www.antlr.org`.
- The RSL2Java tool, [Hja04].

An extension requires the following steps. The directories mentioned are relative to the `source_code` directory, which can be found on the enclosed CD-ROM.

- Extension of the lexer and parser, which means an extension of the ANTLR grammar file `translator\syntacticanalyzer\rsltorslast.g`.
- Extension of the RSL AST specification, `RSLAst_Module.rsl`.
- Extension of the transformer specification, `TransformerRSL1.rsl`.
- Extension of the visitor modules, `translator\lib\RSLAstVisitor.java` and `translator\lib\StringRSLAstVisitor.java`.

When these extensions have been done the transformer must be generated. This is done as follows.

- Generation of the lexer and the parser by executing the command

  `java antlr.Tool rsltorslast.g`

  in the directory `translator\syntacticanalyzer`.
- Translation of the RSL AST into Java by executing the command

  `java translator.Runner2 RSLAst_Module rslast.properties`

  in the `source_code` directory.

- Translation of the transformer specification into Java by executing the command

  ```
  java translator.Runner2 TransformerRSL1 transformer.properties
  ```

  in the `source_code` directory.

- Compiling all the Java source code by executing the command

  ```
  javac translator/lib/*.java
  translator/rslast/*.java
  translator/rsllib/*.java
  translator/*.java
  translator/syntacticanalyzer/*.java
  translator/rsltransformer/*.java
  ```

  in the `source_code` directory.

An overview of the packages of the program and the destinations of the generated Java files can be found in Figure A.1 on the following page.

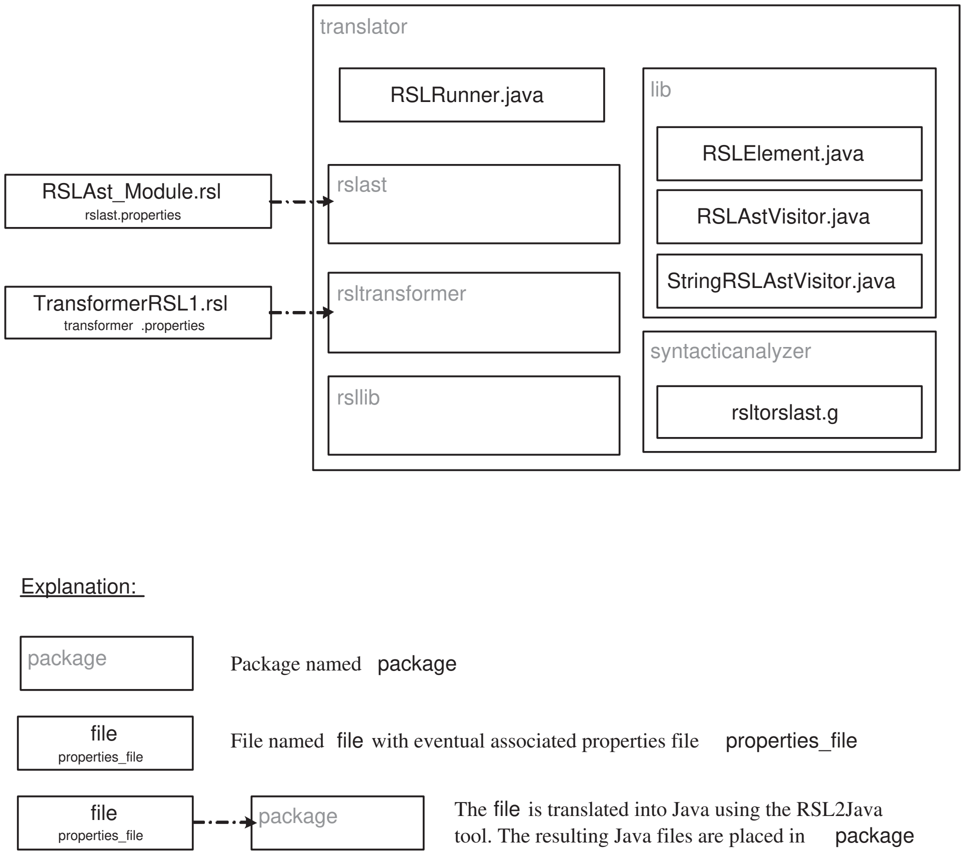

Figure A.1: Overview of the program

Appendix B

Contents of CD-ROM

The enclosed CD-ROM contains the following files. Directories are written as `directory`:

`source_code` contains the source code including specifications in RSL.

`executable_code` contains compiled code for the transformer, compiled for J2SE 5.0.

`specifications` contains the specifications used to generate the program and the specification of the transformer in RSL.

`test_files` contains the test files and the result of the transformation of these when possible.

`example` contains an example RSL specification which can be used to test the set up of the transformer.

`reports` contains .ps and .pdf versions of the final report. The .pdf version is made using LaTeX hyperrefs for ease of reading.

`RSL2Java_translator` contains a compiled version of the RSL2Java tool, compiled for J2SE 5.0. This version is the one used to generate the transformer.

Appendix C

Formal Specifications of Transformations

This appendix contains the specifications of the project written in RSL.

C.1 Formal Specification of the Transformer

Specification C.1 – Transformer.rsl

scheme Transformer =
  extend RSLAst_Module **with**
  class
    type

      /∗****Global maps.****∗/

      /∗Map from typename id to variable id. Given by
      the user.∗/
      TRANS = Id $\overrightarrow{m}$ Id,

      /∗Map from type name id to all involved types
      ids in type expression.∗/
      TYPINGS = Id $\overrightarrow{m}$ Id*,

      /∗Map from function name to corresponding type
      expression.∗/
      FUNC = Id $\overrightarrow{m}$ FuncSpec,
      FuncSpec ::
        type_expr : TypeExpr

```
      value_expr : ValueExpr
      read_list : Access*
      write_list : Access*,

   /*****Local maps.*****/
   /*Local to every value definition.*/

   /*Map from typename id to current corresponding
   value name.*/
   ENV = Id →m Binding,

   /*Map from value name to corresponding type expression.
   */
   TYPES = Binding →m ExpType,
   ExpType == Known(type_expr : TypeExpr) | Unknown,
   AccessResult ::
      accessList : Access*   idSet : Id-set,
   DResult ==
      Transformable(result : Decl) | Not_transformable,
   TRResult ==
      RSLast_transformable(result : RSLAst) |
      RSLast_not_transformable,
   ClassResult ==
      Class_transformable(result : ClassExpr) |
      Class_not_transformable,
   DLResult ==
      DL_transformable(result : Decl*) |
      DL_not_transformable,
   DeclResult ==
      Decl_transformable(result : Decl) |
      Decl_not_transformable,
   MapType :: tedom : ExpType   terange : ExpType

value

   /*Transforms an RSL AST if possible.*/
   /*
   Arguments:
   =======
   rslast: the RSL AST
   schemeid: the new Id of the scheme. Given by
   the user
   trans: the TRANS map. Given by the user
   Results:
```

```
=====
TRResult: the result of the transformation
*/
TRRSLAst : RSLAst × Text × TRANS → TRResult
TRRSLAst(rslast, schemeid, trans) ≡
   case
      TRClassExpr(
          class_expr(schemedef(libmodule(rslast))), trans)
   of
      Class_transformable(ce) →
         RSLast_transformable(
             mk_RSLAst(
                  mk_LibModule(
                      context_list(libmodule(rslast)),
                      mk_SchemeDef(mk_Id(schemeid), ce)))),
      Class_not_transformable → RSLast_not_transformable
   end,

/*Transforms a class expression if possible.*/
/*
Arguments:
=======
ce: the class expression
Results:
=====
ClassResult: the result of the transformation
*/
TRClassExpr : ClassExpr × TRANS → ClassResult
TRClassExpr(ce, trans) ≡
   case ce of
      BasicClassExpr(dl) →
         let
            func = establishFuncMap(dl, trans),
            typings =
               expandTYPINGSMap(
                   getIds(getTypeDecl(dl)),
                   makeTYPINGSMap(getTypeDecl(dl)), trans)
         in
            case
               TRDeclList(dl, typings, func, trans, ⟨⟩)
            of
               DL_transformable(dlres) →
                  Class_transformable(
                      BasicClassExpr(
```

```
                              makeVariables(dlres, trans))),
                    DL_not_transformable →
                       Class_not_transformable
                 end
              end,
           _ → Class_not_transformable
        end,

     /*Transforms a declaration list if possible.*/
     /*
     Arguments:
     =======
     dl: the declaration list
     dlres: the resulting declaration list
     Results:
     =====
     DLResult: the result of the transformation
     */
     TRDeclList :
        Decl* × TYPINGS × FUNC × TRANS × Decl* →
           DLResult
     TRDeclList(dl, typings, func, trans, dlres) ≡
        if dl = ⟨⟩ then DL_transformable(dlres)
        else
           case TRDecl(hd dl, typings, func, trans) of
              Decl_transformable(decl) →
                 TRDeclList(
                    tl dl, typings, func, trans,
                    dlres ⌢ ⟨decl⟩),
              Decl_not_transformable → DL_not_transformable
           end
        end,

     /*Transforms a declaration if possible.*/
     /*
     Arguments:
     =======
     d: the declaration
     Results:
     =====
     DeclResult: the result of the transformation
     */
     TRDecl :
        Decl × TYPINGS × FUNC × TRANS → DeclResult
```

```
TRDecl(d, typings, func, trans) ≡
   case d of
      TypeDecl(tdl) →
         case TRTypeDecl(tdl, typings, func, trans) of
            Transformable(decl) →
               Decl_transformable(decl),
            Not_transformable → Decl_not_transformable
         end,
      ValueDecl(vd) →
         case TRValueDecl(vd, typings, func, trans) of
            Transformable(decl) →
               Decl_transformable(decl),
            Not_transformable → Decl_not_transformable
         end,
      _ → Decl_not_transformable
   end,

/*Makes a variable declaration according to the
trans map.*/
/*
Arguments:
=======
decl: the original declaration list
Results:
=====
Decl_list: the resulting declaration list
*/
makeVariables : Decl* × TRANS → Decl*
makeVariables(decl, trans) ≡
   if decl = ⟨⟩
   then ⟨VariableDecl(makeVariableDeclList(trans))⟩
   else
      case hd decl of
         TypeDecl(tdl) →
            ⟨hd decl⟩ ⁀ makeVariables(tl decl, trans),
         ValueDecl(vdl) →
            ⟨VariableDecl(makeVariableDeclList(trans))⟩ ⁀
            decl
      end
   end,

/*Makes a variable definition list according to
the trans map.*/
/*
```

```
Results:
=====
VariableDef_list: the resulting variable definition
list
*/
makeVariableDeclList : TRANS → VariableDef*
makeVariableDeclList(trans) ≡
   if trans = [ ] then ⟨⟩
   else
      ⟨makeVariableDef(hd trans, trans)⟩ ⁀
      makeVariableDeclList(trans \ {hd trans})
   end,

/*Makes a variable definition from a type name
id.*/
/*
Arguments:
=======
typename: the type name id
Results:
=====
VariableDef: the resulting variable definition
*/
makeVariableDef : Id × TRANS → VariableDef
makeVariableDef(typename, trans) ≡
   SingleVariableDef(
      trans(typename), TypeName(typename),
      NoInitialisation),

/*Transforms a type declaration if possible.*/
/*
Arguments:
=======
tdl: the type declaration
Results:
=====
DResult: the result of the transformation
*/
TRTypeDecl :
   TypeDef* × TYPINGS × FUNC × TRANS → DResult
TRTypeDecl(tdl, typings, func, trans) ≡
   if CheckTypeDefList(tdl, typings, func, trans)
   then Transformable(TypeDecl(tdl))
   else Not_transformable
```

```
   end,

/*Checks for transformability of a type definfition
list.*/
/*
Arguments:
=======
tdl: the list of type definitions
Results:
=====
Bool: true if the list is transformable, false
otherwise
*/
CheckTypeDefList :
   TypeDef* × TYPINGS × FUNC × TRANS → Bool
CheckTypeDefList(tdl, typings, func, trans) ≡
   if tdl = ⟨⟩ then true
   else
      CheckTypeDef(hd tdl, typings, func, trans) ∧
      CheckTypeDefList(tl tdl, typings, func, trans)
   end,

/*Checks for transformability of a type definfition.
*/
/*
Arguments:
=======
td: the type definition
Results:
=====
Bool: true if the type definition is transformable,
false otherwise
*/
CheckTypeDef :
   TypeDef × TYPINGS × FUNC × TRANS → Bool
CheckTypeDef(td, typings, func, trans) ≡
   case td of
      SortDef(id) → id ∉ dom trans,
      VariantDef(id, vl) →
         CheckVariantList(id, vl, typings, func, trans) ∧
         id ∉ elems typings(id),
      ShortRecordDef(id, cl) →
         CheckComponentKindList(
            id, cl, typings, func, trans) ∧
```

```
            id ∉ elems typings(id),
        AbbreviationDef(id, te) →
            CheckTypeExpr(te, typings, func, trans) ∧
            id ∉ elems typings(id)
    end,

/*Checks for transformability of a variant list.
*/
/*
Arguments:
=======
id: the id of the variant definition
vl: the list of variants
Results:
=====
Bool: true if the list is transformable, false
otherwise
*/
CheckVariantList :
    Id × Variant* × TYPINGS × FUNC × TRANS →
        Bool
CheckVariantList(id, vl, typings, func, trans) ≡
    if vl = ⟨⟩ then true
    else
        CheckVariant(id, hd vl, typings, func, trans) ∧
        CheckVariantList(id, tl vl, typings, func, trans)
    end,

/*Checks for transformability of a variant.*/
/*
Arguments:
=======
id: the id of the variant definition
v: the variant
Results:
=====
Bool: true if the variant is transformable, false
otherwise
*/
CheckVariant :
    Id × Variant × TYPINGS × FUNC × TRANS → Bool
CheckVariant(id, v, typings, func, trans) ≡
    case v of
        RecordVariant(c, cl) →
```

```
            CheckComponentKindList(
                id, cl, typings, func, trans),
        _ → true
    end,

/*Checks for transformability of a component kind
list.*/
/*
Arguments:
=======
id: the id of the type definition
cl: the list of component kinds
Results:
=====
Bool: true if the list is transformable, false
otherwise
*/
CheckComponentKindList :
    Id × ComponentKind* × TYPINGS × FUNC × TRANS →
        Bool
CheckComponentKindList(id, cl, typings, func, trans) ≡
    if cl = ⟨⟩ then true
    else
        CheckComponentKind(id, hd cl, typings, func, trans) ∧
        CheckComponentKindList(
            id, tl cl, typings, func, trans)
    end,

/*Checks for transformability of a component kind.
*/
/*
Arguments:
=======
id: the id of the type definition
c: the component kind
Results:
=====
Bool: true if the component kind is transformable,
false otherwise
*/
CheckComponentKind :
    Id × ComponentKind × TYPINGS × FUNC × TRANS →
        Bool
CheckComponentKind(id, c, typings, func, trans) ≡
```

```
    CheckTypeExpr(type_expr(c), typings, func, trans),

/∗Checks for transformability of a type expression.
∗/
/∗
Arguments:
=======
te: the type expression
Results:
=====
Bool: true if the type expression is transformable,
false otherwise
∗/
CheckTypeExpr :
    TypeExpr × TYPINGS × FUNC × TRANS → Bool
CheckTypeExpr(te, typings, func, trans) ≡
    case te of
        TypeLiteral(tn) → true,
        TypeName(tn) → true,
        TypeExprProduct(tep) →
            CheckTypeExprList(tep, typings, func, trans),
        TypeExprSet(tes) →
            case tes of
                FiniteSetTypeExpr(fse) →
                    ∼ containsTRANSName(
                            Known(fse), typings, dom trans),
                InfiniteSetTypeExpr(ise) →
                    ∼ containsTRANSName(
                            Known(ise), typings, dom trans)
            end,
        TypeExprList(les) →
            case les of
                FiniteListTypeExpr(fle) →
                    ∼ containsTRANSName(
                            Known(fle), typings, dom trans),
                InfiniteListTypeExpr(ile) →
                    ∼ containsTRANSName(
                            Known(ile), typings, dom trans)
            end,
        TypeExprMap(tem) →
            case tem of
                FiniteMapTypeExpr(tedom, terange) →
                    ∼ (containsTRANSName(
                            Known(tedom), typings, dom trans) ∨
```

```
                containsTRANSName(
                    Known(terange), typings, dom trans)),
            InfiniteMapTypeExpr(tedom, terange) →
                ∼ (containsTRANSName(
                    Known(tedom), typings, dom trans) ∨
                containsTRANSName(
                    Known(terange), typings, dom trans))
        end,
    FunctionTypeExpr(arg, fa, res) → false,
    SubtypeExpr(st, ve) →
        let
            (b, env, types) =
                CheckValueExpr(
                    ve, Known(TypeLiteral(BOOL)), typings,
                    func, trans, [ ], [ ])
        in
            ∼ containsTRANSName(
                    Known(type_expr(st)), typings, dom trans) ∧
            b
        end,
    BracketedTypeExpr(bte) →
        CheckTypeExpr(bte, typings, func, trans)
end,

/*Checks for transformability of a type expression
list.*/
/*
Arguments:
=======
tel: the type expression list
Results:
=====
Bool: true if the type expression list is transformable,
false otherwise
*/
CheckTypeExprList :
    TypeExpr* × TYPINGS × FUNC × TRANS → Bool
CheckTypeExprList(tel, typings, func, trans) ≡
    if tel = ⟨⟩ then true
    else
        CheckTypeExpr(hd tel, typings, func, trans) ∧
        CheckTypeExprList(tl tel, typings, func, trans)
    end,
```

```
/*Transforms a list of value declarations.*/
/*
Arguments:
=======
vdl: the list of value declarations
Results:
=====
DResult: the result
*/
TRValueDecl :
   ValueDef* × TYPINGS × FUNC × TRANS → DResult
TRValueDecl(vdl, typings, func, trans) ≡
   if CheckValueDefList(vdl, typings, func, trans)
   then
      /*The value declaration list is transformable
      and is transformed.*/
      Transformable(
          ValueDecl(TRValueDefList(vdl, func, trans)))
   else
      /*The value declaration list is not transformable.
      */
      Not_transformable
   end,

/*Checks for transformability of a value definfition
list.*/
/*
Arguments:
=======
vdl: the list of value definitions
Results:
=====
Bool: true if the list is transformable, false
otherwise
*/
CheckValueDefList :
   ValueDef* × TYPINGS × FUNC × TRANS → Bool
CheckValueDefList(vdl, typings, func, trans) ≡
   if vdl = ⟨⟩ then true
   else
      CheckValueDef(hd vdl, typings, func, trans) ∧
      CheckValueDefList(tl vdl, typings, func, trans)
   end,
```

```
/*Checks for transformability of a value definition.
*/
/*
Arguments:
=======
vd: the value definition
Results:
=====
Bool: true if the value definition is transformable,
false otherwise
*/
CheckValueDef :
   ValueDef × TYPINGS × FUNC × TRANS → Bool
CheckValueDef(vd, typings, func, trans) ≡
   case vd of
      ExplicitValueDef(st, ve) →
         if
            containsTRANSName(
                Known(type_expr(st)), typings, dom trans)
         then
            case binding(st) of
               IdBinding(id) →
                  let
                     (b, env, types) =
                        CheckValueExpr(
                            ve, Known(type_expr(st)), typings,
                            func, trans, [ ], [ ])
                  in
                     b
                  end,
               Make_ProductBinding(pb) → false
            end
         else true
         end,
      ExplicitFunctionDef(st, ffa, ve, precond) →
         case type_expr(st) of
            FunctionTypeExpr(arg, fa, res) →
               let
                  /*Establishes ENV and TYPES.*/
                  (b1, ffap, env, types, prlet) =
                     TRFormalFuncAppl(ffa, func, trans),
                  (b2, env′, types′) =
                     CheckOptPreCondition(
                         precond, Known(TypeLiteral(BOOL)),
```

```
                        typings, func, trans, env, types),
                    (b3, env″, types″) =
                      CheckValueExpr(
                        ve, Known(type_expr(res)), typings,
                        func, trans, env, types′) in
                    b1 ∧ b2 ∧ b3 ∧
                    CheckTypeExpr(arg, typings, func, trans) ∧
                    CheckTypeExpr(
                      type_expr(res), typings, func, trans)
                  end
              end
          end,

        /*Checks for transformability of a pre condition.
        */
        /*
        Arguments:
        =======
        precond: the pre condition
        et: the expected type of the value expression
        Results:
        =====
        Bool: true if the precondition is transformable,
        false otherwise
        */
        CheckOptPreCondition :
          OptionalPreCondition × ExpType × TYPINGS ×
          FUNC × TRANS × ENV × TYPES →
            Bool × ENV × TYPES
        CheckOptPreCondition(
           precond, et, typings, func, trans, env, types) ≡
          case precond of
            PreCondition(ve) →
              let
                (b, env′, types′) =
                  CheckValueExpr(
                    ve, et, typings, func, trans, env, types)
              in
                (b ∧ CheckPreCondGen(ve, func, trans, types),
                 env′, types′)
              end,
            NoPreCondition → (true, env, types)
          end,
```

```
/*Checks if a pre condition contains generators
or hidden generators.*/
/*
Arguments:
=======
ve: the value expression of the pre condition
Results:
=====
Bool: true if the precondition does not contain
generators, false otherwise
*/
CheckPreCondGen :
   ValueExpr × FUNC × TRANS × TYPES → Bool
CheckPreCondGen(ve, func, trans, types) ≡
   case ve of
      Make_ValueLiteral(vl) → true,
      Make_ValueOrVariableName(vn) → true,
      Make_BasicExpr(be) → true,
      ProductExpr(vel) →
         CheckPreCondGenProduct(vel, func, trans, types),
      Make_SetExpr(setExpr) →
         CheckPreCondGenSet(setExpr, func, trans, types),
      Make_ListExpr(listExpr) →
         CheckPreCondGenList(listExpr, func, trans, types),
      Make_MapExpr(mapExpr) →
         CheckPreCondGenMap(mapExpr, func, trans, types),
      ApplicationExpr(ave, vl) →
         CheckPreCondApplicationExpr(
             ave, vl, func, trans, types),
      BracketedExpr(bve) →
         CheckPreCondGen(bve, func, trans, types),
      ValueInfixExpr(first, op, second) →
         CheckPreCondGen(first, func, trans, types) ∧
         CheckPreCondGen(second, func, trans, types),
      ValuePrefixExpr(op, operand) →
         CheckPreCondGen(operand, func, trans, types),
      LetExpr(ldl, lve) →
         CheckPreCondGenLetDef(ldl, func, trans, types) ∧
         CheckPreCondGen(lve, func, trans, types),
      Make_IfExpr(ie) →
         CheckPreCondGen(
             condition(ie), func, trans, types) ∧
         CheckPreCondGen(if_case(ie), func, trans, types) ∧
         CheckPreCondGenElsif(
```

```
            elsif_list(ie), func, trans, types) ∧
          CheckPreCondGen(
            else_case(ie), func, trans, types),
        CaseExpr(cond, cbl) →
          CheckPreCondGen(cond, func, trans, types) ∧
          CheckPreCondGenCaseBranch(
            cbl, func, trans, types)
      end,

/*Checks if a value expression list contains generators
or hidden generators.*/
/*
Arguments:
=======
vel: the value expression list
Results:
=====
Bool: true if the value expression list does
not contain generators, false otherwise
*/
CheckPreCondGenProduct :
  ValueExpr* × FUNC × TRANS × TYPES → Bool
CheckPreCondGenProduct(vel, func, trans, types) ≡
  if vel = ⟨⟩ then true
  else
    CheckPreCondGen(hd vel, func, trans, types) ∧
    CheckPreCondGenProduct(tl vel, func, trans, types)
  end,

/*Checks if a set expression contains generators
or hidden generators.*/
/*
Arguments:
=======
se: the set expression
Results:
=====
Bool: true if the set expression does not contain
generators, false otherwise
*/
CheckPreCondGenSet :
  SetExpr × FUNC × TRANS × TYPES → Bool
CheckPreCondGenSet(se, func, trans, types) ≡
  case se of
```

```
        RangedSetExpr(first, second) →
            CheckPreCondGen(first, func, trans, types) ∧
            CheckPreCondGen(second, func, trans, types),
        EnumeratedSetExpr(ovel) →
            case ovel of
                ValueExprList(vel) →
                    CheckPreCondGenProduct(
                        vel, func, trans, types),
                NoValueExprList → true
            end,
        ComprehendedSetExpr(ve, t, or) →
            CheckPreCondGen(ve, func, trans, types) ∧
            CheckPreCondOptRestriction(
                or, func, trans, types)
    end,

/*Checks if a list expression contains generators
or hidden generators.*/
/*
Arguments:
=======
le: the list expression
Results:
=====
Bool: true if the list expression does not contain
generators, false otherwise
*/
CheckPreCondGenList :
    ListExpr × FUNC × TRANS × TYPES → Bool
CheckPreCondGenList(le, func, trans, types) ≡
    case le of
        RangedListExpr(first, second) →
            CheckPreCondGen(first, func, trans, types) ∧
            CheckPreCondGen(second, func, trans, types),
        EnumeratedListExpr(ovel) →
            case ovel of
                ValueExprList(vel) →
                    CheckPreCondGenProduct(
                        vel, func, trans, types),
                NoValueExprList → true
            end,
        ComprehendedListExpr(ve, b, vei, or) →
            CheckPreCondGen(ve, func, trans, types) ∧
            CheckPreCondGen(vei, func, trans, types) ∧
```

```
            CheckPreCondOptRestriction(
                or, func, trans, types)
    end,

/*Checks if a map expression contains generators
or hidden generators.*/
/*
Arguments:
=======
me: the map expression
Results:
=====
Bool: true if the map expression does not contain
generators, false otherwise
*/
CheckPreCondGenMap :
    MapExpr × FUNC × TRANS × TYPES → Bool
CheckPreCondGenMap(me, func, trans, types) ≡
    case me of
        EnumeratedMapExpr(ovel) →
            case ovel of
                ValueExprPairList(vel) →
                    CheckPreCondPairList(vel, func, trans, types),
                NoValueExprPairList → true
            end,
        ComprehendedMapExpr(vep, t, or) →
            CheckPreCondGen(first(vep), func, trans, types) ∧
            CheckPreCondGen(second(vep), func, trans, types) ∧
            CheckPreCondOptRestriction(
                or, func, trans, types)
    end,

/*Checks if an optional restriction contains generators
or hidden generators.*/
/*
Arguments:
=======
or: the optional restriction
Results:
=====
Bool: true if the optional restriction does not
contain generators, false otherwise
*/
CheckPreCondOptRestriction :
```

```
   OptionalRestriction × FUNC × TRANS × TYPES → Bool
CheckPreCondOptRestriction(or, func, trans, types) ≡
   case or of
      Restriction(ve) →
         CheckPreCondGen(ve, func, trans, types),
      NoRestriction → true
   end,

/*Checks if a pair list contains generators or
hidden generators.*/
/*
Arguments:
=======
vepl: the pair list
Results:
=====
Bool: true if the pair list does not contain
generators, false otherwise
*/
CheckPreCondPairList :
   ValueExprPair* × FUNC × TRANS × TYPES → Bool
CheckPreCondPairList(vepl, func, trans, types) ≡
   if vepl = ⟨⟩ then true
   else
      CheckPreCondGen(first(hd vepl), func, trans, types) ∧
      CheckPreCondGen(
         second(hd vepl), func, trans, types) ∧
      CheckPreCondPairList(tl vepl, func, trans, types)
   end,

/*Checks if an application expression contains
generators or hidden generators.*/
/*
Arguments:
=======
ve: the application id
vel: the arguments of the application expression
Results:
=====
Bool: true if the application expression does
not contain generators, false otherwise
*/
CheckPreCondApplicationExpr :
   ValueExpr × ValueExpr* × FUNC × TRANS × TYPES →
```

```
        Bool
CheckPreCondApplicationExpr(
    ve, vel, func, trans, types) ≡
   if CheckPreCondGenProduct(vel, func, trans, types)
   then
      case ve of
         Make_ValueOrVariableName(vn) →
            if id(vn) ∈ dom func
            then
               /*The application expression is a function application.
               */
               if write_list(func(id(vn))) ≠ ⟨⟩
               then false
               else
                  case type_expr(func(id(vn))) of
                     FunctionTypeExpr(arg, fa, res) →
                        ∼ containsGen(
                              vel, typeExprToExpTypeList(arg),
                              trans, func) ∨
                        onlyTOIArgument(vel, trans, types)
                  end
               end
            else  /*List or map application.*/
               true
            end
      end
   else false
   end,

/*Checks if a let def list contains generators
or hidden generators.*/
/*
Arguments:
=======
ldl: the let def list
Results:
=====
Bool: true if the let def list does not contain
generators, false otherwise
*/
CheckPreCondGenLetDef :
   LetDef* × FUNC × TRANS × TYPES → Bool
CheckPreCondGenLetDef(ldl, func, trans, types) ≡
   if ldl = ⟨⟩ then true
```

```
   else
      CheckPreCondGen(
          value_expr(hd ldl), func, trans, types) ∧
      CheckPreCondGenLetDef(tl ldl, func, trans, types)
   end,

/*Checks if an elsif list contains generators
or hidden generators.*/
/*
Arguments:
=======
eil: the elsif list
Results:
=====
Bool: true if the elsif list does not contain
generators, false otherwise
*/
CheckPreCondGenElsif :
   Elsif* × FUNC × TRANS × TYPES → Bool
CheckPreCondGenElsif(eil, func, trans, types) ≡
   if eil = ⟨⟩ then true
   else
      CheckPreCondGen(
          condition(hd eil), func, trans, types) ∧
      CheckPreCondGen(
          elsif_case(hd eil), func, trans, types) ∧
      CheckPreCondGenElsif(tl eil, func, trans, types)
   end,

/*Checks if a case branch list contains generators
or hidden generators.*/
/*
Arguments:
=======
cbl: the case branch list
Results:
=====
Bool: true if the case branch list does not contain
generators, false otherwise
*/
CheckPreCondGenCaseBranch :
   CaseBranch* × FUNC × TRANS × TYPES → Bool
CheckPreCondGenCaseBranch(cbl, func, trans, types) ≡
   if cbl = ⟨⟩ then true
```

```
    else
        CheckPreCondGen(
            value_expr(hd cbl), func, trans, types) ∧
        CheckPreCondGenCaseBranch(
            tl cbl, func, trans, types)
    end,

/*Checks for transformability of a value expression.
*/
/*
Arguments:
=======
ve: the value expression
et: the expected type of the value expression
Results:
=====
Bool: true if the list is transformable, false
otherwise
*/
CheckValueExpr :
    ValueExpr × ExpType × TYPINGS × FUNC × TRANS ×
    ENV × TYPES →
        Bool × ENV × TYPES
CheckValueExpr(
     ve, et, typings, func, trans, env, types) ≡
    case ve of
        Make_ValueLiteral(vl) →
            (true, updateENV(env, et, trans), types),
        Make_ValueOrVariableName(vn) →
            if ∼ isinBinding(IdBinding(id(vn)), dom types)
            then (true, updateENV(env, et, trans), types)
            else
                if
                    checkTRANS(
                        types(IdBinding(id(vn))), dom trans)
                then
                    /*The value expression is of the type of interest.
                    */
                    (isinBinding(IdBinding(id(vn)), rng env),
                     updateENV(env, et, trans), types)
                else
                    /*The value expression is not of the type of interest.
                    */
                    (true, updateENV(env, et, trans), types)
```

```
        end
      end,
    Make_BasicExpr(be) → (true, env, types),
    ProductExpr(vel) →
      CheckValueExprList(
          vel,
          expTypeToExpTypeList(
               removeBrackets(et), len vel), typings,
          func, trans, env, types),
    Make_SetExpr(setExpr) →
      if
        containsTRANSName(
            getSetType(et), typings, dom trans)
      then (false, env, types)
      else
        CheckSetExpr(
            setExpr, getSetType(et), typings, func,
            trans, env, types)
      end,
    Make_ListExpr(listExpr) →
      if
        containsTRANSName(
            getListType(et), typings, dom trans)
      then (false, env, types)
      else
        CheckListExpr(
            listExpr, getListType(et), typings, func,
            trans, env, types)
      end,
    Make_MapExpr(mapExpr) →
      if
        containsTRANSName(
            tedom(getMapType(et)), typings, dom trans) ∨
        containsTRANSName(
            terange(getMapType(et)), typings, dom trans)
      then (false, env, types)
      else
        CheckMapExpr(
            mapExpr, getMapType(et), typings, func,
            trans, env, types)
      end,
    ApplicationExpr(ave, vl) →
      CheckApplicationExpr(
          ave, vl, et, typings, func, trans, env, types),
```

```
BracketedExpr(bve) →
   let
      (b, env′, types′) =
         CheckValueExpr(
             bve, Unknown, typings, func, trans, env,
             types)
   in
      (b, updateENV(env′, et, trans), types′)
   end,
ValueInfixExpr(first, op, second) →
   let
      (b, env′, types′) =
         CheckValueExprList(
             ⟨first, second⟩, ⟨Unknown, Unknown⟩,
             typings, func, trans, env, types)
   in
      (b, updateENV(env′, et, trans), types′)
   end,
ValuePrefixExpr(op, operand) →
   let
      (b, env′, types′) =
         CheckValueExpr(
             operand, Unknown, typings, func, trans,
             env, types)
   in
      (b, updateENV(env′, et, trans), types′)
   end,
LetExpr(ldl, lve) →
   let
      (b, env′, types′) =
         CheckLetDefList(
             ldl, typings, func, trans, env, types),
      (b′, env″, types″) =
         CheckValueExpr(
             lve, et, typings, func, trans, env′,
             types′)
   in
      (b ∧ b′, env″, types″)
   end,
Make_IfExpr(ie) →
   CheckIfExpr(
       ie, et, typings, func, trans, env, types),
CaseExpr(cond, cbl) →
   let
```

```
          (b, env′, types′) =
            CheckValueExpr(
              cond, Unknown, typings, func, trans, env,
              types),
          (b′, env″, types″) =
            CheckCaseBranchList(
              cbl, et, typings, func, trans, env′,
              types′)
        in
          (b ∧ b′, env″, types″)
        end
    end,

/∗Checks for transformability of a list of value
expressions.∗/
/∗
Arguments:
=======
vel: the list of value expressions
etl: the list of the corresponding expected types
of the value expressions
Results:
=====
Bool: true if the list is transformable, false
otherwise
∗/
CheckValueExprList :
  ValueExpr∗ × ExpType∗ × TYPINGS × FUNC ×
  TRANS × ENV × TYPES →
    Bool × ENV × TYPES
CheckValueExprList(
   vel, etl, typings, func, trans, env, types) ≡
  if vel = ⟨⟩ then (true, env, types)
  else
    let
      (b, env′, types′) =
        CheckValueExpr(
          hd vel, hd etl, typings, func, trans, env,
          types)
    in
      if ∼ b then (b, env′, types′)
      else
        CheckValueExprList(
          tl vel, tl etl, typings, func, trans, env′,
```

```
              types′)
          end
        end
      end,

    /*Checks for transformability of a set expression.
    */
    /*
    Arguments:
    =======
    se: the set expression
    et: the expected type of the components of the
    set expression
    Results:
    =====
    Bool: true if the set expression is transformable,
    false otherwise
    */
    CheckSetExpr :
      SetExpr × ExpType × TYPINGS × FUNC × TRANS ×
      ENV × TYPES →
        Bool × ENV × TYPES
    CheckSetExpr(se, et, typings, func, trans, env, types) ≡
      case se of
        RangedSetExpr(fve, sve) →
          CheckValueExprList(
            ⟨fve, sve⟩, ⟨et, et⟩, typings, func,
            trans, env, types),
        EnumeratedSetExpr(ovel) →
          CheckOptValueExprList(
            ovel, et, typings, func, trans, env, types),
        ComprehendedSetExpr(ve, typlist, or) →
          let
            (b1, env1, types1) =
              CheckValueExpr(
                ve, et, typings, func, trans, env, types),
            (b2, env2, types2) =
              CheckOptRestriction(
                or, Known(TypeLiteral(BOOL)), typings,
                func, trans, env1, types1)
          in
            (b1 ∧ b2, env2, types2)
          end
      end,
```

```
/*Checks for transformability of a list expression.
*/
/*
Arguments:
=======
le: the list expression
et: the expected type of the components of the
list expression
Results:
=====
Bool: true if the list expression is transformable,
false otherwise
*/
CheckListExpr :
   ListExpr × ExpType × TYPINGS × FUNC × TRANS ×
   ENV × TYPES →
      Bool × ENV × TYPES
CheckListExpr(le, et, typings, func, trans, env, types) ≡
   case le of
      RangedListExpr(fve, sve) →
         CheckValueExprList(
            ⟨fve, sve⟩, ⟨et, et⟩, typings, func,
            trans, env, types),
      EnumeratedListExpr(ovel) →
         CheckOptValueExprList(
            ovel, et, typings, func, trans, env, types),
      ComprehendedListExpr(ve1, b, ve2, or) →
         let
            (b1, env1, types1) =
               CheckValueExpr(
                  ve1, et, typings, func, trans, env, types),
            (b2, env2, types2) =
               CheckValueExpr(
                  ve2, et, typings, func, trans, env1,
                  types1),
            (b3, env3, types3) =
               CheckOptRestriction(
                  or, Known(TypeLiteral(BOOL)), typings,
                  func, trans, env2, types2)
         in
            (b1 ∧ b2 ∧ b3, env3, types3)
         end
   end,
```

```
/*Checks for transformability of a map expression.
*/
/*
Arguments:
=======
me: the map expression
et: the expected type of the components of the
map expression
Results:
=====
Bool: true if the map expression is transformable,
false otherwise
*/
CheckMapExpr :
   MapExpr × MapType × TYPINGS × FUNC × TRANS ×
   ENV × TYPES →
      Bool × ENV × TYPES
CheckMapExpr(me, et, typings, func, trans, env, types) ≡
   case me of
      EnumeratedMapExpr(ovel) →
         CheckOptValueExprPairList(
             ovel, et, typings, func, trans, env, types),
      ComprehendedMapExpr(vep, typlist, or) →
         let
            (b1, env1, types1) =
               CheckValueExprList(
                   ⟨first(vep), second(vep)⟩,
                   ⟨tedom(et), terange(et)⟩, typings,
                   func, trans, env, types),
            (b2, env2, types2) =
               CheckOptRestriction(
                   or, Known(TypeLiteral(BOOL)), typings,
                   func, trans, env1, types1)
         in
            (b1 ∧ b2, env2, types2)
         end
   end,

/*Checks for transformability of an optional value
expression list.*/
/*
Arguments:
=======
```

```
ovel: the optional value expression list
et: the expected type of the components of the
value expression list
Results:
=====
Bool: true if the optional value expression list
is transformable, false otherwise
*/
CheckOptValueExprList :
   OptionalValueExprList × ExpType × TYPINGS ×
   FUNC × TRANS × ENV × TYPES →
      Bool × ENV × TYPES
CheckOptValueExprList(
    ovel, et, typings, func, trans, env, types) ≡
   case ovel of
      ValueExprList(vel) →
         CheckValueExprList(
             vel, toExpTypeList(et, len vel), typings,
             func, trans, env, types),
      NoValueExprList → (true, env, types)
   end,

/*Checks for transformability of an optional value
expression pair list.*/
/*
Arguments:
=======
ovel: the optional value expression pair list
et: the expected type of the components of the
value expression pair list
Results:
=====
Bool: true if the optional value expression pair
list is transformable, false otherwise
*/
CheckOptValueExprPairList :
   OptionalValueExprPairList × MapType × TYPINGS ×
   FUNC × TRANS × ENV × TYPES →
      Bool × ENV × TYPES
CheckOptValueExprPairList(
    ovel, et, typings, func, trans, env, types) ≡
   case ovel of
      ValueExprPairList(vel) →
         CheckValueExprPairList(
```

```
            vel, et, typings, func, trans, env, types),
        NoValueExprPairList → (true, env, types)
    end,

/*Checks for transformability of a list of value
expression pairs.*/
/*
Arguments:
=======
vel: the list of value expression pairs
et: the corresponding expected types of the value
expression pairs
Results:
=====
Bool: true if the list is transformable, false
otherwise
*/
CheckValueExprPairList :
    ValueExprPair* × MapType × TYPINGS × FUNC ×
    TRANS × ENV × TYPES →
        Bool × ENV × TYPES
CheckValueExprPairList(
     vel, et, typings, func, trans, env, types) ≡
    if vel = ⟨⟩ then (true, env, types)
    else
        let
            (b, env′, types′) =
                CheckValueExprList(
                    ⟨first(hd vel), second(hd vel)⟩,
                    ⟨tedom(et), terange(et)⟩, typings, func,
                    trans, env, types)
        in
            if ∼ b then (b, env′, types′)
            else
                CheckValueExprPairList(
                    tl vel, et, typings, func, trans, env′,
                    types′)
            end
        end
    end,

/*Checks for transformability of an optional restriction.
*/
/*
```

```
Arguments:
=======
or: the optional restriction
et: the expected type of the restriction
Results:
=====
Bool: true if the optional restriction is transformable,
false otherwise
*/
CheckOptRestriction :
   OptionalRestriction × ExpType × TYPINGS × FUNC ×
   TRANS × ENV × TYPES →
      Bool × ENV × TYPES
CheckOptRestriction(
    or, et, typings, func, trans, env, types) ≡
   case or of
      Restriction(ve) →
         CheckValueExpr(
             ve, et, typings, func, trans, env, types),
      NoRestriction → (true, env, types)
   end,

/*Checks for transformability of an application
expression.
*/
/*
Arguments:
=======
ve: the value expression
vel: the arguments of the application expression
et: the expected type of the application expression
Results:
=====
Bool: true if the application expression is transformable,
false otherwise
*/
CheckApplicationExpr :
   ValueExpr × ValueExpr* × ExpType × TYPINGS ×
   FUNC × TRANS × ENV × TYPES →
      Bool × ENV × TYPES
CheckApplicationExpr(
    ve, vel, et, typings, func, trans, env, types) ≡
   case ve of
      Make_ValueOrVariableName(vn) →
```

```
            if id(vn) ∈ dom func
            then
              /*The application expression is a function application.
              */
              case type_expr(func(id(vn))) of
                FunctionTypeExpr(arg, fa, res) →
                  CheckFunctionAppl(
                      ve, vel, typeExprToExpTypeList(arg),
                      typings, func, trans, env, types)
              end
            else
              /*The application expression is a list or map
              application.*/
              CheckListMapAppl(
                  ve, vel, et, typings, func, trans, env,
                  types)
            end,
          _ →
            CheckListMapAppl(
                ve, vel, et, typings, func, trans, env, types)
      end,

    /*Checks if a function application can be transformed.
    */
    /*
    Arguments:
    =======
    ve: the value expression
    vel: the arguments of the function application
    et: the expected types of the arguments of the
    function application
    Results:
    =====
    Bool: true if the function application is transformable,
    false otherwise
    */
    CheckFunctionAppl :
      ValueExpr × ValueExpr* × ExpType* ×
      TYPINGS × FUNC × TRANS × ENV × TYPES →
        Bool × ENV × TYPES
    CheckFunctionAppl(
        ve, vel, etl, typings, func, trans, env, types) ≡
      let
        (b, env′, types′) =
```

```
        CheckValueExprList(
            vel, etl, typings, func, trans, env, types)
  in
    case ve of
      Make_ValueOrVariableName(vn) →
        (b ∧ (len vel ≥ len etl),
         setEnv(write_list(func(id(vn))), env′), types′
        )
    end
  end,

/*Checks if a list or map application can be transformed.
*/
/*
Arguments:
=======
ve: the value expression
vel: the arguments of the list or map application
et: the expected type of the list or map application
Results:
=======
Bool: true if the list or map application is
transformable, false otherwise
*/
CheckListMapAppl :
  ValueExpr × ValueExpr* × ExpType × TYPINGS ×
  FUNC × TRANS × ENV × TYPES →
    Bool × ENV × TYPES
CheckListMapAppl(
   ve, vel, et, typings, func, trans, env, types) ≡
  case ve of
    Make_ValueOrVariableName(vn) →
      let
        /*Checks for transformability the arguments.*/
        (b, env′, types′) =
          CheckValueExprList(
              vel,
              expTypeToExpTypeList(
                  removeBrackets(
                      getListMapTypeArg(
                        types(IdBinding(id(vn))))),
                  len vel), typings, func, trans, env,
              types) in
        if
```

```
                checkTRANS(
                    types(IdBinding(id(vn))), dom trans)
            then
                /*The list or map is of the type of interest.
                */
                (isinBinding(IdBinding(id(vn)), rng env) ∧
                 b ∧
                 (len expTypeToExpTypeList(
                             removeBrackets(
                                getListMapTypeArg(
                                   types(IdBinding(id(vn))))),
                             len vel) = len vel),
                 updateENV(env, et, trans), types′)
            else
                /*The list or map is not of the type of interest.
                */
                (b ∧
                 (len expTypeToExpTypeList(
                             removeBrackets(
                                getListMapTypeArg(
                                   types(IdBinding(id(vn))))),
                             len vel) = len vel),
                 updateENV(env, et, trans), types′)
            end
        end,
      _ → (true, env, types)
  end,

/*Checks for transformability of a list of let
definition
list.*/
/*
Arguments:
=======
ldl: the list of let definitions
Results:
=====
Bool: true if the let defition list can be transformed,
false otherwise
*/
CheckLetDefList :
   LetDef* × TYPINGS × FUNC × TRANS × ENV ×
   TYPES →
      Bool × ENV × TYPES
```

```
CheckLetDefList(ldl, typings, func, trans, env, types) ≡
   if ldl = ⟨⟩ then (true, env, types)
   else
      let
         (b, env′, types′) =
            CheckLetDef(
                hd ldl, typings, func, trans, env, types)
      in
         if b
         then
            CheckLetDefList(
                tl ldl, typings, func, trans, env′, types′)
         else (b, env′, types′)
         end
      end
   end,

/∗Checks for transformability of a list of let
definition.
∗/
/∗
Arguments:
=======
ld: the let definition
Results:
=======
Bool: true if the let defition can be transformed,
false otherwise
∗/
CheckLetDef :
   LetDef × TYPINGS × FUNC × TRANS × ENV × TYPES →
      Bool × ENV × TYPES
CheckLetDef(ld, typings, func, trans, env, types) ≡
   let
      (bb, envb, types′) =
         CheckLetBinding(
             binding(ld), typings, func, trans, env, types)
   in
      if ∼ bb then (false, env, types)
      else
         case value_expr(ld) of
            ApplicationExpr(ve, vel) →
               case ve of
                  Make_ValueOrVariableName(vn) →
```

```
        if id(vn) ∈ dom func
        then
          /*The let definition is a function application.
          */
          case type_expr(func(id(vn))) of
            FunctionTypeExpr(arg, fa, res) →
              let
                (b, env′, types′) =
                  CheckValueExpr(
                    value_expr(ld),
                    Known(type_expr(res)),
                    typings, func, trans, env,
                    types),
                b1 =
                  matchBindingTE(
                    binding(ld), type_expr(res)),
                /*ENV must be updated.*/
                env″ =
                  makeENV(
                    binding(ld),
                    type_expr(res), trans),
                types″ =
                  makeTYPES(
                    binding(ld),
                    Known(type_expr(res)), trans
                  )
              in
                (b ∧ b1, env′ † env″,
                 types′ † types″)
              end
          end
        else
          /*The let definition is a list ot map application.
          */
          CheckValueExpr(
              value_expr(ld), Unknown, typings,
              func, trans, env,
              types †
              makeTYPES(
                binding(ld), Unknown, trans))
        end,
      _ →
        CheckValueExpr(
            value_expr(ld), Unknown, typings,
```

```
                        func, trans, env,
                        types †
                        makeTYPES(binding(ld), Unknown, trans)
                    )
                end,
            _ →
                CheckValueExpr(
                    value_expr(ld), Unknown, typings, func,
                    trans, env,
                    types †
                    makeTYPES(binding(ld), Unknown, trans))
        end
    end
end,

/*Checks for transformability of a let binding.
*/
/*
Arguments:
=======
lb: the let binding
Results:
=======
Bool: true if the let binding can be transformed,
false otherwise
*/
CheckLetBinding :
    LetBinding × TYPINGS × FUNC × TRANS × ENV ×
    TYPES →
        Bool × ENV × TYPES
CheckLetBinding(lb, typings, func, trans, env, types) ≡
    case lb of
        MakeBinding(b) → (true, env, types),
        MakeRecordPattern(vn, pl) →
            CheckPatternList(pl, func, trans, env, types),
        MakeListPatternLet(lp) →
            CheckListPattern(lp, func, trans, env, types)
    end,

/*Checks for transformability of an if expression.
*/
/*
Arguments:
=======
```

```
ie: the if expression
et: the expected type of the if expression
Results:
=======
Bool: true if the if expression can be transformed,
false otherwise
*/
CheckIfExpr :
   IfExpr × ExpType × TYPINGS × FUNC × TRANS ×
   ENV × TYPES →
      Bool × ENV × TYPES
CheckIfExpr(ie, et, typings, func, trans, env, types) ≡
   let
      (b1, env1, types1) =
         CheckValueExpr(
            condition(ie), Known(TypeLiteral(BOOL)),
            typings, func, trans, env, types),
      (b1′, env1′, types1′) =
         CheckValueExpr(
            if_case(ie), et, typings, func, trans, env1,
            types1),
      (b2, env2, types2) =
         CheckElsif(
            elsif_list(ie), et, typings, func, trans,
            env1, types1′),
      (b3, env3, types3) =
         CheckValueExpr(
            else_case(ie), et, typings, func, trans,
            env2, types2)
   in
      (b1 ∧ b1′ ∧ b2 ∧ b3, env3, types3)
   end,

/*Checks for transformability of an elsif list.
*/
/*
Arguments:
=======
eil: the elsif list
et: the expected type of the if expression
Results:
=======
Bool: true if the elsif list can be transformed,
false otherwise
```

```
*/
CheckElsif :
   Elsif* × ExpType × TYPINGS × FUNC × TRANS ×
   ENV × TYPES →
      Bool × ENV × TYPES
CheckElsif(eil, et, typings, func, trans, env, types) ≡
   if eil = ⟨⟩ then (true, env, types)
   else
      let
         (b1, env1, types1) =
            CheckValueExpr(
               condition(hd eil),
               Known(TypeLiteral(BOOL)), typings, func,
               trans, env, types),
         (b2, env2, types2) =
            CheckValueExpr(
               elsif_case(hd eil), et, typings, func,
               trans, env1, types1)
      in
         if (b1 ∧ b2)
         then
            CheckElsif(
               tl eil, et, typings, func, trans, env1,
               types2)
         else (false, env1, types2)
         end
      end
   end,

/*Checks for transformability of a case branch
list.
*/
/*
Arguments:
=======
cbl: the case branch list
et: the expected type of the case expression
Results:
=======
Bool: true if the case branch list can be transformed,
false otherwise
*/
CheckCaseBranchList :
   CaseBranch* × ExpType × TYPINGS × FUNC ×
```

```
    TRANS × ENV × TYPES →
        Bool × ENV × TYPES
CheckCaseBranchList(
    cbl, et, typings, func, trans, env, types) ≡
    if cbl = ⟨⟩ then (true, env, types)
    else
        let
            (b1, env1, types1) =
                CheckPattern(
                    pattern(hd cbl), func, trans, env, types),
            (b2, env2, types2) =
                CheckValueExpr(
                    value_expr(hd cbl), et, typings, func,
                    trans, env1, types1)
        in
            if (b1 ∧ b2)
            then
                CheckCaseBranchList(
                    tl cbl, et, typings, func, trans, env1,
                    types2)
            else (false, env1, types2)
            end
        end
    end,

/*Checks for transformability of a pattern.*/
/*
Arguments:
=======
p: the pattern
Results:
=======
Bool: true if the pattern can be transformed,
false
otherwise
*/
CheckPattern :
    Pattern × FUNC × TRANS × ENV × TYPES →
        Bool × ENV × TYPES
CheckPattern(p, func, trans, env, types) ≡
    case p of
        ValueLiteralPattern(vl) → (true, env, types),
        NamePattern(id, optid) →
            if id ∈ dom func then (false, env, types)
```

```
          else
            if ~ isinBinding(IdBinding(id), dom types)
            then (true, env, types)
            else
              if
                checkTRANS(types(IdBinding(id)), dom trans)
              then
                /*The value expression is of the type of interest.
                */
                (false, env, types)
              else
                /*The value expression is not of the type of interest.
                */
                (true, env, types)
              end
            end
          end,
        WildcardPattern → (true, env, types),
        ProductPattern(pl) →
          CheckPatternList(pl, func, trans, env, types),
        RecordPattern(vn, pl) →
          if id(vn) ∈ dom func then (false, env, types)
          else
            CheckPatternList(pl, func, trans, env, types)
          end,
        MakeListPattern(lp) →
          CheckListPattern(lp, func, trans, env, types)
      end,

/*Checks for transformability of a pattern list.
*/
/*
Arguments:
=======
pl: the pattern list
Results:
=======
Bool: true if the pattern list can be transformed,
false otherwise
*/
CheckPatternList :
   Pattern* × FUNC × TRANS × ENV × TYPES →
      Bool × ENV × TYPES
CheckPatternList(pl, func, trans, env, types) ≡
```

```
    if pl = ⟨⟩ then (true, env, types)
    else
      let
        (b, env1, types1) =
          CheckPattern(hd pl, func, trans, env, types)
      in
        if b
        then
          CheckPatternList(
              tl pl, func, trans, env1, types1)
        else (false, env1, types)
        end
      end
    end,

/*Checks for transformability of a list pattern.
*/
/*
Arguments:
=======
lp: the list pattern
Results:
=======
Bool: true if the lis pattern can be transformed,
false otherwise
*/
CheckListPattern :
    ListPattern × FUNC × TRANS × ENV × TYPES →
      Bool × ENV × TYPES
CheckListPattern(lp, func, trans, env, types) ≡
    case lp of
      Make_EnumeratedListPattern(elp) →
        CheckOptInnerPattern(
            inner_pattern(elp), func, trans, env, types),
      ConcatenatedListPattern(elp, p) →
        let
          (b1, env1, types1) =
            CheckOptInnerPattern(
                inner_pattern(elp), func, trans, env,
                types),
          (b2, env2, types2) =
            CheckPattern(p, func, trans, env1, types1)
        in
          (b1 ∧ b2, env2, types2)
```

```
            end
        end,

    /*Checks for transformability of a optional inner
    pattern.*/
    /*
    Arguments:
    =======
    oip: optional inner pattern
    Results:
    =======
    Bool: true if the optional inner pattern can
    be transformed, false otherwise
    */
    CheckOptInnerPattern :
        OptionalInnerPattern × FUNC × TRANS × ENV × TYPES →
            Bool × ENV × TYPES
    CheckOptInnerPattern(oip, func, trans, env, types) ≡
        case oip of
            InnerPatternList(pl) →
                CheckPatternList(pl, func, trans, env, types),
            NoInnerPattern → (true, env, types)
        end,

    /*Match a type expression against a let binding.
    */
    /*
    Arguments:
    =======
    lb: the let binding
    te: the type expression
    Results:
    =======
    Bool: true if the let binding an the type expression
    can be matched, false otherwise
    */
    matchBindingTE : LetBinding × TypeExpr → Bool
    matchBindingTE(lb, te) ≡
        getLengthLetBinding(lb) ≥ getLengthTE(te),

    /*Transforms a value definition list.*/
    /*
    Arguments:
    =======
```

```
vdl: the value definition list
Result:
=======
ValueDef_list: the resulting imperative version
of the value definition list
*/
TRValueDefList :
   ValueDef* × FUNC × TRANS → ValueDef*
TRValueDefList(vdl, func, trans) ≡
   if vdl = ⟨⟩ then ⟨⟩
   else
      ⟨TRValueDef(hd vdl, func, trans)⟩ ^
      TRValueDefList(tl vdl, func, trans)
   end,

/*Transforms a value definition.*/
/*
Arguments:
=======
vd: the value definition
Result:
=======
ValueDef: the resulting imperative version of
the value definition
*/
TRValueDef : ValueDef × FUNC × TRANS → ValueDef
TRValueDef(vd, func, trans) ≡
   case vd of
      ExplicitValueDef(st, ve) →
         if
            includesTRANSName(
                Known(type_expr(st)), dom trans)
         then
            case binding(st) of
               IdBinding(id) →
                  let
                     (transve, types) =
                        TRValueExpr(
                            ve, Known(type_expr(st)), func,
                            trans, [])
                  in
                     ExplicitFunctionDef(
                        TRSingleTyping(
                            mk_SingleTyping(
```

```
                        binding(st),
                        FunctionTypeExpr(
                          TypeLiteral(UNIT),
                          TOTAL_FUNCTION_ARROW,
                          mk_ResultDesc(
                            NoReadAccessMode,
                            NoWriteAccessMode,
                            type_expr(st)))), func,
                      trans),
                  IdApplication(
                      id,
                      mk_FormalFunctionParameter(⟨⟩)),
                  transve, NoPreCondition)
            end
        end
      else vd
      end,
    ExplicitFunctionDef(st, ffa, ve, precond) →
      case type_expr(st) of
        FunctionTypeExpr(arg, fa, res) →
          let
            /*Establishes ENV and TYPES.*/
            (b, ffap, env, types, prlet) =
              TRFormalFuncAppl(ffa, func, trans),
            types1 = alterTYPESMap(types, prlet),
            (precond′, types2) =
              TROptPreCondition(
                  precond, Known(TypeLiteral(BOOL)),
                  func, trans, types1),
            /*Transforms the value expression.*/
            (transve, types3) =
              TRValueExpr(
                  ve, Known(type_expr(res)), func,
                  trans, types2) in
            if prlet = ⟨⟩
            then
              ExplicitFunctionDef(
                  TRSingleTyping(st, func, trans),
                  ffap, transve, precond′)
            else
              ExplicitFunctionDef(
                  TRSingleTyping(st, func, trans),
                  ffap, LetExpr(prlet, transve),
                  precond′)
```

```
                    end
                end
            end
        end,

    /*Transforms a single typing.*/
    /*
    Arguments:
    =======
    st: the single typing
    Result:
    =======
    ValueDef: the resulting imperative version of
    the single typing
    */
    TRSingleTyping :
        SingleTyping × FUNC × TRANS → SingleTyping
    TRSingleTyping(st, func, trans) ≡
        case binding(st) of
            IdBinding(id) →
                mk_SingleTyping(
                    binding(st),
                    TRTypeExpr(id, type_expr(st), func, trans))
        end,

    /*Transforms a type expression.*/
    /*
    Arguments:
    =======
    id: the id of the corresponding function
    te: the type expression
    Result:
    =======
    TypeExpr: the resulting imperative version of
    the type expression
    */
    TRTypeExpr :
        Id × TypeExpr × FUNC × TRANS → TypeExpr
    TRTypeExpr(id, te, func, trans) ≡
        case te of
            TypeLiteral(literal) → te,
            TypeName(tid) →
                if (tid ∈ dom trans)
                then
```

```
            /*Type expression of a type of interest.*/
            TypeLiteral(UNIT)
         else
            /*Type expression not of a type of interest.*/
            te
         end,
      TypeExprProduct(tel) →
         TypeExprProduct(
            TRTypeExprList(id, tel, func, trans)),
      TypeExprSet(tes) → te,
      TypeExprList(tel) → te,
      TypeExprMap(tem) → te,
      FunctionTypeExpr(fte, fa, rd) →
         TRFunctionDef(id, fte, fa, rd, func, trans),
      BracketedTypeExpr(bte) →
         BracketedTypeExpr(
            TRTypeExpr(id, bte, func, trans))
   end,

/*Transforms a type expression list.*/
/*
Arguments:
=======
id: the id of the corresponding function
tel: the type expression list
Result:
=======
TypeExpr_list: the resulting imperative version
of the type expression list
*/
TRTypeExprList :
   Id × TypeExpr* × FUNC × TRANS → TypeExpr*
TRTypeExprList(id, tel, func, trans) ≡
   if tel = ⟨⟩ then ⟨⟩
   else
      ⟨TRTypeExpr(id, hd tel, func, trans)⟩ ^
      TRTypeExprList(id, tl tel, func, trans)
   end,

/*Transforms a function definition.*/
/*
Arguments:
=======
id: the id of the corresponding function
```

```
te: the type expression of the function argument
fa: the function arrow
rd: the result description of the function
Result:
=======
TypeExpr: the resulting imperative version of
the function definition
*/
TRFunctionDef :
   Id × TypeExpr × FunctionArrow × ResultDesc ×
   FUNC × TRANS →
      TypeExpr
TRFunctionDef(id, te, fa, rd, func, trans) ≡
   let
      rl = read_list(func(id)),
      wl = write_list(func(id)),
      te′ = TRTypeExpr(id, te, func, trans),
      rd′ = TRTypeExpr(id, type_expr(rd), func, trans)
   in
      FunctionTypeExpr(
         truncate(te′), fa,
         mk_ResultDesc(
            makeReadAccessDesc(rl, trans),
            makeWriteAccessDesc(wl, trans),
            truncate(rd′)))
   end,

/*Transforms a formal function application.*/
/*
Arguments:
=======
ffa: the formal function application
Result:
=======
FormalFunctionApplication: the imperative version
of the formal function application
LetDef_list: the resulting let expression
*/
TRFormalFuncAppl :
   FormalFunctionApplication × FUNC × TRANS →
      Bool × FormalFunctionApplication × ENV ×
      TYPES × LetDef*
TRFormalFuncAppl(ffa, func, trans) ≡
   case ffa of
```

```
    IdApplication(id, ffp) →
      case type_expr(func(id)) of
        FunctionTypeExpr(arg, fa, res) →
          let
            /∗Transforms the formal function parameter list.
            ∗/
            (b, ffap, env, types, prlet) =
              TRFormalFuncParam(
                ffp, arg, trans, [ ], [ ], ⟨⟩) in
            (b, IdApplication(id, ffap), env, types,
             prlet)
          end
      end,
    _ → (true, ffa, [ ], [ ], ⟨⟩)
  end,

/∗Transforms a formal function parameter.∗/
/∗
Arguments:
=======
ffpl: the formal function parameter
te: the type of the formal function parameter
prlet: let expression used to access types of
interests that are abbreviation types
Result:
=======
FormalFunctionParameter_list: the imperative
version of the formal function parameter list
LetDef_list: the resulting let expression
∗/
TRFormalFuncParam :
  FormalFunctionParameter × TypeExpr × TRANS ×
  ENV × TYPES × LetDef* →
    Bool × FormalFunctionParameter × ENV × TYPES ×
    LetDef*
TRFormalFuncParam(ffpl, te, trans, env, types, prlet) ≡
  if binding_list(ffpl) = ⟨⟩
  then
    (true, mk_FormalFunctionParameter(⟨⟩), env,
     types, prlet)
  else
    case te of
      TypeLiteral(literal) →
        (true, ffpl, env,
```

```
     types †
     [ makeBinding(binding_list(ffpl)) ↦ Known(te) ],
     prlet),
TypeName(id) →
    if id ∈ dom trans
    then
        /*Formal function parameter of a type of interest.
        */
        (true, mk_FormalFunctionParameter(⟨⟩),
         env †
         [ id ↦ makeBinding(binding_list(ffpl)) ],
         types †
         makeTYPESMap(binding_list(ffpl), te),
         makeProductLet(
                te, makeBinding(binding_list(ffpl)),
                prlet, trans))
    else
        /*Formal function parameter not of a type of interest.
        */
        (true, ffpl, env,
         types †
         [ makeBinding(binding_list(ffpl)) ↦
            Known(te) ], prlet)
    end,
TypeExprProduct(tep) →
    TRFFPProduct(
         ffpl, tep, trans, env, types, prlet),
TypeExprSet(tes) →
    (true, ffpl, env,
     types †
     [ makeBinding(binding_list(ffpl)) ↦ Known(te) ],
     prlet),
TypeExprList(tel) →
    (true, ffpl, env,
     types †
     [ makeBinding(binding_list(ffpl)) ↦ Known(te) ],
     prlet),
TypeExprMap(tem) →
    (true, ffpl, env,
     types †
     [ makeBinding(binding_list(ffpl)) ↦ Known(te) ],
     prlet),
/*Higher order functions.*/
FunctionTypeExpr(tef, fa, rd) →
```

```
            TRFormalFuncParam(
                ffpl, tef, trans, env,
                types †
                [ makeBinding(binding_list(ffpl)) ↦
                   Known(te) ], prlet),
         BracketedTypeExpr(bte) →
            TRFormalFuncParam(
                ffpl, bte, trans, env, types, prlet)
      end
   end,

/*Transforms a formal function parameter list.
*/
/*
Arguments:
=======
ffpl: the formal function parameter list
tel: the corresponding type expression list
prlet: let expression used to access types of
interests that are abbreviation types
Result:
=======
FormalFunctionParameter_list: the imperative
version of the formal function parameter list
LetDef_list: the resulting let expression
*/
TRFFPProduct :
   FormalFunctionParameter × TypeExpr* × TRANS ×
   ENV × TYPES × LetDef* →
      Bool × FormalFunctionParameter × ENV × TYPES ×
      LetDef*
TRFFPProduct(ffpl, tel, trans, env, types, prlet) ≡
   if binding_list(ffpl) = ⟨⟩
   then
      if tel ≠ ⟨⟩
      then
         (false, mk_FormalFunctionParameter(⟨⟩), env,
          types, prlet)
      else
         (true, mk_FormalFunctionParameter(⟨⟩), env,
          types, prlet)
      end
   else
      case hd binding_list(ffpl) of
```

```
Make_ProductBinding(pb) →
  let
    (b, ffpl′, env′, types′, prlet′) =
      TRFormalFuncParam(
        mk_FormalFunctionParameter(
          binding_list(pb)), hd tel, trans,
        env, types, prlet),
    (b1, ffpl″, env″, types″, prlet″) =
      TRFFPProduct(
        mk_FormalFunctionParameter(
          tl binding_list(ffpl)), tl tel,
        trans, env′, types′, prlet′)
  in
    if binding_list(ffpl′) = ⟨⟩
    then
      (b ∧ b1,
       mk_FormalFunctionParameter(
         binding_list(ffpl″)), env″, types″,
       prlet″)
    else
      if len binding_list(ffpl′) = 1
      then
        (b ∧ b1,
         mk_FormalFunctionParameter(
           binding_list(ffpl′) ⌢
           binding_list(ffpl″)), env″,
         types″, prlet″)
      else
        (b ∧ b1,
         mk_FormalFunctionParameter(
           ⟨Make_ProductBinding(
              mk_ProductBinding(
                binding_list(ffpl′)))⟩ ⌢
           binding_list(ffpl″)), env″,
         types″, prlet″)
      end
    end
  end,
_ →
  let
    (b, ffpl′, env′, types′, prlet′) =
      TRFormalFuncParam(
        mk_FormalFunctionParameter(
          ⟨hd binding_list(ffpl)⟩), hd tel,
```

```
                    trans, env, types, prlet),
              (b1, ffpl″, env″, types″, prlet″) =
                 TRFFPProduct(
                    mk_FormalFunctionParameter(
                       tl binding_list(ffpl)), tl tel,
                    trans, env′, types′, prlet′)
           in
              (b ∧ b1,
               mk_FormalFunctionParameter(
                    binding_list(ffpl′) ⌢
                    binding_list(ffpl″)), env″, types″,
               prlet″)
           end
     end
  end,

/*Checks if it is necessary to establish a let
expression.*/
/*
Arguments:
=======
te: the type of the binding
b: the actual binding
prlet: the original value of the LetDef_list
Result:
=======
LetDef_list: the resulting value of the LetDef_
list
*/
makeProductLet :
  TypeExpr × Binding × LetDef* × TRANS →
     LetDef*
makeProductLet(te, b, prlet, trans) ≡
  case te of
     TypeName(tn) →
        case b of
           Make_ProductBinding(pb) →
              prlet ⌢
              ⟨mk_LetDef(
                    MakeBinding(b),
                    Make_ValueOrVariableName(
                       mk_ValueOrVariableName(trans(tn))))⟩,
           _ → prlet
        end
```

```
    end,

/*Transforms a pre condition.*/
/*
Arguments:
=======
precond: the pre condition
et: the expected type of the pre condition
Result:
=======
OptionalPreCondition: the imperative version
of the pre condition
*/
TROptPreCondition :
    OptionalPreCondition × ExpType × FUNC × TRANS ×
    TYPES →
        OptionalPreCondition × TYPES
TROptPreCondition(precond, et, func, trans, types) ≡
    case precond of
        PreCondition(ve) →
            let
                (ve′, types′) =
                    TRValueExpr(ve, et, func, trans, types)
            in
                (PreCondition(ve′), types′)
            end,
        NoPreCondition → (NoPreCondition, types)
    end,

/*Transforms a value expression.*/
/*
Arguments:
=======
ve: the value expression
et: the expected type of the value expression
Result:
=======
ValueExpr: the imperative version of the value
expression
*/
TRValueExpr :
    ValueExpr × ExpType × FUNC × TRANS × TYPES →
        ValueExpr × TYPES
TRValueExpr(ve, et, func, trans, types) ≡
```

```
case ve of
   Make_ValueLiteral(vl) →
      (makeAssignExpr(ve, et, trans), types),
   Make_ValueOrVariableName(vn) →
      if ∼ isinBinding(IdBinding(id(vn)), dom types)
      then
         if id(vn) ∈ dom func
         then  /*Constant of the type of interest.*/
            TRValueExpr(
                ApplicationExpr(ve, ⟨⟩), et, func,
                trans, types)
         else (ve, types)
         end
      else
         if
            checkTRANS(
                types(IdBinding(id(vn))), dom trans)
         then
            /*The value expression is of a type of interest.
            */
            (makeAssignExpr(
                  getTRANS(types(IdBinding(id(vn))), trans),
                  et, trans), types)
         else
            /*The value expression is not of a type of interest.
            */
            (makeAssignExpr(ve, et, trans), types)
         end
      end,
   Make_BasicExpr(be) → (ve, types),
   ProductExpr(vel) →
      let
         (vel′, types′) =
            TRValueExprListProductFunc(
                vel,
                expTypeToExpTypeList(
                     removeBrackets(et), len vel), func,
                trans, types)
      in
         if
            containsGen(
                vel,
                expTypeToExpTypeList(
                     removeBrackets(et), len vel), trans,
```

```
                func)
            then
                (TRProductExpr(
                    vel′,
                    expTypeToExpTypeList(
                        removeBrackets(et), len vel), et,
                    func, trans, types), types′)
            else
                (makeAssignExpr(ProductExpr(vel′), et, trans),
                 types′)
            end
        end,
    Make_SetExpr(se) →
        let
            (se′, types′) =
                TRSetExpr(
                    se, getSetType(et), func, trans, types)
        in
            (makeAssignExpr(Make_SetExpr(se′), et, trans),
             types′)
        end,
    Make_ListExpr(le) →
        let
            (le′, types′) =
                TRListExpr(
                    le, getListType(et), func, trans, types)
        in
            (makeAssignExpr(Make_ListExpr(le′), et, trans),
             types′)
        end,
    Make_MapExpr(me) →
        let
            (me′, types′) =
                TRMapExpr(
                    me, getMapType(et), func, trans, types)
        in
            (makeAssignExpr(Make_MapExpr(me′), et, trans),
             types′)
        end,
    ApplicationExpr(ave, vel) →
        let
            (ve′, types′) =
                TRApplicationExpr(
                    ave, vel, et, func, trans, types)
```

```
    in
        if ∼ isTGen(ve, trans, func)
        then (makeAssignExpr(ve′, et, trans), types′)
        else (ve′, types′)
        end
    end,
BracketedExpr(bve) →
    let
        et′ = getBracketedType(et),
        (bve′, types′) =
            TRValueExpr(bve, et′, func, trans, types)
    in
        (makeAssignExpr(BracketedExpr(bve′), et, trans),
         types′)
    end,
ValueInfixExpr(first, op, second) →
    let
        (first′, types′) =
            TRValueExpr(
                first, Unknown, func, trans, types),
        (second′, types′′) =
            TRValueExpr(
                second, Unknown, func, trans, types′)
    in
        (makeAssignExpr(
               ValueInfixExpr(first′, op, second′), et,
               trans), types′′)
    end,
ValuePrefixExpr(op, operand) →
    let
        (operand′, types′) =
            TRValueExpr(
                operand, Unknown, func, trans, types)
    in
        (makeAssignExpr(
               ValuePrefixExpr(op, operand′), et, trans),
         types′)
    end,
LetExpr(ldl, lve) →
    let
        (ldl, types′) =
            TRLetDefList(ldl, func, trans, types),
        (ve′, types′′) =
            TRValueExpr(lve, et, func, trans, types′)
```

```
            in
                (LetExpr(ldl, ve′), types″)
            end,
        Make_IfExpr(ie) →
            TRIfExpr(ie, et, func, trans, types),
        CaseExpr(cond, cbl) →
            let
                (cond′, types′) =
                    TRValueExpr(
                        cond, Unknown, func, trans, types),
                (cbl′, types″) =
                    TRCaseBranchList(
                        cbl, et, func, trans, types′)
            in
                (CaseExpr(cond′, cbl′), types″)
            end
    end,

/*Transforms a value expression list.*/
/*
Arguments:
=======
vel: the value expression list
et: the expected type of the value expression
list
Result:
=======
ValueExpr_list: the imperative version of the
value expression list
*/
TRValueExprList :
    ValueExpr* × ExpType × FUNC × TRANS × TYPES →
        ValueExpr* × TYPES
TRValueExprList(vel, et, func, trans, types) ≡
    if vel = ⟨⟩ then (⟨⟩, types)
    else
        let
            (ve, types′) =
                TRValueExpr(
                    hd vel, getHead(et), func, trans, types),
            (vel′, types″) =
                TRValueExprList(
                    tl vel, getTail(et), func, trans, types′)
        in
```

```
        (⟨ve⟩ ⌢ vel′, types″)
      end
   end,

/∗Transforms a value expression list of a product
expression or function application.∗/
/∗
Arguments:
=======
vel: the value expression list
et: the expected type of the value expression
list
Result:
=======
ValueExpr_list: the imperative version of the
value expression list
∗/
TRValueExprListProductFunc :
   ValueExpr∗ × ExpType∗ × FUNC × TRANS ×
   TYPES →
      ValueExpr∗ × TYPES
TRValueExprListProductFunc(
    vel, etl, func, trans, types) ≡
   if vel = ⟨⟩ then (⟨⟩, types)
   else
      let
         (ve, types′) =
            TRValueExpr(hd vel, hd etl, func, trans, types),
         (vel′, types″) =
            TRValueExprListProductFunc(
               tl vel, tl etl, func, trans, types′)
      in
         (⟨ve⟩ ⌢ vel′, types″)
      end
   end,

/∗Transforms a product expression.∗/
/∗
Arguments:
=======
vel: the value expression
etl: the expected types of the value expression
in the list
et: the expected type of the product expression
```

```
Result:
=======
ValueExpr: the imperative version of the value
expression list
*/
TRProductExpr :
   ValueExpr* × ExpType* × ExpType × FUNC ×
   TRANS × TYPES →
      ValueExpr
TRProductExpr(vel, etl, et, func, trans, types) ≡
   let
      (b, vl) = makeLetBinding(len vel, 0, ⟨⟩, ⟨⟩),
      vel′ =
         makeValueExprList(vel, etl, vl, trans, func)
   in
      LetExpr(
          ⟨mk_LetDef(
                  MakeBinding(makeBinding(b)),
                  ProductExpr(vel))⟩,
          makeAssignExpr(ProductExpr(vel′), et, trans))
   end,

/*Transforms a set expression.*/
/*
Arguments:
=======
se: the set expression
et: the expected type the components of the set
expression
Result:
=======
SetExpr: the imperative version of the set expression
*/
TRSetExpr :
   SetExpr × ExpType × FUNC × TRANS × TYPES →
      SetExpr × TYPES
TRSetExpr(se, et, func, trans, types) ≡
   case se of
      RangedSetExpr(fve, sve) →
         let
            (fve′, types′) =
               TRValueExpr(fve, et, func, trans, types),
            (sve′, types″) =
               TRValueExpr(sve, et, func, trans, types′)
```

```
            in
               (RangedSetExpr(fve′, sve′), types″)
            end,
         EnumeratedSetExpr(ovel) →
            let
               (ovel′, types′) =
                  TROptValueExprList(
                     ovel, et, func, trans, types)
            in
               (EnumeratedSetExpr(ovel′), types′)
            end,
         ComprehendedSetExpr(ve, typlist, or) →
            let
               (ve′, types1) =
                  TRValueExpr(ve, et, func, trans, types),
               (or′, types2) =
                  TROptRestriction(
                     or, Known(TypeLiteral(BOOL)), func,
                     trans, types1)
            in
               (ComprehendedSetExpr(ve′, typlist, or′), types2
               )
            end
      end,

/∗Transforms a list expression.∗/
/∗
Arguments:
=======
le: the list expression
et: the expected type the components of the list
expression
Result:
=======
ListExpr: the imperative version of the list
expression
∗/
TRListExpr :
   ListExpr × ExpType × FUNC × TRANS × TYPES →
      ListExpr × TYPES
TRListExpr(le, et, func, trans, types) ≡
   case le of
      RangedListExpr(fve, sve) →
         let
```

```
              (fve′, types′) =
                  TRValueExpr(fve, et, func, trans, types),
              (sve′, types″) =
                  TRValueExpr(sve, et, func, trans, types′)
          in
              (RangedListExpr(fve′, sve′), types″)
          end,
      EnumeratedListExpr(ovel) →
          let
              (ovel′, types′) =
                  TROptValueExprList(
                       ovel, et, func, trans, types)
          in
              (EnumeratedListExpr(ovel′), types′)
          end,
      ComprehendedListExpr(ve1, b, ve2, or) →
          let
              (ve1′, types1) =
                  TRValueExpr(ve1, et, func, trans, types),
              (ve2′, types2) =
                  TRValueExpr(ve2, et, func, trans, types1),
              (or′, types3) =
                  TROptRestriction(
                       or, Known(TypeLiteral(BOOL)), func,
                       trans, types2)
          in
              (ComprehendedListExpr(ve1′, b, ve2′, or′),
               types3)
          end
  end,

/∗Transforms a map expression.∗/
/∗
Arguments:
=======
me: the map expression
et: the expected type the components of the map
expression
Result:
=======
MapExpr: the imperative version of the map expression
∗/
TRMapExpr :
   MapExpr × MapType × FUNC × TRANS × TYPES →
```

```
    MapExpr × TYPES
TRMapExpr(me, et, func, trans, types) ≡
  case me of
    EnumeratedMapExpr(ovel) →
      let
        (ovel′, types′) =
          TROptValueExprPairList(
            ovel, et, func, trans, types)
      in
        (EnumeratedMapExpr(ovel′), types′)
      end,
    ComprehendedMapExpr(ve, typlist, or) →
      let
        (vedom, types1) =
          TRValueExpr(
            first(ve), tedom(et), func, trans, types),
        (verange, types2) =
          TRValueExpr(
            second(ve), terange(et), func, trans,
            types1),
        (or′, types3) =
          TROptRestriction(
            or, Known(TypeLiteral(BOOL)), func,
            trans, types2)
      in
        (ComprehendedMapExpr(
           mk_ValueExprPair(vedom, verange), typlist,
           or′), types3)
      end
  end,

/*Transforms an optional value expression list.
*/
/*
Arguments:
=======
ovel: the optional value expression list
et: the expected type the components of the set
expression
Result:
=======
OptionalValueExprList: the imperative version
of the optional value expression list
*/
```

TROptValueExprList :
  OptionalValueExprList × ExpType × FUNC × TRANS ×
  TYPES →
    OptionalValueExprList × TYPES
TROptValueExprList(ovel, et, func, trans, types) ≡
  case ovel **of**
    ValueExprList(vel) →
      let
        (vel′, types′) =
          TRValueExprListOpt(
            vel, et, func, trans, types)
      in
        (ValueExprList(vel′), types′)
      end,
    NoValueExprList → (ovel, types)
  end,

/∗Transforms an optional value expression pair
list.∗/
/∗
Arguments:
=======
ovel: the optional value expression pair list
et: the expected type the components of the map
expression
Result:
=======
OptionalValueExprPairList: the imperative version
of the optional value expression pair list
∗/
TROptValueExprPairList :
  OptionalValueExprPairList × MapType × FUNC ×
  TRANS × TYPES →
    OptionalValueExprPairList × TYPES
TROptValueExprPairList(ovel, et, func, trans, types) ≡
  case ovel **of**
    ValueExprPairList(vel) →
      let
        (vel′, types′) =
          TRValueExprPairListOpt(
            vel, et, func, trans, types)
      in
        (ValueExprPairList(vel′), types′)
      end,

```
      NoValueExprPairList → (ovel, types)
   end,

/*Transforms a value expression list.*/
/*
Arguments:
=======
vel: the value expression list
et: the expected type of the components of the
value expression list
list
Result:
=======
ValueExpr_list: the imperative version of the
value expression list
*/
TRValueExprListOpt :
   ValueExpr* × ExpType × FUNC × TRANS × TYPES →
      ValueExpr* × TYPES
TRValueExprListOpt(vel, et, func, trans, types) ≡
   if vel = ⟨⟩ then (⟨⟩, types)
   else
      let
         (ve, types′) =
            TRValueExpr(hd vel, et, func, trans, types),
         (vel′, types″) =
            TRValueExprListOpt(
               tl vel, et, func, trans, types′)
      in
         (⟨ve⟩ ⌢ vel′, types″)
      end
   end,

/*Transforms a value expression pair list.*/
/*
Arguments:
=======
vel: the value expression pair list
et: the expected type of the components of the
value expression pair list
list
Result:
=======
ValueExprPair_list: the imperative version of
```

```
the value expression pair list
*/
TRValueExprPairListOpt :
   ValueExprPair* × MapType × FUNC × TRANS ×
   TYPES →
      ValueExprPair* × TYPES
TRValueExprPairListOpt(vel, et, func, trans, types) ≡
   if vel = ⟨⟩ then (⟨⟩, types)
   else
      let
         (ve1, types1) =
            TRValueExpr(
               first(hd vel), tedom(et), func, trans, types
            ),
         (ve2, types2) =
            TRValueExpr(
               second(hd vel), terange(et), func, trans,
               types1),
         (vel′, types″) =
            TRValueExprPairListOpt(
               tl vel, et, func, trans, types2)
      in
         (⟨mk_ValueExprPair(ve1, ve2)⟩ ⌢ vel′, types″)
      end
   end,

/*Transforms an optional restriction.*/
/*
Arguments:
=======
or: the optional restriction
et: the expected type the optional restriction
Result:
=======
OptionalRestriction: the imperative version of
the optional restriction
*/
TROptRestriction :
   OptionalRestriction × ExpType × FUNC × TRANS ×
   TYPES →
      OptionalRestriction × TYPES
TROptRestriction(or, et, func, trans, types) ≡
   case or of
      Restriction(ve) →
```

```
        let
          (ve′, types′) =
            TRValueExpr(ve, et, func, trans, types)
        in
          (Restriction(ve′), types′)
        end,
      NoRestriction → (or, types)
    end,

/∗Transforms an application expression.∗/
/∗
Arguments:
=======
ve: the value expression
vel: the value expression list
et: the expected type of the application expression
Result:
=======
ValueExpr: the imperative version of the application
expression
∗/
TRApplicationExpr :
  ValueExpr × ValueExpr∗ × ExpType × FUNC ×
  TRANS × TYPES →
    ValueExpr × TYPES
TRApplicationExpr(ve, vel, et, func, trans, types) ≡
  case ve of
    Make_ValueOrVariableName(vn) →
      if id(vn) ∈ dom func
      then
        case type_expr(func(id(vn))) of
          FunctionTypeExpr(arg, fa, res) →
            /∗The value expression is a function application
            expression.∗/
            let
              (vel′, types′) =
                TRValueExprListProductFunc(
                  vel, expTypeList(arg, len vel),
                  func, trans, types)
            in
              if
                ∼ onlyTOIArgument(vel, trans, types) ∧
                len vel = 1
              then
```

```
                    (makeSequencingExpr(
                        ApplicationExpr(ve, vel′),
                        type_expr(res), et, trans, func),
                     types′)
                 else
                    if
                       containsGen(
                          vel, typeExprToExpTypeList(arg),
                          trans, func) ∧
                       ∼ onlyTOIArgument(vel, trans, types)
                    then
                       (makeSequencingExpr(
                          TRFunctionAppl(
                             ve, vel′,
                             typeExprToExpTypeList(arg),
                             func, trans, types),
                          type_expr(res), et, trans, func),
                        types′)
                    else
                       (makeSequencingExpr(
                          ApplicationExpr(
                             ve,
                             removeTOI(
                                vel′,
                                typeExprToExpTypeList(arg),
                                trans)), type_expr(res),
                          et, trans, func), types′)
                    end
                 end
              end
           end
        else
           /*List or map application.*/
           TRListMapAppl(ve, vel, et, func, trans, types)
        end,
     _ →
        TRListMapAppl(ve, vel, et, func, trans, types)
  end,

/*Transforms a function application.*/
/*
Arguments:
=======
ve: the value expression
```

```
vel: the arguments of the function application
etl: the expected type of the arguments
Result:
=======
ValueExpr: the imperative version of the function
expression
*/
TRFunctionAppl :
   ValueExpr × ValueExpr* × ExpType* ×
   FUNC × TRANS × TYPES →
      ValueExpr
TRFunctionAppl(ve, vel, etl, func, trans, types) ≡
   let
      (b, vl) = makeLetBinding(len vel, 0, ⟨⟩, ⟨⟩),
      vel′ =
         makeValueExprList(vel, etl, vl, trans, func)
   in
      ApplicationExpr(
          ve,
          ⟨LetExpr(
                  ⟨mk_LetDef(
                        MakeBinding(makeBinding(b)),
                        ProductExpr(vel))⟩,
                  ProductExpr(vel′))⟩)
   end,

/*Removes arguments of the type of interest.*/
/*
Arguments:
=======
vel: the arguments of the function application
etl: the expected type of the arguments
Result:
=======
ValueExpr_list: the resulting value expression
list
*/
removeTOI :
   ValueExpr* × ExpType* × TRANS →
      ValueExpr*
removeTOI(vel, etl, trans) ≡
   if vel = ⟨⟩ then ⟨⟩
   else
      case hd etl of
```

```
            Known(te) →
               case te of
                  TypeName(tn) →
                     if tn ∈ dom trans
                     then
                        /*Type of interest.*/
                        removeTOI(tl vel, tl etl, trans)
                     else
                        /*Not type of interest.*/
                        ⟨hd vel⟩ ⌢
                        removeTOI(tl vel, tl etl, trans)
                     end,
                  _ →
                     ⟨hd vel⟩ ⌢
                     removeTOI(tl vel, tl etl, trans)
               end,
            Unknown →
               ⟨hd vel⟩ ⌢ removeTOI(tl vel, tl etl, trans)
         end
      end,

/*Transforms a list or map application.*/
/*
Arguments:
=======
ve: the value expression
vel: the arguments of the list or map application
et: the expected type of the list or map application
Result:
=======
ValueExpr: the imperative version of the list
or map application
*/
TRListMapAppl :
   ValueExpr × ValueExpr* × ExpType × FUNC ×
   TRANS × TYPES →
      ValueExpr × TYPES
TRListMapAppl(ve, vel, et, func, trans, types) ≡
   let
      (vel′, types′) =
         TRValueExprList(vel, et, func, trans, types),
      (ve′, types″) =
         TRValueExpr(ve, et, func, trans, types′)
   in
```

```
        (ApplicationExpr(ve′, vel′), types′ † types″)
    end,

/*Transforms a let definition list.*/
/*
Arguments:
=======
ldl: the let definition list
Result:
=======
LetDeflist: the imperative version of the let
definition list
*/
TRLetDefList :
    LetDef* × FUNC × TRANS × TYPES →
        LetDef* × TYPES
TRLetDefList(ldl, func, trans, types) ≡
    if ldl = ⟨⟩ then (⟨⟩, types)
    else
        let
            (ld, types′) =
                TRLetDef(hd ldl, func, trans, types),
            (ldl′, types″) =
                TRLetDefList(tl ldl, func, trans, types)
        in
            (⟨ld⟩ ^ ldl′, types′ † types″)
        end
    end,

/*Transforms a let definition.*/
/*
Arguments:
=======
ld: the let definition
Result:
=======
LetDef: the imperative version of the let definition
*/
TRLetDef :
    LetDef × FUNC × TRANS × TYPES → LetDef × TYPES
TRLetDef(ld, func, trans, types) ≡
    case value_expr(ld) of
        ApplicationExpr(ve, vel) →
            case ve of
```

```
Make_ValueOrVariableName(vn) →
  if id(vn) ∈ dom func
  then
    /*Function application.*/
    case type_expr(func(id(vn))) of
      FunctionTypeExpr(arg, fa, res) →
        let
          (ve′, types′) =
            TRValueExpr(
              value_expr(ld),
              Known(type_expr(res)), func,
              trans, types),
          /*Updates TYPES.*/
          types″ =
            makeTYPES(
              binding(ld),
              Known(type_expr(res)), trans)
        in
          if
            isGen(
              ve,
              Known(type_expr(func(id(vn)))),
              trans, func)
          then
            if
              ∼ returnsNonTOI(
                  ve,
                  Known(
                    type_expr(func(id(vn)))),
                  trans, func)
            then
              (mk_LetDef(
                  MakeBinding(
                    IdBinding(mk_Id(″dummy″))),
                  ve′), types′ † types″)
            else
              (mk_LetDef(
                  TRLetBinding(
                    binding(ld),
                    type_expr(res), func,
                    trans, types″), ve′),
                types′ † types″)
            end
          else
```

```
                              (mk_LetDef(binding(ld), ve′),
                               types′ † types″)
                          end
                      end
              end
          else /*List or map application.*/
              let
                  (ve′, types′) =
                      TRValueExpr(
                          value_expr(ld), Unknown, func,
                          trans, types)
              in
                  (mk_LetDef(binding(ld), ve′),
                   types′ †
                   makeTYPES(binding(ld), Unknown, trans))
              end
          end,
      _ →
          let
              (ve′, types′) =
                  TRValueExpr(
                      value_expr(ld), Unknown, func, trans,
                      types)
          in
              (mk_LetDef(binding(ld), ve′),
               types′ †
               makeTYPES(binding(ld), Unknown, trans))
          end
      end,
  _ →
      let
          (ve′, types′) =
              TRValueExpr(
                  value_expr(ld), Unknown, func, trans,
                  types)
      in
          (mk_LetDef(binding(ld), ve′),
           types′ †
           makeTYPES(binding(ld), Unknown, trans))
      end
end,

/*Transforms a let binding.*/
/*
```

```
Arguments:
=======
lb: the let binding
te: the type expression of the function result
Result:
=======
LetBinding: the imperative version of the let
binding
*/
TRLetBinding :
   LetBinding × TypeExpr × FUNC × TRANS × TYPES →
      LetBinding
TRLetBinding(lb, te, func, trans, types) ≡
   case lb of
      MakeBinding(b) →
         MakeBinding(makeBinding(TRBinding(b, te, trans))),
      MakeRecordPattern(vn, pl) →
         let
            (pl′, types′) =
               TRPatternList(pl, func, trans, types)
         in
            MakeRecordPattern(vn, pl′)
         end,
      MakeListPatternLet(lp) →
         let
            (lp′, types′) =
               TRListPattern(lp, func, trans, types)
         in
            MakeListPatternLet(lp′)
         end
   end,

/*Transforms a binding.*/
/*
Arguments:
=======
b: the binding
te: the type expression of the binding
Result:
=======
Binding_list: the imperative version of the let
binding in list form. The occurrences of bindings
of the type of interest is removed
*/
```

```
TRBinding :
   Binding × TypeExpr × TRANS → Binding*
TRBinding(b, te, trans) ≡
   case b of
      IdBinding(id) →
         case te of
            TypeName(tn) →
               if tn ∈ dom trans then ⟨⟩ else ⟨b⟩ end,
            _ → ⟨b⟩
         end,
      Make_ProductBinding(pb) →
         TRBindingList(
             binding_list(pb), typeExprToList(te), trans)
   end,

/*Transforms a binding list.*/
/*
Arguments:
=======
bl: the binding list
tel: the corresponding type expression list
Result:
=======
Binding_list: the imperative version of the binding
list
*/
TRBindingList :
   Binding* × TypeExpr* × TRANS →
      Binding*
TRBindingList(bl, tel, trans) ≡
   if bl = ⟨⟩ then ⟨⟩
   else
      TRBinding(hd bl, hd tel, trans) ⌢
      TRBindingList(tl bl, tl tel, trans)
   end,

/*Transforms an if expression.*/
/*
Arguments:
=======
ie: the if expression
et: the expected type of the if expression
Result:
=======
```

```
ValueExpr: the imperative version of the if expression
*/
TRIfExpr :
   IfExpr × ExpType × FUNC × TRANS × TYPES →
      ValueExpr × TYPES
TRIfExpr(ie, et, func, trans, types) ≡
   let
      (ifcond, types1) =
         TRValueExpr(
            condition(ie), Known(TypeLiteral(BOOL)),
            func, trans, types),
      (ifcase, types2) =
         TRValueExpr(if_case(ie), et, func, trans, types1),
      (elsiflist, types3) =
         TRElsif(elsif_list(ie), et, func, trans, types2),
      (elsecase, types4) =
         TRValueExpr(
            else_case(ie), et, func, trans, types3)
   in
      (Make_IfExpr(
            mk_IfExpr(ifcond, ifcase, elsiflist, elsecase)),
       types4)
   end,

/*Transforms an elsif list.*/
/*
Arguments:
=======
eil: the elsif list
et: the expected type of the if expression
Result:
=======
Elsif_list: the imperative version of the elsif
list
*/
TRElsif :
   Elsif* × ExpType × FUNC × TRANS × TYPES →
      Elsif* × TYPES
TRElsif(eil, et, func, trans, types) ≡
   if eil = ⟨⟩ then (⟨⟩, types)
   else
      let
         (ve1, types1) =
            TRValueExpr(
```

```
                condition(hd eil),
                Known(TypeLiteral(BOOL)), func, trans, types
            ),
        (ve2, types2) =
            TRValueExpr(
                elsif_case(hd eil), et, func, trans, types1),
        (elsiftl, types3) =
            TRElsif(tl eil, et, func, trans, types2)
    in
        (⟨mk_Elsif(ve1, ve2)⟩ ⌢ elsiftl, types3)
    end
end,

/*Transforms a case branch list.*/
/*
Arguments:
=======
cbl: the case branch list
et: the expected type of the case expression
Results:
=======
CaseBranch_list: the imperative version of the
case branch list
*/
TRCaseBranchList :
    CaseBranch* × ExpType × FUNC × TRANS × TYPES →
        CaseBranch* × TYPES
TRCaseBranchList(cbl, et, func, trans, types) ≡
    if cbl = ⟨⟩ then (⟨⟩, types)
    else
        let
            (p, types1) =
                TRPattern(pattern(hd cbl), func, trans, types),
            (ve, types2) =
                TRValueExpr(
                    value_expr(hd cbl), et, func, trans, types1),
            (cbl′, types3) =
                TRCaseBranchList(
                    tl cbl, et, func, trans, types2)
        in
            (⟨mk_CaseBranch(p, ve)⟩ ⌢ cbl′, types3)
        end
    end,
```

```
/*Transforms a pattern.*/
/*
Arguments:
=======
p: the pattern
Results:
=======
Pattern: the imperative version of the pattern
*/
TRPattern :
   Pattern × FUNC × TRANS × TYPES →
      Pattern × TYPES
TRPattern(p, func, trans, types) ≡
   case p of
      ValueLiteralPattern(vl) → (p, types),
      NamePattern(id, optid) →
         if ∼ isinBinding(IdBinding(id), dom types)
         then (p, types)
         else
            if checkTRANS(types(IdBinding(id)), dom trans)
            then
               /*The value expression is of the type of interest.
               */
               (NamePattern(
                     getTRANSId(types(IdBinding(id)), trans),
                     optid), types)
            else
               /*The value expression is not of the type of interest.
               */
               (p, types)
            end
         end,
      WildcardPattern → (p, types),
      ProductPattern(pl) →
         let
            (pl′, types′) =
               TRPatternList(pl, func, trans, types)
         in
            (ProductPattern(pl′), types′)
         end,
      RecordPattern(vn, pl) →
         let
            (pl′, types′) =
               TRPatternList(pl, func, trans, types)
```

```
            in
               (RecordPattern(vn, pl′), types′)
            end,
         MakeListPattern(lp) →
            let
               (lp′, types′) =
                  TRListPattern(lp, func, trans, types)
            in
               (MakeListPattern(lp′), types′)
            end
   end,

/*Transforms a pattern list.*/
/*
Arguments:
=======
pl: the pattern list
Results:
=======
Pattern_list: the imperative version of the pattern
list
*/
TRPatternList :
   Pattern* × FUNC × TRANS × TYPES →
      Pattern* × TYPES
TRPatternList(pl, func, trans, types) ≡
   if pl = ⟨⟩ then (⟨⟩, types)
   else
      let
         (p1, types1) =
            TRPattern(hd pl, func, trans, types),
         (pl′, types2) =
            TRPatternList(tl pl, func, trans, types1)
      in
         (⟨p1⟩ ^ pl′, types2)
      end
   end,

/*Transforms a list pattern.*/
/*
Arguments:
=======
lp: the list pattern
Results:
```

```
=======
ListPattern: the imperative version of the list
pattern
*/
TRListPattern :
   ListPattern × FUNC × TRANS × TYPES →
      ListPattern × TYPES
TRListPattern(lp, func, trans, types) ≡
   case lp of
      Make_EnumeratedListPattern(elp) →
         let
            (elp′, types′) =
               TROptInnerPattern(
                  inner_pattern(elp), func, trans, types)
         in
            (Make_EnumeratedListPattern(
                  mk_EnumeratedListPattern(elp′)), types′)
         end,
      ConcatenatedListPattern(elp, p) →
         let
            (elp′, types1) =
               TROptInnerPattern(
                  inner_pattern(elp), func, trans, types),
            (p′, types2) =
               TRPattern(p, func, trans, types1)
         in
            (ConcatenatedListPattern(
                  mk_EnumeratedListPattern(elp′), p′), types2
            )
         end
   end,

/*Transforms an optional inner pattern.*/
/*
Arguments:
=======
oip: optional inner pattern
Results:
=======
OptionalInnerPattern: the imperative version
of the optional inner pattern
*/
TROptInnerPattern :
   OptionalInnerPattern × FUNC × TRANS × TYPES →
```

```
    OptionalInnerPattern × TYPES
TROptInnerPattern(oip, func, trans, types) ≡
  case oip of
    InnerPatternList(pl) →
      let
        (pl′, types′) =
          TRPatternList(pl, func, trans, types)
      in
        (InnerPatternList(pl′), types′)
      end,
    NoInnerPattern → (oip, types)
  end,

/*Establishes an assignment expression.*/
/*
Arguments:
=======
ve: the value expression
et: the corresponding expected type
Results:
=======
ValueExpr: the resulting value expression
*/
makeAssignExpr :
  ValueExpr × ExpType × TRANS → ValueExpr
makeAssignExpr(ve, et, trans) ≡
  case et of
    Known(te) →
      case te of
        TypeName(tn) →
          if tn ∈ dom trans
          then
            /*Type of interest.*/
            AssignExpr(trans(tn), ve)
          else
            /*Not type of interest.*/
            ve
          end,
        _ → ve
      end,
    Unknown → ve
  end,

/***** TypeExpr *****/
```

```
/*Returns a list of type expressions from a type
expression.*/
/*
Arguments:
=======
te: the type expression
Result:
=======
TypeExpr_list: the corresponding list
*/
typeExprToList : TypeExpr → TypeExpr*
typeExprToList(te) ≡
   case te of
      TypeExprProduct(tep) → tep,
      _ → ⟨te⟩
   end,

/*Returns the number of type expressions in a
type expression.*/
/*
Arguments:
=======
te: the type expression
Results:
=====
Int: the number of type expressions
*/
getLengthTE : TypeExpr → Int
getLengthTE(te) ≡
   case te of
      TypeExprProduct(tep) → getLengthTEL(tep),
      BracketedTypeExpr(bte) → getLengthTE(bte),
      _ → 1
   end,

/*Returns the number of type expressions in a
type expression list.*/
/*
Arguments:
=======
tel: the type expression list
Results:
=====
```

```
Int: the number of type expressions
*/
getLengthTEL : TypeExpr* → Int
getLengthTEL(tel) ≡
   if tel = ⟨⟩ then 0
   else getLengthTE(hd tel) + getLengthTEL(tl tel)
   end,

/*Removes superfluous occurrences of UNIT from
a type expression.*/
/*
Arguments:
=======
te: the type expression
Result:
=======
TypeExpr: the resulting type expression
*/
truncate : TypeExpr → TypeExpr
truncate(tel) ≡
   case tel of
      TypeLiteral(literal) → tel,
      TypeName(id) → tel,
      TypeExprProduct(telp) →
         if truncateTypeExprProduct(telp) = ⟨⟩
         then TypeLiteral(UNIT)
         else
            if len truncateTypeExprProduct(telp) = 1
            then hd truncateTypeExprProduct(telp)
            else
               TypeExprProduct(
                   truncateTypeExprProduct(telp))
            end
         end,
      TypeExprSet(tes) → tel,
      TypeExprList(te) → tel,
      TypeExprMap(tem) → tel,
      FunctionTypeExpr(te, fa, rd) →
         FunctionTypeExpr(
             truncate(te), fa,
             mk_ResultDesc(
                   read_access_desc(rd),
                   write_access_desc(rd),
                   truncate(type_expr(rd)))),
```

```
        BracketedTypeExpr(te) →
          case truncate(te) of
            TypeLiteral(literal) →
              case literal of
                UNIT → truncate(te),
                _ → BracketedTypeExpr(truncate(te))
              end,
            _ → BracketedTypeExpr(truncate(te))
          end
      end,

/*Removes superfluous occurrences of UNIT from
a list of type expressions.*/
/*
Arguments:
=======
tel: the type expression list
Result:
=======
TypeExpr_list: the resulting type expression
list
*/
truncateTypeExprProduct :
  TypeExpr* → TypeExpr*
truncateTypeExprProduct(tel) ≡
  if tel = ⟨⟩ then ⟨⟩
  else
    case hd tel of
      TypeLiteral(literal) →
        case literal of
          UNIT → truncateTypeExprProduct(tl tel),
          _ →
            ⟨truncate(hd tel)⟩ ^
            truncateTypeExprProduct(tl tel)
        end,
      _ →
        ⟨truncate(hd tel)⟩ ^
        truncateTypeExprProduct(tl tel)
    end
  end,

/*Compares a type expression and an expected type.
*/
/*
```

```
Arguments:
=======
te: the type expression
et: the expected type
Results:
=====
Bool: true if they are identical, false otherwise
*/
equalsType : TypeExpr × ExpType → Bool
equalsType(te, et) ≡
   case et of
      Known(ete) → equalsTypes(te, ete),
      Unknown → false
   end,

/*Compares two type expressions.*/
/*
Arguments:
=======
te1: the type expression
te2: the type expression
Results:
=====
Bool: true if they are identical, false otherwise
*/
equalsTypes : TypeExpr × TypeExpr → Bool
equalsTypes(te1, te2) ≡
   case te1 of
      TypeLiteral(literal1) →
         case te2 of
            TypeLiteral(literal2) → literal1 = literal2,
            _ → false
         end,
      TypeName(tn1) →
         case te2 of
            TypeName(tn2) → getText(tn1) = getText(tn2),
            _ → false
         end,
      TypeExprProduct(tel1) →
         case te2 of
            TypeExprProduct(tel2) →
               len tel1 = len tel2 ∧
               equalsTypesProduct(tel1, tel2),
            _ → false
```

```
        end,
      TypeExprSet(tes1) →
        case te2 of
          TypeExprSet(tes2) →
            equalsTypesSet(tes1, tes2),
          _ → false
        end,
      TypeExprList(tel1) →
        case te2 of
          TypeExprList(tel2) →
            equalsTypesList(tel1, tel2),
          _ → false
        end,
      TypeExprMap(tem1) →
        case te2 of
          TypeExprMap(tem2) →
            equalsTypesMap(tem1, tem2),
          _ → false
        end,
      FunctionTypeExpr(fte1, fa1, rd1) →
        case te2 of
          FunctionTypeExpr(fte2, fa2, rd2) →
            equalsTypes(fte1, fte2) ∧ fa1 = fa2 ∧
            equalsTypes(type_expr(rd1), type_expr(rd2)),
          _ → false
        end,
      BracketedTypeExpr(bte1) →
        case te2 of
          BracketedTypeExpr(bte2) →
            equalsTypes(bte1, bte2),
          _ → false
        end
    end,

/*Compares two type expression lists.*/
/*
Arguments:
=======
tel1: the type expression list
tel2: the type expression list
Results:
=====
Bool: true if they are identical, false otherwise
*/
```

```
equalsTypesProduct :
   TypeExpr* × TypeExpr* → Bool
equalsTypesProduct(tel1, tel2) ≡
   if tel1 = ⟨⟩ then true
   else
      equalsTypes(hd tel1, hd tel2) ∧
      equalsTypesProduct(tl tel1, tl tel2)
   end,

/*Compares two set type expressions.*/
/*
Arguments:
=======
tes1: the set type expression
tes2: the set type expression
Results:
=====
Bool: true if they are identical, false otherwise
*/
equalsTypesSet : TypeExprSets × TypeExprSets → Bool
equalsTypesSet(tes1, tes2) ≡
   case tes1 of
      FiniteSetTypeExpr(te1) →
         case tes2 of
            FiniteSetTypeExpr(te2) →
               equalsTypes(te1, te2),
            _ → false
         end,
      InfiniteSetTypeExpr(te1) →
         case tes2 of
            InfiniteSetTypeExpr(te2) →
               equalsTypes(te1, te2),
            _ → false
         end
   end,

/*Compares two list type expressions.*/
/*
Arguments:
=======
tel1: the list type expression
tel2: the list type expression
Results:
=====
```

```
Bool: true if they are identical, false otherwise
*/
equalsTypesList :
   TypeExprLists × TypeExprLists → Bool
equalsTypesList(tel1, tel2) ≡
   case tel1 of
      FiniteListTypeExpr(te1) →
         case tel2 of
            FiniteListTypeExpr(te2) →
               equalsTypes(te1, te2),
            _ → false
         end,
      InfiniteListTypeExpr(te1) →
         case tel2 of
            InfiniteListTypeExpr(te2) →
               equalsTypes(te1, te2),
            _ → false
         end
   end,

/*Compares two map type expressions.*/
/*
Arguments:
=======
tel1: the map type expression
tel2: the map type expression
Results:
=====
Bool: true if they are identical, false otherwise
*/
equalsTypesMap : TypeExprMaps × TypeExprMaps → Bool
equalsTypesMap(tem1, tem2) ≡
   case tem1 of
      FiniteMapTypeExpr(tedom1, terange1) →
         case tem2 of
            FiniteMapTypeExpr(tedom2, terange2) →
               equalsTypes(tedom1, tedom2) ∧
               equalsTypes(terange1, terange2),
            _ → false
         end,
      InfiniteMapTypeExpr(tedom1, terange1) →
         case tem2 of
            InfiniteMapTypeExpr(tedom2, terange2) →
               equalsTypes(tedom1, tedom2) ∧
```

```
                equalsTypes(terange1, terange2),
            _ → false
        end
    end,

/***** TypeExpr end *****/

/***** ExpType *****/

/*Gets the head of an expected type which is a
TypeExprProduct.*/
/*
Arguments:
=======
et: the expected type
Results:
=====
ExpType: the head of the TypeExprProduct of the
expected type
*/
getHead : ExpType → ExpType
getHead(et) ≡
    case et of
        Known(te) →
            case te of
                TypeExprProduct(tel) →
                    case hd tel of
                        BracketedTypeExpr(bte) → Known(bte),
                        _ → Known(hd tel)
                    end,
                _ → Unknown
            end,
        Unknown → Unknown
    end,

/*Gets the tail of an expected type which is a
TypeExprProduct.*/
/*
Arguments:
=======
et: the expected type
Results:
=====
ExpType: the tail of the TypeExprProduct of the
```

```
expected type
*/
getTail : ExpType → ExpType
getTail(et) ≡
   case et of
      Known(te) →
         case te of
            TypeExprProduct(tel) →
               Known(TypeExprProduct(tl tel)),
            _ → Unknown
         end,
      Unknown → Unknown
   end,

/*Makes a list of expected types from a type expr.
*/
/*
Arguments:
=======
te: the type expr
length: the length of the resulting list
Results:
=====
ExpType_list: the list of expected types
*/
expTypeList : TypeExpr × Nat → ExpType*
expTypeList(te, length) ≡
   if length = 1 then ⟨Known(te)⟩
   else
      expTypeToExpTypeList(
          removeBrackets(Known(te)), length)
   end,

/*Makes a list of expected types from an expected
types.*/
/*
Arguments:
=======
et: the expected type
length: the length of the resulting list
Results:
=====
ExpType_list: the list of expected types
*/
```

expTypeToExpTypeList : ExpType $\times$ **Nat** $\rightarrow$ ExpType*
expTypeToExpTypeList(et, length) $\equiv$
    if length = 0 **then** $\langle\rangle$
    else
        $\langle$getHead(et)$\rangle$ $\frown$
        expTypeToExpTypeList(getTail(et), length $-$ 1)
    end,

/*Makes a list of expected types from an expected
types.*/
/*
Arguments:
=======
et: the expected type
length: the length of the resulting list
Results:
=====
ExpType_list: the list of expected types
*/
toExpTypeList : ExpType $\times$ **Nat** $\rightarrow$ ExpType*
toExpTypeList(et, length) $\equiv$
    if length = 0 **then** $\langle\rangle$
    else $\langle$et$\rangle$ $\frown$ toExpTypeList(et, length $-$ 1)
    end,

/*Makes a list of expected types from a type expression.
*/
/*
Arguments:
=======
te: the type expression
Results:
=====
ExpType_list: the list of expected types
*/
typeExprToExpTypeList : TypeExpr $\rightarrow$ ExpType*
typeExprToExpTypeList(te) $\equiv$
    case te **of**
        TypeExprProduct(tep) $\rightarrow$
            typeExprListToExpTypeList(tep),
        TypeLiteral(ln) $\rightarrow$
            case ln **of**
                UNIT $\rightarrow$ $\langle\rangle$,
                _ $\rightarrow$ $\langle$Known(te)$\rangle$

```
            end,
        _ → ⟨Known(te)⟩
    end,

/*Makes a list of expected types from a type expression
list.*/
/*
Arguments:
=======
tel: the type expression list
Results:
=====
ExpType_list: the list of expected types
*/
typeExprListToExpTypeList :
    TypeExpr* → ExpType*
typeExprListToExpTypeList(tel) ≡
    if tel = ⟨⟩ then ⟨⟩
    else
        ⟨Known(hd tel)⟩ ^
        typeExprListToExpTypeList(tl tel)
    end,

/*Returns the expected type of the components
of a set expression.*/
/*
Arguments:
=======
et: the expected type of the set expression
Results:
=====
ExpType: the expected type of the componenets
of the set expression
*/
getSetType : ExpType → ExpType
getSetType(et) ≡
    case et of
        Known(te) →
            case te of
                TypeExprSet(tes) →
                    case tes of
                        FiniteSetTypeExpr(fse) → Known(fse),
                        InfiniteSetTypeExpr(ise) → Known(ise)
                    end,
```

```
          _ → Unknown
        end,
      _ → Unknown
    end,

/*Returns the expected type of the components
of a list expression.*/
/*
Arguments:
=======
et: the expected type of the list expression
Results:
=====
ExpType: the expected type of the componenets
of the list expression
*/
getListType : ExpType → ExpType
getListType(et) ≡
  case et of
    Known(te) →
      case te of
        TypeExprList(tes) →
          case tes of
            FiniteListTypeExpr(fse) → Known(fse),
            InfiniteListTypeExpr(ise) → Known(ise)
          end,
        _ → Unknown
      end,
    _ → Unknown
  end,

/*Returns the expected types of the components
of a map expression.*/
/*
Arguments:
=======
et: the expected type of the map expression
Results:
=====
MapType: the expected type of the components
of the map expression
*/
getMapType : ExpType → MapType
getMapType(et) ≡
```

```
    case et of
        Known(te) →
            case te of
                TypeExprMap(tem) →
                    case tem of
                        FiniteMapTypeExpr(tedom, terange) →
                            mk_MapType(Known(tedom), Known(terange)),
                        FiniteMapTypeExpr(tedom, terange) →
                            mk_MapType(Known(tedom), Known(terange))
                    end,
                _ → mk_MapType(Unknown, Unknown)
            end,
        _ → mk_MapType(Unknown, Unknown)
    end,

/*Gets the expected type of a bracketed type.
*/
/*
Arguments:
=======
et: the expected type
Results:
=====
ExpType: the expected type of the if bracketed,
Unknown otherwise
*/
getBracketedType : ExpType → ExpType
getBracketedType(et) ≡
    case et of
        Known(te) →
            case te of
                BracketedTypeExpr(bte) → Known(bte),
                _ → Unknown
            end,
        Unknown → Unknown
    end,

/*Gets the expected type of the list or map application
expression arguments.
*/
/*
Arguments:
=======
et: the expected type of the list or map application
```

```
Results:
=====
ExpType: the expected type of the arguments
*/
getListMapTypeArg : ExpType → ExpType
getListMapTypeArg(et) ≡
  case et of
    Known(te) →
      case te of
        TypeExprList(tel) →
          Known(TypeExprProduct(⟨TypeLiteral(INT)⟩)),
        TypeExprMap(tem) →
          case tem of
            FiniteMapTypeExpr(tedom, terange) →
              Known(TypeExprProduct(⟨tedom⟩)),
            InfiniteMapTypeExpr(tedom, terange) →
              Known(TypeExprProduct(⟨tedom⟩))
          end,
        _ → et
      end,
    Unknown → Unknown
  end,

/*Removes outer brackets from a expected type.
*/
/*
Arguments:
=======
et: the expected type
Result:
=======
ExpType: the resulting expected type
*/
removeBrackets : ExpType → ExpType
removeBrackets(et) ≡
  case et of
    Known(te) →
      case te of
        BracketedTypeExpr(bte) → Known(bte),
        _ → et
      end,
    _ → et
  end,
```

```
/***** ExpType end *****/

/***** ValueExpr *****/

/*Turns a value expression list into a value expression.
*/
/*
Arguments:
=======
vel: the value expression list
Result:
=======
ValueExpr: the resulting value expression
*/
toValueExpr : ValueExpr* → ValueExpr
toValueExpr(vel) ≡
   if len vel = 1 then hd vel else ProductExpr(vel) end,

/*Removes proper Tgenerators from a value expression
list.*/
/*
Arguments:
=======
vel: the value expression list
etl: the corresponding list of expected types
vl: the value expression list which is shortened
Results:
=====
ValueExpr_list: the corresponding value expression
list
*/
makeValueExprList :
   ValueExpr* × ExpType* × ValueExpr* ×
   TRANS × FUNC →
      ValueExpr*
makeValueExprList(vel, etl, vl, trans, func) ≡
   if vel = ⟨⟩ then ⟨⟩
   else
      if
         isGen(hd vel, hd etl, trans, func) ∧
         ∼ returnsNonTOI(hd vel, hd etl, trans, func)
      then
         makeValueExprList(
            tl vel, tl etl, tl vl, trans, func)
```

```
        else
            ⟨hd vl⟩ ⌢
            makeValueExprList(
                tl vel, tl etl, tl vl, trans, func)
        end
    end,

/*Establishes sequencing expression when appropriate.
*/
/*
Arguments:
=======
ve: the value expression
te: the type of the value expression
et: the expected type of the value expression
Results:
=====
ValueExpr: the resulting value expression list
*/
makeSequencingExpr :
    ValueExpr × TypeExpr × ExpType × TRANS × FUNC →
        ValueExpr
makeSequencingExpr(ve, te, et, trans, func) ≡
    if
        isGen(ve, et, trans, func) ∧
        ∼ returnsNonTOI(ve, et, trans, func) ∧
        ∼ equalsType(te, et)
    then
        BracketedExpr(
            SequencingExpr(
                ve,
                toValueExpr(getTOIReturnsList(te, trans))))
    else ve
    end,

/*∗∗∗∗ ValueExpr end ∗∗∗∗*/

/*∗∗∗∗ Generators and observers ∗∗∗∗*/

/*Returns true if the value expression list contains
generators, false otherwise.
*/
/*
Arguments:
```

```
=======
vel: the original value expression list
etl: the corresponding list of expected types
Result:
=======
Bool: the result
*/
containsGen :
   ValueExpr* × ExpType* × TRANS × FUNC →
      Bool
containsGen(vel, etl, trans, func) ≡
   if vel = ⟨⟩ then false
   else
      if
         isGen(hd vel, hd etl, trans, func) ∧
         ∼ returnsNonTOI(hd vel, hd etl, trans, func)
      then true
      else containsGen(tl vel, tl etl, trans, func)
      end
   end,

/*Checks if a value expression is a generator.
*/
/*
Arguments:
=======
ve: the value expression
et: the expected type of the value expression
Result:
=======
Bool: true if the value expression is a generator,
false otherwise
*/
isGen : ValueExpr × ExpType × TRANS × FUNC → Bool
isGen(ve, et, trans, func) ≡
   case ve of
      ApplicationExpr(vea, vel) →
         case vea of
            Make_ValueOrVariableName(vn) →
               write_list(func(id(vn))) ≠ ⟨⟩,
            _ → false
         end,
      _ →
         case et of
```

```
            Known(te) →
               case te of
                  TypeName(tn) → tn ∈ dom trans,
                  TypeExprProduct(tel) →
                     isGenProduct(ve, tel, trans, func),
                  FunctionTypeExpr(arg, fa, res) →
                     isGen(
                         ve, Known(type_expr(res)), trans, func
                     ),
                  BracketedTypeExpr(bte) →
                     isGen(ve, Known(bte), trans, func),
                  _ → false
               end,
            Unknown → false
         end
   end,

/*Checks if a value expression is a generator.
*/
/*
Arguments:
=======
ve: the value expression
tel: the type of the value expression
Result:
=======
Bool: true if the value expression is a generator,
false otherwise
*/
isGenProduct :
   ValueExpr × TypeExpr* × TRANS × FUNC → Bool
isGenProduct(ve, tel, trans, func) ≡
   if tel = ⟨⟩ then false
   else
      if isGen(ve, Known(hd tel), trans, func) then true
      else isGenProduct(ve, tl tel, trans, func)
      end
   end,

/*Checks if a value expression is a generator.
*/
/*
Arguments:
=======
```

```
ve: the value expression
Result:
=======
Bool: true if the value expression is a generator,
false otherwise
*/
isTGen : ValueExpr × TRANS × FUNC → Bool
isTGen(ve, trans, func) ≡
  case ve of
    ApplicationExpr(vea, vel) →
      case vea of
        Make_ValueOrVariableName(vn) →
          if id(vn) ∈ dom func
          then
            case type_expr(func(id(vn))) of
              FunctionTypeExpr(arg, fa, res) →
                case type_expr(res) of
                  TypeName(tn) → tn ∈ dom trans,
                  BracketedTypeExpr(bte) →
                   case bte of
                    TypeName(tn) →
                     tn ∈ dom trans,
                    _ → false
                   end,
                  _ → false
                end,
              _ → false
            end
          else false
          end,
        _ → false
      end,
    _ → false
  end,

/*Checks if a value expression list only consists
of type names of the types of interest.*/
/*
Arguments:
=======
vel: the value expression list
Result:
=======
Bool: true if the value expression only consists
```

```
of type names
of the type of interest, false otherwise
*/
onlyTOIArgument :
   ValueExpr* × TRANS × TYPES → Bool
onlyTOIArgument(vel, trans, types) ≡
   if vel = ⟨⟩ then true
   else
      case hd vel of
         Make_ValueOrVariableName(vn) →
            if isinBinding(IdBinding(id(vn)), dom types)
            then
               if
                  checkTRANS(
                     types(IdBinding(id(vn))), dom trans)
               then onlyTOIArgument(tl vel, trans, types)
               else false
               end
            else false
            end,
         _ → false
      end
   end,

/*Checks if a value expression returns other values
than values of the type of interest.*/
/*
Arguments:
=======
ve: the value expression
et: the expected type of the value expression
Result:
=======
Bool: true if the value expression returns values
not of the type of interest, false otherwise
*/
returnsNonTOI :
   ValueExpr × ExpType × TRANS × FUNC → Bool
returnsNonTOI(ve, et, trans, func) ≡
   case ve of
      ApplicationExpr(vea, vel) →
         case vea of
            Make_ValueOrVariableName(vn) →
               case type_expr(func(id(vn))) of
```

```
                    FunctionTypeExpr(arg, fa, res) →
                        getNonTOIReturnsList(
                            type_expr(res), trans) ≠ ⟨⟩
                end,
                _ → true
            end,
        _ →
            case et of
                Known(te) →
                    case te of
                        TypeName(tn) → ∼ (tn ∈ dom trans),
                        TypeExprProduct(tel) →
                            getNonTOIReturnsList(te, trans) ≠ ⟨⟩,
                        FunctionTypeExpr(arg, fa, res) →
                            getNonTOIReturnsList(
                                type_expr(res), trans) ≠ ⟨⟩,
                        _ → true
                    end,
                Unknown → true
            end
    end,

/*Gets a list of types that are not of a type
of interest from a type expression.*/
/*
Arguments:
=======
te: the type expression
Result:
=======
TypeExpr_list: the list of types that are not
of a type of interest
*/
getNonTOIReturnsList :
    TypeExpr × TRANS → TypeExpr*
getNonTOIReturnsList(te, trans) ≡
    case te of
        TypeLiteral(literal) → ⟨te⟩,
        TypeName(id) →
            if (id ∈ dom trans) then ⟨⟩ else ⟨te⟩ end,
        TypeExprProduct(tep) →
            getNonTOIProduct(tep, trans),
        TypeExprSet(tes) → ⟨te⟩,
        TypeExprList(tel) → ⟨te⟩,
```

```
        TypeExprMap(tem) → ⟨te⟩,
        FunctionTypeExpr(arg, fa, res) →
            getNonTOIReturnsList(type_expr(res), trans),
        BracketedTypeExpr(bte) →
            getNonTOIReturnsList(bte, trans)
    end,

/*Gets a list of types that are not of a type
of interest from a type expression list.*/
/*
Arguments:
=======
tel: the type expression list
Result:
=======
TypeExpr_list: the list of types that are not
of a type of interest
*/
getNonTOIProduct :
    TypeExpr* × TRANS → TypeExpr*
getNonTOIProduct(tel, trans) ≡
    if tel = ⟨⟩ then ⟨⟩
    else
        getNonTOIReturnsList(hd tel, trans) ^
        getNonTOIProduct(tl tel, trans)
    end,

/*Gets the variables corresponding to the type
expression.*/
/*
Arguments:
=======
te: the type expression
Result:
=======
ValueExpr_list: the return values
*/
getTOIReturnsList :
    TypeExpr × TRANS → ValueExpr*
getTOIReturnsList(te, trans) ≡
    case te of
        TypeLiteral(literal) → ⟨⟩,
        TypeName(id) →
            if id ∈ dom trans
```

```
            then
               ⟨Make_ValueOrVariableName(
                     mk_ValueOrVariableName(trans(id)))⟩
            else ⟨⟩
            end,
         TypeExprProduct(tep) →
            getTOIReturnsListProduct(tep, trans),
         TypeExprSet(tes) → ⟨⟩,
         TypeExprList(tel) → ⟨⟩,
         TypeExprMap(tem) → ⟨⟩,
         FunctionTypeExpr(arg, fa, res) →
            getTOIReturnsList(type_expr(res), trans),
         BracketedTypeExpr(bte) →
            getTOIReturnsList(bte, trans)
      end,

/*Gets the variables corresponding to the type
expression.*/
/*
Arguments:
=======
tel: the type expression list
Result:
=======
ValueExpr_list: the result
*/
getTOIReturnsListProduct :
   TypeExpr* × TRANS → ValueExpr*
getTOIReturnsListProduct(tel, trans) ≡
   if tel = ⟨⟩ then ⟨⟩
   else
      getTOIReturnsList(hd tel, trans) ⌢
      getTOIReturnsListProduct(tl tel, trans)
   end,

/*Gets the type names corresponding to the type
expression.*/
/*
Arguments:
=======
te: the type expression
Result:
=======
Access_list: the return values
```

```
*/
getTOIReturnsListTN : TypeExpr × TRANS → Access*
getTOIReturnsListTN(te, trans) ≡
   case te of
      TypeLiteral(literal) → ⟨⟩,
      TypeName(id) →
         if id ∈ dom trans
         then
            ⟨AccessValueOrVariableName(
                    mk_ValueOrVariableName(id))⟩
         else ⟨⟩
         end,
      TypeExprProduct(tep) →
         getTOIReturnsListProductTN(tep, trans),
      TypeExprSet(tes) → ⟨⟩,
      TypeExprList(tel) → ⟨⟩,
      TypeExprMap(tem) → ⟨⟩,
      FunctionTypeExpr(arg, fa, res) →
         getTOIReturnsListTN(type_expr(res), trans),
      BracketedTypeExpr(bte) →
         getTOIReturnsListTN(bte, trans)
   end,

/*Gets the type names corresponding to the type
expression.*/
/*
Arguments:
=======
tel: the type expression list
Result:
=======
ValueExpr_list: the result
*/
getTOIReturnsListProductTN :
   TypeExpr* × TRANS → Access*
getTOIReturnsListProductTN(tel, trans) ≡
   if tel = ⟨⟩ then ⟨⟩
   else
      getTOIReturnsListTN(hd tel, trans) ⌢
      getTOIReturnsListProductTN(tl tel, trans)
   end,

/***** Generators and observers end *****/
```

```
/***** Access *****/

/*Makes a read access description from an access
list.*/
/*
Arguments:
=======
al: the access list
Result:
=======
OptionalReadAccessDesc: the resulting read access
description
*/
makeReadAccessDesc :
   Access* × TRANS → OptionalReadAccessDesc
makeReadAccessDesc(al, trans) ≡
   case al of
      ⟨⟩ → NoReadAccessMode,
      _ → ReadAccessDesc(toVariableList(al, trans))
   end,

/*Makes a write access description from an access
list.*/
/*
Arguments:
=======
al: the access list
Result:
=======
OptionalWriteAccessDesc: the resulting write
access description
*/
makeWriteAccessDesc :
   Access* × TRANS → OptionalWriteAccessDesc
makeWriteAccessDesc(rs, trans) ≡
   case rs of
      ⟨⟩ → NoWriteAccessMode,
      _ → WriteAccessDesc(toVariableList(rs, trans))
   end,

/*Makes a access list of variable names from an
access list of type names.*/
/*
Arguments:
```

```
=======
al: the access list
Result:
=======
Access_list: the resulting access list
*/
toVariableList : Access* × TRANS → Access*
toVariableList(al, trans) ≡
  if al = ⟨⟩ then ⟨⟩
  else
    case hd al of
      AccessValueOrVariableName(vn) →
        if id(vn) ∈ dom trans
        then
          ⟨AccessValueOrVariableName(
               mk_ValueOrVariableName(trans(id(vn))))⟩ ⌢
          toVariableList(tl al, trans)
        else toVariableList(tl al, trans)
        end
    end
  end,

/*Removes duplets from an access list.*/
/*
Arguments:
=======
al: the access list
Result:
=======
Access_list: the resulting access list
*/
removeDuplets : Access* → Access*
removeDuplets(al) ≡
  if len al = card elems al then al
  else
    if hd al ∈ elems tl al
    then removeDuplets(tl al)
    else ⟨hd al⟩ ⌢ removeDuplets(tl al)
    end
  end,

/*Gets an access list from a type expression.
*/
/*
```

```
Arguments:
=======
te: the type expression
Result:
=======
Access_list: the corresponding access list
*/
getAccess : TypeExpr × TRANS → Access*
getAccess(te, trans) ≡
   case te of
      TypeLiteral(literal) → ⟨⟩,
      TypeName(id) →
         if (id ∈ dom trans)
         then
            /*Type of interest.*/
            ⟨AccessValueOrVariableName(
                   mk_ValueOrVariableName(id))⟩
         else
            /*Not type of interest.*/
            ⟨⟩
         end,
      TypeExprProduct(tep) → getAccessList(tep, trans),
      TypeExprSet(tes) → ⟨⟩,
      TypeExprList(tel) → ⟨⟩,
      TypeExprMap(tem) → ⟨⟩,
      FunctionTypeExpr(arg, fa, res) → ⟨⟩,
      BracketedTypeExpr(bte) → getAccess(bte, trans)
   end,

/*Returns an access list from a type expression
list.*/
/*
Arguments:
=======
tel: the type expression list
Result:
=======
Access_list: the corresponding access list
*/
getAccessList : TypeExpr* × TRANS → Access*
getAccessList(tel, trans) ≡
   if tel = ⟨⟩ then ⟨⟩
   else
      getAccess(hd tel, trans) ⌢
```

```
        getAccessList(tl tel, trans)
    end,

/*Returns an access list of the values of types
of interest read from during evaluation of pre
condition.*/
/*
Arguments:
=======
idset: the functions already evaluated
precond: the pre condition
et: the expected type of the value expression
Result:
=======
AccessResult: the corresponding access result
*/
getAccessObsOptPreCondition :
    Id-set × OptionalPreCondition × ExpType × FUNC ×
    TRANS →
        AccessResult
getAccessObsOptPreCondition(
     idset, precond, et, func, trans) ≡
    case precond of
        PreCondition(ve) →
            getAccessObs(idset, ve, et, func, trans),
        NoPreCondition → mk_AccessResult(⟨⟩, idset)
    end,

/*Returns an access list of the values of types
of interest read from during evaluation of a
function.*/
/*
Arguments:
=======
idset: the functions already evaluated
ve: the value expression
et: the expected type of the value expression
Result:
=======
AccessResult: the corresponding access result
*/
getAccessObs :
    Id-set × ValueExpr × ExpType × FUNC × TRANS →
        AccessResult
```

```
getAccessObs(idset, ve, et, func, trans) ≡
  case ve of
    Make_ValueLiteral(vl) →
      mk_AccessResult(⟨⟩, idset),
    Make_ValueOrVariableName(vn) →
      mk_AccessResult(⟨⟩, idset),
    Make_BasicExpr(be) → mk_AccessResult(⟨⟩, idset),
    ProductExpr(vel) →
      getAccessObsList(idset, vel, et, func, trans),
    Make_SetExpr(se) →
      getAccessObsSetExpr(
         idset, se, getSetType(et), func, trans),
    Make_ListExpr(le) →
      getAccessObsListExpr(
         idset, le, getListType(et), func, trans),
    Make_MapExpr(me) →
      getAccessObsMapExpr(
         idset, me, getMapType(et), func, trans),
    ApplicationExpr(ave, vl) →
      case ave of
        Make_ValueOrVariableName(vn) →
          if id(vn) ∈ dom func
          then
            if id(vn) ∈ idset
            then
              case type_expr(func(id(vn))) of
                FunctionTypeExpr(arg, fa, res) →
                  /*Already evaluated.*/
                  let
                   ar =
                     getAccessObsListList(
                       idset, vl,
                       expTypeList(arg, len vl),
                       func, trans)
                  in
                   mk_AccessResult(
                     removeDuplets(
                       accessList(ar) ^
                       getSequencingAccess(
                         ve, type_expr(res), et,
                         trans, func)), idSet(ar))
                  end
              end
            else
```

```
                    /*Not evaluated yet.*/
                    case type_expr(func(id(vn))) of
                      FunctionTypeExpr(arg, fa, res) →
                        let
                          ar =
                            getAccessObs(
                              idset ∪ {id(vn)},
                              value_expr(func(id(vn))),
                              Known(type_expr(res)), func,
                              trans),
                          ar′ =
                            getAccessObsListList(
                              idset, vl,
                              expTypeList(arg, len vl),
                              func, trans)
                        in
                          mk_AccessResult(
                            removeDuplets(
                              getAccess(arg, trans) ^
                              accessList(ar) ^
                              accessList(ar′) ^
                              getSequencingAccess(
                                ve, type_expr(res), et,
                                trans, func)),
                            idSet(ar) ∪ idSet(ar′))
                        end
                    end
                  end
                else
                  getAccessObsList(
                     idset, vl, et, func, trans)
                end,
            _ →
                getAccessObsList(idset, vl, et, func, trans)
        end,
      BracketedExpr(bve) →
        getAccessObs(idset, bve, et, func, trans),
      ValueInfixExpr(first, op, second) →
        getAccessObsList(
           idset, ⟨first, second⟩, Unknown, func, trans
        ),
      ValuePrefixExpr(op, operand) →
        getAccessObs(
           idset, operand, Unknown, func, trans),
```

```
LetExpr(ldl, lve) →
   let
      ar =
         getAccessObsLetDefList(
            idset, ldl, func, trans),
      ar′ =
         getAccessObs(idSet(ar), lve, et, func, trans)
   in
      mk_AccessResult(
         removeDuplets(
            accessList(ar) ⌢ accessList(ar′)),
         idSet(ar′))
   end,
Make_IfExpr(ie) →
   getAccessObsListList(
      idset,
      ⟨condition(ie)⟩ ⌢
      elsifToList(elsif_list(ie)) ⌢ ⟨else_case(ie)
      ⟩,
      ⟨Known(TypeLiteral(BOOL))⟩ ⌢
      elsifToTypeList(elsif_list(ie), et) ⌢ ⟨et⟩,
      func, trans),
CaseExpr(cond, cbl) →
   let
      ar =
         getAccessObs(
            idset, cond, Known(TypeLiteral(BOOL)),
            func, trans),
      ar′ =
         getAccessObsCaseBranch(
            idSet(ar), cbl, et, func, trans)
   in
      mk_AccessResult(
         removeDuplets(
            accessList(ar) ⌢ accessList(ar′)),
         idSet(ar′))
   end
end,

/*Returns the type names accessed when a sequencing
expression must be established.*/
/*
Arguments:
=======
```

```
te: the type of the value expression
Results:
=====
Access_list: the resulting access list
*/
getSequencingAccess :
   ValueExpr × TypeExpr × ExpType × TRANS × FUNC →
      Access*
getSequencingAccess(ve, te, et, trans, func) ≡
   if
      isTGen(ve, trans, func) ∧
      ∼ returnsNonTOI(ve, et, trans, func) ∧
      ∼ equalsType(te, et)
   then getTOIReturnsListTN(te, trans)
   else ⟨⟩
   end,

/*Returns an access list of the values of types
of interest read from during evaluation of a
set expression.*/
/*
Arguments:
=======
idset: the functions already evaluated
se: the set expression
et: the expected type of the set expression
Result:
=======
AccessResult: the corresponding access result
*/
getAccessObsSetExpr :
   Id-set × SetExpr × ExpType × FUNC × TRANS →
      AccessResult
getAccessObsSetExpr(idset, se, et, func, trans) ≡
   case se of
      RangedSetExpr(fve, sve) →
         getAccessObsListList(
             idset, ⟨fve, sve⟩, ⟨et, et⟩, func, trans),
      EnumeratedSetExpr(ovel) →
         getAccessObsOptVEL(idset, ovel, et, func, trans),
      ComprehendedSetExpr(ve, typlist, or) →
         let
            ar = getAccessObs(idset, ve, et, func, trans),
            ar′ =
```

```
                getAccessObsList(
                    idSet(ar), getVEOptRestriction(or),
                    Known(TypeLiteral(BOOL)), func, trans)
            in
                mk_AccessResult(
                    removeDuplets(
                        accessList(ar) ⌢ accessList(ar′)),
                    idSet(ar′))
            end
    end,

/*Returns an access list of the values of types
of interest read from during evaluation of a
list expression.*/
/*
Arguments:
=======
idset: the functions already evaluated
le: the list expression
et: the expected type of the list expression
Result:
=======
AccessResult: the corresponding access result
*/
getAccessObsListExpr :
    Id-set × ListExpr × ExpType × FUNC × TRANS →
        AccessResult
getAccessObsListExpr(idset, le, et, func, trans) ≡
    case le of
        RangedListExpr(fve, sve) →
            getAccessObsListList(
                idset, ⟨fve, sve⟩, ⟨et, et⟩, func, trans),
        EnumeratedListExpr(ovel) →
            getAccessObsOptVEL(idset, ovel, et, func, trans),
        ComprehendedListExpr(ve1, b, ve2, or) →
            getAccessObsListList(
                idset,
                ⟨ve1⟩ ⌢ ⟨ve2⟩ ⌢ getVEOptRestriction(or),
                ⟨et, et, Known(TypeLiteral(BOOL))⟩, func,
                trans)
    end,

/*Returns an access list of the values of types
of interest read from during evaluation of a
```

```
map expression.*/
/*
Arguments:
=======
idset: the functions already evaluated
me: the map expression
et: the expected type of the map expression
Result:
=======
AccessResult: the corresponding access result
*/
getAccessObsMapExpr :
   Id-set × MapExpr × MapType × FUNC × TRANS →
      AccessResult
getAccessObsMapExpr(idset, me, et, func, trans) ≡
   case me of
      EnumeratedMapExpr(ovel) →
         getAccessObsOptVEPL(idset, ovel, et, func, trans),
      ComprehendedMapExpr(ve, typlist, or) →
         getAccessObsListList(
             idset,
             ⟨first(ve), second(ve)⟩ ^
             getVEOptRestriction(or),
             ⟨tedom(et), terange(et),
                Known(TypeLiteral(BOOL))⟩, func, trans)
   end,

/*Returns an access list of the values of types
of interest read from during evaluation of an
optional value expression list.*/
/*
Arguments:
=======
idset: the functions already evaluated
ovel: the optional value expression list
et: the expected type of the optional value expressions
list
Result:
=======
AccessResult: the corresponding access result
*/
getAccessObsOptVEL :
   Id-set × OptionalValueExprList × ExpType ×
   FUNC × TRANS →
```

```
        AccessResult
getAccessObsOptVEL(idset, ovel, et, func, trans) ≡
    case ovel of
        ValueExprList(vel) →
            getAccessObsList(idset, vel, et, func, trans),
        NoValueExprList → mk_AccessResult(⟨⟩, idset)
    end,

/*Returns an access list of the values of types
of interest read from during evaluation of an
optional value expression pair list.*/
/*
Arguments:
=======
idset: the functions already evaluated
ovel: the optional value expression pair list
et: the expected type of the optional value expression
pair list
Result:
=======
AccessResult: the corresponding access result
*/
getAccessObsOptVEPL :
    Id-set × OptionalValueExprPairList × MapType ×
    FUNC × TRANS →
        AccessResult
getAccessObsOptVEPL(idset, ovel, et, func, trans) ≡
    case ovel of
        ValueExprPairList(vel) →
            getAccessObsListList(
                idset, pairListToList(vel),
                pairListToTypeList(vel, et), func, trans),
        NoValueExprPairList →
            mk_AccessResult(⟨⟩, idset)
    end,

/*Turns a value expression pair list into a value
expression list.*/
/*
Arguments:
=======
vepl: the value expression pair list
Result:
=======
```

ValueExpr_list: the resulting value expression
list
*/
pairListToList : ValueExprPair* → ValueExpr*
pairListToList(vepl) ≡
  if vepl = ⟨⟩ **then** ⟨⟩
  else
    ⟨first(**hd** vepl), second(**hd** vepl)⟩ ⌢
    pairListToList(**tl** vepl)
  end,

/*Turns a value expression pair list into an expected
type list.*/
/*
Arguments:
=======
vepl: the value expression pair list
et: the expected type of the value expression
pair
Result:
=======
ValueExpr_list: the resulting value expression
list
*/
pairListToTypeList :
  ValueExprPair* × MapType → ExpType*
pairListToTypeList(vepl, et) ≡
  if vepl = ⟨⟩ **then** ⟨⟩
  else
    ⟨tedom(et), terange(et)⟩ ⌢
    pairListToTypeList(**tl** vepl, et)
  end,

/*Returns the constituent value expression of
an optional restriction if any.*/
/*
Arguments:
=======
or: the optional restriction
Result:
=======
ValueExpr_list: the resulting value expression
list
*/

getVEOptRestriction :
  OptionalRestriction → ValueExpr^*
getVEOptRestriction(or) ≡
  case or **of**
    Restriction(ve) → ⟨ve⟩,
    NoRestriction → ⟨⟩
  end,

/*Returns a list of value expressions made from
an elsif list
*/
/*
Arguments:
=======
eil: the elsif list
Result:
=======
ValueExpr_list: the resulting value expression
list
*/
elsifToList : Elsif^* → ValueExpr^*
elsifToList(eil) ≡
  if eil = ⟨⟩ **then** ⟨⟩
  else
    ⟨condition(**hd** eil)⟩ ⌢ ⟨elsif_case(**hd** eil)⟩ ⌢
    elsifToList(**tl** eil)
  end,

/*Returns a list of expected types made from an
elsif list.*/
/*
Arguments:
=======
eil: the elsif list
et: the expected type of the if expression
Result:
=======
ExpType_list: the resulting expected type list
*/
elsifToTypeList :
  Elsif^* × ExpType → ExpType^*
elsifToTypeList(eil, et) ≡
  if eil = ⟨⟩ **then** ⟨⟩
  else

```
        ⟨Known(TypeLiteral(BOOL))⟩ ⌢ ⟨et⟩ ⌢
        elsifToTypeList(tl eil, et)
    end,

/*Returns an access list of the values of types
of interest read from during evaluation of a
function.*/
/*
Arguments:
=======
idset: the functions already evaluated
vel: the value expression list
et: the expected type of the arguments of the
function
Result:
=======
AccessResult: the corresponding access result
*/
getAccessObsList :
    Id-set × ValueExpr* × ExpType × FUNC × TRANS →
        AccessResult
getAccessObsList(idset, vel, et, func, trans) ≡
    if vel = ⟨⟩ then mk_AccessResult(⟨⟩, idset)
    else
        let
            ar =
                getAccessObs(
                    idset, hd vel, getHead(et), func, trans),
            ar′ =
                getAccessObsList(
                    idSet(ar), tl vel, getTail(et), func, trans)
        in
            mk_AccessResult(
                removeDuplets(
                    accessList(ar) ⌢ accessList(ar′)),
                idSet(ar′))
        end
    end,

/*Returns an access list of the values of types
of interest read from during evaluation of a
function.*/
/*
Arguments:
```

```
=======
idset: the functions already evaluated
vel: the value expression list
etl: the list expected types of the value expression
list
Result:
=======
AccessResult: the corresponding access result
*/
getAccessObsListList :
   Id-set × ValueExpr* × ExpType* × FUNC ×
   TRANS →
      AccessResult
getAccessObsListList(idset, vel, etl, func, trans) ≡
   if vel = ⟨⟩ then mk_AccessResult(⟨⟩, idset)
   else
      let
         ar =
            getAccessObs(
               idset, hd vel, hd etl, func, trans),
         ar′ =
            getAccessObsListList(
               idSet(ar), tl vel, tl etl, func, trans)
      in
         mk_AccessResult(
            removeDuplets(
               accessList(ar) ⁀ accessList(ar′)),
            idSet(ar′))
      end
   end,

/*Returns an access list of the values of types
of interest read from during evaluation of a
function.*/
/*
Arguments:
=======
idset: the functions already evaluated
ldl: the let def list
Result:
=======
AccessResult: the corresponding access result
*/
getAccessObsLetDefList :
```

```
    Id-set × LetDef* × FUNC × TRANS →
        AccessResult
getAccessObsLetDefList(idset, ldl, func, trans) ≡
    if ldl = ⟨⟩ then mk_AccessResult(⟨⟩, idset)
    else
        let
            ar =
                getAccessObs(
                    idset, value_expr(hd ldl), Unknown, func,
                    trans),
            ar′ =
                getAccessObsLetDefList(
                    idSet(ar), tl ldl, func, trans)
        in
            mk_AccessResult(
                removeDuplets(
                    accessList(ar) ^ accessList(ar′)),
                idSet(ar′))
        end
    end,

/*Returns an access list of the values of types
of interest read from during evaluation of a
case branch list.*/
/*
Arguments:
=======
idset: the functions already evaluated
cbl: the case branch list
et: the expected type of the case expression
Result:
=======
AccessResult: the corresponding access result
*/
getAccessObsCaseBranch :
    Id-set × CaseBranch* × ExpType × FUNC × TRANS →
        AccessResult
getAccessObsCaseBranch(idset, cbl, et, func, trans) ≡
    if cbl = ⟨⟩ then mk_AccessResult(⟨⟩, idset)
    else
        let
            ar =
                getAccessObs(
                    idset, value_expr(hd cbl), et, func, trans),
```

```
          ar′ =
            getAccessObsCaseBranch(
              idSet(ar), tl cbl, et, func, trans)
        in
          mk_AccessResult(
            removeDuplets(
               accessList(ar) ⌢ accessList(ar′)),
            idSet(ar′))
        end
      end,

  /*Returns an access list of the values of types
  of interest written to during evaluation of a
  function.*/
  /*
  Arguments:
  =======
  idset: the functions already evaluated
  ve: the value expression
  Result:
  =======
  AccessResult: the corresponding access result
  */
  getAccessGen :
    Id-set × ValueExpr × ExpType × FUNC × TRANS →
      AccessResult
  getAccessGen(idset, ve, et, func, trans) ≡
    case ve of
      Make_ValueLiteral(vl) →
        mk_AccessResult(⟨⟩, idset),
      Make_ValueOrVariableName(vn) →
        if id(vn) ∈ dom func
        then
          getAccessGen(
            idset, ApplicationExpr(ve, ⟨⟩), et, func,
            trans)
        else
          mk_AccessResult(
            getAssignExpr(ve, et, trans), idset)
        end,
      Make_BasicExpr(be) → mk_AccessResult(⟨⟩, idset),
      ProductExpr(vel) →
        getAccessGenList(
          idset, vel,
```

```
            expTypeToExpTypeList(
                  removeBrackets(et), len vel), func, trans),
    Make_SetExpr(se) →
       getAccessGenSetExpr(
            idset, se, getSetType(et), func, trans),
    Make_ListExpr(le) →
       getAccessGenListExpr(
            idset, le, getListType(et), func, trans),
    Make_MapExpr(me) →
       getAccessGenMapExpr(
            idset, me, getMapType(et), func, trans),
    ApplicationExpr(ave, vl) →
       case ave of
          Make_ValueOrVariableName(vn) →
             if id(vn) ∈ dom func
             then
                if id(vn) ∈ idset
                then
                   case type_expr(func(id(vn))) of
                      FunctionTypeExpr(arg, fa, res) →
                         /*Already evaluated.*/
                         getAccessGenList(
                           idset, vl,
                           expTypeList(arg, len vl), func,
                           trans)
                   end
                else
                   /*Not evaluated yet.*/
                   case type_expr(func(id(vn))) of
                      FunctionTypeExpr(arg, fa, res) →
                         let
                           ar =
                             getAccessGen(
                                idset ∪ {id(vn)},
                                value_expr(func(id(vn))),
                                Known(type_expr(res)), func,
                                trans),
                           ar′ =
                             getAccessGenList(
                                idset, vl,
                                expTypeList(arg, len vl),
                                func, trans)
                         in
                           mk_AccessResult(
```

```
                                  removeDuplets(
                                    getAccess(
                                      type_expr(res), trans) ⌢
                                    accessList(ar) ⌢
                                    accessList(ar′)),
                                  idSet(ar) ∪ idSet(ar′))
                              end
                         end
                      end
                    else
                      getAccessGenList(
                          idset, vl,
                          expTypeToExpTypeList(
                              removeBrackets(et), len vl), func,
                          trans)
                    end,
                  _ →
                    getAccessGenList(
                        idset, vl,
                        expTypeToExpTypeList(
                            removeBrackets(et), len vl), func,
                        trans)
                end,
            BracketedExpr(bve) →
                getAccessGen(idset, bve, et, func, trans),
            ValueInfixExpr(first, op, second) →
                getAccessGenList(
                    idset, ⟨first, second⟩,
                    ⟨Unknown, Unknown⟩, func, trans),
            ValuePrefixExpr(op, operand) →
                getAccessGen(
                    idset, operand, Unknown, func, trans),
            LetExpr(ldl, lve) →
                let
                    ar =
                        getAccessGenLetDefList(
                            idset, ldl, func, trans),
                    ar′ =
                        getAccessGen(idSet(ar), lve, et, func, trans)
                in
                    mk_AccessResult(
                        removeDuplets(
                            accessList(ar) ⌢ accessList(ar′)),
                        idSet(ar′))
```

```
          end,
        Make_IfExpr(ie) →
          getAccessGenListList(
              idset,
              ⟨condition(ie)⟩ ⌢
              elsifToList(elsif_list(ie)) ⌢ ⟨else_case(ie)
              ⟩,
              ⟨Known(TypeLiteral(BOOL))⟩ ⌢
              elsifToTypeList(elsif_list(ie), et) ⌢ ⟨et⟩,
              func, trans),
        CaseExpr(cond, cbl) →
          let
            ar =
              getAccessGen(
                  idset, cond, Known(TypeLiteral(BOOL)),
                  func, trans),
            ar′ =
              getAccessGenCaseBranch(
                  idSet(ar), cbl, et, func, trans)
          in
            mk_AccessResult(
              removeDuplets(
                  accessList(ar) ⌢ accessList(ar′)),
              idSet(ar′))
          end
    end,

/*Checks if an assignment expression should be
established.*/
/*
Arguments:
=======
ve: the value expression
et: the corresponding expected type
Results:
=======
Bool: true if an assignment expression must be
established, false otherwise
*/
getAssignExpr :
    ValueExpr × ExpType × TRANS → Access*
getAssignExpr(ve, et, trans) ≡
    case et of
      Known(te) →
```

```
            case te of
               TypeName(tn) →
                  if tn ∈ dom trans
                  then
                     /*Type of interest.*/
                     ⟨AccessValueOrVariableName(
                           mk_ValueOrVariableName(tn))⟩
                  else
                     /*Not type of interest.*/
                     ⟨⟩
                  end,
               _ → ⟨⟩
            end,
         Unknown → ⟨⟩
      end,

   /*Returns an access list of the values of types
   of interest written to during evaluation of a
   set expression.*/
   /*
   Arguments:
   =======
   idset: the functions already evaluated
   se: the set expression
   Result:
   =======
   AccessResult: the corresponding access result
   */
   getAccessGenSetExpr :
      Id-set × SetExpr × ExpType × FUNC × TRANS →
         AccessResult
   getAccessGenSetExpr(idset, se, et, func, trans) ≡
      case se of
         RangedSetExpr(fve, sve) →
            getAccessGenListList(
               idset, ⟨fve, sve⟩, ⟨et, et⟩, func, trans),
         EnumeratedSetExpr(ovel) →
            getAccessGenOptVEL(idset, ovel, et, func, trans),
         ComprehendedSetExpr(ve, typlist, or) →
            let
               ar = getAccessGen(idset, ve, et, func, trans),
               ar′ =
                  getAccessGenList(
                     idSet(ar), getVEOptRestriction(or),
```

```
                expTypeToExpTypeList(
                    removeBrackets(
                        Known(TypeLiteral(BOOL))), 1),
                func, trans)
        in
            mk_AccessResult(
                removeDuplets(
                    accessList(ar) ⌢ accessList(ar′)),
                idSet(ar′))
        end
    end,

/*Returns an access list of the values of types
of interest written to during evaluation of a
list expression.*/
/*
Arguments:
=======
idset: the functions already evaluated
le: the list expression
Result:
=======
AccessResult: the corresponding access result
*/
getAccessGenListExpr :
    Id-set × ListExpr × ExpType × FUNC × TRANS →
        AccessResult
getAccessGenListExpr(idset, le, et, func, trans) ≡
    case le of
        RangedListExpr(fve, sve) →
            getAccessGenListList(
                idset, ⟨fve, sve⟩, ⟨et, et⟩, func, trans),
        EnumeratedListExpr(ovel) →
            getAccessGenOptVEL(idset, ovel, et, func, trans),
        ComprehendedListExpr(ve1, b, ve2, or) →
            getAccessGenListList(
                idset,
                ⟨ve1⟩ ⌢ ⟨ve2⟩ ⌢ getVEOptRestriction(or),
                ⟨et, et, Known(TypeLiteral(BOOL))⟩, func,
                trans)
    end,

/*Returns an access list of the values of types
of interest written to during evaluation of a
```

```
map expression.*/
/*
Arguments:
=======
idset: the functions already evaluated
me: the map expression
Result:
=======
AccessResult: the corresponding access result
*/
getAccessGenMapExpr :
   Id-set × MapExpr × MapType × FUNC × TRANS →
      AccessResult
getAccessGenMapExpr(idset, me, et, func, trans) ≡
   case me of
      EnumeratedMapExpr(ovel) →
         getAccessGenOptVEPL(idset, ovel, et, func, trans),
      ComprehendedMapExpr(ve, typlist, or) →
         getAccessGenListList(
            idset,
            ⟨first(ve), second(ve)⟩ ^
            getVEOptRestriction(or),
            ⟨tedom(et), terange(et),
               Known(TypeLiteral(BOOL))⟩, func, trans)
   end,

/*Returns an access list of the values of types
of interest written to during evaluation of an
optional value expression list.*/
/*
Arguments:
=======
idset: the functions already evaluated
ovel: the optional value expression list
Result:
=======
AccessResult: the corresponding access result
*/
getAccessGenOptVEL :
   Id-set × OptionalValueExprList × ExpType ×
   FUNC × TRANS →
      AccessResult
getAccessGenOptVEL(idset, ovel, et, func, trans) ≡
   case ovel of
```

```
    ValueExprList(vel) →
      getAccessGenList(
        idset, vel,
        expTypeToExpTypeList(
          removeBrackets(et), len vel), func, trans),
    NoValueExprList → mk_AccessResult(⟨⟩, idset)
  end,

/*Returns an access list of the values of types
of interest written to during evaluation of an
optional value expression pair list.*/
/*
Arguments:
=======
idset: the functions already evaluated
ovel: the optional value expression pair list
Result:
=======
AccessResult: the corresponding access result
*/
getAccessGenOptVEPL :
  Id-set × OptionalValueExprPairList × MapType ×
  FUNC × TRANS →
    AccessResult
getAccessGenOptVEPL(idset, ovel, et, func, trans) ≡
  case ovel of
    ValueExprPairList(vel) →
      getAccessGenListList(
        idset, pairListToList(vel),
        pairListToTypeList(vel, et), func, trans),
    NoValueExprPairList →
      mk_AccessResult(⟨⟩, idset)
  end,

/*Returns an access list of the values of types
of interest read from during evaluation of a
function.*/
/*
Arguments:
=======
idset: the functions already evaluated
vel: the value expression list
Result:
=======
```

```
AccessResult: the corresponding access result
*/
getAccessGenList :
   Id-set × ValueExpr* × ExpType* × FUNC ×
   TRANS →
      AccessResult
getAccessGenList(idset, vel, etl, func, trans) ≡
   if vel = ⟨⟩ then mk_AccessResult(⟨⟩, idset)
   else
      let
         ar =
            getAccessGen(
               idset, hd vel, hd etl, func, trans),
         ar′ =
            getAccessGenList(
               idSet(ar), tl vel, tl etl, func, trans)
      in
         mk_AccessResult(
            removeDuplets(
               accessList(ar) ⌢ accessList(ar′)),
            idSet(ar′))
      end
   end,

/*Returns an access list of the values of types
of interest read from during evaluation of a
function.*/
/*
Arguments:
=======
idset: the functions already evaluated
vel: the value expression list
etl: the list expected types of the value expression
list
Result:
=======
AccessResult: the corresponding access result
*/
getAccessGenListList :
   Id-set × ValueExpr* × ExpType* × FUNC ×
   TRANS →
      AccessResult
getAccessGenListList(idset, vel, etl, func, trans) ≡
   if vel = ⟨⟩ then mk_AccessResult(⟨⟩, idset)
```

```
    else
      let
        ar =
          getAccessGen(
             idset, hd vel, hd etl, func, trans),
        ar′ =
          getAccessGenListList(
             idSet(ar), tl vel, tl etl, func, trans)
      in
        mk_AccessResult(
           removeDuplets(
                accessList(ar) ⌢ accessList(ar′)),
           idSet(ar′))
      end
    end,

/*Returns an access list of the values of types
of interest read from during evaluation of a
function.*/
/*
Arguments:
=======
idset: the functions already evaluated
ldl: the let definition list
Result:
=======
AccessResult: the corresponding access result
*/
getAccessGenLetDefList :
  Id-set × LetDef* × FUNC × TRANS →
    AccessResult
getAccessGenLetDefList(idset, ldl, func, trans) ≡
  if ldl = ⟨⟩ then mk_AccessResult(⟨⟩, idset)
  else
    let
      ar =
        getAccessGen(
           idset, value_expr(hd ldl), Unknown, func,
           trans),
      ar′ =
        getAccessGenLetDefList(
           idSet(ar), tl ldl, func, trans)
    in
      mk_AccessResult(
```

```
            removeDuplets(
                  accessList(ar) ⌢ accessList(ar′)),
            idSet(ar′))
      end
   end,

/*Returns an access list of the values of types
of interest read from during evaluation of a
case branch list.*/
/*
Arguments:
=======
idset: the functions already evaluated
cbl: the case branch list
Result:
=======
AccessResult: the corresponding access result
*/
getAccessGenCaseBranch :
   Id-set × CaseBranch* × ExpType × FUNC × TRANS →
      AccessResult
getAccessGenCaseBranch(idset, cbl, et, func, trans) ≡
   if cbl = ⟨⟩ then mk_AccessResult(⟨⟩, idset)
   else
      let
         ar =
            getAccessGen(
                idset, value_expr(hd cbl), et, func, trans),
         ar′ =
            getAccessGenCaseBranch(
                idSet(ar), tl cbl, et, func, trans)
      in
         mk_AccessResult(
            removeDuplets(
                  accessList(ar) ⌢ accessList(ar′)),
            idSet(ar′))
      end
   end,

/***** Access end *****/

/***** Bindings *****/

/*Makes a product binding from a list of bindings.
```

```
*/
/*
Arguments:
=======
bl: the binding list
Result:
=======
Binding: the corresponding binding
*/
makeBinding : Binding* → Binding
makeBinding(bl) ≡
   if len bl = 1 then hd bl
   else Make_ProductBinding(mk_ProductBinding(bl))
   end,

/*Makes a binding list from a value expression
list.*/
/*
Arguments:
=======
length: length of the binding list
counter: the number of bindings made so far
bl: the binding list
vel: the value expression list
Results:
=====
Binding_list: the resulting binding list
ValueExpr_list: the corresponding value expression
list
*/
makeLetBinding :
   Nat × Nat × Binding* × ValueExpr* →
      Binding* × ValueExpr*
makeLetBinding(length, counter, bl, vel) ≡
   if length = 0 then (bl, vel)
   else
      let id = makeId(counter) in
         makeLetBinding(
            length − 1, counter + 1,
            bl ⌢ ⟨IdBinding(id)⟩,
            vel ⌢
            ⟨Make_ValueOrVariableName(
                     mk_ValueOrVariableName(id))⟩)
      end
```

```
    end,

/*Returns the length of a let binding.*/
/*
Arguments:
=======
lb: the let binding
Results:
=======
Int: the length of the let binding
*/
getLengthLetBinding : LetBinding → Int
getLengthLetBinding(lb) ≡
    case lb of
        MakeBinding(b) → getLengthBinding(b),
        MakeRecordPattern(vn, p) → 1,
        MakeListPatternLet(lp) → 1
    end,

/*Returns the length of a binding.*/
/*
Arguments:
=======
b: the binding
Results:
=======
Int: the length of the binding
*/
getLengthBinding : Binding → Int
getLengthBinding(b) ≡
    case b of
        IdBinding(id) → 1,
        Make_ProductBinding(pb) →
            getLengthBindingList(binding_list(pb))
    end,

/*Returns the number of parameters of a binding
list.*/
/*
Arguments:
=======
bl: the binding list
Results:
=====
```

```
Int: the number of parameters
*/
getLengthBindingList : Binding* → Int
getLengthBindingList(bl) ≡
   if bl = ⟨⟩ then 0
   else
      case hd bl of
         IdBinding(id) → 1 + getLengthBindingList(tl bl),
         Make_ProductBinding(pb) →
            getLengthBindingList(binding_list(pb)) +
            getLengthBindingList(tl bl)
      end
   end,

/*Checks if a binding is in a binding set.*/
/*
Arguments:
=======
b: the binding
bl: the binding set
Results:
=====
Bool: true if the binding is in the binding set,
false otherwise
*/
isinBinding : Binding × Binding-set → Bool
isinBinding(b, bl) ≡
   if bl = {} then false
   else
      case hd bl of
         IdBinding(id) →
            if getText(id) = getTextBinding(b) then true
            else isinBinding(b, bl \ {hd bl})
            end,
         Make_ProductBinding(pb) →
            isinBinding(b, elems binding_list(pb)) ∨
            isinBinding(b, bl \ {hd bl})
      end
   end,

/*Returns the text of an id binding.*/
/*
Arguments:
=======
```

```
b: the binding
Results:
=====
Text: the text of the binding
*/
getTextBinding : Binding → Text
getTextBinding(b) ≡
   case b of
      IdBinding(id) → getText(id)
   end,

/***** Bindings end *****/

/***** TRANS *****/

/*Checks if the id of the expected type is in
the Idset.*/
/*
Arguments:
=======
et: the expected type of the value expression
idset: the set of ids, either the domain or range
of TRANS
Results:
=====
Bool: true if the expected type is in the idset,
false otherwise
*/
checkTRANS : ExpType × Id-set → Bool
checkTRANS(et, idset) ≡
   case et of
      Known(te) →
         case te of
            TypeName(tn) → tn ∈ idset,
            _ → false
         end,
      Unknown → false
   end,

/*Checks if the ids of the expected type is in
the Idset.*/
/*
Arguments:
=======
```

```
et: the expected type of the value expression
idset: the set of ids, either the domain or range
of TRANS
Results:
=====
Bool: true if the expected type is in the idset,
false otherwise
*/
containsTRANSName :
   ExpType × TYPINGS × Id-set → Bool
containsTRANSName(et, typings, idset) ≡
   case et of
      Known(te) →
         case te of
            TypeLiteral(tn) → false,
            TypeName(tn) →
               tn ∈ idset ∨
               (idset ∩ elems typings(tn)) ≠ {},
            TypeExprProduct(tel) →
               containsTRANSNameList(tel, typings, idset),
            TypeExprSet(tes) →
               case tes of
                  FiniteSetTypeExpr(fte) →
                     containsTRANSName(
                          Known(fte), typings, idset),
                  InfiniteSetTypeExpr(ite) →
                     containsTRANSName(
                          Known(ite), typings, idset)
               end,
            TypeExprList(tel) →
               case tel of
                  FiniteListTypeExpr(fte) →
                     containsTRANSName(
                          Known(fte), typings, idset),
                  InfiniteListTypeExpr(ite) →
                     containsTRANSName(
                          Known(ite), typings, idset)
               end,
            TypeExprMap(tem) →
               case tem of
                  FiniteMapTypeExpr(tedom, terange) →
                     containsTRANSName(
                          Known(tedom), typings, idset) ∨
                     containsTRANSName(
```

```
                    Known(terange), typings, idset),
                InfiniteMapTypeExpr(tedom, terange) →
                  containsTRANSName(
                    Known(tedom), typings, idset) ∨
                  containsTRANSName(
                    Known(terange), typings, idset)
              end,
            FunctionTypeExpr(arg, fa, res) → false,
            BracketedTypeExpr(bte) →
              containsTRANSName(Known(bte), typings, idset)
          end,
        Unknown → false
      end,

/*Checks if one of the ids of the type expression
list is in the Idset.*/
/*
Arguments:
=======
tel: the type expressionlist
idset: the set of ids, either the domain or range
of TRANS
Results:
=====
Bool: true if on of the ids of the type expression
list is in the idset, false otherwise
*/
containsTRANSNameList :
  TypeExpr* × TYPINGS × Id-set → Bool
containsTRANSNameList(tel, typings, idset) ≡
  if tel = ⟨⟩ then false
  else
    containsTRANSName(Known(hd tel), typings, idset) ∨
    containsTRANSNameList(tl tel, typings, idset)
  end,

/*Checks if the ids of the expected type is in
the Idset.*/
/*
Arguments:
=======
et: the expected type of the value expression
idset: the set of ids, either the domain or range
of TRANS
```

```
Results:
=====
Bool: true if the expected type is in the idset,
false otherwise
*/
includesTRANSName : ExpType × Id-set → Bool
includesTRANSName(et, idset) ≡
   case et of
      Known(te) →
         case te of
            TypeLiteral(tn) → false,
            TypeName(tn) → tn ∈ idset,
            TypeExprProduct(tel) →
               includesTRANSNameList(tel, idset),
            TypeExprSet(tes) →
               case tes of
                  FiniteSetTypeExpr(fte) →
                     includesTRANSName(Known(fte), idset),
                  InfiniteSetTypeExpr(ite) →
                     includesTRANSName(Known(ite), idset)
               end,
            TypeExprList(tel) →
               case tel of
                  FiniteListTypeExpr(fte) →
                     includesTRANSName(Known(fte), idset),
                  InfiniteListTypeExpr(ite) →
                     includesTRANSName(Known(ite), idset)
               end,
            TypeExprMap(tem) →
               case tem of
                  FiniteMapTypeExpr(tedom, terange) →
                     includesTRANSName(Known(tedom), idset) ∨
                     includesTRANSName(Known(terange), idset),
                  InfiniteMapTypeExpr(tedom, terange) →
                     includesTRANSName(Known(tedom), idset) ∨
                     includesTRANSName(Known(terange), idset)
               end,
            FunctionTypeExpr(arg, fa, res) → false,
            BracketedTypeExpr(bte) →
               includesTRANSName(Known(bte), idset)
         end,
      Unknown → false
   end,
```

```
/*Checks if one of the ids of the type expression
list is in the Idset.*/
/*
Arguments:
=======
tel: the type expression list
idset: the set of ids, either the domain or range
of TRANS
Results:
=====
Bool: true if on of the ids of the type expression
list is in the idset, false otherwise
*/
includesTRANSNameList :
   TypeExpr* × Id-set → Bool
includesTRANSNameList(tel, idset) ≡
   if tel = ⟨⟩ then false
   else
      includesTRANSName(Known(hd tel), idset) ∨
      includesTRANSNameList(tl tel, idset)
   end,

/*Gets the variable name corresponding to a type
of interest.
Pre_condition: et = Known(TypeName(id)).*/
/*
Arguments:
=======
et: the expected type
Result:
=======
ValueExpr: the corresponding ValueOrVariableName
*/
getTRANS : ExpType × TRANS → ValueExpr
getTRANS(et, trans) ≡
   case et of
      Known(te) →
         case te of
            TypeName(tn) →
               Make_ValueOrVariableName(
                  mk_ValueOrVariableName(trans(tn)))
         end
   end,
```

```
/*Gets the id corresponding to a type of interest.
Pre_condition: et = Known(TypeName(id)).*/
/*
Arguments:
=======
et: the expected type
Result:
=======
Id: the corresponding Id
*/
getTRANSId : ExpType × TRANS → Id
getTRANSId(et, trans) ≡
   case et of
      Known(te) →
         case te of
            TypeName(tn) → trans(tn)
         end
   end,

/***** TRANS end *****/

/***** TYPINGS *****/

/*Makes a TYPINGS map from a type definition list.
*/
/*
Arguments:
=======
tdl: the type definition list
*/
makeTYPINGSMap : TypeDef* → TYPINGS
makeTYPINGSMap(tdl) ≡
   if tdl = ⟨⟩ then [ ]
   else
      makeTYPINGSEntrance(hd tdl) †
      makeTYPINGSMap(tl tdl)
   end,

/*Makes a TYPINGS map entrance from a type definition.
*/
/*
Arguments:
=======
td: the type definition
```

```
*/
makeTYPINGSEntrance : TypeDef → TYPINGS
makeTYPINGSEntrance(td) ≡
   case td of
      SortDef(id) → [ id ↦ ⟨⟩ ],
      VariantDef(id, vl) →
         [ id ↦ getTYPINGSVariantList(vl) ],
      ShortRecordDef(id, ckl) →
         [ id ↦ getTYPINGSComponentKindList(ckl) ],
      AbbreviationDef(id, te) →
         [ id ↦ getTYPINGSTypeExpr(te) ]
   end,

/*Gets an id list to the TYPINGS map from a variant
list.*/
/*
Arguments:
=======
vl: the variant list
Results:
=====
Id_list: the resulting list of ids
*/
getTYPINGSVariantList : Variant* → Id*
getTYPINGSVariantList(vl) ≡
   if vl = ⟨⟩ then ⟨⟩
   else
      removeDupletsId(
          getTYPINGSVariant(hd vl) ⌢
          getTYPINGSVariantList(tl vl))
   end,

/*Gets an id list to the TYPINGS map from a variant.
*/
/*
Arguments:
=======
v: the variant
Results:
=====
Id_list: the resulting list of ids
*/
getTYPINGSVariant : Variant → Id*
getTYPINGSVariant(v) ≡
```

```
    case v of
       Make_Constructor(c) → ⟨⟩,
       RecordVariant(c, ckl) →
          getTYPINGSComponentKindList(ckl)
    end,

/*Gets an id list to the TYPINGS map from a component
kind list.*/
/*
Arguments:
=======
ckl: the component kind list
Results:
=====
Id_list: the resulting list of ids
*/
getTYPINGSComponentKindList :
    ComponentKind* → Id*
getTYPINGSComponentKindList(ckl) ≡
    if ckl = ⟨⟩ then ⟨⟩
    else
       removeDupletsId(
           getTYPINGSTypeExpr(type_expr(hd ckl)) ^
           getTYPINGSComponentKindList(tl ckl))
    end,

/*Gets an id list to the TYPINGS map from a name
or wildcard list.*/
/*
Arguments:
=======
nwl: the name or wildcard list
Results:
=====
Id_list: the resulting list of ids
*/
getTYPINGSNameWildcardList :
    NameOrWildcard* → Id*
getTYPINGSNameWildcardList(nwl) ≡
    if nwl = ⟨⟩ then ⟨⟩
    else
       removeDupletsId(
           getTYPINGSNameWildcard(hd nwl) ^
           getTYPINGSNameWildcardList(tl nwl))
```

```
    end,

/*Gets an id list to the TYPINGS map from a name
or wildcard.*/
/*
Arguments:
=======
nw: the name or wildcard
Results:
=====
Id_list: the resulting list of ids
*/
getTYPINGSNameWildcard : NameOrWildcard → Id*
getTYPINGSNameWildcard(nw) ≡
    case nw of
        Name(var) → ⟨id(var)⟩,
        Wildcard → ⟨⟩
    end,

/*Gets an id list to the TYPINGS map from a type
expr.*/
/*
Arguments:
=======
te: the type expression
Results:
=====
Id_list: the resulting list of ids
*/
getTYPINGSTypeExpr : TypeExpr → Id*
getTYPINGSTypeExpr(te) ≡
    case te of
        TypeLiteral(tn) → ⟨⟩,
        TypeName(id) → ⟨id⟩,
        TypeExprProduct(tep) →
            getTYPINGSTypeExprList(tep),
        TypeExprSet(tes) →
            case tes of
                FiniteSetTypeExpr(fse) →
                    getTYPINGSTypeExpr(fse),
                InfiniteSetTypeExpr(ise) →
                    getTYPINGSTypeExpr(ise)
            end,
        TypeExprList(les) →
```

```
            case les of
                FiniteListTypeExpr(fle) →
                    getTYPINGSTypeExpr(fle),
                InfiniteListTypeExpr(ile) →
                    getTYPINGSTypeExpr(ile)
            end,
        TypeExprMap(tem) →
            case tem of
                FiniteMapTypeExpr(tedom, terange) →
                    removeDupletsId(
                        getTYPINGSTypeExpr(tedom) ⌢
                        getTYPINGSTypeExpr(terange)),
                InfiniteMapTypeExpr(tedom, terange) →
                    removeDupletsId(
                        getTYPINGSTypeExpr(tedom) ⌢
                        getTYPINGSTypeExpr(terange))
            end,
        FunctionTypeExpr(arg, fa, res) →
            removeDupletsId(
                getTYPINGSTypeExpr(arg) ⌢
                getTYPINGSTypeExpr(type_expr(res))),
        SubtypeExpr(st, ve) →
            getTYPINGSTypeExpr(type_expr(st)),
        BracketedTypeExpr(bte) → getTYPINGSTypeExpr(bte)
    end,

/*Gets an id list to the TYPINGS map from a type
expression list.*/
/*
Arguments:
=======
tel: the type expression list
Results:
=====
Id_list: the resulting list of ids
*/
getTYPINGSTypeExprList : TypeExpr* → Id*
getTYPINGSTypeExprList(tel) ≡
    if tel = ⟨⟩ then ⟨⟩
    else
        removeDupletsId(
            getTYPINGSTypeExpr(hd tel) ⌢
            getTYPINGSTypeExprList(tl tel))
    end,
```

```
/*Gets a type definition list from a declaration
list.*/
/*
Arguments:
=======
dl: the declaration list
Result:
=======
TypeDef_list: the corresponding type definition
list
*/
getTypeDecl : Decl* → TypeDef*
getTypeDecl(dl) ≡
   if dl = ⟨⟩ then ⟨⟩
   else
      case hd dl of
         TypeDecl(vdl) → vdl ⌢ getTypeDecl(tl dl),
         _ → getTypeDecl(tl dl)
      end
   end,

/*Gets the ids of a list of type definitions.
*/
/*
Arguments:
=======
tdl: the list of type definitions
Results:
=====
Id_list: the resulting list of ids
*/
getIds : TypeDef* → Id*
getIds(tdl) ≡
   if tdl = ⟨⟩ then ⟨⟩
   else
      case hd tdl of
         SortDef(id) → ⟨id⟩ ⌢ getIds(tl tdl),
         VariantDef(id, vl) → ⟨id⟩ ⌢ getIds(tl tdl),
         ShortRecordDef(id, ckl) →
            ⟨id⟩ ⌢ getIds(tl tdl),
         AbbreviationDef(id, te) →
            ⟨id⟩ ⌢ getIds(tl tdl)
      end
```

```
   end,

/*Expands the TYPINGS map.*/
/*
Arguments:
=======
idl: the list of type def ids
*/
expandTYPINGSMap :
   Id* × TYPINGS × TRANS → TYPINGS
expandTYPINGSMap(idl, typings, trans) ≡
   if idl = ⟨⟩ then typings
   else
      expandTYPINGSMap(
         tl idl,
         typings †
         [ hd idl ↦
            removeNonTOI(
               getIdList(hd idl, {}, typings), dom trans) ],
         trans)
   end,

/*Removes ids not of the type of interest from
an id list.*/
/*
Arguments:
=======
idl: the list of ids
ids: a set of type names of the type of interest
Results:
=====
Id_list: the resulting list of ids
*/
removeNonTOI : Id* × Id-set → Id*
removeNonTOI(idl, ids) ≡
   if idl = ⟨⟩ then ⟨⟩
   else
      if hd idl ∉ ids then removeNonTOI(tl idl, ids)
      else ⟨hd idl⟩ ⌢ removeNonTOI(tl idl, ids)
      end
   end,

/*Gets the ids which is part an id′s type.*/
/*
```

```
Arguments:
=======
id: the list id
ids: the ids already evaluated
Results:
=====
Id_list: the resulting list of ids
*/
getIdList : Id × Id-set × TYPINGS → Id*
getIdList(id, ids, typings) ≡
   getIdListIdList(typings(id), ids ∪ {id}, typings),

/*Gets the ids which is part of the types of an
id list.*/
/*
Arguments:
=======
idl: the id list
ids: the ids already evaluated
Results:
=====
Id_list: the resulting list of ids
*/
getIdListIdList :
   Id* × Id-set × TYPINGS → Id*
getIdListIdList(idl, ids, typings) ≡
   if idl = ⟨⟩ then ⟨⟩
   else
      if hd idl ∉ ids
      then
         ⟨hd idl⟩ ⌢ getIdList(hd idl, ids, typings) ⌢
         getIdListIdList(
            tl idl, ids ∪ {hd idl}, typings)
      else
         removeDupletsId(
            ⟨hd idl⟩ ⌢
            getIdListIdList(tl idl, ids, typings))
      end
   end,

/***** TYPINGS end *****/

/***** FUNC *****/
```

```
/*Establishes the FUNC map from a value declaration
list.*/
/*
Arguments:
=======
dl: the value declaration list
*/
establishFuncMap : Decl* × TRANS → FUNC
establishFuncMap(dl, trans) ≡
   let vdl = getValueDecl(dl) in
      expandFuncMap(
          vdl, makeFuncMap(vdl, [ ], trans), trans)
   end,

/*Gets a value definition list from a declaration
list.*/
/*
Arguments:
=======
dl: the declaration list
Result:
=======
ValueDef_list: the corresponding value definition
list
*/
getValueDecl : Decl* → ValueDef*
getValueDecl(dl) ≡
   if dl = ⟨⟩ then ⟨⟩
   else
      case hd dl of
         ValueDecl(vdl) → vdl ⌢ getValueDecl(tl dl),
         _ → getValueDecl(tl dl)
      end
   end,

/*Establishes the FUNC map from a value definition
list.*/
/*
Arguments:
=======
vdl: the value definition list
*/
makeFuncMap : ValueDef* × FUNC × TRANS → FUNC
makeFuncMap(vdl, func, trans) ≡
```

```
    if vdl = ⟨⟩ then func
    else
       makeFuncMap(
           tl vdl, func † getMapEntrance(hd vdl, trans),
           trans)
    end,

/*Estalishes an entrance in the FUNC map based
on a value definition.*/
/*
Arguments:
=======
vd: the value definition
*/
getMapEntrance : ValueDef × TRANS → FUNC
getMapEntrance(vd, trans) ≡
    case vd of
       ExplicitValueDef(st, ve) →
          if
             includesTRANSName(
                 Known(type_expr(st)), dom trans)
          then
             case binding(st) of
                IdBinding(id) →
                   [id ↦
                    mk_FuncSpec(
                        FunctionTypeExpr(
                           TypeLiteral(UNIT),
                           TOTAL_FUNCTION_ARROW,
                           mk_ResultDesc(
                              NoReadAccessMode,
                              NoWriteAccessMode, type_expr(st))
                        ), ve, ⟨⟩, ⟨⟩)],
                Make_ProductBinding(pb) → []
             end
          else []
          end,
       ExplicitFunctionDef(st, ffa, ve, precond) →
          case binding(st) of
             IdBinding(id) →
                [id ↦
                 mk_FuncSpec(type_expr(st), ve, ⟨⟩, ⟨⟩)]
          end
    end,
```

```
/*Expands the FUNC map with read and write lists.
*/
/*
Arguments:
=======
vdl: the value definition list
*/
expandFuncMap : ValueDef* × FUNC × TRANS → FUNC
expandFuncMap(vdl, func, trans) ≡
   if vdl = ⟨⟩ then func
   else
      expandFuncMap(
          tl vdl,
          func † expandMapEntrance(hd vdl, func, trans),
          trans)
   end,

/*Expands the an entrance in the FUNC map with
read and write lists.*/
/*
Arguments:
=======
vd: the value definition
*/
expandMapEntrance : ValueDef × FUNC × TRANS → FUNC
expandMapEntrance(vd, func, trans) ≡
   case vd of
      ExplicitValueDef(st, ve) →
         if
            includesTRANSName(
                Known(type_expr(st)), dom trans)
         then
            case binding(st) of
               IdBinding(id) →
                  [id ↦
                   mk_FuncSpec(
                        FunctionTypeExpr(
                            TypeLiteral(UNIT),
                            TOTAL_FUNCTION_ARROW,
                            mk_ResultDesc(
                               NoReadAccessMode,
                               NoWriteAccessMode, type_expr(st))
                        ), ve, ⟨⟩,
```

```
                              removeDuplets(
                                getAccess(type_expr(st), trans) ⌢
                                accessList(
                                  getAccessGen(
                                    {id}, ve,
                                    Known(type_expr(st)), func,
                                    trans))))],
                  Make_ProductBinding(pb) → [ ]
                end
              else [ ]
              end,
          ExplicitFunctionDef(st, ffa, ve, precond) →
              case binding(st) of
                IdBinding(id) →
                  case type_expr(st) of
                    FunctionTypeExpr(arg, fa, res) →
                      [id ↦
                       mk_FuncSpec(
                           type_expr(st), ve,
                           removeDuplets(
                             getAccess(arg, trans) ⌢
                             accessList(
                               getAccessObs(
                                 {id}, ve,
                                 Known(type_expr(res)),
                                 func, trans)) ⌢
                             accessList(
                               getAccessObsOptPreCondition(
                                 {id}, precond,
                                 Known(TypeLiteral(BOOL)),
                                 func, trans))),
                           removeDuplets(
                             getAccess(type_expr(res), trans) ⌢
                             accessList(
                               getAccessGen(
                                 {id}, ve,
                                 Known(type_expr(res)),
                                 func, trans))))]
                  end
              end
        end,

/***** FUNC end *****/
```

```
/∗**** ENV ****∗/

/∗Updates the environment. Types of interests
that have been written to must be removed from
the ENV.∗/
/∗
Arguments:
=======
te: the expected type of the value expression
∗/
updateENV : ENV × ExpType × TRANS → ENV
updateENV(env, et, trans) ≡
   case et of
      Known(te) →
         case te of
            TypeName(tn) →
               if tn ∈ dom trans
               then
                  /∗Type of interest.∗/
                  env \ {tn}
               else
                  /∗Not type of interest.∗/
                  env
               end,
            _ → env
         end,
      Unknown → env
   end,

/∗Sets ENV.∗/
/∗
Arguments:
=======
al: the list of types of interest written to
∗/
setEnv : Access* × ENV → ENV
setEnv(al, env) ≡
   if al = ⟨⟩ then env
   else
      case hd al of
         AccessValueOrVariableName(vn) →
            setEnv(tl al, env \ {id(vn)}),
         _ → setEnv(tl al, env)
      end
```

```
    end,

/*Updates the environment according to the let
binding.
*/
/*
Arguments:
=======
lb: the let binding
te: the type expression of the let binding
*/
makeENV : LetBinding × TypeExpr × TRANS → ENV
makeENV(lb, te, trans) ≡
    case lb of
        MakeBinding(b) → makeENVBinding(b, te, trans),
        _ → ([ ])
    end,

/*Makes the appropriate ENV binding.*/
/*
Arguments:
=======
b: the binding
te: the corresponding type
*/
makeENVBinding : Binding × TypeExpr × TRANS → ENV
makeENVBinding(b, te, trans) ≡
    case b of
        IdBinding(idb) →
            case te of
                TypeName(tn) →
                    if tn ∈ dom trans then ([ tn ↦ b ])
                    else ([ ])
                    end,
                _ → ([ ])
            end,
        Make_ProductBinding(pb) →
            makeENVBindingList(
                binding_list(pb), typeExprToList(te), trans)
    end,

/*Makes the appropriate ENV bindings from a binding
list.*/
/*
```

```
Arguments:
=======
bl: the binding list
tel: the corresponding type expression list
*/
makeENVBindingList :
   Binding* × TypeExpr* × TRANS → ENV
makeENVBindingList(bl, tel, trans) ≡
   if bl = ⟨⟩ then ([ ])
   else
      makeENVBinding(hd bl, hd tel, trans) †
      makeENVBindingList(tl bl, tl tel, trans)
   end,

/***** ENV end *****/

/***** TYPES *****/

/*Updates TYPES according to the let binding.
*/
/*
Arguments:
=======
lb: the let binding
et: the expected type of the let binding
*/
makeTYPES : LetBinding × ExpType × TRANS → TYPES
makeTYPES(lb, et, trans) ≡
   case lb of
      MakeBinding(b) → makeTYPESBinding(b, et, trans),
      _ → ([ ])
   end,

/*Makes the appropriate TYPES binding.*/
/*
Arguments:
=======
b: the binding
et: the corresponding expected type
*/
makeTYPESBinding :
   Binding × ExpType × TRANS → TYPES
makeTYPESBinding(b, et, trans) ≡
   case b of
```

```
        IdBinding(id) → [ b ↦ et ],
        Make_ProductBinding(pb) →
          makeTYPESBindingList(
            binding_list(pb),
            expTypeToExpTypeList(
              removeBrackets(et), len binding_list(pb)),
            trans)
  end,

/*Makes the appropriate TYPES binding from a binding
list.*/
/*
Arguments:
=======
b: the binding list
etl: the corresponding expected types
*/
makeTYPESBindingList :
  Binding* × ExpType* × TRANS → TYPES
makeTYPESBindingList(bl, etl, trans) ≡
  if bl = ⟨⟩ then ([ ])
  else
    makeTYPESBinding(hd bl, hd etl, trans) †
    makeTYPESBindingList(tl bl, tl etl, trans)
  end,

/*Makes the appropriate TYPES binding from a binding
list.*/
/*
Arguments:
=======
b: the binding list
te: the corresponding types
*/
makeTYPESMap : Binding* × TypeExpr → TYPES
makeTYPESMap(bl, te) ≡
  if bl = ⟨⟩ then [ ]
  else
    case hd bl of
      IdBinding(id) →
        [ hd bl ↦ Known(te) ] †
        makeTYPESMap(tl bl, te),
      Make_ProductBinding(bil) →
        makeTYPESMap(binding_list(bil), te) †
```

```
            makeTYPESMap(tl bl, te)
        end
    end,

/*Alters TYPES when a let def list is established.
*/
/*
Arguments:
=======
ldl: the let def list
*/
alterTYPESMap : TYPES × LetDef* → TYPES
alterTYPESMap(types, ldl) ≡
    if ldl = ⟨⟩ then types
    else
        case binding(hd ldl) of
            MakeBinding(b) →
                case b of
                    IdBinding(id) →
                        alterTYPESMap(
                            types † [ b ↦ Unknown ], tl ldl),
                    Make_ProductBinding(bl) →
                        alterTYPESMap(
                            alterTYPESMapList(
                                types, binding_list(bl)), tl ldl)
                end
        end
    end,

/*Alters TYPES from a binding list.*/
/*
Arguments:
=======
bl: the binding list
*/
alterTYPESMapList : TYPES × Binding* → TYPES
alterTYPESMapList(types, bl) ≡
    if bl = ⟨⟩ then types
    else
        case hd bl of
            IdBinding(id) →
                alterTYPESMapList(
                    types † [ hd bl ↦ Unknown ], tl bl),
            Make_ProductBinding(bil) →
```

```
            alterTYPESMapList(
                alterTYPESMapList(types, binding_list(bil)),
                tl bl)
        end
    end,

/***** TYPES end *****/

/***** Id *****/

/*Removes duplets from an id list.*/
/*
Arguments:
=======
il: the id list
Result:
=======
Id_list: the resulting id list
*/
removeDupletsId : Id* → Id*
removeDupletsId(il) ≡
    if len il = card elems il then il
    else
        if hd il ∈ elems tl il
        then removeDupletsId(tl il)
        else ⟨hd il⟩ ⌢ removeDupletsId(tl il)
        end
    end,

/*Makes a new Id using the number.*/
/*
Arguments:
=======
no: the number identifying the Id
Results:
=====
Id: the new Id
*/
makeId : Nat → Id
makeId(no) ≡ mk_Id("pa_" ⌢ natToText(no)),

/*Translates Nat into Text.*/
/*
Arguments:
```

```
=======
no: the Nat
Results:
=====
Text: the corresponding Text
*/
natToText : Nat → Text
natToText(no) ≡
   case no of
      0 → "0",
      1 → "1",
      2 → "2",
      3 → "3",
      4 → "4",
      5 → "5",
      6 → "6",
      7 → "7",
      8 → "8",
      9 → "9"
   end

/***** Id end *****/

end
```

Appendix D

Specification of Transformer in RSL_1

This appendix contains the specifications of the project written within RSL_1.

D.1 Formal Specification of the RSL AST

Specification D.1 – RSLAst_Module.rsl

```
scheme RSLAst_Module =
  class
    type
      RSLAst :: libmodule : LibModule,
      LibModule ::
        context_list : Id*   schemedef : SchemeDef,
      SchemeDef :: id : Id   class_expr : ClassExpr,
      ClassExpr ==
        BasicClassExpr(declaration_list : Decl*) |
        ExtendingClassExpr(
            base_class : ClassExpr,
            extension_class : ClassExpr) |
        SchemeInstantiation(id : Id),
      Decl ==
        TypeDecl(type_def_list : TypeDef*) |
        ValueDecl(value_def_list : ValueDef*) |
        VariableDecl(variable_def_list : VariableDef*) |
        TestDecl(test_def_list : TestDef*),

      /*Type Definitions*/
```

```
TypeDef ==
   SortDef(sd_id : Id) |
   VariantDef(id : Id, variant_list : Variant*) |
   UnionDef(
       ud_id : Id,
       name_or_wildcard_list : NameOrWildcard*) |
   ShortRecordDef(
       srd_id : Id,
       component_kind_string : ComponentKind*) |
   AbbreviationDef(abbr_id : Id, type_expr : TypeExpr),
Variant ==
   Make_Constructor(constructor : Constructor) |
   RecordVariant(
       record_constructor : Constructor,
       component_kind_list : ComponentKind*),
Constructor :: id : Id,
ComponentKind ::
   optional_destructor : OptionalDestructor
   type_expr : TypeExpr
   optional_reconstructor : OptionalReconstructor,
OptionalDestructor ==
   Destructor(id : Id) | NoDestructor,
OptionalReconstructor ==
   Reconstructor(id : Id) | NoReconstructor,
NameOrWildcard ==
   Name(var : ValueOrVariableName) | Wildcard,

/*Value Definitions*/
ValueDef ==
   ExplicitValueDef(
       single_typing : SingleTyping,
       value_expr : ValueExpr) |
   ExplicitFunctionDef(
       fun_single_typing : SingleTyping,
       formal_function_application :
           FormalFunctionApplication,
       fun_value_expr : ValueExpr,
       pre_cond : OptionalPreCondition),
OptionalPreCondition ==
   PreCondition(cond : ValueExpr) | NoPreCondition,
Typing ==
   Make_SingleTyping(single_typing : SingleTyping) |
   Make_MultipleTyping(multiple_typing : MultipleTyping),
SingleTyping ::
```

```
   binding : Binding   type_expr : TypeExpr,
MultipleTyping ::
   binding_list : Binding*   type_expr : TypeExpr,
FormalFunctionApplication ==
   IdApplication(
       id : Id,
       formal_function_parameter :
          FormalFunctionParameter),
FormalFunctionParameter :: binding_list : Binding*,
Binding ==
   IdBinding(id : Id) |
   Make_ProductBinding(prod_binding : ProductBinding),
ProductBinding :: binding_list : Binding*,

/*Variable Definitions*/
VariableDef ==
   SingleVariableDef(
       id : Id,
       type_expr : TypeExpr,
       optional_initialisation : OptionalInitialisation) |
   MultipleVariableDef(
       id_list : Id*, m_type_expr : TypeExpr),
OptionalInitialisation ==
   Initialisation(value_expr : ValueExpr) |
   NoInitialisation,

/*Test Definitions*/
TestDef == TestCase(id : Id, value_expr : ValueExpr),

/*Type Expressions*/
TypeExpr ==
   TypeLiteral(type_literal : TypeLiterals) |
   TypeName(id : Id) |
   TypeExprProduct(component_list : TypeExpr*) |
   TypeExprSet(type_expr_set : TypeExprSets) |
   TypeExprList(type_expr_list : TypeExprLists) |
   TypeExprMap(type_expr_map : TypeExprMaps) |
   FunctionTypeExpr(
       type_expr_argument : TypeExpr,
       function_arrow : FunctionArrow,
       type_expr_result : ResultDesc) |
   SubtypeExpr(
       single_typing : SingleTyping,
       restriction : ValueExpr) |
```

```
    BracketedTypeExpr(b_type_expr : TypeExpr),
TypeLiterals ==
    UNIT | INT | NAT | REAL | BOOL | CHAR | TEXT,
TypeExprSets ==
    FiniteSetTypeExpr(type_expr : TypeExpr) |
    InfiniteSetTypeExpr(i_type_expr : TypeExpr),
TypeExprLists ==
    FiniteListTypeExpr(type_expr : TypeExpr) |
    InfiniteListTypeExpr(i_type_expr : TypeExpr),
TypeExprMaps ==
    FiniteMapTypeExpr(
        type_expr_dom : TypeExpr,
        type_expr_range : TypeExpr) |
    InfiniteMapTypeExpr(
        i_type_expr_dom : TypeExpr,
        i_type_expr_range : TypeExpr),
FunctionArrow ==
    TOTAL_FUNCTION_ARROW | PARTIAL_FUNCTION_ARROW,
ResultDesc ::
    read_access_desc : OptionalReadAccessDesc
    write_access_desc : OptionalWriteAccessDesc
    type_expr : TypeExpr,
OptionalReadAccessDesc ==
    ReadAccessDesc(access_list : Access*) |
    NoReadAccessMode,
OptionalWriteAccessDesc ==
    WriteAccessDesc(access_list : Access*) |
    NoWriteAccessMode,
Access ==
    AccessValueOrVariableName(
        variable_name : ValueOrVariableName),

/*Value Expression*/
ValueExpr ==
    Make_ValueLiteral(value_literal : ValueLiteral) |
    Make_ValueOrVariableName(
        value_or_variable_name : ValueOrVariableName) |
    Make_BasicExpr(basic_expr : BasicExpr) |
    ProductExpr(value_expr_list : ValueExpr*) |
    Make_SetExpr(set_expr : SetExpr) |
    Make_ListExpr(list_expr : ListExpr) |
    Make_MapExpr(map_expr : MapExpr) |
    ApplicationExpr(
        value_expr : ValueExpr,
```

```
        appl_value_expr_list : ValueExpr*) |
    DisambiguationExpr(
        dis_value_expr : ValueExpr,
        dis_type_expr : TypeExpr) |
    BracketedExpr(bracketed_expr : ValueExpr) |
    ValueInfixExpr(
        left : ValueExpr,
        op : InfixOperator,
        right : ValueExpr) |
    ValuePrefixExpr(
        op : PrefixOperator, operand : ValueExpr) |
    AssignExpr(id : Id, assign_value_expr : ValueExpr) |
    SequencingExpr(first : ValueExpr, second : ValueExpr) |
    LetExpr(
        let_def_list : LetDef*,
        let_value_expr : ValueExpr) |
    Make_IfExpr(if_expr : IfExpr) |
    CaseExpr(
        condition : ValueExpr,
        case_branch_list : CaseBranch*),
ValueLiteral ==
    ValueLiteralInteger(getTextInteger : Text) |
    ValueLiteralReal(getTextReal : Text) |
    ValueLiteralBool(getTextBool : Text) |
    ValueLiteralChar(getTextChar : Text) |
    ValueLiteralText(getTextText : Text),
ValueOrVariableName :: id : Id,
BasicExpr == CHAOS,
SetExpr ==
    RangedSetExpr(first : ValueExpr, second : ValueExpr) |
    EnumeratedSetExpr(
        value_expr_list : OptionalValueExprList) |
    ComprehendedSetExpr(
        value_expr : ValueExpr,
        typing_list : Typing*,
        restriction : OptionalRestriction),
ListExpr ==
    RangedListExpr(first : ValueExpr, second : ValueExpr) |
    EnumeratedListExpr(
        value_expr_list : OptionalValueExprList) |
    ComprehendedListExpr(
        value_expr : ValueExpr,
        binding : Binding,
        in_value_expr : ValueExpr,
```

```
            restriction : OptionalRestriction),
    MapExpr ==
        EnumeratedMapExpr(
            value_expr_pair_list : OptionalValueExprPairList) |
        ComprehendedMapExpr(
            value_expr_pair : ValueExprPair,
            typing_list : Typing*,
            restriction : OptionalRestriction),
    OptionalValueExprList ==
        ValueExprList(value_expr_list : ValueExpr*) |
        NoValueExprList,
    OptionalRestriction ==
        Restriction(value_expr : ValueExpr) | NoRestriction,
    ValueExprPair ::
        first : ValueExpr   second : ValueExpr,
    OptionalValueExprPairList ==
        ValueExprPairList(pair_list : ValueExprPair*) |
        NoValueExprPairList,
    InfixOperator ==
        PLUS |
        MINUS |
        EQUAL |
        NOTEQUAL |
        LT |
        GT |
        LTE |
        GTE |
        HAT |
        STAR |
        SLASH |
        BACKSLASH |
        EXP |
        SUBSET |
        PROPSUBSET |
        REVSUBSET |
        REVPROPSUBSET |
        MEMBER |
        NOTMEMBER |
        UNION |
        INTER |
        OVERRIDE |
        COMPOSITION |
        IMPLIES |
        OR |
```

```
    AND,
PrefixOperator ==
    ABS |
    INTCAST |
    REALCAST |
    LEN |
    INDS |
    ELEMS |
    HD |
    TL |
    NOT |
    PREFIXMINUS |
    PREFIXPLUS |
    CARD |
    DOM |
    RNG,
LetDef ::
    binding : LetBinding   value_expr : ValueExpr,
LetBinding ==
    MakeBinding(binding : Binding) |
    MakeRecordPattern(
        value_or_variable_name : ValueOrVariableName,
        inner_pattern_list : Pattern*) |
    MakeListPatternLet(list_pattern : ListPattern),
ListPattern ==
    Make_EnumeratedListPattern(
        enum_list_pattern : EnumeratedListPattern) |
    ConcatenatedListPattern(
        c_enum_list_pattern : EnumeratedListPattern,
        c_inner_pattern : Pattern),
EnumeratedListPattern ::
    inner_pattern : OptionalInnerPattern,
OptionalInnerPattern ==
    InnerPatternList(pattern_list : Pattern*) |
    NoInnerPattern,
Pattern ==
    ValueLiteralPattern(value_literal : ValueLiteral) |
    NamePattern(id : Id, value_initializer : OptionalId) |
    WildcardPattern |
    ProductPattern(p_inner_pattern_list : Pattern*) |
    RecordPattern(
        value_or_variable_name : ValueOrVariableName,
        inner_pattern_list : Pattern*) |
    MakeListPattern(list_pattern : ListPattern),
```

```
    IfExpr ::
        condition : ValueExpr
        if_case : ValueExpr
        elsif_list : Elsif*
        else_case : ValueExpr,
    Elsif ::
        condition : ValueExpr   elsif_case : ValueExpr,
    CaseBranch ::
        pattern : Pattern   value_expr : ValueExpr,

    /*Common*/
    OptionalId == Make_Id(id : Id) | NoOptionalId,
    Id :: getText : Text,
    Trans :: type_id : Id   var_id : Id
end
```

D.2 Formal Specification of the Transformer in RSL_1

Specification D.2 – TransformerRSL1.rsl

```
scheme TransformerRSL1 =
  extend RSLAst_Module with
  class
    type
      TRANSMapEntrance :: first : Id   second : Id,
      TRANS :: map : TRANSMapEntrance*,
      TYPINGSMapEntrance :: type_id : Id   id_list : Id*,
      TYPINGS :: map : TYPINGSMapEntrance*,
      FUNCMapEntrance :: first : Id   second : FuncSpec,
      FuncSpec ::
        type_expr : TypeExpr
        value_expr : ValueExpr
        read_list : Access*
        write_list : Access*,
      FUNC :: map : FUNCMapEntrance*,
      ENVMapEntrance :: first : Id   second : Binding,
      ENV :: map : ENVMapEntrance*,
      TYPESMapEntrance :: first : Binding   second : ExpType,
      TYPES :: map : TYPESMapEntrance*,
      ExpType == Known(type_expr : TypeExpr) | Unknown,
      AccessResult ::
        accessList : Access*   accessIdSet : Id*,
      DResult ==
        Transformable(result : Decl) | Not_transformable,
      TRResult ==
        RSLast_transformable(result : RSLAst) |
        RSLast_not_transformable,
      ClassResult ==
        Class_transformable(result : ClassExpr) |
        Class_not_transformable,
      DLResult ==
        DL_transformable(result : Decl*) |
        DL_not_transformable,
      DeclResult ==
        Decl_transformable(result : Decl) |
        Decl_not_transformable,
      MapType :: tedom : ExpType   terange : ExpType,
      BINDING_VE ::
```

bindingList : Binding*
valueExprList : ValueExpr*,
BOOL_FFA_ENV_TYPES_LDL ::
bool : **Bool**
formalfa : FormalFunctionApplication
envMap : ENV
typesMap : TYPES
letdefList : LetDef*,
BOOL_ENV_TYPES ::
bool : **Bool** envMap : ENV typesMap : TYPES,
BOOL_ENV :: bool : **Bool** envMap : ENV,
PRECOND_TYPES ::
preCond : OptionalPreCondition typesMap : TYPES,
VE_TYPES :: valueExpr : ValueExpr typesMap : TYPES,
VEL_TYPES ::
valueExprList : ValueExpr* typesMap : TYPES,
VEPL_TYPES ::
valueExprPairList : ValueExprPair*
typesMap : TYPES,
BOOL_FFP_ENV_TYPES_LDL ::
bool : **Bool**
formalfp : FormalFunctionParameter
envMap : ENV
typesMap : TYPES
letdefList : LetDef*,
LDL_ENV :: letdefList : LetDef* envMap : ENV,
BINDINGLIST_VEL ::
bindingList : Binding*
valueExprList : ValueExpr*,
LDL_TYPES ::
letdefList : LetDef* typesMap : TYPES,
LD_TYPES :: letdef : LetDef typesMap : TYPES,
EIL_TYPES :: elsIfList : Elsif* typesMap : TYPES,
CBL_TYPES ::
caseBranchList : CaseBranch* typesMap : TYPES,
PATTERN_TYPES :: pattern : Pattern typesMap : TYPES,
PL_TYPES ::
patternList : Pattern* typesMap : TYPES,
LP_TYPES ::
listPattern : ListPattern typesMap : TYPES,
OIP_TYPES ::
optInnerPattern : OptionalInnerPattern
typesMap : TYPES,
SE_TYPES :: setExpr : SetExpr typesMap : TYPES,

```
    OVEL_TYPES ::
       optValueExprList : OptionalValueExprList
       typesMap : TYPES,
    OVEPL_TYPES ::
       optValueExprPairList : OptionalValueExprPairList
       typesMap : TYPES,
    OR_TYPES ::
       optRestriction : OptionalRestriction
       typesMap : TYPES,
    LE_TYPES :: listExpr : ListExpr   typesMap : TYPES,
    ME_TYPES :: mapExpr : MapExpr   typesMap : TYPES

value
    TRRSLAst : RSLAst × Text × TRANS → TRResult
    TRRSLAst(rslast, schemeid, trans) ≡
       case
          TRClassExpr(
              class_expr(schemedef(libmodule(rslast))), trans)
       of
          Class_transformable(ce) →
             RSLast_transformable(
                 mk_RSLAst(
                    mk_LibModule(
                       context_list(libmodule(rslast)),
                       mk_SchemeDef(mk_Id(schemeid), ce)))),
          Class_not_transformable → RSLast_not_transformable
       end,

    TRClassExpr : ClassExpr × TRANS → ClassResult
    TRClassExpr(ce, trans) ≡
       case ce of
          BasicClassExpr(dl) →
             case
                TRDeclList(
                    dl,
                    expandTYPINGSMap(
                       getIds(getTypeDecl(dl)),
                       makeTYPINGSMap(getTypeDecl(dl)), trans),
                    establishFuncMap(dl, trans), trans, ⟨⟩)
             of
                DL_transformable(dlres) →
                   Class_transformable(
                      BasicClassExpr(
                         makeVariables(dlres, trans))),
```

```
                DL_not_transformable → Class_not_transformable
            end,
        _ → Class_not_transformable
    end,

TRDeclList :
    Decl* × TYPINGS × FUNC × TRANS × Decl* →
        DLResult
TRDeclList(dl, typings, func, trans, dlres) ≡
    if dl = ⟨⟩ then DL_transformable(dlres)
    else
        case TRDecl(hd dl, typings, func, trans) of
            Decl_transformable(decl) →
                TRDeclList(
                    tl dl, typings, func, trans,
                    dlres ⌢ ⟨decl⟩),
            Decl_not_transformable → DL_not_transformable
        end
    end,

TRDecl :
    Decl × TYPINGS × FUNC × TRANS → DeclResult
TRDecl(d, typings, func, trans) ≡
    case d of
        TypeDecl(tdl) →
            case TRTypeDecl(tdl, typings, func, trans) of
                Transformable(decl) →
                    Decl_transformable(decl),
                Not_transformable → Decl_not_transformable
            end,
        ValueDecl(vd) →
            case TRValueDecl(vd, typings, func, trans) of
                Transformable(decl) →
                    Decl_transformable(decl),
                Not_transformable → Decl_not_transformable
            end,
        _ → Decl_not_transformable
    end,

makeVariables : Decl* × TRANS → Decl*
makeVariables(decl, trans) ≡
    if decl = ⟨⟩
    then ⟨VariableDecl(makeVariableDeclList(trans))⟩
    else
```

```
    case hd decl of
      TypeDecl(tdl) →
        ⟨hd decl⟩ ⌢ makeVariables(tl decl, trans),
      ValueDecl(vdl) →
        ⟨VariableDecl(makeVariableDeclList(trans))⟩ ⌢
        decl
    end
  end,

makeVariableDeclList : TRANS → VariableDef*
makeVariableDeclList(trans) ≡
  if getTRANSMapEntranceList(trans) = ⟨⟩ then ⟨⟩
  else
    ⟨makeVariableDef(
          first(getHeadTRANS(trans)), trans)⟩ ⌢
    makeVariableDeclList(
       removeTRANS(trans, first(getHeadTRANS(trans))))
  end,

makeVariableDef : Id × TRANS → VariableDef
makeVariableDef(typename, trans) ≡
  SingleVariableDef(
     getMapValueTRANS(trans, typename),
     TypeName(typename), NoInitialisation),

TRTypeDecl :
  TypeDef* × TYPINGS × FUNC × TRANS → DResult
TRTypeDecl(tdl, typings, func, trans) ≡
  if CheckTypeDefList(tdl, typings, func, trans)
  then Transformable(TypeDecl(tdl))
  else Not_transformable
  end,

CheckTypeDefList :
  TypeDef* × TYPINGS × FUNC × TRANS → Bool
CheckTypeDefList(tdl, typings, func, trans) ≡
  if tdl = ⟨⟩ then true
  else
    and(
       CheckTypeDef(hd tdl, typings, func, trans),
       CheckTypeDefList(tl tdl, typings, func, trans))
  end,

CheckTypeDef :
```

```
   TypeDef × TYPINGS × FUNC × TRANS → Bool
CheckTypeDef(td, typings, func, trans) ≡
   case td of
      SortDef(id) → not(isinId(id, domainTRANS(trans))),
      VariantDef(id, vl) →
         and(
            CheckVariantList(id, vl, typings, func, trans),
            notInTYPINGS(id, typings, trans)),
      ShortRecordDef(id, cl) →
         and(
            CheckComponentKindList(
                  id, cl, typings, func, trans),
            notInTYPINGS(id, typings, trans)),
      AbbreviationDef(id, te) →
         and(
            CheckTypeExpr(te, typings, func, trans),
            notInTYPINGS(id, typings, trans))
   end,

CheckVariantList :
   Id × Variant* × TYPINGS × FUNC × TRANS →
      Bool
CheckVariantList(id, vl, typings, func, trans) ≡
   if vl = ⟨⟩ then true
   else
      and(
         CheckVariant(id, hd vl, typings, func, trans),
         CheckVariantList(
               id, tl vl, typings, func, trans))
   end,

CheckVariant :
   Id × Variant × TYPINGS × FUNC × TRANS → Bool
CheckVariant(id, v, typings, func, trans) ≡
   case v of
      RecordVariant(c, cl) →
         CheckComponentKindList(
            id, cl, typings, func, trans),
      _ → true
   end,

CheckComponentKindList :
   Id × ComponentKind* × TYPINGS × FUNC × TRANS →
      Bool
```

```
CheckComponentKindList(id, cl, typings, func, trans) ≡
   if cl = ⟨⟩ then true
   else
      and(
         CheckComponentKind(
               id, hd cl, typings, func, trans),
         CheckComponentKindList(
               id, tl cl, typings, func, trans))
   end,

CheckComponentKind :
   Id × ComponentKind × TYPINGS × FUNC × TRANS →
      Bool
CheckComponentKind(id, c, typings, func, trans) ≡
   CheckTypeExpr(type_expr(c), typings, func, trans),

CheckTypeExpr :
   TypeExpr × TYPINGS × FUNC × TRANS → Bool
CheckTypeExpr(te, typings, func, trans) ≡
   case te of
      TypeLiteral(tn) → true,
      TypeName(tn) → true,
      TypeExprProduct(tep) →
         CheckTypeExprList(tep, typings, func, trans),
      TypeExprSet(tes) →
         case tes of
            FiniteSetTypeExpr(fse) →
               not(
                  containsTRANSName(
                        Known(fse), typings,
                        domainTRANS(trans))),
            InfiniteSetTypeExpr(ise) →
               not(
                  containsTRANSName(
                        Known(ise), typings,
                        domainTRANS(trans)))
         end,
      TypeExprList(les) →
         case les of
            FiniteListTypeExpr(fle) →
               not(
                  containsTRANSName(
                        Known(fle), typings,
                        domainTRANS(trans))),
```

```
            InfiniteListTypeExpr(ile) →
               not(
                  containsTRANSName(
                        Known(ile), typings,
                        domainTRANS(trans)))
         end,
      TypeExprMap(tem) →
         case tem of
            FiniteMapTypeExpr(tedom, terange) →
               not(
                  or(containsTRANSName(
                           Known(tedom), typings,
                           domainTRANS(trans)),
                        containsTRANSName(
                           Known(terange), typings,
                           domainTRANS(trans)))),
            InfiniteMapTypeExpr(tedom, terange) →
               not(
                  or(containsTRANSName(
                           Known(tedom), typings,
                           domainTRANS(trans)),
                        containsTRANSName(
                           Known(terange), typings,
                           domainTRANS(trans))))
         end,
      FunctionTypeExpr(arg, fa, res) → false,
      SubtypeExpr(st, ve) →
         not(
            and(
               containsTRANSName(
                  Known(type_expr(st)), typings,
                  domainTRANS(trans)),
               bool(
                  CheckValueExpr(
                     ve, Known(TypeLiteral(BOOL)),
                     typings, func, trans, mk_ENV(⟨⟩),
                     mk_TYPES(⟨⟩)))))),
      BracketedTypeExpr(bte) →
         CheckTypeExpr(bte, typings, func, trans)
   end,

CheckTypeExprList :
   TypeExpr* × TYPINGS × FUNC × TRANS → Bool
CheckTypeExprList(tel, typings, func, trans) ≡
```

```
  if tel = ⟨⟩ then true
  else
    and(
      CheckTypeExpr(hd tel, typings, func, trans),
      CheckTypeExprList(tl tel, typings, func, trans))
  end,

TRValueDecl :
  ValueDef* × TYPINGS × FUNC × TRANS → DResult
TRValueDecl(vdl, typings, func, trans) ≡
  if CheckValueDefList(vdl, typings, func, trans)
  then
    /*The value declaration list is transformable
    and is transformed.*/
    Transformable(
      ValueDecl(TRValueDefList(vdl, func, trans)))
  else
    /*The value declaration list is not transformable.
    */
    Not_transformable
  end,

CheckValueDefList :
  ValueDef* × TYPINGS × FUNC × TRANS → Bool
CheckValueDefList(vdl, typings, func, trans) ≡
  if vdl = ⟨⟩ then true
  else
    and(
      CheckValueDef(hd vdl, typings, func, trans),
      CheckValueDefList(tl vdl, typings, func, trans))
  end,

CheckValueDef :
  ValueDef × TYPINGS × FUNC × TRANS → Bool
CheckValueDef(vd, typings, func, trans) ≡
  case vd of
    ExplicitValueDef(st, ve) →
      if
        containsTRANSName(
          Known(type_expr(st)), typings,
          domainTRANS(trans))
      then
        case binding(st) of
          IdBinding(id) →
```

```
            bool(
               CheckValueExpr(
                  ve, Known(type_expr(st)), typings,
                  func, trans, mk_ENV(⟨⟩),
                  mk_TYPES(⟨⟩))),
          Make_ProductBinding(pb) → false
        end
     else true
     end,
  ExplicitFunctionDef(st, ffa, ve, precond) →
     case type_expr(st) of
        FunctionTypeExpr(arg, fa, res) →
          /∗Establishes ENV and TYPES.∗/
          and(
             and(
                and(
                   and(
                     bool(
                        TRFormalFuncAppl(
                          ffa, func, trans)),
                     bool(
                        CheckOptPreCondition(
                          precond,
                          Known(TypeLiteral(BOOL)),
                          typings, func, trans,
                          envMap(
                             TRFormalFuncAppl(
                               ffa, func, trans)),
                          typesMap(
                             TRFormalFuncAppl(
                               ffa, func, trans))))),
                   bool(
                     CheckValueExpr(
                        ve, Known(type_expr(res)),
                        typings, func, trans,
                        envMap(
                          TRFormalFuncAppl(
                             ffa, func, trans)),
                        typesMap(
                          CheckOptPreCondition(
                             precond,
                             Known(TypeLiteral(BOOL)),
                             typings, func, trans,
                             envMap(
```

```
                                    TRFormalFuncAppl(
                                      ffa, func, trans)),
                                  typesMap(
                                    TRFormalFuncAppl(
                                      ffa, func, trans))
                                )))))),
                          CheckTypeExpr(
                              arg, typings, func, trans)),
                      CheckTypeExpr(
                          type_expr(res), typings, func, trans))
        end
    end,

CheckOptPreCondition :
    OptionalPreCondition × ExpType × TYPINGS ×
    FUNC × TRANS × ENV × TYPES →
        BOOL_ENV_TYPES
CheckOptPreCondition(
     precond, et, typings, func, trans, env, types) ≡
    case precond of
        PreCondition(ve) →
            mk_BOOL_ENV_TYPES(
                and(
                      bool(
                          CheckValueExpr(
                                ve, et, typings, func, trans, env,
                                types)),
                      CheckPreCondGen(ve, func, trans, types)),
                envMap(
                      CheckValueExpr(
                          ve, et, typings, func, trans, env, types
                      )),
                typesMap(
                      CheckValueExpr(
                          ve, et, typings, func, trans, env, types
                      ))),
        NoPreCondition →
            mk_BOOL_ENV_TYPES(true, env, types)
    end,

CheckPreCondGen :
    ValueExpr × FUNC × TRANS × TYPES → Bool
CheckPreCondGen(ve, func, trans, types) ≡
    case ve of
```

```
    Make_ValueLiteral(vl) → true,
    Make_ValueOrVariableName(vn) → true,
    Make_BasicExpr(be) → true,
    ProductExpr(vel) →
      CheckPreCondGenProduct(vel, func, trans, types),
    Make_SetExpr(setExpr) →
      CheckPreCondGenSet(setExpr, func, trans, types),
    Make_ListExpr(listExpr) →
      CheckPreCondGenList(listExpr, func, trans, types),
    Make_MapExpr(mapExpr) →
      CheckPreCondGenMap(mapExpr, func, trans, types),
    ApplicationExpr(ave, vl) →
      CheckPreCondApplicationExpr(
          ave, vl, func, trans, types),
    BracketedExpr(bve) →
      CheckPreCondGen(bve, func, trans, types),
    ValueInfixExpr(first, op, second) →
      and(
          CheckPreCondGen(first, func, trans, types),
          CheckPreCondGen(second, func, trans, types)),
    ValuePrefixExpr(op, operand) →
      CheckPreCondGen(operand, func, trans, types),
    LetExpr(ldl, lve) →
      and(
          CheckPreCondGenLetDef(ldl, func, trans, types),
          CheckPreCondGen(lve, func, trans, types)),
    Make_IfExpr(ie) →
      and(
          and(
              and(
                  CheckPreCondGen(
                      condition(ie), func, trans, types),
                  CheckPreCondGen(
                      if_case(ie), func, trans, types)),
              CheckPreCondGenElsif(
                  elsif_list(ie), func, trans, types)),
          CheckPreCondGen(
              else_case(ie), func, trans, types)),
    CaseExpr(cond, cbl) →
      and(
          CheckPreCondGen(cond, func, trans, types),
          CheckPreCondGenCaseBranch(
              cbl, func, trans, types))
end,
```

```
CheckPreCondGenProduct :
   ValueExpr* × FUNC × TRANS × TYPES → Bool
CheckPreCondGenProduct(vel, func, trans, types) ≡
   if vel = ⟨⟩ then true
   else
      and(
          CheckPreCondGen(hd vel, func, trans, types),
          CheckPreCondGenProduct(
               tl vel, func, trans, types))
   end,

CheckPreCondGenSet :
   SetExpr × FUNC × TRANS × TYPES → Bool
CheckPreCondGenSet(se, func, trans, types) ≡
   case se of
      RangedSetExpr(first, second) →
         and(
             CheckPreCondGen(first, func, trans, types),
             CheckPreCondGen(second, func, trans, types)),
      EnumeratedSetExpr(ovel) →
         case ovel of
            ValueExprList(vel) →
               CheckPreCondGenProduct(
                   vel, func, trans, types),
            NoValueExprList → true
         end,
      ComprehendedSetExpr(ve, t, or) →
         and(
             CheckPreCondGen(ve, func, trans, types),
             CheckPreCondOptRestriction(
                  or, func, trans, types))
   end,

CheckPreCondGenList :
   ListExpr × FUNC × TRANS × TYPES → Bool
CheckPreCondGenList(le, func, trans, types) ≡
   case le of
      RangedListExpr(first, second) →
         and(
             CheckPreCondGen(first, func, trans, types),
             CheckPreCondGen(second, func, trans, types)),
      EnumeratedListExpr(ovel) →
         case ovel of
```

```
              ValueExprList(vel) →
                 CheckPreCondGenProduct(
                    vel, func, trans, types),
              NoValueExprList → true
            end,
         ComprehendedListExpr(ve, b, vei, or) →
            and(
              and(
                   CheckPreCondGen(ve, func, trans, types),
                   CheckPreCondGen(vei, func, trans, types)),
              CheckPreCondOptRestriction(
                   or, func, trans, types))
      end,

   CheckPreCondGenMap :
      MapExpr × FUNC × TRANS × TYPES → Bool
   CheckPreCondGenMap(me, func, trans, types) ≡
      case me of
         EnumeratedMapExpr(ovel) →
            case ovel of
              ValueExprPairList(vel) →
                 CheckPreCondPairList(vel, func, trans, types),
              NoValueExprPairList → true
            end,
         ComprehendedMapExpr(vep, t, or) →
            and(
              and(
                   CheckPreCondGen(
                      first(vep), func, trans, types),
                   CheckPreCondGen(
                      second(vep), func, trans, types)),
              CheckPreCondOptRestriction(
                   or, func, trans, types))
      end,

   CheckPreCondOptRestriction :
      OptionalRestriction × FUNC × TRANS × TYPES → Bool
   CheckPreCondOptRestriction(opr, func, trans, types) ≡
      case opr of
         Restriction(ve) →
            CheckPreCondGen(ve, func, trans, types),
         NoRestriction → true
      end,
```

```
CheckPreCondPairList :
   ValueExprPair* × FUNC × TRANS × TYPES → Bool
CheckPreCondPairList(vepl, func, trans, types) ≡
   if vepl = ⟨⟩ then true
   else
      and(
         and(
               CheckPreCondGen(
                  first(hd vepl), func, trans, types),
               CheckPreCondGen(
                  second(hd vepl), func, trans, types)),
         CheckPreCondPairList(
               tl vepl, func, trans, types))
   end,

CheckPreCondApplicationExpr :
   ValueExpr × ValueExpr* × FUNC × TRANS × TYPES →
      Bool
CheckPreCondApplicationExpr(
    ve, vel, func, trans, types) ≡
   if CheckPreCondGenProduct(vel, func, trans, types)
   then
      case ve of
         Make_ValueOrVariableName(vn) →
            if isinId(id(vn), domainFUNC(func))
            then
               if
                  not(
                      lengthAccess(
                           write_list(
                              getMapValueFUNC(func, id(vn)))) =
                        0)
               then false
               else
                  case
                     type_expr(getMapValueFUNC(func, id(vn)))
                  of
                     FunctionTypeExpr(arg, fa, res) →
                        or(not(
                              containsGen(
                                 vel,
                                 typeExprToExpTypeList(arg),
                                 trans, func)),
                           onlyTOIArgument(vel, trans, types))
```

```
                    end
                  end
                else true
                end
            end
          else false
          end,

        CheckPreCondGenLetDef :
          LetDef* × FUNC × TRANS × TYPES → Bool
        CheckPreCondGenLetDef(ldl, func, trans, types) ≡
          if ldl = ⟨⟩ then true
          else
            and(
                CheckPreCondGen(
                    value_expr(hd ldl), func, trans, types),
                CheckPreCondGenLetDef(
                    tl ldl, func, trans, types))
          end,

        CheckPreCondGenElsif :
          Elsif* × FUNC × TRANS × TYPES → Bool
        CheckPreCondGenElsif(eil, func, trans, types) ≡
          if eil = ⟨⟩ then true
          else
            and(
                and(
                    CheckPreCondGen(
                        condition(hd eil), func, trans, types),
                    CheckPreCondGen(
                        elsif_case(hd eil), func, trans, types)),
                CheckPreCondGenElsif(tl eil, func, trans, types)
            )
          end,

        CheckPreCondGenCaseBranch :
          CaseBranch* × FUNC × TRANS × TYPES → Bool
        CheckPreCondGenCaseBranch(cbl, func, trans, types) ≡
          if cbl = ⟨⟩ then true
          else
            and(
                CheckPreCondGen(
                    value_expr(hd cbl), func, trans, types),
                CheckPreCondGenCaseBranch(
```

```
                tl cbl, func, trans, types))
    end,

  CheckValueExpr :
    ValueExpr × ExpType × TYPINGS × FUNC × TRANS ×
    ENV × TYPES →
      BOOL_ENV_TYPES
  CheckValueExpr(
     ve, et, typings, func, trans, env, types) ≡
    case ve of
      Make_ValueLiteral(vl) →
        mk_BOOL_ENV_TYPES(
           true, updateENV(env, et, trans), types),
      Make_ValueOrVariableName(vn) →
        if
          not(
             isinBinding(
                 IdBinding(id(vn)), domainTYPES(types)))
        then
          mk_BOOL_ENV_TYPES(
             true, updateENV(env, et, trans), types)
        else
          if
            checkTRANS(
               getMapValueTYPES(types, IdBinding(id(vn))),
               domainTRANS(trans))
          then
            /∗The value expression is of the type of interest.
            ∗/
            mk_BOOL_ENV_TYPES(
               isinBinding(
                   IdBinding(id(vn)), rangeENV(env)),
               updateENV(env, et, trans), types)
          else
            /∗The value expression is not of the type of interest.
            ∗/
            mk_BOOL_ENV_TYPES(
               true, updateENV(env, et, trans), types)
          end
        end,
      Make_BasicExpr(be) →
        mk_BOOL_ENV_TYPES(true, env, types),
      ProductExpr(vel) →
        CheckValueExprList(
```

```
        vel,
        expTypeToExpTypeList(
            removeBrackets(et), lengthVE(vel)),
        typings, func, trans, env, types),
Make_SetExpr(setExpr) →
  if
    containsTRANSName(
        getSetType(et), typings, domainTRANS(trans))
  then mk_BOOL_ENV_TYPES(false, env, types)
  else
    CheckSetExpr(
        setExpr, getSetType(et), typings, func,
        trans, env, types)
  end,
Make_ListExpr(listExpr) →
  if
    containsTRANSName(
        getListType(et), typings,
        domainTRANS(trans))
  then mk_BOOL_ENV_TYPES(false, env, types)
  else
    CheckListExpr(
        listExpr, getListType(et), typings, func,
        trans, env, types)
  end,
Make_MapExpr(mapExpr) →
  if
    or(containsTRANSName(
            tedom(getMapType(et)), typings,
            domainTRANS(trans)),
        containsTRANSName(
            terange(getMapType(et)), typings,
            domainTRANS(trans)))
  then mk_BOOL_ENV_TYPES(false, env, types)
  else
    CheckMapExpr(
        mapExpr, getMapType(et), typings, func,
        trans, env, types)
  end,
ApplicationExpr(ave, vl) →
  CheckApplicationExpr(
      ave, vl, et, typings, func, trans, env, types),
BracketedExpr(bve) →
  mk_BOOL_ENV_TYPES(
```

```
            bool(
                CheckValueExpr(
                    bve, Unknown, typings, func, trans,
                    env, types)),
            updateENV(
                envMap(
                    CheckValueExpr(
                        bve, Unknown, typings, func, trans,
                        env, types)), et, trans),
            typesMap(
                CheckValueExpr(
                    bve, Unknown, typings, func, trans,
                    env, types))),
    ValueInfixExpr(first, op, second) →
        mk_BOOL_ENV_TYPES(
            bool(
                CheckValueExprList(
                    ⟨first, second⟩,
                    ⟨Unknown, Unknown⟩, typings, func,
                    trans, env, types)),
            updateENV(
                envMap(
                    CheckValueExprList(
                        ⟨first, second⟩,
                        ⟨Unknown, Unknown⟩, typings, func,
                        trans, env, types)), et, trans),
            typesMap(
                CheckValueExprList(
                    ⟨first, second⟩,
                    ⟨Unknown, Unknown⟩, typings, func,
                    trans, env, types))),
    ValuePrefixExpr(op, operand) →
        mk_BOOL_ENV_TYPES(
            bool(
                CheckValueExpr(
                    operand, Unknown, typings, func, trans,
                    env, types)),
            updateENV(
                envMap(
                    CheckValueExpr(
                        operand, Unknown, typings, func,
                        trans, env, types)), et, trans),
            typesMap(
                CheckValueExpr(
```

```
                    operand, Unknown, typings, func, trans,
                    env, types))),
          LetExpr(ldl, lve) →
            mk_BOOL_ENV_TYPES(
              and(
                  bool(
                    CheckLetDefList(
                        ldl, typings, func, trans, env, types
                    )),
                  bool(
                    CheckValueExpr(
                        lve, et, typings, func, trans,
                        envMap(
                          CheckLetDefList(
                            ldl, typings, func, trans,
                            env, types)),
                        typesMap(
                          CheckLetDefList(
                            ldl, typings, func, trans,
                            env, types))))),
              envMap(
                  CheckValueExpr(
                    lve, et, typings, func, trans,
                    envMap(
                        CheckLetDefList(
                          ldl, typings, func, trans, env,
                          types)),
                    typesMap(
                        CheckLetDefList(
                          ldl, typings, func, trans, env,
                          types)))),
              typesMap(
                  CheckValueExpr(
                    lve, et, typings, func, trans,
                    envMap(
                        CheckLetDefList(
                          ldl, typings, func, trans, env,
                          types)),
                    typesMap(
                        CheckLetDefList(
                          ldl, typings, func, trans, env,
                          types))))),
          Make_IfExpr(ie) →
            CheckIfExpr(
```

```
                ie, et, typings, func, trans, env, types),
        CaseExpr(cond, cbl) →
            mk_BOOL_ENV_TYPES(
                and(
                    bool(
                        CheckValueExpr(
                            cond, Unknown, typings, func, trans,
                            env, types)),
                    bool(
                        CheckCaseBranchList(
                            cbl, et, typings, func, trans,
                            envMap(
                                CheckValueExpr(
                                    cond, Unknown, typings, func,
                                    trans, env, types)),
                            typesMap(
                                CheckValueExpr(
                                    cond, Unknown, typings, func,
                                    trans, env, types))))),
                envMap(
                    CheckCaseBranchList(
                        cbl, et, typings, func, trans,
                        envMap(
                            CheckValueExpr(
                                cond, Unknown, typings, func,
                                trans, env, types)),
                        typesMap(
                            CheckValueExpr(
                                cond, Unknown, typings, func,
                                trans, env, types)))),
                typesMap(
                    CheckCaseBranchList(
                        cbl, et, typings, func, trans,
                        envMap(
                            CheckValueExpr(
                                cond, Unknown, typings, func,
                                trans, env, types)),
                        typesMap(
                            CheckValueExpr(
                                cond, Unknown, typings, func,
                                trans, env, types)))))
    end,

CheckValueExprList :
```

```
    ValueExpr* × ExpType* × TYPINGS × FUNC ×
    TRANS × ENV × TYPES →
      BOOL_ENV_TYPES
  CheckValueExprList(
    vel, etl, typings, func, trans, env, types) ≡
    if vel = ⟨⟩
    then mk_BOOL_ENV_TYPES(true, env, types)
    else
      if
        not(
          bool(
              CheckValueExpr(
                hd vel, hd etl, typings, func, trans,
                env, types)))
      then
        mk_BOOL_ENV_TYPES(
          bool(
              CheckValueExpr(
                hd vel, hd etl, typings, func, trans,
                env, types)),
          envMap(
              CheckValueExpr(
                hd vel, hd etl, typings, func, trans,
                env, types)),
          typesMap(
              CheckValueExpr(
                hd vel, hd etl, typings, func, trans,
                env, types)))
      else
        CheckValueExprList(
          tl vel, tl etl, typings, func, trans,
          envMap(
              CheckValueExpr(
                hd vel, hd etl, typings, func, trans,
                env, types)),
          typesMap(
              CheckValueExpr(
                hd vel, hd etl, typings, func, trans,
                env, types)))
      end
    end,

  CheckSetExpr :
    SetExpr × ExpType × TYPINGS × FUNC × TRANS ×
```

```
    ENV × TYPES →
      BOOL_ENV_TYPES
  CheckSetExpr(se, et, typings, func, trans, env, types) ≡
    case se of
      RangedSetExpr(fve, sve) →
        CheckValueExprList(
          ⟨fve, sve⟩, ⟨et, et⟩, typings, func,
          trans, env, types),
      EnumeratedSetExpr(ovel) →
        CheckOptValueExprList(
          ovel, et, typings, func, trans, env, types),
      ComprehendedSetExpr(ve, typlist, or) →
        mk_BOOL_ENV_TYPES(
          and(
              bool(
                CheckValueExpr(
                    ve, et, typings, func, trans, env,
                    types)),
              bool(
                CheckOptRestriction(
                    or, Known(TypeLiteral(BOOL)),
                    typings, func, trans,
                    envMap(
                      CheckValueExpr(
                        ve, et, typings, func, trans,
                        env, types)),
                    typesMap(
                      CheckValueExpr(
                        ve, et, typings, func, trans,
                        env, types))))),
          envMap(
              CheckOptRestriction(
                or, Known(TypeLiteral(BOOL)), typings,
                func, trans,
                envMap(
                    CheckValueExpr(
                      ve, et, typings, func, trans,
                      env, types)),
                typesMap(
                    CheckValueExpr(
                      ve, et, typings, func, trans,
                      env, types)))),
          typesMap(
              CheckOptRestriction(
```

```
                        or, Known(TypeLiteral(BOOL)), typings,
                        func, trans,
                        envMap(
                            CheckValueExpr(
                              ve, et, typings, func, trans,
                              env, types)),
                        typesMap(
                            CheckValueExpr(
                              ve, et, typings, func, trans,
                              env, types)))))
    end,

  CheckListExpr :
    ListExpr × ExpType × TYPINGS × FUNC × TRANS ×
    ENV × TYPES →
      BOOL_ENV_TYPES
  CheckListExpr(le, et, typings, func, trans, env, types) ≡
    case le of
      RangedListExpr(fve, sve) →
        CheckValueExprList(
            ⟨fve, sve⟩, ⟨et, et⟩, typings, func,
            trans, env, types),
      EnumeratedListExpr(ovel) →
        CheckOptValueExprList(
            ovel, et, typings, func, trans, env, types),
      ComprehendedListExpr(ve1, b, ve2, or) →
        mk_BOOL_ENV_TYPES(
            and(
                and(
                    bool(
                        CheckValueExpr(
                          ve1, et, typings, func, trans,
                          env, types)),
                    bool(
                        CheckValueExpr(
                          ve2, et, typings, func, trans,
                          envMap(
                            CheckValueExpr(
                              ve1, et, typings, func,
                              trans, env, types)),
                          typesMap(
                            CheckValueExpr(
                              ve1, et, typings, func,
                              trans, env, types))))),
```

```
          bool(
             CheckOptRestriction(
                 or, Known(TypeLiteral(BOOL)),
                 typings, func, trans,
                 envMap(
                   CheckValueExpr(
                     ve2, et, typings, func, trans,
                     envMap(
                       CheckValueExpr(
                         ve1, et, typings, func,
                         trans, env, types)),
                     typesMap(
                       CheckValueExpr(
                         ve1, et, typings, func,
                         trans, env, types)))),
                 typesMap(
                   CheckValueExpr(
                     ve2, et, typings, func, trans,
                     envMap(
                       CheckValueExpr(
                         ve1, et, typings, func,
                         trans, env, types)),
                     typesMap(
                       CheckValueExpr(
                         ve1, et, typings, func,
                         trans, env, types))))))),
      envMap(
          CheckOptRestriction(
             or, Known(TypeLiteral(BOOL)), typings,
             func, trans,
             envMap(
                 CheckValueExpr(
                   ve2, et, typings, func, trans,
                   envMap(
                     CheckValueExpr(
                       ve1, et, typings, func,
                       trans, env, types)),
                   typesMap(
                     CheckValueExpr(
                       ve1, et, typings, func,
                       trans, env, types)))),
             typesMap(
                 CheckValueExpr(
                   ve2, et, typings, func, trans,
```

```
                                    envMap(
                                      CheckValueExpr(
                                        ve1, et, typings, func,
                                        trans, env, types)),
                                    typesMap(
                                      CheckValueExpr(
                                        ve1, et, typings, func,
                                        trans, env, types)))))),
                  typesMap(
                      CheckOptRestriction(
                        or, Known(TypeLiteral(BOOL)), typings,
                        func, trans,
                        envMap(
                            CheckValueExpr(
                              ve2, et, typings, func, trans,
                              envMap(
                                CheckValueExpr(
                                  ve1, et, typings, func,
                                  trans, env, types)),
                              typesMap(
                                CheckValueExpr(
                                  ve1, et, typings, func,
                                  trans, env, types)))),
                        typesMap(
                            CheckValueExpr(
                              ve2, et, typings, func, trans,
                              envMap(
                                CheckValueExpr(
                                  ve1, et, typings, func,
                                  trans, env, types)),
                              typesMap(
                                CheckValueExpr(
                                  ve1, et, typings, func,
                                  trans, env, types)))))))
  end,

CheckMapExpr :
  MapExpr × MapType × TYPINGS × FUNC × TRANS ×
  ENV × TYPES →
    BOOL_ENV_TYPES
CheckMapExpr(me, et, typings, func, trans, env, types) ≡
  case me of
    EnumeratedMapExpr(ovel) →
      CheckOptValueExprPairList(
```

```
            ovel, et, typings, func, trans, env, types),
        ComprehendedMapExpr(vep, typlist, or) →
          mk_BOOL_ENV_TYPES(
            and(
                bool(
                  CheckValueExprList(
                      ⟨first(vep), second(vep)⟩,
                      ⟨tedom(et), terange(et)⟩, typings,
                      func, trans, env, types)),
                bool(
                  CheckOptRestriction(
                      or, Known(TypeLiteral(BOOL)),
                      typings, func, trans,
                      envMap(
                        CheckValueExprList(
                          ⟨first(vep), second(vep)⟩,
                          ⟨tedom(et), terange(et)⟩,
                          typings, func, trans, env,
                          types)),
                      typesMap(
                        CheckValueExprList(
                          ⟨first(vep), second(vep)⟩,
                          ⟨tedom(et), terange(et)⟩,
                          typings, func, trans, env,
                          types))))),
            envMap(
                CheckOptRestriction(
                  or, Known(TypeLiteral(BOOL)), typings,
                  func, trans,
                  envMap(
                      CheckValueExprList(
                        ⟨first(vep), second(vep)⟩,
                        ⟨tedom(et), terange(et)⟩,
                        typings, func, trans, env, types)),
                  typesMap(
                      CheckValueExprList(
                        ⟨first(vep), second(vep)⟩,
                        ⟨tedom(et), terange(et)⟩,
                        typings, func, trans, env, types))
                )),
            typesMap(
                CheckOptRestriction(
                  or, Known(TypeLiteral(BOOL)), typings,
                  func, trans,
```

```
                    envMap(
                        CheckValueExprList(
                          ⟨first(vep), second(vep)⟩,
                          ⟨tedom(et), terange(et)⟩,
                          typings, func, trans, env, types)),
                    typesMap(
                        CheckValueExprList(
                          ⟨first(vep), second(vep)⟩,
                          ⟨tedom(et), terange(et)⟩,
                          typings, func, trans, env, types))
                )))
    end,

  CheckOptValueExprList :
    OptionalValueExprList × ExpType × TYPINGS ×
    FUNC × TRANS × ENV × TYPES →
      BOOL_ENV_TYPES
  CheckOptValueExprList(
     ovel, et, typings, func, trans, env, types) ≡
    case ovel of
      ValueExprList(vel) →
        CheckValueExprList(
            vel, toExpTypeList(et, lengthVE(vel)),
            typings, func, trans, env, types),
      NoValueExprList →
        mk_BOOL_ENV_TYPES(true, env, types)
    end,

  CheckOptValueExprPairList :
    OptionalValueExprPairList × MapType × TYPINGS ×
    FUNC × TRANS × ENV × TYPES →
      BOOL_ENV_TYPES
  CheckOptValueExprPairList(
     ovel, et, typings, func, trans, env, types) ≡
    case ovel of
      ValueExprPairList(vel) →
        CheckValueExprPairList(
            vel, et, typings, func, trans, env, types),
      NoValueExprPairList →
        mk_BOOL_ENV_TYPES(true, env, types)
    end,

  CheckValueExprPairList :
    ValueExprPair* × MapType × TYPINGS × FUNC ×
```

```
    TRANS × ENV × TYPES →
      BOOL_ENV_TYPES
  CheckValueExprPairList(
    vel, et, typings, func, trans, env, types) ≡
    if vel = ⟨⟩
    then mk_BOOL_ENV_TYPES(true, env, types)
    else
      if
        not(
          bool(
            CheckValueExprList(
              ⟨first(hd vel), second(hd vel)⟩,
              ⟨tedom(et), terange(et)⟩, typings,
              func, trans, env, types)))
      then
        mk_BOOL_ENV_TYPES(
          bool(
            CheckValueExprList(
              ⟨first(hd vel), second(hd vel)⟩,
              ⟨tedom(et), terange(et)⟩, typings,
              func, trans, env, types)),
          envMap(
            CheckValueExprList(
              ⟨first(hd vel), second(hd vel)⟩,
              ⟨tedom(et), terange(et)⟩, typings,
              func, trans, env, types)),
          typesMap(
            CheckValueExprList(
              ⟨first(hd vel), second(hd vel)⟩,
              ⟨tedom(et), terange(et)⟩, typings,
              func, trans, env, types)))
      else
        CheckValueExprPairList(
          tl vel, et, typings, func, trans,
          envMap(
            CheckValueExprList(
              ⟨first(hd vel), second(hd vel)⟩,
              ⟨tedom(et), terange(et)⟩, typings,
              func, trans, env, types)),
          typesMap(
            CheckValueExprList(
              ⟨first(hd vel), second(hd vel)⟩,
              ⟨tedom(et), terange(et)⟩, typings,
              func, trans, env, types)))
```

```
        end
    end,

CheckOptRestriction :
    OptionalRestriction × ExpType × TYPINGS × FUNC ×
    TRANS × ENV × TYPES →
        BOOL_ENV_TYPES
CheckOptRestriction(
     optr, et, typings, func, trans, env, types) ≡
    case optr of
        Restriction(ve) →
            CheckValueExpr(
                 ve, et, typings, func, trans, env, types),
        NoRestriction →
            mk_BOOL_ENV_TYPES(true, env, types)
    end,

CheckApplicationExpr :
    ValueExpr × ValueExpr* × ExpType × TYPINGS ×
    FUNC × TRANS × ENV × TYPES →
        BOOL_ENV_TYPES
CheckApplicationExpr(
     ve, vel, et, typings, func, trans, env, types) ≡
    case ve of
        Make_ValueOrVariableName(vn) →
            if isinId(id(vn), domainFUNC(func))
            then
                /*The application expression is a function application.
                */
                case
                    type_expr(getMapValueFUNC(func, id(vn)))
                of
                    FunctionTypeExpr(arg, fa, res) →
                        CheckFunctionAppl(
                             ve, vel, typeExprToExpTypeList(arg),
                             typings, func, trans, env, types)
                end
            else
                /*The application expression is a list or map
                application.*/
                CheckListMapAppl(
                     ve, vel, et, typings, func, trans, env,
                     types)
            end,
```

```
    _ →
      CheckListMapAppl(
          ve, vel, et, typings, func, trans, env, types)
  end,

CheckFunctionAppl :
  ValueExpr × ValueExpr* × ExpType* ×
  TYPINGS × FUNC × TRANS × ENV × TYPES →
    BOOL_ENV_TYPES
CheckFunctionAppl(
   ve, vel, etl, typings, func, trans, env, types) ≡
  case ve of
    Make_ValueOrVariableName(vn) →
      mk_BOOL_ENV_TYPES(
          and(
                bool(
                    CheckValueExprList(
                          vel, etl, typings, func, trans, env,
                          types)),
                greaterEqual(
                    lengthVE(vel), lengthET(etl), 0)),
          setEnv(
                write_list(getMapValueFUNC(func, id(vn))),
                envMap(
                    CheckValueExprList(
                          vel, etl, typings, func, trans, env,
                          types))),
          typesMap(
                CheckValueExprList(
                    vel, etl, typings, func, trans, env,
                    types)))
  end,

CheckListMapAppl :
  ValueExpr × ValueExpr* × ExpType × TYPINGS ×
  FUNC × TRANS × ENV × TYPES →
    BOOL_ENV_TYPES
CheckListMapAppl(
   ve, vel, et, typings, func, trans, env, types) ≡
  case ve of
    Make_ValueOrVariableName(vn) →
      if
        checkTRANS(
            getMapValueTYPES(types, IdBinding(id(vn))),
```

```
        domainTRANS(trans))
then
    /*The list or map is of the type of interest.
    */
    mk_BOOL_ENV_TYPES(
        and(
            and(
                isinBinding(
                    IdBinding(id(vn)), rangeENV(env)),
                bool(
                    CheckValueExprList(
                        vel,
                        expTypeToExpTypeList(
                            removeBrackets(
                                getListMapTypeArg(
                                    getMapValueTYPES(
                                        types,
                                        IdBinding(id(vn))))),
                            lengthVE(vel)), typings,
                        func, trans, env, types))),
            (lengthET(
                expTypeToExpTypeList(
                    removeBrackets(
                        getListMapTypeArg(
                            getMapValueTYPES(
                                types, IdBinding(id(vn)))
                        )), lengthVE(vel))) =
                lengthVE(vel))),
        updateENV(env, et, trans),
        typesMap(
            CheckValueExprList(
                vel,
                expTypeToExpTypeList(
                    removeBrackets(
                        getListMapTypeArg(
                            getMapValueTYPES(
                                types, IdBinding(id(vn))))
                    ), lengthVE(vel)), typings, func,
                trans, env, types)))
else
    /*The list or map is not of the type of interest.
    */
    mk_BOOL_ENV_TYPES(
        and(
```

```
                    bool(
                      CheckValueExprList(
                        vel,
                        expTypeToExpTypeList(
                          removeBrackets(
                            getListMapTypeArg(
                              getMapValueTYPES(
                                types,
                                IdBinding(id(vn))))),
                          lengthVE(vel)), typings, func,
                        trans, env, types)),
                    (lengthET(
                        expTypeToExpTypeList(
                          removeBrackets(
                            getListMapTypeArg(
                              getMapValueTYPES(
                                types, IdBinding(id(vn)))
                            )), lengthVE(vel))) =
                      lengthVE(vel))),
                 updateENV(env, et, trans),
                 typesMap(
                    CheckValueExprList(
                      vel,
                      expTypeToExpTypeList(
                        removeBrackets(
                          getListMapTypeArg(
                            getMapValueTYPES(
                              types, IdBinding(id(vn))))
                        ), lengthVE(vel)), typings, func,
                      trans, env, types)))
        end,
      _ → mk_BOOL_ENV_TYPES(true, env, types)
    end,

  CheckLetDefList :
    LetDef* × TYPINGS × FUNC × TRANS × ENV ×
    TYPES →
      BOOL_ENV_TYPES
  CheckLetDefList(ldl, typings, func, trans, env, types) ≡
    if ldl = ⟨⟩
    then mk_BOOL_ENV_TYPES(true, env, types)
    else
      if
        bool(
```

```
            CheckLetDef(
                 hd ldl, typings, func, trans, env, types))
      then
         CheckLetDefList(
            tl ldl, typings, func, trans,
            envMap(
                 CheckLetDef(
                    hd ldl, typings, func, trans, env, types
                 )),
            typesMap(
                 CheckLetDef(
                    hd ldl, typings, func, trans, env, types
                 )))
      else
         mk_BOOL_ENV_TYPES(
            bool(
                 CheckLetDef(
                    hd ldl, typings, func, trans, env, types
                 )),
            envMap(
                 CheckLetDef(
                    hd ldl, typings, func, trans, env, types
                 )),
            typesMap(
                 CheckLetDef(
                    hd ldl, typings, func, trans, env, types
                 )))
      end
   end,

CheckLetDef :
   LetDef × TYPINGS × FUNC × TRANS × ENV × TYPES →
      BOOL_ENV_TYPES
CheckLetDef(ld, typings, func, trans, env, types) ≡
   if
      not(
          bool(
               CheckLetBinding(
                   binding(ld), typings, func, trans, env,
                   types)))
   then mk_BOOL_ENV_TYPES(false, env, types)
   else
      case value_expr(ld) of
         ApplicationExpr(ve, vel) →
```

```
case ve of
  Make_ValueOrVariableName(vn) →
    if isinId(id(vn), domainFUNC(func))
    then
      /*The let definition is a function application.
      */
      case
        type_expr(
            getMapValueFUNC(func, id(vn)))
      of
        FunctionTypeExpr(arg, fa, res) →
          mk_BOOL_ENV_TYPES(
            and(
              bool(
                CheckValueExpr(
                  value_expr(ld),
                  Known(type_expr(res)),
                  typings, func, trans,
                  env, types)),
              matchBindingTE(
                binding(ld), type_expr(res))
            ),
            overrideENV(
              envMap(
                CheckValueExpr(
                  value_expr(ld),
                  Known(type_expr(res)),
                  typings, func, trans,
                  env, types)),
              makeENV(
                binding(ld),
                type_expr(res), trans)),
            overrideTYPES(
              typesMap(
                CheckValueExpr(
                  value_expr(ld),
                  Known(type_expr(res)),
                  typings, func, trans,
                  env, types)),
              makeTYPES(
                binding(ld),
                Known(type_expr(res)), trans
              )))
      end
```

```
                    else
                      /*The let definition is a list ot map application.
                      */
                      CheckValueExpr(
                        value_expr(ld), Unknown, typings,
                        func, trans, env,
                        overrideTYPES(
                          types,
                          makeTYPES(
                            binding(ld), Unknown, trans)))
                    end,
                  _ →
                    CheckValueExpr(
                      value_expr(ld), Unknown, typings, func,
                      trans, env,
                      overrideTYPES(
                        types,
                        makeTYPES(
                          binding(ld), Unknown, trans)))
                end,
              _ →
                CheckValueExpr(
                  value_expr(ld), Unknown, typings, func,
                  trans, env,
                  overrideTYPES(
                    types,
                    makeTYPES(binding(ld), Unknown, trans)))
          end
        end,

    CheckLetBinding :
      LetBinding × TYPINGS × FUNC × TRANS × ENV ×
      TYPES →
        BOOL_ENV_TYPES
    CheckLetBinding(lb, typings, func, trans, env, types) ≡
      case lb of
        MakeBinding(b) →
          mk_BOOL_ENV_TYPES(true, env, types),
        MakeRecordPattern(vn, pl) →
          CheckPatternList(pl, func, trans, env, types),
        MakeListPatternLet(lp) →
          CheckListPattern(lp, func, trans, env, types)
      end,
```

```
CheckIfExpr :
   IfExpr × ExpType × TYPINGS × FUNC × TRANS ×
   ENV × TYPES →
      BOOL_ENV_TYPES
CheckIfExpr(ie, et, typings, func, trans, env, types) ≡
   mk_BOOL_ENV_TYPES(
       and(
            and(
                and(
                     bool(
                         CheckValueExpr(
                             condition(ie),
                             Known(TypeLiteral(BOOL)), typings,
                             func, trans, env, types)),
                     bool(
                         CheckValueExpr(
                             if_case(ie), et, typings, func,
                             trans,
                             envMap(
                                CheckValueExpr(
                                   condition(ie),
                                   Known(TypeLiteral(BOOL)),
                                   typings, func, trans, env,
                                   types)),
                             typesMap(
                                CheckValueExpr(
                                   condition(ie),
                                   Known(TypeLiteral(BOOL)),
                                   typings, func, trans, env,
                                   types))))),
                bool(
                     CheckElsif(
                         elsif_list(ie), et, typings, func,
                         trans,
                         envMap(
                             CheckValueExpr(
                                condition(ie),
                                Known(TypeLiteral(BOOL)),
                                typings, func, trans, env, types
                             )),
                         typesMap(
                             CheckValueExpr(
                                if_case(ie), et, typings, func,
                                trans,
```

```
                                envMap(
                                   CheckValueExpr(
                                      condition(ie),
                                      Known(TypeLiteral(BOOL)),
                                      typings, func, trans,
                                      env, types)),
                                typesMap(
                                   CheckValueExpr(
                                      condition(ie),
                                      Known(TypeLiteral(BOOL)),
                                      typings, func, trans,
                                      env, types))))))),
              bool(
                 CheckValueExpr(
                       else_case(ie), et, typings, func, trans,
                       envMap(
                          CheckElsif(
                             elsif_list(ie), et, typings, func,
                             trans,
                             envMap(
                                CheckValueExpr(
                                   condition(ie),
                                   Known(TypeLiteral(BOOL)),
                                   typings, func, trans, env,
                                   types)),
                             typesMap(
                                CheckValueExpr(
                                   if_case(ie), et, typings,
                                   func, trans,
                                   envMap(
                                      CheckValueExpr(
                                         condition(ie),
                                         Known(
                                            TypeLiteral(BOOL)),
                                         typings, func, trans,
                                         env, types)),
                                   typesMap(
                                      CheckValueExpr(
                                         condition(ie),
                                         Known(
                                            TypeLiteral(BOOL)),
                                         typings, func, trans,
                                         env, types)))))),
                       typesMap(
```

```
                    CheckElsif(
                        elsif_list(ie), et, typings, func,
                        trans,
                        envMap(
                          CheckValueExpr(
                            condition(ie),
                            Known(TypeLiteral(BOOL)),
                            typings, func, trans, env,
                            types)),
                        typesMap(
                          CheckValueExpr(
                            if_case(ie), et, typings,
                            func, trans,
                            envMap(
                              CheckValueExpr(
                                condition(ie),
                                Known(
                                  TypeLiteral(BOOL)),
                                typings, func, trans,
                                env, types)),
                            typesMap(
                              CheckValueExpr(
                                condition(ie),
                                Known(
                                  TypeLiteral(BOOL)),
                                typings, func, trans,
                                env, types))))))))),
envMap(
    CheckValueExpr(
        else_case(ie), et, typings, func, trans,
        envMap(
            CheckElsif(
                elsif_list(ie), et, typings, func,
                trans,
                envMap(
                    CheckValueExpr(
                      condition(ie),
                      Known(TypeLiteral(BOOL)),
                      typings, func, trans, env, types
                    )),
                typesMap(
                    CheckValueExpr(
                      if_case(ie), et, typings, func,
                      trans,
```

```
                                          envMap(
                                            CheckValueExpr(
                                              condition(ie),
                                              Known(TypeLiteral(BOOL)),
                                              typings, func, trans,
                                              env, types)),
                                          typesMap(
                                            CheckValueExpr(
                                              condition(ie),
                                              Known(TypeLiteral(BOOL)),
                                              typings, func, trans,
                                              env, types)))))),
                          typesMap(
                               CheckElsif(
                                   elsif_list(ie), et, typings, func,
                                   trans,
                                   envMap(
                                       CheckValueExpr(
                                         condition(ie),
                                         Known(TypeLiteral(BOOL)),
                                         typings, func, trans, env, types
                                       )),
                                   typesMap(
                                       CheckValueExpr(
                                         if_case(ie), et, typings, func,
                                         trans,
                                         envMap(
                                           CheckValueExpr(
                                             condition(ie),
                                             Known(TypeLiteral(BOOL)),
                                             typings, func, trans,
                                             env, types)),
                                         typesMap(
                                           CheckValueExpr(
                                             condition(ie),
                                             Known(TypeLiteral(BOOL)),
                                             typings, func, trans,
                                             env, types)))))))),
                 typesMap(
                      CheckValueExpr(
                          else_case(ie), et, typings, func, trans,
                          envMap(
                               CheckElsif(
                                   elsif_list(ie), et, typings, func,
```

```
                trans,
                envMap(
                    CheckValueExpr(
                        condition(ie),
                        Known(TypeLiteral(BOOL)),
                        typings, func, trans, env, types
                    )),
                typesMap(
                    CheckValueExpr(
                        if_case(ie), et, typings, func,
                        trans,
                        envMap(
                            CheckValueExpr(
                                condition(ie),
                                Known(TypeLiteral(BOOL)),
                                typings, func, trans,
                                env, types)),
                        typesMap(
                            CheckValueExpr(
                                condition(ie),
                                Known(TypeLiteral(BOOL)),
                                typings, func, trans,
                                env, types)))))),
        typesMap(
            CheckElsif(
                elsif_list(ie), et, typings, func,
                trans,
                envMap(
                    CheckValueExpr(
                        condition(ie),
                        Known(TypeLiteral(BOOL)),
                        typings, func, trans, env, types
                    )),
                typesMap(
                    CheckValueExpr(
                        if_case(ie), et, typings, func,
                        trans,
                        envMap(
                            CheckValueExpr(
                                condition(ie),
                                Known(TypeLiteral(BOOL)),
                                typings, func, trans,
                                env, types)),
                        typesMap(
```

```
                                    CheckValueExpr(
                                      condition(ie),
                                      Known(TypeLiteral(BOOL)),
                                      typings, func, trans,
                                      env, types))))))))),

CheckElsif :
  Elsif* × ExpType × TYPINGS × FUNC × TRANS ×
  ENV × TYPES →
    BOOL_ENV_TYPES
CheckElsif(eil, et, typings, func, trans, env, types) ≡
  if eil = ⟨⟩
  then mk_BOOL_ENV_TYPES(true, env, types)
  else
    if
      and(
        bool(
            CheckValueExpr(
              condition(hd eil),
              Known(TypeLiteral(BOOL)), typings,
              func, trans, env, types)),
        bool(
            CheckValueExpr(
              elsif_case(hd eil), et, typings, func,
              trans,
              envMap(
                  CheckValueExpr(
                    condition(hd eil),
                    Known(TypeLiteral(BOOL)),
                    typings, func, trans, env, types)),
              typesMap(
                  CheckValueExpr(
                    condition(hd eil),
                    Known(TypeLiteral(BOOL)),
                    typings, func, trans, env, types))
            )))
    then
      CheckElsif(
        tl eil, et, typings, func, trans,
        envMap(
            CheckValueExpr(
              condition(hd eil),
              Known(TypeLiteral(BOOL)), typings,
              func, trans, env, types)),
```

```
            typesMap(
                CheckValueExpr(
                    elsif_case(hd eil), et, typings, func,
                    trans,
                    envMap(
                        CheckValueExpr(
                            condition(hd eil),
                            Known(TypeLiteral(BOOL)),
                            typings, func, trans, env, types)),
                    typesMap(
                        CheckValueExpr(
                            condition(hd eil),
                            Known(TypeLiteral(BOOL)),
                            typings, func, trans, env, types))
                )))
      else
         mk_BOOL_ENV_TYPES(
            false,
            envMap(
                CheckValueExpr(
                    condition(hd eil),
                    Known(TypeLiteral(BOOL)), typings,
                    func, trans, env, types)),
            typesMap(
                CheckValueExpr(
                    elsif_case(hd eil), et, typings, func,
                    trans,
                    envMap(
                        CheckValueExpr(
                            condition(hd eil),
                            Known(TypeLiteral(BOOL)),
                            typings, func, trans, env, types)),
                    typesMap(
                        CheckValueExpr(
                            condition(hd eil),
                            Known(TypeLiteral(BOOL)),
                            typings, func, trans, env, types))
                )))
      end
   end,

CheckCaseBranchList :
   CaseBranch* × ExpType × TYPINGS × FUNC ×
   TRANS × ENV × TYPES →
```

```
        BOOL_ENV_TYPES
    CheckCaseBranchList(
      cbl, et, typings, func, trans, env, types) ≡
      if cbl = ⟨⟩
      then mk_BOOL_ENV_TYPES(true, env, types)
      else
        if
          and(
            bool(
              CheckPattern(
                pattern(hd cbl), func, trans, env, types
              )),
            bool(
              CheckValueExpr(
                value_expr(hd cbl), et, typings, func,
                trans,
                envMap(
                  CheckPattern(
                    pattern(hd cbl), func, trans,
                    env, types)),
                typesMap(
                  CheckPattern(
                    pattern(hd cbl), func, trans,
                    env, types)))))
        then
          CheckCaseBranchList(
            tl cbl, et, typings, func, trans,
            envMap(
              CheckPattern(
                pattern(hd cbl), func, trans, env, types
              )),
            typesMap(
              CheckValueExpr(
                value_expr(hd cbl), et, typings, func,
                trans,
                envMap(
                  CheckPattern(
                    pattern(hd cbl), func, trans,
                    env, types)),
                typesMap(
                  CheckPattern(
                    pattern(hd cbl), func, trans,
                    env, types)))))
        else
```

```
            mk_BOOL_ENV_TYPES(
              false,
              envMap(
                  CheckPattern(
                    pattern(hd cbl), func, trans, env, types
                  )),
              typesMap(
                  CheckValueExpr(
                    value_expr(hd cbl), et, typings, func,
                    trans,
                    envMap(
                        CheckPattern(
                          pattern(hd cbl), func, trans,
                          env, types)),
                    typesMap(
                        CheckPattern(
                          pattern(hd cbl), func, trans,
                          env, types)))))
        end
      end,

    CheckPattern :
      Pattern × FUNC × TRANS × ENV × TYPES →
        BOOL_ENV_TYPES
    CheckPattern(p, func, trans, env, types) ≡
      case p of
        ValueLiteralPattern(vl) →
          mk_BOOL_ENV_TYPES(true, env, types),
        NamePattern(id, optid) →
          if isinId(id, domainFUNC(func))
          then mk_BOOL_ENV_TYPES(false, env, types)
          else
            if
              not(
                 isinBinding(
                     IdBinding(id), domainTYPES(types)))
            then mk_BOOL_ENV_TYPES(true, env, types)
            else
              if
                checkTRANS(
                  getMapValueTYPES(types, IdBinding(id)),
                  domainTRANS(trans))
              then
                /*The value expression is of the type of interest.
```

```
                  */
                  mk_BOOL_ENV_TYPES(false, env, types)
               else
                  /*The value expression is not of the type of interest.
                  */
                  mk_BOOL_ENV_TYPES(true, env, types)
               end
            end
         end,
      WildcardPattern →
         mk_BOOL_ENV_TYPES(true, env, types),
      ProductPattern(pl) →
         CheckPatternList(pl, func, trans, env, types),
      RecordPattern(vn, pl) →
         if isinId(id(vn), domainFUNC(func))
         then mk_BOOL_ENV_TYPES(false, env, types)
         else
            CheckPatternList(pl, func, trans, env, types)
         end,
      MakeListPattern(lp) →
         CheckListPattern(lp, func, trans, env, types)
   end,

CheckPatternList :
   Pattern* × FUNC × TRANS × ENV × TYPES →
      BOOL_ENV_TYPES
CheckPatternList(pl, func, trans, env, types) ≡
   if pl = ⟨⟩
   then mk_BOOL_ENV_TYPES(true, env, types)
   else
      if
         bool(
             CheckPattern(hd pl, func, trans, env, types))
      then
         CheckPatternList(
             tl pl, func, trans,
             envMap(
                  CheckPattern(
                       hd pl, func, trans, env, types)),
             typesMap(
                  CheckPattern(
                       hd pl, func, trans, env, types)))
      else
         mk_BOOL_ENV_TYPES(
```

```
                false,
                envMap(
                    CheckPattern(
                        hd pl, func, trans, env, types)), types)
        end
    end,

CheckListPattern :
    ListPattern × FUNC × TRANS × ENV × TYPES →
        BOOL_ENV_TYPES
CheckListPattern(lp, func, trans, env, types) ≡
    case lp of
        Make_EnumeratedListPattern(elp) →
            CheckOptInnerPattern(
                inner_pattern(elp), func, trans, env, types),
        ConcatenatedListPattern(elp, p) →
            mk_BOOL_ENV_TYPES(
                and(
                    bool(
                        CheckOptInnerPattern(
                            inner_pattern(elp), func, trans,
                            env, types)),
                    bool(
                        CheckPattern(
                            p, func, trans,
                            envMap(
                                CheckOptInnerPattern(
                                    inner_pattern(elp), func,
                                    trans, env, types)),
                            typesMap(
                                CheckOptInnerPattern(
                                    inner_pattern(elp), func,
                                    trans, env, types))))),
                envMap(
                    CheckPattern(
                        p, func, trans,
                        envMap(
                            CheckOptInnerPattern(
                                inner_pattern(elp), func, trans,
                                env, types)),
                        typesMap(
                            CheckOptInnerPattern(
                                inner_pattern(elp), func, trans,
                                env, types)))),
```

```
                typesMap(
                    CheckPattern(
                        p, func, trans,
                        envMap(
                            CheckOptInnerPattern(
                                inner_pattern(elp), func, trans,
                                env, types)),
                        typesMap(
                            CheckOptInnerPattern(
                                inner_pattern(elp), func, trans,
                                env, types)))))
    end,

CheckOptInnerPattern :
    OptionalInnerPattern × FUNC × TRANS × ENV × TYPES →
        BOOL_ENV_TYPES
CheckOptInnerPattern(oip, func, trans, env, types) ≡
    case oip of
        InnerPatternList(pl) →
            CheckPatternList(pl, func, trans, env, types),
        NoInnerPattern →
            mk_BOOL_ENV_TYPES(true, env, types)
    end,

matchBindingTE : LetBinding × TypeExpr → Bool
matchBindingTE(lb, te) ≡
    greaterEqual(
        getLengthLetBinding(lb), getLengthTE(te), 0),

TRValueDefList :
    ValueDef* × FUNC × TRANS → ValueDef*
TRValueDefList(vdl, func, trans) ≡
    if vdl = ⟨⟩ then ⟨⟩
    else
        ⟨TRValueDef(hd vdl, func, trans)⟩ ^
        TRValueDefList(tl vdl, func, trans)
    end,

TRValueDef : ValueDef × FUNC × TRANS → ValueDef
TRValueDef(vd, func, trans) ≡
    case vd of
        ExplicitValueDef(st, ve) →
            if
                includesTRANSName(
```

```
                Known(type_expr(st)), domainTRANS(trans))
        then
          case binding(st) of
            IdBinding(id) →
              ExplicitFunctionDef(
                TRSingleTyping(
                    mk_SingleTyping(
                      binding(st),
                      FunctionTypeExpr(
                        TypeLiteral(UNIT),
                        TOTAL_FUNCTION_ARROW,
                        mk_ResultDesc(
                          NoReadAccessMode,
                          NoWriteAccessMode,
                          type_expr(st)))), func,
                    trans),
                IdApplication(
                    id, mk_FormalFunctionParameter(⟨⟩)),
                valueExpr(
                    TRValueExpr(
                      ve, Known(type_expr(st)), func,
                      trans, mk_TYPES(⟨⟩))),
                NoPreCondition)
          end
        else vd
        end,
      ExplicitFunctionDef(st, ffa, ve, precond) →
        case type_expr(st) of
          FunctionTypeExpr(arg, fa, res) →
            if
              lengthLetDefList(
                letdefList(
                    TRFormalFuncAppl(ffa, func, trans))) =
                0
            then
              ExplicitFunctionDef(
                TRSingleTyping(st, func, trans),
                formalfa(
                    TRFormalFuncAppl(ffa, func, trans)),
                valueExpr(
                    TRValueExpr(
                      ve, Known(type_expr(res)), func,
                      trans,
                      typesMap(
```

```
                        TROptPreCondition(
                          precond,
                          Known(TypeLiteral(BOOL)),
                          func, trans,
                          alterTYPESMap(
                            typesMap(
                              TRFormalFuncAppl(
                                ffa, func, trans)),
                            letdefList(
                              TRFormalFuncAppl(
                                ffa, func, trans)))
                        )))),
              preCond(
                    TROptPreCondition(
                      precond,
                      Known(TypeLiteral(BOOL)), func,
                      trans,
                      alterTYPESMap(
                        typesMap(
                          TRFormalFuncAppl(
                            ffa, func, trans)),
                        letdefList(
                          TRFormalFuncAppl(
                            ffa, func, trans))))))
        else
          ExplicitFunctionDef(
              TRSingleTyping(st, func, trans),
              formalfa(
                    TRFormalFuncAppl(ffa, func, trans)),
              LetExpr(
                    letdefList(
                      TRFormalFuncAppl(ffa, func, trans)
                    ),
                    valueExpr(
                      TRValueExpr(
                        ve, Known(type_expr(res)),
                        func, trans,
                        typesMap(
                          TROptPreCondition(
                            precond,
                            Known(TypeLiteral(BOOL)),
                            func, trans,
                            alterTYPESMap(
                              typesMap(
```

```
                                          TRFormalFuncAppl(
                                            ffa, func, trans
                                          )),
                                        letdefList(
                                          TRFormalFuncAppl(
                                            ffa, func, trans
                                          )))))))),
                        preCond(
                              TROptPreCondition(
                                precond,
                                Known(TypeLiteral(BOOL)), func,
                                trans,
                                alterTYPESMap(
                                  typesMap(
                                    TRFormalFuncAppl(
                                      ffa, func, trans)),
                                  letdefList(
                                    TRFormalFuncAppl(
                                      ffa, func, trans))))))
                end
            end
        end,

    TRSingleTyping :
        SingleTyping × FUNC × TRANS → SingleTyping
    TRSingleTyping(st, func, trans) ≡
        case binding(st) of
            IdBinding(id) →
                mk_SingleTyping(
                    binding(st),
                    TRTypeExpr(id, type_expr(st), func, trans))
        end,

    TRTypeExpr :
        Id × TypeExpr × FUNC × TRANS → TypeExpr
    TRTypeExpr(id, te, func, trans) ≡
        case te of
            TypeLiteral(literal) → te,
            TypeName(tid) →
                if isinId(tid, domainTRANS(trans))
                then
                    /*Type expression of a type of interest.*/
                    TypeLiteral(UNIT)
                else
```

```
            /*Type expression not of a type of interest.*/
            te
          end,
        TypeExprProduct(tel) →
          TypeExprProduct(
              TRTypeExprList(id, tel, func, trans)),
        TypeExprSet(tes) → te,
        TypeExprList(tel) → te,
        TypeExprMap(tem) → te,
        FunctionTypeExpr(fte, fa, rd) →
          TRFunctionDef(id, fte, fa, rd, func, trans),
        BracketedTypeExpr(bte) →
          BracketedTypeExpr(
              TRTypeExpr(id, bte, func, trans))
      end,

    TRTypeExprList :
      Id × TypeExpr* × FUNC × TRANS → TypeExpr*
    TRTypeExprList(id, tel, func, trans) ≡
      if tel = ⟨⟩ then ⟨⟩
      else
        ⟨TRTypeExpr(id, hd tel, func, trans)⟩ ^
        TRTypeExprList(id, tl tel, func, trans)
      end,

    TRFunctionDef :
      Id × TypeExpr × FunctionArrow × ResultDesc ×
      FUNC × TRANS →
        TypeExpr
    TRFunctionDef(id, te, fa, rd, func, trans) ≡
      FunctionTypeExpr(
         truncate(TRTypeExpr(id, te, func, trans)), fa,
         mk_ResultDesc(
            makeReadAccessDesc(
               read_list(getMapValueFUNC(func, id)), trans),
            makeWriteAccessDesc(
               write_list(getMapValueFUNC(func, id)), trans
            ),
            truncate(
               TRTypeExpr(id, type_expr(rd), func, trans)))
      ),

    TRFormalFuncAppl :
      FormalFunctionApplication × FUNC × TRANS →
```

```
        BOOL_FFA_ENV_TYPES_LDL
    TRFormalFuncAppl(ffa, func, trans) ≡
      case ffa of
        IdApplication(id, ffp) →
          case type_expr(getMapValueFUNC(func, id)) of
            FunctionTypeExpr(arg, fa, res) →
              mk_BOOL_FFA_ENV_TYPES_LDL(
                bool(
                    TRFormalFuncParam(
                      ffp, arg, trans, mk_ENV(⟨⟩),
                      mk_TYPES(⟨⟩), ⟨⟩)),
                IdApplication(
                    id,
                    formalfp(
                      TRFormalFuncParam(
                        ffp, arg, trans, mk_ENV(⟨⟩),
                        mk_TYPES(⟨⟩), ⟨⟩))),
                envMap(
                    TRFormalFuncParam(
                      ffp, arg, trans, mk_ENV(⟨⟩),
                      mk_TYPES(⟨⟩), ⟨⟩)),
                typesMap(
                    TRFormalFuncParam(
                      ffp, arg, trans, mk_ENV(⟨⟩),
                      mk_TYPES(⟨⟩), ⟨⟩)),
                letdefList(
                    TRFormalFuncParam(
                      ffp, arg, trans, mk_ENV(⟨⟩),
                      mk_TYPES(⟨⟩), ⟨⟩)))
          end,
        _ →
          mk_BOOL_FFA_ENV_TYPES_LDL(
              true, ffa, mk_ENV(⟨⟩), mk_TYPES(⟨⟩), ⟨⟩)
      end,

    TRFormalFuncParam :
      FormalFunctionParameter × TypeExpr × TRANS ×
      ENV × TYPES × LetDef* →
        BOOL_FFP_ENV_TYPES_LDL
    TRFormalFuncParam(ffpl, te, trans, env, types, prlet) ≡
      if lengthBinding(binding_list(ffpl)) = 0
      then
        mk_BOOL_FFP_ENV_TYPES_LDL(
            true, mk_FormalFunctionParameter(⟨⟩), env,
```

```
            types, prlet)
   else
      case te of
         TypeLiteral(literal) →
            mk_BOOL_FFP_ENV_TYPES_LDL(
               true, ffpl, env,
               overrideTYPES(
                  types,
                  mk_TYPES(
                     ⟨mk_TYPESMapEntrance(
                        makeBinding(binding_list(ffpl)),
                        Known(te))⟩)), prlet),
         TypeName(id) →
            if isinId(id, domainTRANS(trans))
            then
               mk_BOOL_FFP_ENV_TYPES_LDL(
                  true, mk_FormalFunctionParameter(⟨⟩),
                  overrideENV(
                     env,
                     mk_ENV(
                        ⟨mk_ENVMapEntrance(
                           id,
                           makeBinding(binding_list(ffpl))
                        )⟩)),
                  overrideTYPES(
                     types,
                     makeTYPESMap(binding_list(ffpl), te)),
                  makeProductLet(
                     te, makeBinding(binding_list(ffpl)),
                     prlet, trans))
            else
               mk_BOOL_FFP_ENV_TYPES_LDL(
                  true, ffpl, env,
                  overrideTYPES(
                     types,
                     mk_TYPES(
                        ⟨mk_TYPESMapEntrance(
                           makeBinding(binding_list(ffpl)),
                           Known(te))⟩)), prlet)
            end,
         TypeExprProduct(tep) →
            TRFFPProduct(
               ffpl, tep, trans, env, types, prlet),
         TypeExprSet(tes) →
```

```
                mk_BOOL_FFP_ENV_TYPES_LDL(
                    true, ffpl, env,
                    overrideTYPES(
                        types,
                        mk_TYPES(
                            ⟨mk_TYPESMapEntrance(
                                makeBinding(binding_list(ffpl)),
                                Known(te))⟩)), prlet),
            TypeExprList(tel) →
                mk_BOOL_FFP_ENV_TYPES_LDL(
                    true, ffpl, env,
                    overrideTYPES(
                        types,
                        mk_TYPES(
                            ⟨mk_TYPESMapEntrance(
                                makeBinding(binding_list(ffpl)),
                                Known(te))⟩)), prlet),
            TypeExprMap(tem) →
                mk_BOOL_FFP_ENV_TYPES_LDL(
                    true, ffpl, env,
                    overrideTYPES(
                        types,
                        mk_TYPES(
                            ⟨mk_TYPESMapEntrance(
                                makeBinding(binding_list(ffpl)),
                                Known(te))⟩)), prlet),
            FunctionTypeExpr(tef, fa, rd) →
                TRFormalFuncParam(
                    ffpl, tef, trans, env,
                    overrideTYPES(
                        types,
                        mk_TYPES(
                            ⟨mk_TYPESMapEntrance(
                                makeBinding(binding_list(ffpl)),
                                Known(te))⟩)), prlet),
            BracketedTypeExpr(bte) →
                TRFormalFuncParam(
                    ffpl, bte, trans, env, types, prlet)
        end
    end,

TRFFPProduct :
    FormalFunctionParameter × TypeExpr* × TRANS ×
    ENV × TYPES × LetDef* →
```

```
    BOOL_FFP_ENV_TYPES_LDL
TRFFPProduct(ffpl, tel, trans, env, types, prlet) ≡
  if lengthBinding(binding_list(ffpl)) = 0
  then
    if not(tel = ⟨⟩)
    then
      mk_BOOL_FFP_ENV_TYPES_LDL(
          false, mk_FormalFunctionParameter(⟨⟩), env,
          types, prlet)
    else
      mk_BOOL_FFP_ENV_TYPES_LDL(
          true, mk_FormalFunctionParameter(⟨⟩), env,
          types, prlet)
    end
  else
    case hd binding_list(ffpl) of
      Make_ProductBinding(pb) →
        if
          emptyList(
              binding_list(
                  formalfp(
                      TRFormalFuncParam(
                        mk_FormalFunctionParameter(
                          binding_list(pb)), hd tel,
                        trans, env, types, prlet))))
        then
          mk_BOOL_FFP_ENV_TYPES_LDL(
              and(
                  bool(
                      TRFormalFuncParam(
                        mk_FormalFunctionParameter(
                          binding_list(pb)), hd tel,
                        trans, env, types, prlet)),
                  bool(
                      TRFFPProduct(
                        mk_FormalFunctionParameter(
                          tl binding_list(ffpl)),
                        tl tel, trans,
                        envMap(
                          TRFormalFuncParam(
                            mk_FormalFunctionParameter(
                              binding_list(pb)),
                            hd tel, trans, env, types,
                            prlet)),
```

```
                    typesMap(
                      TRFormalFuncParam(
                        mk_FormalFunctionParameter(
                          binding_list(pb)),
                        hd tel, trans, env, types,
                        prlet)),
                    letdefList(
                      TRFormalFuncParam(
                        mk_FormalFunctionParameter(
                          binding_list(pb)),
                        hd tel, trans, env, types,
                        prlet))))),
          mk_FormalFunctionParameter(
               binding_list(
                  formalfp(
                    TRFFPProduct(
                      mk_FormalFunctionParameter(
                        tl binding_list(ffpl)),
                      tl tel, trans,
                      envMap(
                        TRFormalFuncParam(
                          mk_FormalFunctionParameter(
                            binding_list(pb)),
                          hd tel, trans, env,
                          types, prlet)),
                      typesMap(
                        TRFormalFuncParam(
                          mk_FormalFunctionParameter(
                            binding_list(pb)),
                          hd tel, trans, env,
                          types, prlet)),
                      letdefList(
                        TRFormalFuncParam(
                          mk_FormalFunctionParameter(
                            binding_list(pb)),
                          hd tel, trans, env,
                          types, prlet)))))),
          envMap(
               TRFFPProduct(
                  mk_FormalFunctionParameter(
                    tl binding_list(ffpl)), tl tel,
                  trans,
                  envMap(
                    TRFormalFuncParam(
```

```
                    mk_FormalFunctionParameter(
                      binding_list(pb)), hd tel,
                    trans, env, types, prlet)),
                typesMap(
                  TRFormalFuncParam(
                    mk_FormalFunctionParameter(
                      binding_list(pb)), hd tel,
                    trans, env, types, prlet)),
                letdefList(
                  TRFormalFuncParam(
                    mk_FormalFunctionParameter(
                      binding_list(pb)), hd tel,
                    trans, env, types, prlet)))),
        typesMap(
            TRFFPProduct(
                mk_FormalFunctionParameter(
                  tl binding_list(ffpl)), tl tel,
                trans,
                envMap(
                  TRFormalFuncParam(
                    mk_FormalFunctionParameter(
                      binding_list(pb)), hd tel,
                    trans, env, types, prlet)),
                typesMap(
                  TRFormalFuncParam(
                    mk_FormalFunctionParameter(
                      binding_list(pb)), hd tel,
                    trans, env, types, prlet)),
                letdefList(
                  TRFormalFuncParam(
                    mk_FormalFunctionParameter(
                      binding_list(pb)), hd tel,
                    trans, env, types, prlet)))),
        letdefList(
            TRFFPProduct(
                mk_FormalFunctionParameter(
                  tl binding_list(ffpl)), tl tel,
                trans,
                envMap(
                  TRFormalFuncParam(
                    mk_FormalFunctionParameter(
                      binding_list(pb)), hd tel,
                    trans, env, types, prlet)),
                typesMap(
```

```
                              TRFormalFuncParam(
                                mk_FormalFunctionParameter(
                                  binding_list(pb)), hd tel,
                                trans, env, types, prlet)),
                            letdefList(
                              TRFormalFuncParam(
                                mk_FormalFunctionParameter(
                                  binding_list(pb)), hd tel,
                                trans, env, types, prlet)))))
          else
            if
              lengthBinding(
                binding_list(
                  formalfp(
                    TRFormalFuncParam(
                      mk_FormalFunctionParameter(
                        binding_list(pb)), hd tel,
                      trans, env, types, prlet)))) =
                1
            then
              mk_BOOL_FFP_ENV_TYPES_LDL(
                and(
                  bool(
                    TRFormalFuncParam(
                      mk_FormalFunctionParameter(
                        binding_list(pb)), hd tel,
                      trans, env, types, prlet)),
                  bool(
                    TRFFPProduct(
                      mk_FormalFunctionParameter(
                        tl binding_list(ffpl)),
                      tl tel, trans,
                      envMap(
                        TRFormalFuncParam(
                          mk_FormalFunctionParameter(
                            binding_list(pb)),
                          hd tel, trans, env,
                          types, prlet)),
                      typesMap(
                        TRFormalFuncParam(
                          mk_FormalFunctionParameter(
                            binding_list(pb)),
                          hd tel, trans, env,
                          types, prlet)),
```

```
                letdefList(
                   TRFormalFuncParam(
                      mk_FormalFunctionParameter(
                         binding_list(pb)),
                      hd tel, trans, env,
                      types, prlet))))),
    mk_FormalFunctionParameter(
          binding_list(
             formalfp(
                TRFormalFuncParam(
                   mk_FormalFunctionParameter(
                      binding_list(pb)),
                   hd tel, trans, env, types,
                   prlet))) ^
          binding_list(
             formalfp(
                TRFFPProduct(
                   mk_FormalFunctionParameter(
                      tl binding_list(ffpl)),
                   tl tel, trans,
                   envMap(
                      TRFormalFuncParam(
                         mk_FormalFunctionParameter(
                            binding_list(pb)),
                         hd tel, trans, env,
                         types, prlet)),
                   typesMap(
                      TRFormalFuncParam(
                         mk_FormalFunctionParameter(
                            binding_list(pb)),
                         hd tel, trans, env,
                         types, prlet)),
                   letdefList(
                      TRFormalFuncParam(
                         mk_FormalFunctionParameter(
                            binding_list(pb)),
                         hd tel, trans, env,
                         types, prlet)))))),
    envMap(
          TRFFPProduct(
             mk_FormalFunctionParameter(
                tl binding_list(ffpl)),
             tl tel, trans,
             envMap(
```

```
                TRFormalFuncParam(
                  mk_FormalFunctionParameter(
                    binding_list(pb)),
                  hd tel, trans, env, types,
                  prlet)),
              typesMap(
                TRFormalFuncParam(
                  mk_FormalFunctionParameter(
                    binding_list(pb)),
                  hd tel, trans, env, types,
                  prlet)),
              letdefList(
                TRFormalFuncParam(
                  mk_FormalFunctionParameter(
                    binding_list(pb)),
                  hd tel, trans, env, types,
                  prlet)))),
        typesMap(
            TRFFPProduct(
              mk_FormalFunctionParameter(
                tl binding_list(ffpl)),
              tl tel, trans,
              envMap(
                TRFormalFuncParam(
                  mk_FormalFunctionParameter(
                    binding_list(pb)),
                  hd tel, trans, env, types,
                  prlet)),
              typesMap(
                TRFormalFuncParam(
                  mk_FormalFunctionParameter(
                    binding_list(pb)),
                  hd tel, trans, env, types,
                  prlet)),
              letdefList(
                TRFormalFuncParam(
                  mk_FormalFunctionParameter(
                    binding_list(pb)),
                  hd tel, trans, env, types,
                  prlet)))),
        letdefList(
            TRFFPProduct(
              mk_FormalFunctionParameter(
                tl binding_list(ffpl)),
```

```
                                    tl tel, trans,
                                    envMap(
                                      TRFormalFuncParam(
                                        mk_FormalFunctionParameter(
                                          binding_list(pb)),
                                        hd tel, trans, env, types,
                                        prlet)),
                                    typesMap(
                                      TRFormalFuncParam(
                                        mk_FormalFunctionParameter(
                                          binding_list(pb)),
                                        hd tel, trans, env, types,
                                        prlet)),
                                    letdefList(
                                      TRFormalFuncParam(
                                        mk_FormalFunctionParameter(
                                          binding_list(pb)),
                                        hd tel, trans, env, types,
                                        prlet)))))
                     else
                        mk_BOOL_FFP_ENV_TYPES_LDL(
                           and(
                                 bool(
                                   TRFormalFuncParam(
                                     mk_FormalFunctionParameter(
                                       binding_list(pb)), hd tel,
                                     trans, env, types, prlet)),
                                 bool(
                                   TRFFPProduct(
                                     mk_FormalFunctionParameter(
                                       tl binding_list(ffpl)),
                                     tl tel, trans,
                                     envMap(
                                       TRFormalFuncParam(
                                         mk_FormalFunctionParameter(
                                           binding_list(pb)),
                                         hd tel, trans, env,
                                         types, prlet)),
                                     typesMap(
                                       TRFormalFuncParam(
                                         mk_FormalFunctionParameter(
                                           binding_list(pb)),
                                         hd tel, trans, env,
                                         types, prlet)),
```

```
                        letdefList(
                          TRFormalFuncParam(
                            mk_FormalFunctionParameter(
                              binding_list(pb)),
                            hd tel, trans, env,
                            types, prlet))))),
        mk_FormalFunctionParameter(
          ⟨Make_ProductBinding(
              mk_ProductBinding(
                binding_list(
                  formalfp(
                    TRFormalFuncParam(
                      mk_FormalFunctionParameter(
                        binding_list(pb)),
                      hd tel, trans, env,
                      types, prlet))))))⟩ ^
          binding_list(
            formalfp(
              TRFFPProduct(
                mk_FormalFunctionParameter(
                  tl binding_list(ffpl)),
                tl tel, trans,
                envMap(
                  TRFormalFuncParam(
                    mk_FormalFunctionParameter(
                      binding_list(pb)),
                    hd tel, trans, env,
                    types, prlet)),
                typesMap(
                  TRFormalFuncParam(
                    mk_FormalFunctionParameter(
                      binding_list(pb)),
                    hd tel, trans, env,
                    types, prlet)),
                letdefList(
                  TRFormalFuncParam(
                    mk_FormalFunctionParameter(
                      binding_list(pb)),
                    hd tel, trans, env,
                    types, prlet)))))),
        envMap(
          TRFFPProduct(
            mk_FormalFunctionParameter(
              tl binding_list(ffpl)),
```

```
                  tl tel, trans,
                  envMap(
                     TRFormalFuncParam(
                        mk_FormalFunctionParameter(
                           binding_list(pb)),
                        hd tel, trans, env, types,
                        prlet)),
                  typesMap(
                     TRFormalFuncParam(
                        mk_FormalFunctionParameter(
                           binding_list(pb)),
                        hd tel, trans, env, types,
                        prlet)),
                  letdefList(
                     TRFormalFuncParam(
                        mk_FormalFunctionParameter(
                           binding_list(pb)),
                        hd tel, trans, env, types,
                        prlet)))),
         typesMap(
               TRFFPProduct(
                  mk_FormalFunctionParameter(
                     tl binding_list(ffpl)),
                  tl tel, trans,
                  envMap(
                     TRFormalFuncParam(
                        mk_FormalFunctionParameter(
                           binding_list(pb)),
                        hd tel, trans, env, types,
                        prlet)),
                  typesMap(
                     TRFormalFuncParam(
                        mk_FormalFunctionParameter(
                           binding_list(pb)),
                        hd tel, trans, env, types,
                        prlet)),
                  letdefList(
                     TRFormalFuncParam(
                        mk_FormalFunctionParameter(
                           binding_list(pb)),
                        hd tel, trans, env, types,
                        prlet)))),
         letdefList(
               TRFFPProduct(
```

```
                        mk_FormalFunctionParameter(
                          tl binding_list(ffpl)),
                        tl tel, trans,
                        envMap(
                          TRFormalFuncParam(
                            mk_FormalFunctionParameter(
                              binding_list(pb)),
                            hd tel, trans, env, types,
                            prlet)),
                        typesMap(
                          TRFormalFuncParam(
                            mk_FormalFunctionParameter(
                              binding_list(pb)),
                            hd tel, trans, env, types,
                            prlet)),
                        letdefList(
                          TRFormalFuncParam(
                            mk_FormalFunctionParameter(
                              binding_list(pb)),
                            hd tel, trans, env, types,
                            prlet)))))
          end
        end,
      _ →
        mk_BOOL_FFP_ENV_TYPES_LDL(
          and(
            bool(
              TRFormalFuncParam(
                mk_FormalFunctionParameter(
                  ⟨hd binding_list(ffpl)⟩),
                hd tel, trans, env, types, prlet)),
            bool(
              TRFFPProduct(
                mk_FormalFunctionParameter(
                  tl binding_list(ffpl)), tl tel,
                trans,
                envMap(
                  TRFormalFuncParam(
                    mk_FormalFunctionParameter(
                      ⟨hd binding_list(ffpl)⟩),
                    hd tel, trans, env, types,
                    prlet)),
                typesMap(
                  TRFormalFuncParam(
```

```
                              mk_FormalFunctionParameter(
                                 ⟨hd binding_list(ffpl)⟩),
                              hd tel, trans, env, types,
                              prlet)),
                          letdefList(
                            TRFormalFuncParam(
                              mk_FormalFunctionParameter(
                                 ⟨hd binding_list(ffpl)⟩),
                              hd tel, trans, env, types,
                              prlet))))),
                mk_FormalFunctionParameter(
                      binding_list(
                          formalfp(
                              TRFormalFuncParam(
                                mk_FormalFunctionParameter(
                                   ⟨hd binding_list(ffpl)⟩),
                                hd tel, trans, env, types, prlet
                              ))) ^
                      binding_list(
                          formalfp(
                              TRFFPProduct(
                                mk_FormalFunctionParameter(
                                   tl binding_list(ffpl)),
                                tl tel, trans,
                                envMap(
                                   TRFormalFuncParam(
                                      mk_FormalFunctionParameter(
                                         ⟨hd binding_list(ffpl)
                                         ⟩), hd tel, trans,
                                      env, types, prlet)),
                                typesMap(
                                   TRFormalFuncParam(
                                      mk_FormalFunctionParameter(
                                         ⟨hd binding_list(ffpl)
                                         ⟩), hd tel, trans,
                                      env, types, prlet)),
                                letdefList(
                                   TRFormalFuncParam(
                                      mk_FormalFunctionParameter(
                                         ⟨hd binding_list(ffpl)
                                         ⟩), hd tel, trans,
                                      env, types, prlet)))))),
                envMap(
                      TRFFPProduct(
```

```
                mk_FormalFunctionParameter(
                    tl binding_list(ffpl)), tl tel,
                trans,
                envMap(
                    TRFormalFuncParam(
                        mk_FormalFunctionParameter(
                            ⟨hd binding_list(ffpl)⟩),
                        hd tel, trans, env, types, prlet
                    )),
                typesMap(
                    TRFormalFuncParam(
                        mk_FormalFunctionParameter(
                            ⟨hd binding_list(ffpl)⟩),
                        hd tel, trans, env, types, prlet
                    )),
                letdefList(
                    TRFormalFuncParam(
                        mk_FormalFunctionParameter(
                            ⟨hd binding_list(ffpl)⟩),
                        hd tel, trans, env, types, prlet
                    )))),
        typesMap(
            TRFFPProduct(
                mk_FormalFunctionParameter(
                    tl binding_list(ffpl)), tl tel,
                trans,
                envMap(
                    TRFormalFuncParam(
                        mk_FormalFunctionParameter(
                            ⟨hd binding_list(ffpl)⟩),
                        hd tel, trans, env, types, prlet
                    )),
                typesMap(
                    TRFormalFuncParam(
                        mk_FormalFunctionParameter(
                            ⟨hd binding_list(ffpl)⟩),
                        hd tel, trans, env, types, prlet
                    )),
                letdefList(
                    TRFormalFuncParam(
                        mk_FormalFunctionParameter(
                            ⟨hd binding_list(ffpl)⟩),
                        hd tel, trans, env, types, prlet
                    )))),
```

```
                letdefList(
                    TRFFPProduct(
                        mk_FormalFunctionParameter(
                            tl binding_list(ffpl)), tl tel,
                        trans,
                        envMap(
                            TRFormalFuncParam(
                              mk_FormalFunctionParameter(
                                ⟨hd binding_list(ffpl)⟩),
                              hd tel, trans, env, types, prlet
                            )),
                        typesMap(
                            TRFormalFuncParam(
                              mk_FormalFunctionParameter(
                                ⟨hd binding_list(ffpl)⟩),
                              hd tel, trans, env, types, prlet
                            )),
                        letdefList(
                            TRFormalFuncParam(
                              mk_FormalFunctionParameter(
                                ⟨hd binding_list(ffpl)⟩),
                              hd tel, trans, env, types, prlet
                            )))))
      end
   end,

makeProductLet :
   TypeExpr × Binding × LetDef* × TRANS →
      LetDef*
makeProductLet(te, b, prlet, trans) ≡
   case te of
      TypeName(tn) →
         case b of
            Make_ProductBinding(pb) →
               prlet ⌢
               ⟨mk_LetDef(
                     MakeBinding(b),
                     Make_ValueOrVariableName(
                           mk_ValueOrVariableName(
                             getMapValueTRANS(trans, tn))))⟩,
            _ → prlet
         end
   end,
```

```
TROptPreCondition :
   OptionalPreCondition × ExpType × FUNC × TRANS ×
   TYPES →
      PRECOND_TYPES
TROptPreCondition(precond, et, func, trans, types) ≡
   case precond of
      PreCondition(ve) →
         mk_PRECOND_TYPES(
            PreCondition(
                 valueExpr(
                     TRValueExpr(ve, et, func, trans, types))
            ),
            typesMap(
                 TRValueExpr(ve, et, func, trans, types))),
      NoPreCondition →
         mk_PRECOND_TYPES(NoPreCondition, types)
   end,

TRValueExpr :
   ValueExpr × ExpType × FUNC × TRANS × TYPES →
      VE_TYPES
TRValueExpr(ve, et, func, trans, types) ≡
   case ve of
      Make_ValueLiteral(vl) →
         mk_VE_TYPES(makeAssignExpr(ve, et, trans), types),
      Make_ValueOrVariableName(vn) →
         if
            not(
               isinBinding(
                    IdBinding(id(vn)), domainTYPES(types)))
         then
            if isinId(id(vn), domainFUNC(func))
            then
               TRValueExpr(
                    ApplicationExpr(ve, ⟨⟩), et, func,
                    trans, types)
            else mk_VE_TYPES(ve, types)
            end
         else
            if
               checkTRANS(
                    getMapValueTYPES(types, IdBinding(id(vn))),
                    domainTRANS(trans))
            then
```

```
            /*The value expression is of a type of interest.
            */
            mk_VE_TYPES(
                makeAssignExpr(
                    getTRANS(
                        getMapValueTYPES(
                          types, IdBinding(id(vn))), trans),
                    et, trans), types)
        else
            /*The value expression is not of a type of interest.
            */
            mk_VE_TYPES(
                makeAssignExpr(ve, et, trans), types)
        end
    end,
  Make_BasicExpr(be) → mk_VE_TYPES(ve, types),
  ProductExpr(vel) →
    if
      containsGen(
        vel,
        expTypeToExpTypeList(
            removeBrackets(et), lengthVE(vel)),
        trans, func)
    then
      mk_VE_TYPES(
        TRProductExpr(
            valueExprList(
                TRValueExprListProductFunc(
                    vel,
                    expTypeToExpTypeList(
                      removeBrackets(et),
                      lengthVE(vel)), func, trans,
                    types)),
            expTypeToExpTypeList(
                removeBrackets(et), lengthVE(vel)),
            et, func, trans, types),
        typesMap(
            TRValueExprListProductFunc(
                vel,
                expTypeToExpTypeList(
                    removeBrackets(et), lengthVE(vel)),
                func, trans, types)))
    else
      mk_VE_TYPES(
```

```
            makeAssignExpr(
                ProductExpr(
                    valueExprList(
                        TRValueExprListProductFunc(
                          vel,
                          expTypeToExpTypeList(
                            removeBrackets(et),
                            lengthVE(vel)), func, trans,
                          types))), et, trans),
            typesMap(
                TRValueExprListProductFunc(
                    vel,
                    expTypeToExpTypeList(
                        removeBrackets(et), lengthVE(vel)),
                    func, trans, types)))
        end,
    Make_SetExpr(se) →
        mk_VE_TYPES(
            makeAssignExpr(
                Make_SetExpr(
                    setExpr(
                        TRSetExpr(
                          se, getSetType(et), func, trans,
                          types))), et, trans),
            typesMap(
                TRSetExpr(
                    se, getSetType(et), func, trans, types))
        ),
    Make_ListExpr(le) →
        mk_VE_TYPES(
            makeAssignExpr(
                Make_ListExpr(
                    listExpr(
                        TRListExpr(
                          le, getListType(et), func, trans,
                          types))), et, trans),
            typesMap(
                TRListExpr(
                    le, getListType(et), func, trans, types)
            )),
    Make_MapExpr(me) →
        mk_VE_TYPES(
            makeAssignExpr(
                Make_MapExpr(
```

```
                mapExpr(
                    TRMapExpr(
                      me, getMapType(et), func, trans,
                      types))), et, trans),
        typesMap(
            TRMapExpr(
                me, getMapType(et), func, trans, types))
    ),
ApplicationExpr(ave, vel) →
    if not(isTGen(ve, trans, func))
    then
        mk_VE_TYPES(
            makeAssignExpr(
                valueExpr(
                    TRApplicationExpr(
                        ave, vel, et, func, trans, types)),
                et, trans),
            typesMap(
                TRApplicationExpr(
                    ave, vel, et, func, trans, types)))
    else
        TRApplicationExpr(
            ave, vel, et, func, trans, types)
    end,
BracketedExpr(bve) →
    mk_VE_TYPES(
        makeAssignExpr(
            BracketedExpr(
                valueExpr(
                    TRValueExpr(
                      bve, getBracketedType(et), func,
                      trans, types))), et, trans),
        typesMap(
            TRValueExpr(
                bve, getBracketedType(et), func, trans,
                types))),
ValueInfixExpr(first, op, second) →
    mk_VE_TYPES(
        makeAssignExpr(
            ValueInfixExpr(
                valueExpr(
                    TRValueExpr(
                      first, Unknown, func, trans, types
                    )), op,
```

```
                    valueExpr(
                        TRValueExpr(
                          second, Unknown, func, trans,
                          typesMap(
                            TRValueExpr(
                              first, Unknown, func,
                              trans, types))))), et, trans
            ),
            typesMap(
                TRValueExpr(
                    second, Unknown, func, trans,
                    typesMap(
                        TRValueExpr(
                          first, Unknown, func, trans, types
                        ))))),
      ValuePrefixExpr(op, operand) →
        mk_VE_TYPES(
            makeAssignExpr(
                ValuePrefixExpr(
                    op,
                    valueExpr(
                        TRValueExpr(
                          operand, Unknown, func, trans,
                          types))), et, trans),
            typesMap(
                TRValueExpr(
                    operand, Unknown, func, trans, types))),
      LetExpr(ldl, lve) →
        mk_VE_TYPES(
            LetExpr(
                letdefList(
                    TRLetDefList(ldl, func, trans, types)),
                valueExpr(
                    TRValueExpr(
                        lve, et, func, trans,
                        typesMap(
                          TRLetDefList(
                            ldl, func, trans, types))))),
            typesMap(
                TRValueExpr(
                    lve, et, func, trans,
                    typesMap(
                        TRLetDefList(ldl, func, trans, types)
                    )))),
```

```
            Make_IfExpr(ie) →
              TRIfExpr(ie, et, func, trans, types),
            CaseExpr(cond, cbl) →
              mk_VE_TYPES(
                CaseExpr(
                    valueExpr(
                      TRValueExpr(
                          cond, Unknown, func, trans, types)),
                    caseBranchList(
                      TRCaseBranchList(
                          cbl, et, func, trans,
                          typesMap(
                            TRValueExpr(
                              cond, Unknown, func, trans,
                              types))))),
                typesMap(
                    TRCaseBranchList(
                      cbl, et, func, trans,
                      typesMap(
                          TRValueExpr(
                            cond, Unknown, func, trans, types)
                      ))))
          end,

      TRValueExprList :
          ValueExpr* × ExpType × FUNC × TRANS × TYPES →
            VEL_TYPES
      TRValueExprList(vel, et, func, trans, types) ≡
          if vel = ⟨⟩ then mk_VEL_TYPES(⟨⟩, types)
          else
            mk_VEL_TYPES(
              ⟨getValueExpr(
                    TRValueExpr(
                      hd vel, getHead(et), func, trans, types)
                )⟩ ^
              valueExprList(
                  TRValueExprList(
                      tl vel, getTail(et), func, trans,
                      typesMap(
                          TRValueExpr(
                              hd vel, getHead(et), func, trans,
                              types)))),
              typesMap(
                  TRValueExprList(
```

```
                    tl vel, getTail(et), func, trans,
                    typesMap(
                        TRValueExpr(
                            hd vel, getHead(et), func, trans,
                            types)))))
    end,

TRValueExprListProductFunc :
    ValueExpr* × ExpType* × FUNC × TRANS ×
    TYPES →
        VEL_TYPES
TRValueExprListProductFunc(
    vel, etl, func, trans, types) ≡
    if vel = ⟨⟩ then mk_VEL_TYPES(⟨⟩, types)
    else
        mk_VEL_TYPES(
            concatVE(
                valueExpr(
                    TRValueExpr(
                        hd vel, hd etl, func, trans, types)),
                valueExprList(
                    TRValueExprListProductFunc(
                        tl vel, tl etl, func, trans,
                        typesMap(
                            TRValueExpr(
                                hd vel, hd etl, func, trans,
                                types))))),
            typesMap(
                TRValueExprListProductFunc(
                    tl vel, tl etl, func, trans,
                    typesMap(
                        TRValueExpr(
                            hd vel, hd etl, func, trans, types))
                )))
    end,

TRProductExpr :
    ValueExpr* × ExpType* × ExpType × FUNC ×
    TRANS × TYPES →
        ValueExpr
TRProductExpr(vel, etl, et, func, trans, types) ≡
    LetExpr(
        ⟨mk_LetDef(
                MakeBinding(
```

```
                    makeBinding(
                        bindingList(
                            makeLetBinding(
                              lengthVE(vel), 0, ⟨⟩, ⟨⟩)))),
                 ProductExpr(vel))⟩,
           makeAssignExpr(
               ProductExpr(
                  makeValueExprList(
                      vel, etl,
                      valueExprList(
                         makeLetBinding(
                             lengthVE(vel), 0, ⟨⟩, ⟨⟩)),
                      trans, func)), et, trans)),

TRSetExpr :
  SetExpr × ExpType × FUNC × TRANS × TYPES →
    SE_TYPES
TRSetExpr(se, et, func, trans, types) ≡
  case se of
    RangedSetExpr(fve, sve) →
      mk_SE_TYPES(
         RangedSetExpr(
             valueExpr(
                TRValueExpr(fve, et, func, trans, types)
             ),
             valueExpr(
                TRValueExpr(
                    sve, et, func, trans,
                    typesMap(
                      TRValueExpr(
                        fve, et, func, trans, types))))
         ),
         typesMap(
             TRValueExpr(
                sve, et, func, trans,
                typesMap(
                    TRValueExpr(
                      fve, et, func, trans, types))))),
    EnumeratedSetExpr(ovel) →
      mk_SE_TYPES(
         EnumeratedSetExpr(
             optValueExprList(
                TROptValueExprList(
                    ovel, et, func, trans, types))),
```

```
                typesMap(
                      TROptValueExprList(
                          ovel, et, func, trans, types))),
        ComprehendedSetExpr(ve, typlist, or) →
            mk_SE_TYPES(
                ComprehendedSetExpr(
                      valueExpr(
                          TRValueExpr(ve, et, func, trans, types)),
                      typlist,
                      optRestriction(
                          TROptRestriction(
                                or, Known(TypeLiteral(BOOL)), func,
                                trans,
                                typesMap(
                                   TRValueExpr(
                                      ve, et, func, trans, types))))),
                typesMap(
                      TROptRestriction(
                          or, Known(TypeLiteral(BOOL)), func,
                          trans,
                          typesMap(
                                TRValueExpr(
                                   ve, et, func, trans, types)))))
    end,

TRListExpr :
    ListExpr × ExpType × FUNC × TRANS × TYPES →
        LE_TYPES
TRListExpr(le, et, func, trans, types) ≡
    case le of
        RangedListExpr(fve, sve) →
            mk_LE_TYPES(
                RangedListExpr(
                      valueExpr(
                          TRValueExpr(fve, et, func, trans, types)
                      ),
                      valueExpr(
                          TRValueExpr(
                                sve, et, func, trans,
                                typesMap(
                                   TRValueExpr(
                                      fve, et, func, trans, types))))
                ),
                typesMap(
```

```
                TRValueExpr(
                    sve, et, func, trans,
                    typesMap(
                        TRValueExpr(
                          fve, et, func, trans, types))))),
EnumeratedListExpr(ovel) →
   mk_LE_TYPES(
      EnumeratedListExpr(
          optValueExprList(
             TROptValueExprList(
                 ovel, et, func, trans, types))),
      typesMap(
          TROptValueExprList(
             ovel, et, func, trans, types))),
ComprehendedListExpr(ve1, b, ve2, or) →
   mk_LE_TYPES(
      ComprehendedListExpr(
          valueExpr(
             TRValueExpr(ve1, et, func, trans, types)
          ), b,
          valueExpr(
             TRValueExpr(
                 ve2, et, func, trans,
                 typesMap(
                   TRValueExpr(
                     ve1, et, func, trans, types)))),
          optRestriction(
             TROptRestriction(
                 or, Known(TypeLiteral(BOOL)), func,
                 trans,
                 typesMap(
                   TRValueExpr(
                     ve2, et, func, trans,
                     typesMap(
                       TRValueExpr(
                         ve1, et, func, trans,
                         types))))))),
      typesMap(
          TROptRestriction(
             or, Known(TypeLiteral(BOOL)), func,
             trans,
             typesMap(
                 TRValueExpr(
                   ve2, et, func, trans,
```

```
                              typesMap(
                                 TRValueExpr(
                                    ve1, et, func, trans, types)
                              ))))))
   end,

TRMapExpr :
   MapExpr × MapType × FUNC × TRANS × TYPES →
      ME_TYPES
TRMapExpr(me, et, func, trans, types) ≡
   case me of
      EnumeratedMapExpr(ovel) →
         mk_ME_TYPES(
            EnumeratedMapExpr(
                 optValueExprPairList(
                    TROptValueExprPairList(
                         ovel, et, func, trans, types))),
            typesMap(
                 TROptValueExprPairList(
                    ovel, et, func, trans, types))),
      ComprehendedMapExpr(ve, typlist, or) →
         mk_ME_TYPES(
            ComprehendedMapExpr(
                 mk_ValueExprPair(
                    valueExpr(
                         TRValueExpr(
                           first(ve), tedom(et), func,
                           trans, types)),
                    valueExpr(
                         TRValueExpr(
                           second(ve), terange(et), func,
                           trans,
                           typesMap(
                              TRValueExpr(
                                 first(ve), tedom(et), func,
                                 trans, types))))), typlist,
                 optRestriction(
                    TROptRestriction(
                         or, Known(TypeLiteral(BOOL)), func,
                         trans,
                         typesMap(
                           TRValueExpr(
                              second(ve), terange(et), func,
                              trans,
```

```
                                    typesMap(
                                      TRValueExpr(
                                        first(ve), tedom(et),
                                        func, trans, types))))))),
                   typesMap(
                        TROptRestriction(
                           or, Known(TypeLiteral(BOOL)), func,
                           trans,
                           typesMap(
                                TRValueExpr(
                                  second(ve), terange(et), func,
                                  trans,
                                  typesMap(
                                    TRValueExpr(
                                      first(ve), tedom(et), func,
                                      trans, types)))))))
          end,

       TROptValueExprList :
          OptionalValueExprList × ExpType × FUNC × TRANS ×
          TYPES →
             OVEL_TYPES
       TROptValueExprList(ovel, et, func, trans, types) ≡
          case ovel of
             ValueExprList(vel) →
                mk_OVEL_TYPES(
                   ValueExprList(
                        valueExprList(
                           TRValueExprListOpt(
                                vel, et, func, trans, types))),
                   typesMap(
                        TRValueExprListOpt(
                           vel, et, func, trans, types))),
             NoValueExprList → mk_OVEL_TYPES(ovel, types)
          end,

       TROptValueExprPairList :
          OptionalValueExprPairList × MapType × FUNC ×
          TRANS × TYPES →
             OVEPL_TYPES
       TROptValueExprPairList(ovel, et, func, trans, types) ≡
          case ovel of
             ValueExprPairList(vel) →
                mk_OVEPL_TYPES(
```

```
            ValueExprPairList(
                valueExprPairList(
                    TRValueExprPairListOpt(
                        vel, et, func, trans, types))),
            typesMap(
                TRValueExprPairListOpt(
                    vel, et, func, trans, types))),
        NoValueExprPairList → mk_OVEPL_TYPES(ovel, types)
    end,

TRValueExprListOpt :
    ValueExpr* × ExpType × FUNC × TRANS × TYPES →
        VEL_TYPES
TRValueExprListOpt(vel, et, func, trans, types) ≡
    if vel = ⟨⟩ then mk_VEL_TYPES(⟨⟩, types)
    else
        mk_VEL_TYPES(
            ⟨getValueExpr(
                    TRValueExpr(hd vel, et, func, trans, types)
                )⟩ ^
            valueExprList(
                TRValueExprList(
                    tl vel, et, func, trans,
                    typesMap(
                        TRValueExpr(
                            hd vel, et, func, trans, types)))),
            typesMap(
                TRValueExprList(
                    tl vel, et, func, trans,
                    typesMap(
                        TRValueExpr(
                            hd vel, et, func, trans, types)))))
    end,

TRValueExprPairListOpt :
    ValueExprPair* × MapType × FUNC × TRANS ×
    TYPES →
        VEPL_TYPES
TRValueExprPairListOpt(vel, et, func, trans, types) ≡
    if vel = ⟨⟩ then mk_VEPL_TYPES(⟨⟩, types)
    else
        mk_VEPL_TYPES(
            ⟨mk_ValueExprPair(
                    valueExpr(
```

```
                    TRValueExpr(
                        first(hd vel), tedom(et), func,
                        trans, types)),
                  valueExpr(
                    TRValueExpr(
                        second(hd vel), terange(et), func,
                        trans,
                        typesMap(
                          TRValueExpr(
                            first(hd vel), tedom(et),
                            func, trans, types))))))⟩ ⌢
            valueExprPairList(
                TRValueExprPairListOpt(
                  tl vel, et, func, trans,
                  typesMap(
                      TRValueExpr(
                        second(hd vel), terange(et), func,
                        trans,
                        typesMap(
                          TRValueExpr(
                            first(hd vel), tedom(et),
                            func, trans, types))))))),
            typesMap(
                TRValueExprPairListOpt(
                  tl vel, et, func, trans,
                  typesMap(
                      TRValueExpr(
                        second(hd vel), terange(et), func,
                        trans,
                        typesMap(
                          TRValueExpr(
                            first(hd vel), tedom(et),
                            func, trans, types)))))))
    end,

  TROptRestriction :
    OptionalRestriction × ExpType × FUNC × TRANS ×
    TYPES →
      OR_TYPES
  TROptRestriction(optr, et, func, trans, types) ≡
    case optr of
      Restriction(ve) →
        mk_OR_TYPES(
          Restriction(
```

```
                valueExpr(
                    TRValueExpr(ve, et, func, trans, types))
            ),
            typesMap(
                TRValueExpr(ve, et, func, trans, types))),
      NoRestriction → mk_OR_TYPES(optr, types)
   end,

TRApplicationExpr :
   ValueExpr × ValueExpr* × ExpType × FUNC ×
   TRANS × TYPES →
      VE_TYPES
TRApplicationExpr(ve, vel, et, func, trans, types) ≡
   case ve of
      Make_ValueOrVariableName(vn) →
         if isinId(id(vn), domainFUNC(func))
         then
            /*The value expression is a function application
            expression.*/
            case
               type_expr(getMapValueFUNC(func, id(vn)))
            of
               FunctionTypeExpr(arg, fa, res) →
                  if
                     and(
                        not(
                           onlyTOIArgument(vel, trans, types)),
                        (lengthVE(vel) = 1))
                  then
                     mk_VE_TYPES(
                        makeSequencingExpr(
                           ApplicationExpr(
                              ve,
                              valueExprList(
                                 TRValueExprListProductFunc(
                                    vel,
                                    expTypeList(
                                       arg, lengthVE(vel)),
                                    func, trans, types))),
                           type_expr(res), et, trans, func),
                        typesMap(
                           TRValueExprListProductFunc(
                              vel,
                              expTypeList(arg, lengthVE(vel)),
```

```
                    func, trans, types)))
else
  if
    and(
      containsGen(
        vel, typeExprToExpTypeList(arg),
        trans, func),
      not(
        onlyTOIArgument(
          vel, trans, types)))
  then
    mk_VE_TYPES(
      makeSequencingExpr(
        TRFunctionAppl(
          ve,
          valueExprList(
            TRValueExprListProductFunc(
              vel,
              expTypeList(
                arg, lengthVE(vel)),
              func, trans, types)),
          typeExprToExpTypeList(arg),
          func, trans, types),
        type_expr(res), et, trans, func),
      typesMap(
        TRValueExprListProductFunc(
          vel,
          expTypeList(
            arg, lengthVE(vel)), func,
          trans, types)))
  else
    mk_VE_TYPES(
      makeSequencingExpr(
        ApplicationExpr(
          ve,
          removeTOI(
            valueExprList(
              TRValueExprListProductFunc(
                vel,
                expTypeList(
                  arg,
                  lengthVE(vel)),
                func, trans, types)),
            typeExprToExpTypeList(arg),
```

```
                                        trans)), type_expr(res),
                                    et, trans, func),
                                  typesMap(
                                    TRValueExprListProductFunc(
                                      vel,
                                      expTypeList(
                                        arg, lengthVE(vel)), func,
                                      trans, types)))
                            end
                          end
                      end
                    else
                      /*List or map application.*/
                      TRListMapAppl(ve, vel, et, func, trans, types)
                    end,
                _ →
                    TRListMapAppl(ve, vel, et, func, trans, types)
            end,

        TRFunctionAppl :
            ValueExpr × ValueExpr* × ExpType* ×
            FUNC × TRANS × TYPES →
                ValueExpr
        TRFunctionAppl(ve, vel, etl, func, trans, types) ≡
            ApplicationExpr(
                ve,
                ⟨LetExpr(
                        ⟨mk_LetDef(
                                MakeBinding(
                                    makeBinding(
                                      bindingList(
                                        makeLetBinding(
                                          lengthVE(vel), 0, ⟨⟩, ⟨⟩
                                        )))), ProductExpr(vel))⟩,
                        ProductExpr(
                            makeValueExprList(
                                vel, etl,
                                valueExprList(
                                    makeLetBinding(
                                      lengthVE(vel), 0, ⟨⟩, ⟨⟩)),
                                trans, func))))⟩),

        removeTOI :
            ValueExpr* × ExpType* × TRANS →
```

```
        ValueExpr*
    removeTOI(vel, etl, trans) ≡
      if vel = ⟨⟩ then ⟨⟩
      else
        case hd etl of
          Known(te) →
            case te of
              TypeName(tn) →
                if isinId(tn, domainTRANS(trans))
                then
                  /*Type of interest.*/
                  removeTOI(tl vel, tl etl, trans)
                else
                  /*Not type of interest.*/
                  ⟨hd vel⟩ ^
                  removeTOI(tl vel, tl etl, trans)
                end,
              _ →
                ⟨hd vel⟩ ^
                removeTOI(tl vel, tl etl, trans)
            end,
          Unknown →
            ⟨hd vel⟩ ^ removeTOI(tl vel, tl etl, trans)
        end
      end,

    TRListMapAppl :
      ValueExpr × ValueExpr* × ExpType × FUNC ×
      TRANS × TYPES →
        VE_TYPES
    TRListMapAppl(ve, vel, et, func, trans, types) ≡
      mk_VE_TYPES(
         ApplicationExpr(
            valueExpr(
               TRValueExpr(
                  ve, et, func, trans,
                  typesMap(
                     TRValueExprList(
                        vel, et, func, trans, types)))),
            valueExprList(
               TRValueExprList(vel, et, func, trans, types)
            )),
         overrideTYPES(
            typesMap(
```

```
                TRValueExprList(vel, et, func, trans, types)
            ),
            typesMap(
                TRValueExpr(
                    ve, et, func, trans,
                    typesMap(
                        TRValueExprList(
                            vel, et, func, trans, types)))))),

TRLetDefList :
   LetDef* × FUNC × TRANS × TYPES → LDL_TYPES
TRLetDefList(ldl, func, trans, types) ≡
   if ldl = ⟨⟩ then mk_LDL_TYPES(ldl, types)
   else
      mk_LDL_TYPES(
         makeLetDefList(
              letdef(TRLetDef(hd ldl, func, trans, types))) ^
         letdefList(
              TRLetDefList(tl ldl, func, trans, types)),
         overrideTYPES(
              typesMap(
                  TRLetDef(hd ldl, func, trans, types)),
              typesMap(
                  TRLetDefList(tl ldl, func, trans, types)))
      )
   end,

TRLetDef :
   LetDef × FUNC × TRANS × TYPES → LD_TYPES
TRLetDef(ld, func, trans, types) ≡
   case value_expr(ld) of
      ApplicationExpr(ve, vel) →
         case ve of
            Make_ValueOrVariableName(vn) →
               if isinId(id(vn), domainFUNC(func))
               then
                  /*Function application.*/
                  case
                     type_expr(getMapValueFUNC(func, id(vn)))
                  of
                     FunctionTypeExpr(arg, fa, res) →
                        if
                           isGen(
                             ve,
```

```
        Known(
          type_expr(
            getMapValueFUNC(
              func, id(vn)))), trans,
        func)
    then
      if
        not(
          returnsNonTOI(
            ve,
            Known(
              type_expr(
                getMapValueFUNC(
                  func, id(vn)))),
            trans, func))
      then
        mk_LD_TYPES(
          mk_LetDef(
            MakeBinding(
              IdBinding(mk_Id("dummy"))),
            valueExpr(
              TRValueExpr(
                value_expr(ld),
                Known(type_expr(res)),
                func, trans, types))),
          overrideTYPES(
            typesMap(
              TRValueExpr(
                value_expr(ld),
                Known(type_expr(res)),
                func, trans, types)),
            makeTYPES(
              binding(ld),
              Known(type_expr(res)),
              trans)))
      else
        mk_LD_TYPES(
          mk_LetDef(
            TRLetBinding(
              binding(ld),
              type_expr(res), func,
              trans,
              makeTYPES(
                binding(ld),
```

```
                                    Known(type_expr(res)),
                                    trans)),
                                valueExpr(
                                  TRValueExpr(
                                    value_expr(ld),
                                    Known(type_expr(res)),
                                    func, trans, types))),
                              overrideTYPES(
                                typesMap(
                                  TRValueExpr(
                                    value_expr(ld),
                                    Known(type_expr(res)),
                                    func, trans, types)),
                                makeTYPES(
                                  binding(ld),
                                  Known(type_expr(res)),
                                  trans)))
                          end
                        else
                          mk_LD_TYPES(
                            mk_LetDef(
                              binding(ld),
                              valueExpr(
                                TRValueExpr(
                                  value_expr(ld),
                                  Known(type_expr(res)),
                                  func, trans, types))),
                            overrideTYPES(
                              typesMap(
                                TRValueExpr(
                                  value_expr(ld),
                                  Known(type_expr(res)),
                                  func, trans, types)),
                              makeTYPES(
                                binding(ld),
                                Known(type_expr(res)), trans
                              )))
                        end
                  end
                else  /*List or map application.*/
                  mk_LD_TYPES(
                      mk_LetDef(
                            binding(ld),
                            valueExpr(
```

```
                              TRValueExpr(
                                value_expr(ld), Unknown, func,
                                trans, types))),
                      overrideTYPES(
                            typesMap(
                              TRValueExpr(
                                value_expr(ld), Unknown, func,
                                trans, types)),
                            makeTYPES(
                              binding(ld), Unknown, trans)))
                end,
              _ →
                mk_LD_TYPES(
                    mk_LetDef(
                          binding(ld),
                          valueExpr(
                              TRValueExpr(
                                value_expr(ld), Unknown, func,
                                trans, types))),
                    overrideTYPES(
                          typesMap(
                              TRValueExpr(
                                value_expr(ld), Unknown, func,
                                trans, types)),
                          makeTYPES(binding(ld), Unknown, trans))
                )
          end,
        _ →
          mk_LD_TYPES(
              mk_LetDef(
                    binding(ld),
                    valueExpr(
                        TRValueExpr(
                              value_expr(ld), Unknown, func,
                              trans, types))),
              overrideTYPES(
                    typesMap(
                        TRValueExpr(
                              value_expr(ld), Unknown, func,
                              trans, types)),
                    makeTYPES(binding(ld), Unknown, trans)))
    end,

TRLetBinding :
```

```
    LetBinding × TypeExpr × FUNC × TRANS × TYPES →
      LetBinding
  TRLetBinding(lb, te, func, trans, types) ≡
    case lb of
      MakeBinding(b) →
        MakeBinding(makeBinding(TRBinding(b, te, trans))),
      MakeRecordPattern(vn, pl) →
        MakeRecordPattern(
          vn,
          patternList(
            TRPatternList(pl, func, trans, types))),
      MakeListPatternLet(lp) →
        MakeListPatternLet(
          listPattern(
            TRListPattern(lp, func, trans, types)))
    end,

  TRBinding :
    Binding × TypeExpr × TRANS → Binding*
  TRBinding(b, te, trans) ≡
    case b of
      IdBinding(id) →
        case te of
          TypeName(tn) →
            if isinId(tn, domainTRANS(trans)) then ⟨⟩
            else ⟨b⟩
            end,
          _ → ⟨b⟩
        end,
      Make_ProductBinding(pb) →
        TRBindingList(
          binding_list(pb), typeExprToList(te), trans)
    end,

  TRBindingList :
    Binding* × TypeExpr* × TRANS →
      Binding*
  TRBindingList(bl, tel, trans) ≡
    if bl = ⟨⟩ then ⟨⟩
    else
      TRBinding(hd bl, hd tel, trans) ⌢
      TRBindingList(tl bl, tl tel, trans)
    end,
```

```
TRIfExpr :
   IfExpr × ExpType × FUNC × TRANS × TYPES →
      VE_TYPES
TRIfExpr(ie, et, func, trans, types) ≡
   mk_VE_TYPES(
      Make_IfExpr(
         mk_IfExpr(
            valueExpr(
               TRValueExpr(
                  condition(ie),
                  Known(TypeLiteral(BOOL)), func,
                  trans, types)),
            valueExpr(
               TRValueExpr(
                  if_case(ie), et, func, trans,
                  typesMap(
                     TRValueExpr(
                        condition(ie),
                        Known(TypeLiteral(BOOL)), func,
                        trans, types)))),
            elsIfList(
               TRElsif(
                  elsif_list(ie), et, func, trans,
                  typesMap(
                     TRValueExpr(
                        if_case(ie), et, func, trans,
                        typesMap(
                           TRValueExpr(
                              condition(ie),
                              Known(TypeLiteral(BOOL)),
                              func, trans, types)))))),
            valueExpr(
               TRValueExpr(
                  else_case(ie), et, func, trans,
                  typesMap(
                     TRElsif(
                        elsif_list(ie), et, func,
                        trans,
                        typesMap(
                           TRValueExpr(
                              if_case(ie), et, func,
                              trans,
                              typesMap(
                                 TRValueExpr(
```

```
                                                  condition(ie),
                                                  Known(
                                                    TypeLiteral(BOOL)
                                                  ), func, trans,
                                                  types))))))))),
          typesMap(
               TRValueExpr(
                    else_case(ie), et, func, trans,
                    typesMap(
                         TRElsif(
                              elsif_list(ie), et, func, trans,
                              typesMap(
                                   TRValueExpr(
                                     if_case(ie), et, func, trans,
                                     typesMap(
                                       TRValueExpr(
                                         condition(ie),
                                         Known(TypeLiteral(BOOL)),
                                         func, trans, types))))))))
     ),

   TRElsif :
     Elsif* × ExpType × FUNC × TRANS × TYPES →
        EIL_TYPES
   TRElsif(eil, et, func, trans, types) ≡
     if eil = ⟨⟩ then mk_EIL_TYPES(⟨⟩, types)
     else
        mk_EIL_TYPES(
          ⟨mk_Elsif(
                 valueExpr(
                      TRValueExpr(
                           condition(hd eil),
                           Known(TypeLiteral(BOOL)), func,
                           trans, types)),
                 valueExpr(
                      TRValueExpr(
                           elsif_case(hd eil), et, func, trans,
                           typesMap(
                             TRValueExpr(
                               condition(hd eil),
                               Known(TypeLiteral(BOOL)),
                               func, trans, types))))))⟩ ⌢
          elsIfList(
               TRElsif(
```

```
                    tl eil, et, func, trans,
                    typesMap(
                        TRValueExpr(
                            elsif_case(hd eil), et, func,
                            trans,
                            typesMap(
                              TRValueExpr(
                                condition(hd eil),
                                Known(TypeLiteral(BOOL)),
                                func, trans, types)))))),
              typesMap(
                  TRElsif(
                    tl eil, et, func, trans,
                    typesMap(
                        TRValueExpr(
                            elsif_case(hd eil), et, func,
                            trans,
                            typesMap(
                              TRValueExpr(
                                condition(hd eil),
                                Known(TypeLiteral(BOOL)),
                                func, trans, types)))))))
      end,

    TRCaseBranchList :
      CaseBranch* × ExpType × FUNC × TRANS × TYPES →
        CBL_TYPES
    TRCaseBranchList(cbl, et, func, trans, types) ≡
      if cbl = ⟨⟩ then mk_CBL_TYPES(cbl, types)
      else
        mk_CBL_TYPES(
            ⟨mk_CaseBranch(
                  pattern(
                      TRPattern(
                          pattern(hd cbl), func, trans, types)),
                  valueExpr(
                      TRValueExpr(
                          value_expr(hd cbl), et, func, trans,
                          typesMap(
                            TRPattern(
                              pattern(hd cbl), func, trans,
                              types))))))⟩ ⌢
            caseBranchList(
                TRCaseBranchList(
```

```
                    tl cbl, et, func, trans,
                    typesMap(
                         TRValueExpr(
                              value_expr(hd cbl), et, func,
                              trans,
                              typesMap(
                                 TRPattern(
                                    pattern(hd cbl), func, trans,
                                    types)))))),
           typesMap(
                TRCaseBranchList(
                    tl cbl, et, func, trans,
                    typesMap(
                         TRValueExpr(
                              value_expr(hd cbl), et, func,
                              trans,
                              typesMap(
                                 TRPattern(
                                    pattern(hd cbl), func, trans,
                                    types)))))))
   end,

TRPattern :
   Pattern × FUNC × TRANS × TYPES → PATTERN_TYPES
TRPattern(p, func, trans, types) ≡
   case p of
      ValueLiteralPattern(vl) →
         mk_PATTERN_TYPES(p, types),
      NamePattern(id, optid) →
         if
            not(
                isinBinding(
                      IdBinding(id), domainTYPES(types)))
         then mk_PATTERN_TYPES(p, types)
         else
            if
               checkTRANS(
                    getMapValueTYPES(types, IdBinding(id)),
                    domainTRANS(trans))
            then
               /*The value expression is of the type of interest.
               */
               mk_PATTERN_TYPES(
                    NamePattern(
```

```
                    getTRANSId(
                      getMapValueTYPES(
                        types, IdBinding(id)), trans),
                    optid), types)
            else
              /*The value expression is not of the type of interest.
              */
              mk_PATTERN_TYPES(p, types)
            end
          end,
        WildcardPattern → mk_PATTERN_TYPES(p, types),
        ProductPattern(pl) →
          mk_PATTERN_TYPES(
            ProductPattern(
              patternList(
                TRPatternList(pl, func, trans, types))),
            typesMap(
              TRPatternList(pl, func, trans, types))),
        RecordPattern(vn, pl) →
          mk_PATTERN_TYPES(
            RecordPattern(
              vn,
              patternList(
                TRPatternList(pl, func, trans, types))),
            typesMap(
              TRPatternList(pl, func, trans, types))),
        MakeListPattern(lp) →
          mk_PATTERN_TYPES(
            MakeListPattern(
              listPattern(
                TRListPattern(lp, func, trans, types))),
            typesMap(
              TRListPattern(lp, func, trans, types)))
      end,

    TRPatternList :
      Pattern* × FUNC × TRANS × TYPES → PL_TYPES
    TRPatternList(pl, func, trans, types) ≡
      if pl = ⟨⟩ then mk_PL_TYPES(⟨⟩, types)
      else
        mk_PL_TYPES(
          ⟨getPattern(
              pattern(
                TRPattern(hd pl, func, trans, types)))⟩ ⌢
```

```
            patternList(
                TRPatternList(
                    tl pl, func, trans,
                    typesMap(
                        TRPattern(hd pl, func, trans, types)))),
            typesMap(
                TRPatternList(
                    tl pl, func, trans,
                    typesMap(
                        TRPattern(hd pl, func, trans, types))))
        )
    end,

TRListPattern :
    ListPattern × FUNC × TRANS × TYPES → LP_TYPES
TRListPattern(lp, func, trans, types) ≡
    case lp of
        Make_EnumeratedListPattern(elp) →
            mk_LP_TYPES(
                Make_EnumeratedListPattern(
                    mk_EnumeratedListPattern(
                        optInnerPattern(
                            TROptInnerPattern(
                                inner_pattern(elp), func, trans,
                                types)))),
                typesMap(
                    TROptInnerPattern(
                        inner_pattern(elp), func, trans, types))
            ),
        ConcatenatedListPattern(elp, p) →
            mk_LP_TYPES(
                ConcatenatedListPattern(
                    mk_EnumeratedListPattern(
                        optInnerPattern(
                            TROptInnerPattern(
                                inner_pattern(elp), func, trans,
                                types))),
                    pattern(
                        TRPattern(
                            p, func, trans,
                            typesMap(
                                TROptInnerPattern(
                                    inner_pattern(elp), func,
                                    trans, types))))),
```

```
            typesMap(
                TRPattern(
                    p, func, trans,
                    typesMap(
                        TROptInnerPattern(
                            inner_pattern(elp), func, trans,
                            types)))))
    end,

TROptInnerPattern :
    OptionalInnerPattern × FUNC × TRANS × TYPES →
        OIP_TYPES
TROptInnerPattern(oip, func, trans, types) ≡
    case oip of
        InnerPatternList(pl) →
            mk_OIP_TYPES(
                InnerPatternList(
                    patternList(
                        TRPatternList(pl, func, trans, types))),
                typesMap(
                    TRPatternList(pl, func, trans, types))),
        NoInnerPattern → mk_OIP_TYPES(oip, types)
    end,

makeAssignExpr :
    ValueExpr × ExpType × TRANS → ValueExpr
makeAssignExpr(ve, et, trans) ≡
    case et of
        Known(te) →
            case te of
                TypeName(tn) →
                    if isinId(tn, domainTRANS(trans))
                    then
                        /*Type of interest.*/
                        AssignExpr(getMapValueTRANS(trans, tn), ve)
                    else
                        /*Not type of interest.*/
                        ve
                    end,
                _ → ve
            end,
        Unknown → ve
    end,
```

```
/***** TypeExpr *****/

typeExprToList : TypeExpr → TypeExpr*
typeExprToList(te) ≡
   case te of
      TypeExprProduct(tep) → tep,
      _ → ⟨te⟩
   end,

getLengthTE : TypeExpr → Int
getLengthTE(te) ≡
   case te of
      TypeExprProduct(tep) → getLengthTEL(tep),
      BracketedTypeExpr(bte) → getLengthTE(bte),
      _ → 1
   end,

getLengthTEL : TypeExpr* → Int
getLengthTEL(tel) ≡
   if tel = ⟨⟩ then 0
   else getLengthTE(hd tel) + getLengthTEL(tl tel)
   end,

truncate : TypeExpr → TypeExpr
truncate(tel) ≡
   case tel of
      TypeLiteral(literal) → tel,
      TypeName(id) → tel,
      TypeExprProduct(telp) →
         if truncateTypeExprProduct(telp) = ⟨⟩
         then TypeLiteral(UNIT)
         else
            if lengthTE(truncateTypeExprProduct(telp)) = 1
            then hd truncateTypeExprProduct(telp)
            else
               TypeExprProduct(
                    truncateTypeExprProduct(telp))
            end
         end,
      TypeExprSet(tes) → tel,
      TypeExprList(te) → tel,
      TypeExprMap(tem) → tel,
      FunctionTypeExpr(te, fa, rd) →
         FunctionTypeExpr(
```

```
              truncate(te), fa,
              mk_ResultDesc(
                   read_access_desc(rd),
                   write_access_desc(rd),
                   truncate(type_expr(rd)))),
       BracketedTypeExpr(te) →
          case truncate(te) of
             TypeLiteral(literal) →
                case literal of
                   UNIT → truncate(te),
                   _ → BracketedTypeExpr(truncate(te))
                end,
             _ → BracketedTypeExpr(truncate(te))
          end
    end,

truncateTypeExprProduct :
    TypeExpr* → TypeExpr*
truncateTypeExprProduct(tel) ≡
    if tel = ⟨⟩ then ⟨⟩
    else
       case hd tel of
          TypeLiteral(literal) →
             case literal of
                UNIT → truncateTypeExprProduct(tl tel),
                _ →
                   ⟨truncate(hd tel)⟩ ⌢
                   truncateTypeExprProduct(tl tel)
             end,
          _ →
             ⟨truncate(hd tel)⟩ ⌢
             truncateTypeExprProduct(tl tel)
       end
    end,

equalsType : TypeExpr × ExpType → Bool
equalsType(te, et) ≡
    case et of
       Known(ete) → equalsTypes(te, ete),
       Unknown → false
    end,

equalsTypes : TypeExpr × TypeExpr → Bool
equalsTypes(te1, te2) ≡
```

```
case te1 of
   TypeLiteral(literal1) →
      case te2 of
         TypeLiteral(literal2) → literal1 = literal2,
         _ → false
      end,
   TypeName(tn1) →
      case te2 of
         TypeName(tn2) →
            equalsText(getText(tn1), getText(tn2)),
         _ → false
      end,
   TypeExprProduct(tel1) →
      case te2 of
         TypeExprProduct(tel2) →
            and(
                lengthTE(tel1) = lengthTE(tel2),
                equalsTypesProduct(tel1, tel2)),
         _ → false
      end,
   TypeExprSet(tes1) →
      case te2 of
         TypeExprSet(tes2) →
            equalsTypesSet(tes1, tes2),
         _ → false
      end,
   TypeExprList(tel1) →
      case te2 of
         TypeExprList(tel2) →
            equalsTypesList(tel1, tel2),
         _ → false
      end,
   TypeExprMap(tem1) →
      case te2 of
         TypeExprMap(tem2) →
            equalsTypesMap(tem1, tem2),
         _ → false
      end,
   FunctionTypeExpr(fte1, fa1, rd1) →
      case te2 of
         FunctionTypeExpr(fte2, fa2, rd2) →
            and(
                and(equalsTypes(fte1, fte2), fa1 = fa2),
                equalsTypes(
```

```
                        type_expr(rd1), type_expr(rd2))),
            _ → false
          end,
        BracketedTypeExpr(bte1) →
          case te2 of
            BracketedTypeExpr(bte2) →
              equalsTypes(bte1, bte2),
            _ → false
          end
      end,

  equalsTypesProduct :
      TypeExpr* × TypeExpr* → Bool
  equalsTypesProduct(tel1, tel2) ≡
      if tel1 = ⟨⟩ then true
      else
        and(
          equalsTypes(hd tel1, hd tel2),
          equalsTypesProduct(tl tel1, tl tel2))
      end,

  equalsTypesSet : TypeExprSets × TypeExprSets → Bool
  equalsTypesSet(tes1, tes2) ≡
      case tes1 of
        FiniteSetTypeExpr(te1) →
          case tes2 of
            FiniteSetTypeExpr(te2) →
              equalsTypes(te1, te2),
            _ → false
          end,
        InfiniteSetTypeExpr(te1) →
          case tes2 of
            InfiniteSetTypeExpr(te2) →
              equalsTypes(te1, te2),
            _ → false
          end
      end,

  equalsTypesList :
      TypeExprLists × TypeExprLists → Bool
  equalsTypesList(tel1, tel2) ≡
      case tel1 of
        FiniteListTypeExpr(te1) →
          case tel2 of
```

```
                FiniteListTypeExpr(te2) →
                    equalsTypes(te1, te2),
                _ → false
            end,
        InfiniteListTypeExpr(te1) →
            case tel2 of
                InfiniteListTypeExpr(te2) →
                    equalsTypes(te1, te2),
                _ → false
            end
    end,

equalsTypesMap : TypeExprMaps × TypeExprMaps → Bool
equalsTypesMap(tem1, tem2) ≡
    case tem1 of
        FiniteMapTypeExpr(tedom1, terange1) →
            case tem2 of
                FiniteMapTypeExpr(tedom2, terange2) →
                    and(
                        equalsTypes(tedom1, tedom2),
                        equalsTypes(terange1, terange2)),
                _ → false
            end,
        InfiniteMapTypeExpr(tedom1, terange1) →
            case tem2 of
                InfiniteMapTypeExpr(tedom2, terange2) →
                    and(
                        equalsTypes(tedom1, tedom2),
                        equalsTypes(terange1, terange2)),
                _ → false
            end
    end,

/***** TypeExpr end *****/

/***** ExpType *****/

getHead : ExpType → ExpType
getHead(et) ≡
    case et of
        Known(te) →
            case te of
                TypeExprProduct(tel) →
                    case hd tel of
```

```
                BracketedTypeExpr(bte) → Known(bte),
                _ → Known(hd tel)
              end,
            _ → Unknown
          end,
        Unknown → Unknown
      end,

  getTail : ExpType → ExpType
  getTail(et) ≡
      case et of
        Known(te) →
          case te of
            TypeExprProduct(tel) →
              Known(TypeExprProduct(tl tel)),
            _ → Unknown
          end,
        Unknown → Unknown
      end,

  expTypeList : TypeExpr × Int → ExpType*
  expTypeList(te, length) ≡
      if length = 1 then ⟨Known(te)⟩
      else
        expTypeToExpTypeList(
            removeBrackets(Known(te)), length)
      end,

  expTypeToExpTypeList : ExpType × Int → ExpType*
  expTypeToExpTypeList(et, length) ≡
      if equals(length, 0) then ⟨⟩
      else
        ⟨getHead(et)⟩ ⌢
        expTypeToExpTypeList(getTail(et), minusOne(length))
      end,

  toExpTypeList : ExpType × Int → ExpType*
  toExpTypeList(et, length) ≡
      if equals(length, 0) then ⟨⟩
      else ⟨et⟩ ⌢ toExpTypeList(et, minusOne(length))
      end,

  typeExprToExpTypeList : TypeExpr → ExpType*
  typeExprToExpTypeList(te) ≡
```

```
    case te of
      TypeExprProduct(tep) →
        typeExprListToExpTypeList(tep),
      TypeLiteral(ln) →
        case ln of
          UNIT → ⟨⟩,
          _ → ⟨Known(te)⟩
        end,
      _ → ⟨Known(te)⟩
    end,

  typeExprListToExpTypeList :
    TypeExpr* → ExpType*
  typeExprListToExpTypeList(tel) ≡
    if tel = ⟨⟩ then ⟨⟩
    else
      ⟨Known(hd tel)⟩ ^
      typeExprListToExpTypeList(tl tel)
    end,

  getSetType : ExpType → ExpType
  getSetType(et) ≡
    case et of
      Known(te) →
        case te of
          TypeExprSet(tes) →
            case tes of
              FiniteSetTypeExpr(fse) → Known(fse),
              InfiniteSetTypeExpr(ise) → Known(ise)
            end,
          _ → Unknown
        end,
      _ → Unknown
    end,

  getListType : ExpType → ExpType
  getListType(et) ≡
    case et of
      Known(te) →
        case te of
          TypeExprList(tes) →
            case tes of
              FiniteListTypeExpr(fse) → Known(fse),
              InfiniteListTypeExpr(ise) → Known(ise)
```

```
                end,
            _ → Unknown
        end,
    _ → Unknown
  end,

getMapType : ExpType → MapType
getMapType(et) ≡
  case et of
    Known(te) →
      case te of
        TypeExprMap(tem) →
          case tem of
            FiniteMapTypeExpr(tedom, terange) →
              mk_MapType(Known(tedom), Known(terange)),
            FiniteMapTypeExpr(tedom, terange) →
              mk_MapType(Known(tedom), Known(terange))
          end,
        _ → mk_MapType(Unknown, Unknown)
      end,
    _ → mk_MapType(Unknown, Unknown)
  end,

getBracketedType : ExpType → ExpType
getBracketedType(et) ≡
  case et of
    Known(te) →
      case te of
        BracketedTypeExpr(bte) → Known(bte),
        _ → Unknown
      end,
    Unknown → Unknown
  end,

getListMapTypeArg : ExpType → ExpType
getListMapTypeArg(et) ≡
  case et of
    Known(te) →
      case te of
        TypeExprList(tel) →
          Known(TypeExprProduct(⟨TypeLiteral(INT)⟩)),
        TypeExprMap(tem) →
          case tem of
            FiniteMapTypeExpr(tedom, terange) →
```

```
                    Known(TypeExprProduct(⟨tedom⟩)),
                  InfiniteMapTypeExpr(tedom, terange) →
                    Known(TypeExprProduct(⟨tedom⟩))
                end,
              _ → et
            end,
          Unknown → Unknown
        end,

removeBrackets : ExpType → ExpType
removeBrackets(et) ≡
    case et of
      Known(te) →
        case te of
          BracketedTypeExpr(bte) → Known(bte),
          _ → et
        end,
      _ → et
    end,

/***** ExpType end *****/

/***** ValueExpr *****/

toValueExpr : ValueExpr* → ValueExpr
toValueExpr(vel) ≡
    if lengthVE(vel) = 1 then hd vel
    else ProductExpr(vel)
    end,

makeValueExprList :
    ValueExpr* × ExpType* × ValueExpr* ×
    TRANS × FUNC →
      ValueExpr*
makeValueExprList(vel, etl, vl, trans, func) ≡
    if vel = ⟨⟩ then vel
    else
      if
        and(
          isGen(hd vel, hd etl, trans, func),
          not(
            returnsNonTOI(hd vel, hd etl, trans, func))
        )
      then
```

```
            makeValueExprList(
                 tl vel, tl etl, tl vl, trans, func)
        else
            ⟨hd vl⟩ ⌢
            makeValueExprList(
                 tl vel, tl etl, tl vl, trans, func)
        end
    end,

makeSequencingExpr :
    ValueExpr × TypeExpr × ExpType × TRANS × FUNC →
        ValueExpr
makeSequencingExpr(ve, te, et, trans, func) ≡
    if
        and(
            and(
                isGen(ve, et, trans, func),
                not(returnsNonTOI(ve, et, trans, func))),
            not(equalsType(te, et)))
    then
        BracketedExpr(
            SequencingExpr(
                ve,
                toValueExpr(getTOIReturnsList(te, trans))))
    else ve
    end,

/*∗∗∗∗ ValueExpr end ∗∗∗∗*/

/*∗∗∗∗ Generators and observers ∗∗∗∗*/

containsGen :
    ValueExpr* × ExpType* × TRANS × FUNC →
        Bool
containsGen(vel, etl, trans, func) ≡
    if vel = ⟨⟩ then false
    else
        if
            and(
                isGen(hd vel, hd etl, trans, func),
                not(
                    returnsNonTOI(hd vel, hd etl, trans, func))
            )
        then true
```

```
      else containsGen(tl vel, tl etl, trans, func)
      end
    end,

  isGen : ValueExpr × ExpType × TRANS × FUNC → Bool
  isGen(ve, et, trans, func) ≡
    case ve of
      ApplicationExpr(vea, vel) →
        case vea of
          Make_ValueOrVariableName(vn) →
            not(
              lengthAccess(
                write_list(
                  getMapValueFUNC(func, id(vn)))) = 0),
          _ → false
        end,
      _ →
        case et of
          Known(te) →
            case te of
              TypeName(tn) →
                isinId(tn, domainTRANS(trans)),
              TypeExprProduct(tel) →
                isGenProduct(ve, tel, trans, func),
              FunctionTypeExpr(arg, fa, res) →
                isGen(
                  ve, Known(type_expr(res)), trans, func
                ),
              BracketedTypeExpr(bte) →
                isGen(ve, Known(bte), trans, func),
              _ → false
            end,
          Unknown → false
        end
    end,

  isGenProduct :
    ValueExpr × TypeExpr* × TRANS × FUNC → Bool
  isGenProduct(ve, tel, trans, func) ≡
    if tel = ⟨⟩ then false
    else
      if isGen(ve, Known(hd tel), trans, func) then true
      else isGenProduct(ve, tl tel, trans, func)
      end
```

```
    end,

  isTGen : ValueExpr × TRANS × FUNC → Bool
  isTGen(ve, trans, func) ≡
    case ve of
      ApplicationExpr(vea, vel) →
        case vea of
          Make_ValueOrVariableName(vn) →
            if isinId(id(vn), domainFUNC(func))
            then
              case
                type_expr(getMapValueFUNC(func, id(vn)))
              of
                FunctionTypeExpr(arg, fa, res) →
                  case type_expr(res) of
                    TypeName(tn) →
                     isinId(tn, domainTRANS(trans)),
                    BracketedTypeExpr(bte) →
                     case bte of
                      TypeName(tn) →
                       isinId(tn, domainTRANS(trans)),
                      _ → false
                     end,
                    _ → false
                  end,
                _ → false
              end
            else false
            end,
          _ → false
        end,
      _ → false
    end,

  onlyTOIArgument :
    ValueExpr* × TRANS × TYPES → Bool
  onlyTOIArgument(vel, trans, types) ≡
    if vel = ⟨⟩ then true
    else
      case hd vel of
        Make_ValueOrVariableName(vn) →
          if
            isinBinding(
              IdBinding(id(vn)), domainTYPES(types))
```

```
            then
               if
                  checkTRANS(
                     getMapValueTYPES(
                        types, IdBinding(id(vn))),
                     domainTRANS(trans))
               then onlyTOIArgument(tl vel, trans, types)
               else false
               end
            else false
            end,
         _ → false
      end
   end,

returnsNonTOI :
   ValueExpr × ExpType × TRANS × FUNC → Bool
returnsNonTOI(ve, et, trans, func) ≡
   case ve of
      ApplicationExpr(vea, vel) →
         case vea of
            Make_ValueOrVariableName(vn) →
               case
                  type_expr(getMapValueFUNC(func, id(vn)))
               of
                  FunctionTypeExpr(arg, fa, res) →
                     not(
                        getNonTOIReturnsList(
                           type_expr(res), trans) = ⟨⟩)
               end,
            _ → true
         end,
      _ →
         case et of
            Known(te) →
               case te of
                  TypeName(tn) →
                     not(isinId(tn, domainTRANS(trans))),
                  TypeExprProduct(tel) →
                     not(
                        getNonTOIReturnsList(te, trans) = ⟨⟩
                     ),
                  FunctionTypeExpr(arg, fa, res) →
                     not(
```

```
                    getNonTOIReturnsList(
                        type_expr(res), trans) = ⟨⟩),
                _ → true
              end,
            Unknown → true
          end
      end,

    getNonTOIReturnsList :
      TypeExpr × TRANS → TypeExpr*
    getNonTOIReturnsList(te, trans) ≡
      case te of
        TypeLiteral(literal) → ⟨te⟩,
        TypeName(id) →
          if isinId(id, domainTRANS(trans)) then ⟨⟩
          else ⟨te⟩
          end,
        TypeExprProduct(tep) →
          getNonTOIProduct(tep, trans),
        TypeExprSet(tes) → ⟨te⟩,
        TypeExprList(tel) → ⟨te⟩,
        TypeExprMap(tem) → ⟨te⟩,
        FunctionTypeExpr(arg, fa, res) →
          getNonTOIReturnsList(type_expr(res), trans),
        BracketedTypeExpr(bte) →
          getNonTOIReturnsList(bte, trans)
      end,

    getNonTOIProduct :
      TypeExpr* × TRANS → TypeExpr*
    getNonTOIProduct(tel, trans) ≡
      if tel = ⟨⟩ then ⟨⟩
      else
        getNonTOIReturnsList(hd tel, trans) ^
        getNonTOIProduct(tl tel, trans)
      end,

    getTOIReturnsList :
      TypeExpr × TRANS → ValueExpr*
    getTOIReturnsList(te, trans) ≡
      case te of
        TypeLiteral(literal) → ⟨⟩,
        TypeName(id) →
          if isinId(id, domainTRANS(trans))
```

```
            then
               ⟨Make_ValueOrVariableName(
                      mk_ValueOrVariableName(
                           getMapValueTRANS(trans, id)))⟩
            else ⟨⟩
            end,
         TypeExprProduct(tep) →
            getTOIReturnsListProduct(tep, trans),
         TypeExprSet(tes) → ⟨⟩,
         TypeExprList(tel) → ⟨⟩,
         TypeExprMap(tem) → ⟨⟩,
         FunctionTypeExpr(arg, fa, res) →
            getTOIReturnsList(type_expr(res), trans),
         BracketedTypeExpr(bte) →
            getTOIReturnsList(bte, trans)
      end,

   getTOIReturnsListProduct :
      TypeExpr* × TRANS → ValueExpr*
   getTOIReturnsListProduct(tel, trans) ≡
      if tel = ⟨⟩ then ⟨⟩
      else
         getTOIReturnsList(hd tel, trans) ⌢
         getTOIReturnsListProduct(tl tel, trans)
      end,

   getTOIReturnsListTN : TypeExpr × TRANS → Access*
   getTOIReturnsListTN(te, trans) ≡
      case te of
         TypeLiteral(literal) → ⟨⟩,
         TypeName(id) →
            if isinId(id, domainTRANS(trans))
            then
               ⟨AccessValueOrVariableName(
                      mk_ValueOrVariableName(id))⟩
            else ⟨⟩
            end,
         TypeExprProduct(tep) →
            getTOIReturnsListProductTN(tep, trans),
         TypeExprSet(tes) → ⟨⟩,
         TypeExprList(tel) → ⟨⟩,
         TypeExprMap(tem) → ⟨⟩,
         FunctionTypeExpr(arg, fa, res) →
            getTOIReturnsListTN(type_expr(res), trans),
```

```
    BracketedTypeExpr(bte) →
      getTOIReturnsListTN(bte, trans)
  end,

getTOIReturnsListProductTN :
  TypeExpr* × TRANS → Access*
getTOIReturnsListProductTN(tel, trans) ≡
  if tel = ⟨⟩ then ⟨⟩
  else
    getTOIReturnsListTN(hd tel, trans) ⌢
    getTOIReturnsListProductTN(tl tel, trans)
  end,

/***** Generators and observers end *****/

/***** Access *****/

makeReadAccessDesc :
  Access* × TRANS → OptionalReadAccessDesc
makeReadAccessDesc(al, trans) ≡
  if al = ⟨⟩ then NoReadAccessMode
  else ReadAccessDesc(toVariableList(al, trans))
  end,

makeWriteAccessDesc :
  Access* × TRANS → OptionalWriteAccessDesc
makeWriteAccessDesc(rs, trans) ≡
  if rs = ⟨⟩ then NoWriteAccessMode
  else WriteAccessDesc(toVariableList(rs, trans))
  end,

toVariableList : Access* × TRANS → Access*
toVariableList(al, trans) ≡
  if al = ⟨⟩ then al
  else
    case hd al of
      AccessValueOrVariableName(vn) →
        if isinId(id(vn), domainTRANS(trans))
        then
          ⟨AccessValueOrVariableName(
                mk_ValueOrVariableName(
                    getMapValueTRANS(trans, id(vn))))⟩ ⌢
          toVariableList(tl al, trans)
        else toVariableList(tl al, trans)
```

```
            end
        end
    end,

removeDuplets : Access* → Access*
removeDuplets(al) ≡
    if
        lengthAccess(al) =
            lengthAccess(elementsAccess(al))
    then al
    else
        if isinAccess(hd al, elementsAccess(tl al))
        then removeDuplets(tl al)
        else ⟨hd al⟩ ⌢ removeDuplets(tl al)
        end
    end,

getAccess : TypeExpr × TRANS → Access*
getAccess(te, trans) ≡
    case te of
        TypeLiteral(literal) → ⟨⟩,
        TypeName(id) →
            if isinId(id, domainTRANS(trans))
            then
                /*Type of interest.*/
                ⟨AccessValueOrVariableName(
                        mk_ValueOrVariableName(id))⟩
            else
                /*Not type of interest.*/
                ⟨⟩
            end,
        TypeExprProduct(tep) → getAccessList(tep, trans),
        TypeExprSet(tes) → ⟨⟩,
        TypeExprList(tel) → ⟨⟩,
        TypeExprMap(tem) → ⟨⟩,
        FunctionTypeExpr(arg, fa, res) → ⟨⟩,
        BracketedTypeExpr(bte) → getAccess(bte, trans)
    end,

getAccessList : TypeExpr* × TRANS → Access*
getAccessList(tel, trans) ≡
    if tel = ⟨⟩ then ⟨⟩
    else
        getAccess(hd tel, trans) ⌢
```

```
        getAccessList(tl tel, trans)
    end,

getAccessObsOptPreCondition :
    Id* × OptionalPreCondition × ExpType ×
    FUNC × TRANS →
        AccessResult
getAccessObsOptPreCondition(
     idset, precond, et, func, trans) ≡
    case precond of
        PreCondition(ve) →
            getAccessObs(idset, ve, et, func, trans),
        NoPreCondition → mk_AccessResult(⟨⟩, idset)
    end,

getAccessObs :
    Id* × ValueExpr × ExpType × FUNC × TRANS →
        AccessResult
getAccessObs(idset, ve, et, func, trans) ≡
    case ve of
        Make_ValueLiteral(vl) →
            mk_AccessResult(⟨⟩, idset),
        Make_ValueOrVariableName(vn) →
            mk_AccessResult(⟨⟩, idset),
        Make_BasicExpr(be) → mk_AccessResult(⟨⟩, idset),
        ProductExpr(vel) →
            getAccessObsList(idset, vel, et, func, trans),
        Make_SetExpr(se) →
            getAccessObsSetExpr(
                idset, se, getSetType(et), func, trans),
        Make_ListExpr(le) →
            getAccessObsListExpr(
                idset, le, getListType(et), func, trans),
        Make_MapExpr(me) →
            getAccessObsMapExpr(
                idset, me, getMapType(et), func, trans),
        ApplicationExpr(ave, vl) →
            case ave of
                Make_ValueOrVariableName(vn) →
                    if isinId(id(vn), domainFUNC(func))
                    then
                        if isinId(id(vn), idset)
                        then
                            case
```

```
            type_expr(
                getMapValueFUNC(func, id(vn)))
        of
            FunctionTypeExpr(arg, fa, res) →
                /*Already evaluated.*/
                mk_AccessResult(
                    removeDuplets(
                        accessList(
                            getAccessObsListList(
                                idset, vl,
                                expTypeList(
                                    arg, lengthVE(vl)),
                                func, trans)) ^
                        getSequencingAccess(
                            ve, type_expr(res), et,
                            trans, func)),
                    getAccessIdSet(
                        getAccessObsListList(
                            idset, vl,
                            expTypeList(
                                arg, lengthVE(vl)),
                            func, trans)))
        end
    else
        /*Not evaluated yet.*/
        case
            type_expr(
                getMapValueFUNC(func, id(vn)))
        of
            FunctionTypeExpr(arg, fa, res) →
                mk_AccessResult(
                    removeDuplets(
                        (getAccess(arg, trans) ^
                         accessList(
                            getAccessObs(
                                addId(id(vn), idset),
                                value_expr(
                                    getMapValueFUNC(
                                        func, id(vn))),
                                Known(type_expr(res)),
                                func, trans))) ^
                        accessList(
                            getAccessObsListList(
                                idset, vl,
```

```
                                    expTypeList(
                                      arg, lengthVE(vl)),
                                    func, trans)) ⌢
                                getSequencingAccess(
                                  ve, type_expr(res), et,
                                  trans, func)),
                              unionId(
                                getAccessIdSet(
                                  getAccessObs(
                                    addId(id(vn), idset),
                                    value_expr(
                                      getMapValueFUNC(
                                        func, id(vn))),
                                    Known(type_expr(res)),
                                    func, trans)),
                                getAccessIdSet(
                                  getAccessObsListList(
                                    idset, vl,
                                    expTypeList(
                                      arg, lengthVE(vl)),
                                    func, trans))))
                  end
                end
              else
                getAccessObsList(
                    idset, vl, et, func, trans)
              end,
            _ →
              getAccessObsList(idset, vl, et, func, trans)
          end,
        BracketedExpr(bve) →
          getAccessObs(idset, bve, et, func, trans),
        ValueInfixExpr(first, op, second) →
          getAccessObsList(
              idset, ⟨first, second⟩, Unknown, func, trans
          ),
        ValuePrefixExpr(op, operand) →
          getAccessObs(
              idset, operand, Unknown, func, trans),
        LetExpr(ldl, lve) →
          mk_AccessResult(
              removeDuplets(
                    accessList(
                        getAccessObsLetDefList(
```

```
                        idset, ldl, func, trans)) ⌢
                accessList(
                    getAccessObs(
                        accessIdSet(
                            getAccessObsLetDefList(
                                idset, ldl, func, trans)),
                        lve, et, func, trans))),
            accessIdSet(
                getAccessObs(
                    accessIdSet(
                        getAccessObsLetDefList(
                            idset, ldl, func, trans)), lve,
                    et, func, trans))),
    Make_IfExpr(ie) →
        getAccessObsListList(
            idset, ifExprToList(ie),
            ⟨Known(TypeLiteral(BOOL))⟩ ⌢
            elsifToTypeList(elsif_list(ie), et) ⌢ ⟨et⟩,
            func, trans),
    CaseExpr(cond, cbl) →
        mk_AccessResult(
            removeDuplets(
                accessList(
                    getAccessObs(
                        idset, cond,
                        Known(TypeLiteral(BOOL)), func, trans
                    )) ⌢
                accessList(
                    getAccessObsCaseBranch(
                        accessIdSet(
                            getAccessObs(
                                idset, cond,
                                Known(TypeLiteral(BOOL)),
                                func, trans)), cbl, et, func,
                        trans))),
            accessIdSet(
                getAccessObsCaseBranch(
                    accessIdSet(
                        getAccessObs(
                            idset, cond,
                            Known(TypeLiteral(BOOL)), func,
                            trans)), cbl, et, func, trans)))
end,
```

```
getSequencingAccess :
   ValueExpr × TypeExpr × ExpType × TRANS × FUNC →
      Access*
getSequencingAccess(ve, te, et, trans, func) ≡
   if
      and(
          and(
               isTGen(ve, trans, func),
               not(returnsNonTOI(ve, et, trans, func))),
          not(equalsType(te, et)))
   then getTOIReturnsListTN(te, trans)
   else ⟨⟩
   end,

getAccessObsSetExpr :
   Id* × SetExpr × ExpType × FUNC × TRANS →
      AccessResult
getAccessObsSetExpr(idset, se, et, func, trans) ≡
   case se of
      RangedSetExpr(fve, sve) →
         getAccessObsListList(
             idset, ⟨fve, sve⟩, ⟨et, et⟩, func, trans),
      EnumeratedSetExpr(ovel) →
         getAccessObsOptVEL(idset, ovel, et, func, trans),
      ComprehendedSetExpr(ve, typlist, or) →
         mk_AccessResult(
             removeDuplets(
                  accessList(
                      getAccessObs(idset, ve, et, func, trans)
                  ) ⌢
                  accessList(
                      getAccessObsList(
                           getAccessIdSet(
                              getAccessObs(
                                 idset, ve, et, func, trans)),
                           getVEOptRestriction(or),
                           Known(TypeLiteral(BOOL)), func, trans
                      ))),
             getAccessIdSet(
                  getAccessObsList(
                      getAccessIdSet(
                           getAccessObs(
                              idset, ve, et, func, trans)),
                      getVEOptRestriction(or),
```

```
                    Known(TypeLiteral(BOOL)), func, trans)))
    end,

getAccessObsListExpr :
    Id* × ListExpr × ExpType × FUNC × TRANS →
        AccessResult
getAccessObsListExpr(idset, le, et, func, trans) ≡
    case le of
        RangedListExpr(fve, sve) →
            getAccessObsListList(
                idset, ⟨fve, sve⟩, ⟨et, et⟩, func, trans),
        EnumeratedListExpr(ovel) →
            getAccessObsOptVEL(idset, ovel, et, func, trans),
        ComprehendedListExpr(ve1, b, ve2, or) →
            getAccessObsListList(
                idset,
                ⟨ve1⟩ ^ ⟨ve2⟩ ^ getVEOptRestriction(or),
                ⟨et, et, Known(TypeLiteral(BOOL))⟩, func,
                trans)
    end,

getAccessObsMapExpr :
    Id* × MapExpr × MapType × FUNC × TRANS →
        AccessResult
getAccessObsMapExpr(idset, me, et, func, trans) ≡
    case me of
        EnumeratedMapExpr(ovel) →
            getAccessObsOptVEPL(idset, ovel, et, func, trans),
        ComprehendedMapExpr(ve, typlist, or) →
            getAccessObsListList(
                idset,
                getValueExprList(ve) ^
                getVEOptRestriction(or),
                ⟨tedom(et), terange(et),
                    Known(TypeLiteral(BOOL))⟩, func, trans)
    end,

getAccessObsOptVEL :
    Id* × OptionalValueExprList × ExpType ×
    FUNC × TRANS →
        AccessResult
getAccessObsOptVEL(idset, ovel, et, func, trans) ≡
    case ovel of
        ValueExprList(vel) →
```

```
        getAccessObsList(idset, vel, et, func, trans),
      NoValueExprList → mk_AccessResult(⟨⟩, idset)
   end,

getAccessObsOptVEPL :
   Id* × OptionalValueExprPairList × MapType ×
   FUNC × TRANS →
      AccessResult
getAccessObsOptVEPL(idset, ovel, et, func, trans) ≡
   case ovel of
      ValueExprPairList(vel) →
         getAccessObsListList(
            idset, pairListToList(vel),
            pairListToTypeList(vel, et), func, trans),
      NoValueExprPairList →
         mk_AccessResult(⟨⟩, idset)
   end,

pairListToList : ValueExprPair* → ValueExpr*
pairListToList(vepl) ≡
   if vepl = ⟨⟩ then ⟨⟩
   else
      ⟨getFirstFromPair(hd vepl),
         getSecondFromPair(hd vepl)⟩ ^
      pairListToList(tl vepl)
   end,

pairListToTypeList :
   ValueExprPair* × MapType → ExpType*
pairListToTypeList(vepl, et) ≡
   if vepl = ⟨⟩ then ⟨⟩
   else
      ⟨getExpType(tedom(et)), getExpType(terange(et))⟩ ^
      pairListToTypeList(tl vepl, et)
   end,

getVEOptRestriction :
   OptionalRestriction → ValueExpr*
getVEOptRestriction(optr) ≡
   case optr of
      Restriction(ve) → ⟨ve⟩,
      NoRestriction → ⟨⟩
   end,
```

```
elsifToList : Elsif* → ValueExpr*
elsifToList(eil) ≡
   if eil = ⟨⟩ then emptyVE()
   else
      ⟨getElsifCondition(hd eil)⟩ ⌢
      ⟨getElsifCase(hd eil)⟩ ⌢ elsifToList(tl eil)
   end,

elsifToTypeList :
   Elsif* × ExpType → ExpType*
elsifToTypeList(eil, et) ≡
   if eil = ⟨⟩ then ⟨⟩
   else
      ⟨Known(TypeLiteral(BOOL))⟩ ⌢ ⟨et⟩ ⌢
      elsifToTypeList(tl eil, et)
   end,

getAccessObsList :
   Id* × ValueExpr* × ExpType × FUNC × TRANS →
      AccessResult
getAccessObsList(idset, vel, et, func, trans) ≡
   if vel = ⟨⟩ then mk_AccessResult(⟨⟩, idset)
   else
      mk_AccessResult(
          removeDuplets(
               accessList(
                  getAccessObs(
                        idset, hd vel, getHead(et), func, trans
                  )) ⌢
               accessList(
                  getAccessObsList(
                        accessIdSet(
                           getAccessObs(
                              idset, hd vel, getHead(et),
                              func, trans)), tl vel,
                        getTail(et), func, trans))),
          accessIdSet(
               getAccessObsList(
                  accessIdSet(
                        getAccessObs(
                           idset, hd vel, getHead(et), func,
                           trans)), tl vel, getTail(et), func,
                  trans)))
   end,
```

```
getAccessObsListList :
   Id* × ValueExpr* × ExpType* × FUNC ×
   TRANS →
      AccessResult
getAccessObsListList(idset, vel, etl, func, trans) ≡
   if vel = ⟨⟩ then mk_AccessResult(⟨⟩, idset)
   else
      mk_AccessResult(
         removeDuplets(
               accessList(
                  getAccessObs(
                        idset, hd vel, hd etl, func, trans)) ^
               accessList(
                  getAccessObsListList(
                        accessIdSet(
                           getAccessObs(
                              idset, hd vel, hd etl, func,
                              trans)), tl vel, tl etl, func,
                        trans))),
         accessIdSet(
               getAccessObsListList(
                  accessIdSet(
                        getAccessObs(
                           idset, hd vel, hd etl, func, trans)),
                  tl vel, tl etl, func, trans)))
   end,

getAccessObsLetDefList :
   Id* × LetDef* × FUNC × TRANS →
      AccessResult
getAccessObsLetDefList(idset, ldl, func, trans) ≡
   if ldl = ⟨⟩ then mk_AccessResult(⟨⟩, idset)
   else
      mk_AccessResult(
         removeDuplets(
               accessList(
                  getAccessObs(
                        idset, value_expr(hd ldl), Unknown,
                        func, trans)) ^
               accessList(
                  getAccessObsLetDefList(
                        accessIdSet(
                           getAccessObs(
```

```
                                    idset, value_expr(hd ldl),
                                    Unknown, func, trans)), tl ldl,
                              func, trans))),
                  accessIdSet(
                        getAccessObsLetDefList(
                              accessIdSet(
                                    getAccessObs(
                                          idset, value_expr(hd ldl), Unknown,
                                          func, trans)), tl ldl, func, trans))
            )
      end,

getAccessObsCaseBranch :
      Id* × CaseBranch* × ExpType × FUNC ×
      TRANS →
            AccessResult
getAccessObsCaseBranch(idset, cbl, et, func, trans) ≡
      if cbl = ⟨⟩ then mk_AccessResult(⟨⟩, idset)
      else
            mk_AccessResult(
                  removeDuplets(
                        accessList(
                              getAccessObs(
                                    idset, value_expr(hd cbl), et, func,
                                    trans)) ^
                        accessList(
                              getAccessObsCaseBranch(
                                    accessIdSet(
                                          getAccessObs(
                                                idset, value_expr(hd cbl), et,
                                                func, trans)), tl cbl, et, func,
                                    trans))),
                  accessIdSet(
                        getAccessObsCaseBranch(
                              accessIdSet(
                                    getAccessObs(
                                          idset, value_expr(hd cbl), et,
                                          func, trans)), tl cbl, et, func,
                              trans)))
      end,

getAccessGen :
      Id* × ValueExpr × ExpType × FUNC × TRANS →
            AccessResult
```

```
getAccessGen(idset, ve, et, func, trans) ≡
  case ve of
    Make_ValueLiteral(vl) →
      mk_AccessResult(⟨⟩, idset),
    Make_ValueOrVariableName(vn) →
      if isinId(id(vn), domainFUNC(func))
      then
        getAccessGen(
           idset, ApplicationExpr(ve, ⟨⟩), et, func,
           trans)
      else
        mk_AccessResult(
           getAssignExpr(ve, et, trans), idset)
      end,
    Make_BasicExpr(be) → mk_AccessResult(⟨⟩, idset),
    ProductExpr(vel) →
      getAccessGenList(
         idset, vel,
         expTypeToExpTypeList(
            removeBrackets(et), lengthVE(vel)), func,
         trans),
    Make_SetExpr(se) →
      getAccessGenSetExpr(
         idset, se, getSetType(et), func, trans),
    Make_ListExpr(le) →
      getAccessGenListExpr(
         idset, le, getListType(et), func, trans),
    Make_MapExpr(me) →
      getAccessGenMapExpr(
         idset, me, getMapType(et), func, trans),
    ApplicationExpr(ave, vl) →
      case ave of
        Make_ValueOrVariableName(vn) →
          if isinId(id(vn), domainFUNC(func))
          then
            if isinId(id(vn), idset)
            then
              case
                type_expr(
                   getMapValueFUNC(func, id(vn)))
              of
                FunctionTypeExpr(arg, fa, res) →
                  /∗Already evaluated.∗/
                  getAccessGenList(
```

```
                idset, vl,
                expTypeList(arg, lengthVE(vl)),
                func, trans)
       end
   else
       /∗Not evaluated yet.∗/
       case
          type_expr(
              getMapValueFUNC(func, id(vn)))
       of
          FunctionTypeExpr(arg, fa, res) →
             mk_AccessResult(
                removeDuplets(
                   (getAccess(
                      type_expr(res), trans) ^
                    accessList(
                      getAccessGen(
                         addId(id(vn), idset),
                         value_expr(
                            getMapValueFUNC(
                               func, id(vn))),
                         Known(type_expr(res)),
                         func, trans))) ^
                   accessList(
                      getAccessGenList(
                         idset, vl,
                         expTypeList(
                            arg, lengthVE(vl)),
                         func, trans))),
                unionId(
                   getAccessIdSet(
                      getAccessGen(
                         addId(id(vn), idset),
                         value_expr(
                            getMapValueFUNC(
                               func, id(vn))),
                         Known(type_expr(res)),
                         func, trans)),
                   getAccessIdSet(
                      getAccessGenList(
                         idset, vl,
                         expTypeList(
                            arg, lengthVE(vl)),
                         func, trans))))
```

```
                end
            end
        else
            getAccessGenList(
                idset, vl,
                expTypeToExpTypeList(
                    removeBrackets(et), lengthVE(vl)),
                func, trans)
        end,
    _ →
        getAccessGenList(
            idset, vl,
            expTypeToExpTypeList(
                removeBrackets(et), lengthVE(vl)),
            func, trans)
  end,
BracketedExpr(bve) →
  getAccessGen(idset, bve, et, func, trans),
ValueInfixExpr(first, op, second) →
  getAccessGenList(
      idset, ⟨first, second⟩,
      ⟨Unknown, Unknown⟩, func, trans),
ValuePrefixExpr(op, operand) →
  getAccessGen(
      idset, operand, Unknown, func, trans),
LetExpr(ldl, lve) →
  mk_AccessResult(
      removeDuplets(
          accessList(
              getAccessGenLetDefList(
                  idset, ldl, func, trans)) ^
          accessList(
              getAccessGen(
                  accessIdSet(
                    getAccessGenLetDefList(
                      idset, ldl, func, trans)),
                  lve, et, func, trans))),
      accessIdSet(
          getAccessGen(
              accessIdSet(
                  getAccessGenLetDefList(
                    idset, ldl, func, trans)), lve,
              et, func, trans))),
Make_IfExpr(ie) →
```

```
            getAccessGenListList(
                idset, ifExprToList(ie),
                ⟨Known(TypeLiteral(BOOL))⟩ ⌢
                elsifToTypeList(elsif_list(ie), et) ⌢ ⟨et⟩,
                func, trans),
        CaseExpr(cond, cbl) →
            mk_AccessResult(
                removeDuplets(
                    accessList(
                        getAccessGen(
                            idset, cond,
                            Known(TypeLiteral(BOOL)), func, trans
                        )) ⌢
                    accessList(
                        getAccessGenCaseBranch(
                            accessIdSet(
                              getAccessGen(
                                idset, cond,
                                Known(TypeLiteral(BOOL)),
                                func, trans)), cbl, et, func,
                            trans))),
                accessIdSet(
                    getAccessGenCaseBranch(
                        accessIdSet(
                            getAccessGen(
                              idset, cond,
                              Known(TypeLiteral(BOOL)), func,
                              trans)), cbl, et, func, trans)))
    end,

getAssignExpr :
    ValueExpr × ExpType × TRANS → Access*
getAssignExpr(ve, et, trans) ≡
    case et of
        Known(te) →
            case te of
                TypeName(tn) →
                    if isinId(tn, domainTRANS(trans))
                    then
                        /*Type of interest.*/
                        ⟨AccessValueOrVariableName(
                                mk_ValueOrVariableName(tn))⟩
                    else
                        /*Not type of interest.*/
```

```
                emptyAccess()
              end,
            _ → emptyAccess()
          end,
        Unknown → emptyAccess()
      end,

    getAccessGenSetExpr :
      Id* × SetExpr × ExpType × FUNC × TRANS →
        AccessResult
    getAccessGenSetExpr(idset, se, et, func, trans) ≡
      case se of
        RangedSetExpr(fve, sve) →
          getAccessGenListList(
            idset, ⟨fve, sve⟩, ⟨et, et⟩, func, trans),
        EnumeratedSetExpr(ovel) →
          getAccessGenOptVEL(idset, ovel, et, func, trans),
        ComprehendedSetExpr(ve, typlist, or) →
          mk_AccessResult(
            removeDuplets(
                accessList(
                  getAccessGen(idset, ve, et, func, trans)
                ) ⌢
                accessList(
                  getAccessGenList(
                      getAccessIdSet(
                        getAccessGen(
                          idset, ve, et, func, trans)),
                      getVEOptRestriction(or),
                      expTypeToExpTypeList(
                        removeBrackets(
                          Known(TypeLiteral(BOOL))), 1),
                      func, trans))),
            getAccessIdSet(
                getAccessGenList(
                  getAccessIdSet(
                      getAccessGen(
                        idset, ve, et, func, trans)),
                  getVEOptRestriction(or),
                  expTypeToExpTypeList(
                      removeBrackets(
                        Known(TypeLiteral(BOOL))), 1),
                  func, trans)))
      end,
```

```
getAccessGenListExpr :
   Id* × ListExpr × ExpType × FUNC × TRANS →
      AccessResult
getAccessGenListExpr(idset, le, et, func, trans) ≡
   case le of
      RangedListExpr(fve, sve) →
         getAccessGenListList(
            idset, ⟨fve, sve⟩, ⟨et, et⟩, func, trans),
      EnumeratedListExpr(ovel) →
         getAccessGenOptVEL(idset, ovel, et, func, trans),
      ComprehendedListExpr(ve1, b, ve2, or) →
         getAccessGenListList(
            idset,
            ⟨ve1⟩ ⁀ ⟨ve2⟩ ⁀ getVEOptRestriction(or),
            ⟨et, et, Known(TypeLiteral(BOOL))⟩, func,
            trans)
   end,

getAccessGenMapExpr :
   Id* × MapExpr × MapType × FUNC × TRANS →
      AccessResult
getAccessGenMapExpr(idset, me, et, func, trans) ≡
   case me of
      EnumeratedMapExpr(ovel) →
         getAccessGenOptVEPL(idset, ovel, et, func, trans),
      ComprehendedMapExpr(ve, typlist, or) →
         getAccessGenListList(
            idset,
            getValueExprList(ve) ⁀
            getVEOptRestriction(or),
            ⟨tedom(et), terange(et),
               Known(TypeLiteral(BOOL))⟩, func, trans)
   end,

getAccessGenOptVEL :
   Id* × OptionalValueExprList × ExpType ×
   FUNC × TRANS →
      AccessResult
getAccessGenOptVEL(idset, ovel, et, func, trans) ≡
   case ovel of
      ValueExprList(vel) →
         getAccessGenList(
            idset, vel,
```

```
                expTypeToExpTypeList(
                    removeBrackets(et), lengthVE(vel)), func,
                trans),
        NoValueExprList → mk_AccessResult(⟨⟩, idset)
    end,

getAccessGenOptVEPL :
    Id* × OptionalValueExprPairList × MapType ×
    FUNC × TRANS →
        AccessResult
getAccessGenOptVEPL(idset, ovel, et, func, trans) ≡
    case ovel of
        ValueExprPairList(vel) →
            getAccessGenListList(
                idset, pairListToList(vel),
                pairListToTypeList(vel, et), func, trans),
        NoValueExprPairList →
            mk_AccessResult(⟨⟩, idset)
    end,

getAccessGenList :
    Id* × ValueExpr* × ExpType* × FUNC ×
    TRANS →
        AccessResult
getAccessGenList(idset, vel, etl, func, trans) ≡
    if vel = ⟨⟩ then mk_AccessResult(⟨⟩, idset)
    else
        mk_AccessResult(
            removeDuplets(
                accessList(
                    getAccessGen(
                        idset, hd vel, hd etl, func, trans)) ⌢
                accessList(
                    getAccessGenList(
                        accessIdSet(
                            getAccessGen(
                                idset, hd vel, hd etl, func,
                                trans)), tl vel, tl etl, func,
                        trans))),
            accessIdSet(
                getAccessGenList(
                    accessIdSet(
                        getAccessGen(
                            idset, hd vel, hd etl, func, trans)),
```

```
                    tl vel, tl etl, func, trans)))
    end,

  getAccessGenListList :
    Id* × ValueExpr* × ExpType* × FUNC ×
    TRANS →
      AccessResult
  getAccessGenListList(idset, vel, etl, func, trans) ≡
    if vel = ⟨⟩ then mk_AccessResult(⟨⟩, idset)
    else
      mk_AccessResult(
        removeDuplets(
            accessList(
              getAccessGen(
                  idset, hd vel, hd etl, func, trans)) ⌢
            accessList(
              getAccessGenListList(
                  accessIdSet(
                    getAccessGen(
                      idset, hd vel, hd etl, func,
                      trans)), tl vel, tl etl, func,
                  trans))),
        accessIdSet(
            getAccessGenListList(
              accessIdSet(
                  getAccessGen(
                    idset, hd vel, hd etl, func, trans)),
              tl vel, tl etl, func, trans)))
    end,

  getAccessGenLetDefList :
    Id* × LetDef* × FUNC × TRANS →
      AccessResult
  getAccessGenLetDefList(idset, ldl, func, trans) ≡
    if ldl = ⟨⟩ then mk_AccessResult(⟨⟩, idset)
    else
      mk_AccessResult(
        removeDuplets(
            accessList(
              getAccessGen(
                  idset, value_expr(hd ldl), Unknown,
                  func, trans)) ⌢
            accessList(
              getAccessGenLetDefList(
```

```
                        accessIdSet(
                            getAccessGen(
                              idset, value_expr(hd ldl),
                              Unknown, func, trans)), tl ldl,
                        func, trans))),
           accessIdSet(
               getAccessGenLetDefList(
                   accessIdSet(
                        getAccessGen(
                            idset, value_expr(hd ldl), Unknown,
                            func, trans)), tl ldl, func, trans))
       )
    end,

getAccessGenCaseBranch :
    Id* × CaseBranch* × ExpType × FUNC ×
    TRANS →
       AccessResult
getAccessGenCaseBranch(idset, cbl, et, func, trans) ≡
    if cbl = ⟨⟩ then mk_AccessResult(⟨⟩, idset)
    else
       mk_AccessResult(
           removeDuplets(
               accessList(
                   getAccessGen(
                        idset, value_expr(hd cbl), et, func,
                        trans)) ^
               accessList(
                   getAccessGenCaseBranch(
                        accessIdSet(
                            getAccessGen(
                              idset, value_expr(hd cbl), et,
                              func, trans)), tl cbl, et, func,
                        trans))),
           accessIdSet(
               getAccessGenCaseBranch(
                   accessIdSet(
                        getAccessGen(
                            idset, value_expr(hd cbl), et,
                            func, trans)), tl cbl, et, func,
                   trans)))
    end,

/***** Access end *****/
```

```
/***** Bindings *****/

makeBinding : Binding* → Binding
makeBinding(bl) ≡
   if lengthBinding(bl) = 1 then hd bl
   else Make_ProductBinding(mk_ProductBinding(bl))
   end,

makeLetBinding :
   Int × Int × Binding* × ValueExpr* →
      BINDING_VE
makeLetBinding(length, counter, b, vel) ≡
   if length = 0 then mk_BINDING_VE(b, vel)
   else
      makeLetBinding(
          minusOne(length), counter + 1,
          b ^ ⟨IdBinding(makeId(counter))⟩,
          vel ^
          ⟨Make_ValueOrVariableName(
                   mk_ValueOrVariableName(makeId(counter)))⟩)
   end,

getLengthLetBinding : LetBinding → Int
getLengthLetBinding(lb) ≡
   case lb of
      MakeBinding(b) → getLengthBinding(b),
      MakeRecordPattern(vn, p) → 1,
      MakeListPatternLet(lp) → 1
   end,

getLengthBinding : Binding → Int
getLengthBinding(b) ≡
   case b of
      IdBinding(id) → 1,
      Make_ProductBinding(pb) →
         getLengthBindingList(binding_list(pb))
   end,

getLengthBindingList : Binding* → Int
getLengthBindingList(bl) ≡
   if bl = ⟨⟩ then 0
   else
      case hd bl of
```

```
            IdBinding(id) → 1 + getLengthBindingList(tl bl),
            Make_ProductBinding(pb) →
               getLengthBindingList(binding_list(pb)) +
               getLengthBindingList(tl bl)
         end
      end,

   isinBinding : Binding × Binding* → Bool
   isinBinding(b, bl) ≡
      if bl = ⟨⟩ then false
      else
         case hd bl of
            IdBinding(id) →
               if getHeadTextBinding(bl) = getTextBinding(b)
               then true
               else isinBinding(b, tl bl)
               end,
            Make_ProductBinding(pb) →
               or(isinBinding(b, binding_list(pb)),
                  isinBinding(b, tl bl))
         end
      end,

   getTextBinding : Binding → Text
   getTextBinding(b) ≡
      case b of
         IdBinding(id) → getText(id)
      end,

   /***** Bindings end *****/

   /***** TRANS *****/

   checkTRANS : ExpType × Id* → Bool
   checkTRANS(et, idset) ≡
      case et of
         Known(te) →
            case te of
               TypeName(tn) → isinId(tn, idset),
               _ → false
            end,
         Unknown → false
      end,
```

```
containsTRANSName :
   ExpType × TYPINGS × Id* → Bool
containsTRANSName(et, typings, idset) ≡
   case et of
      Known(te) →
         case te of
            TypeLiteral(tn) → false,
            TypeName(tn) →
               or(isinId(tn, idset),
                   not(
                        (interId(
                              idset,
                              elementsId(
                                 getMapValueTYPINGS(typings, tn))
                        )) = ⟨⟩)),
            TypeExprProduct(tel) →
               containsTRANSNameList(tel, typings, idset),
            TypeExprSet(tes) →
               case tes of
                  FiniteSetTypeExpr(fte) →
                     containsTRANSName(
                          Known(fte), typings, idset),
                  InfiniteSetTypeExpr(ite) →
                     containsTRANSName(
                          Known(ite), typings, idset)
               end,
            TypeExprList(tel) →
               case tel of
                  FiniteListTypeExpr(fte) →
                     containsTRANSName(
                          Known(fte), typings, idset),
                  InfiniteListTypeExpr(ite) →
                     containsTRANSName(
                          Known(ite), typings, idset)
               end,
            TypeExprMap(tem) →
               case tem of
                  FiniteMapTypeExpr(tedom, terange) →
                     or(containsTRANSName(
                              Known(tedom), typings, idset),
                          containsTRANSName(
                              Known(terange), typings, idset)),
                  InfiniteMapTypeExpr(tedom, terange) →
                     or(containsTRANSName(
```

```
                        Known(tedom), typings, idset),
                     containsTRANSName(
                        Known(terange), typings, idset))
              end,
           FunctionTypeExpr(arg, fa, res) → false,
           BracketedTypeExpr(bte) →
              containsTRANSName(Known(bte), typings, idset)
        end,
     Unknown → false
  end,

containsTRANSNameList :
  TypeExpr* × TYPINGS × Id* → Bool
containsTRANSNameList(tel, typings, idset) ≡
  if tel = ⟨⟩ then false
  else
     or(containsTRANSName(Known(hd tel), typings, idset),
        containsTRANSNameList(tl tel, typings, idset))
  end,

includesTRANSName : ExpType × Id* → Bool
includesTRANSName(et, idset) ≡
  case et of
     Known(te) →
        case te of
           TypeLiteral(tn) → false,
           TypeName(tn) → isinId(tn, idset),
           TypeExprProduct(tel) →
              includesTRANSNameList(tel, idset),
           TypeExprSet(tes) →
              case tes of
                 FiniteSetTypeExpr(fte) →
                    includesTRANSName(Known(fte), idset),
                 InfiniteSetTypeExpr(ite) →
                    includesTRANSName(Known(ite), idset)
              end,
           TypeExprList(tel) →
              case tel of
                 FiniteListTypeExpr(fte) →
                    includesTRANSName(Known(fte), idset),
                 InfiniteListTypeExpr(ite) →
                    includesTRANSName(Known(ite), idset)
              end,
           TypeExprMap(tem) →
```

```
                case tem of
                    FiniteMapTypeExpr(tedom, terange) →
                        or(includesTRANSName(Known(tedom), idset),
                            includesTRANSName(
                                Known(terange), idset)),
                    InfiniteMapTypeExpr(tedom, terange) →
                        or(includesTRANSName(Known(tedom), idset),
                            includesTRANSName(
                                Known(terange), idset))
                end,
            FunctionTypeExpr(arg, fa, res) → false,
            BracketedTypeExpr(bte) →
                includesTRANSName(Known(bte), idset)
        end,
    Unknown → false
end,

includesTRANSNameList :
    TypeExpr* × Id* → Bool
includesTRANSNameList(tel, idset) ≡
    if tel = ⟨⟩ then false
    else
        or(includesTRANSName(Known(hd tel), idset),
            includesTRANSNameList(tl tel, idset))
    end,

getTRANS : ExpType × TRANS → ValueExpr
getTRANS(et, trans) ≡
    case et of
        Known(te) →
            case te of
                TypeName(tn) →
                    Make_ValueOrVariableName(
                        mk_ValueOrVariableName(
                            getMapValueTRANS(trans, tn)))
            end
    end,

getTRANSId : ExpType × TRANS → Id
getTRANSId(et, trans) ≡
    case et of
        Known(te) →
            case te of
                TypeName(tn) → getMapValueTRANS(trans, tn)
```

```
        end
    end,

/***** TRANS map *****/

domainTRANS : TRANS → Id*
domainTRANS(trans) ≡
    if getTRANSMapEntranceList(trans) = ⟨⟩ then ⟨⟩
    else
        ⟨getFirstTRANS(hd getTRANSMapEntranceList(trans))
        ⟩ ⌢
        domainTRANS(
            mk_TRANS(tl getTRANSMapEntranceList(trans)))
    end,

getMapValueTRANS : TRANS × Id → Id
getMapValueTRANS(trans, id) ≡
    if
        getTextIdTRANS(
            getFirstTRANS(hd getTRANSMapEntranceList(trans))
        ) = getTextIdTRANS(id)
    then
        getSecondTRANS(hd getTRANSMapEntranceList(trans))
    else
        getMapValueTRANS(
            mk_TRANS(tl getTRANSMapEntranceList(trans)), id)
    end,

getFirstTRANS : TRANSMapEntrance → Id
getFirstTRANS(me) ≡ first(me),

getSecondTRANS : TRANSMapEntrance → Id
getSecondTRANS(me) ≡ second(me),

getTextIdTRANS : Id → Text
getTextIdTRANS(id) ≡ getText(id),

getTRANSMapEntranceList :
    TRANS → TRANSMapEntrance*
getTRANSMapEntranceList(trans) ≡ map(trans),

removeTRANS : TRANS × Id → TRANS
removeTRANS(trans, id) ≡
    if getTRANSMapEntranceList(trans) = ⟨⟩ then trans
```

```
    else
      if
        getTextId(
            getFirstTRANS(
                 hd getTRANSMapEntranceList(trans))) =
           getTextId(id)
      then
        removeTRANS(
            mk_TRANS(tl getTRANSMapEntranceList(trans)),
            id)
      else
        mk_TRANS(
            ⟨getHeadTRANS(trans)⟩ ⌢
            getTRANSMapEntranceList(
                 removeTRANS(
                     mk_TRANS(
                          tl getTRANSMapEntranceList(trans)),
                     id)))
      end
    end,

  getHeadTRANS : TRANS → TRANSMapEntrance
  getHeadTRANS(trans) ≡
    hd getTRANSMapEntranceList(trans),

  /***** TRANS end *****/

  /***** TYPINGS *****/

  makeTYPINGSMap : TypeDef* → TYPINGS
  makeTYPINGSMap(tdl) ≡
    if tdl = ⟨⟩ then mk_TYPINGS(⟨⟩)
    else
      overrideTYPINGS(
          makeTYPINGSMap(tl tdl),
          makeTYPINGSEntrance(hd tdl))
    end,

  makeTYPINGSEntrance : TypeDef → TYPINGS
  makeTYPINGSEntrance(td) ≡
    case td of
      SortDef(id) →
        mk_TYPINGS(⟨mk_TYPINGSMapEntrance(id, ⟨⟩)⟩),
      VariantDef(id, vl) →
```

```
            mk_TYPINGS(
                ⟨mk_TYPINGSMapEntrance(
                        id, getTYPINGSVariantList(vl))⟩),
        ShortRecordDef(id, ckl) →
            mk_TYPINGS(
                ⟨mk_TYPINGSMapEntrance(
                        id, getTYPINGSComponentKindList(ckl))⟩),
        AbbreviationDef(id, te) →
            mk_TYPINGS(
                ⟨mk_TYPINGSMapEntrance(
                        id, getTYPINGSTypeExpr(te))⟩)
    end,

getTYPINGSVariantList : Variant* → Id*
getTYPINGSVariantList(vl) ≡
    if vl = ⟨⟩ then emptyId()
    else
        removeDupletsId(
            getTYPINGSVariant(hd vl) ⌢
            getTYPINGSVariantList(tl vl))
    end,

getTYPINGSVariant : Variant → Id*
getTYPINGSVariant(v) ≡
    case v of
        Make_Constructor(c) → ⟨⟩,
        RecordVariant(c, ckl) →
            getTYPINGSComponentKindList(ckl)
    end,

getTYPINGSComponentKindList :
    ComponentKind* → Id*
getTYPINGSComponentKindList(ckl) ≡
    if ckl = ⟨⟩ then emptyId()
    else
        removeDupletsId(
            getTYPINGSTypeExpr(type_expr(hd ckl)) ⌢
            getTYPINGSComponentKindList(tl ckl))
    end,

getTYPINGSNameWildcardList :
    NameOrWildcard* → Id*
getTYPINGSNameWildcardList(nwl) ≡
    if nwl = ⟨⟩ then emptyId()
```

```
    else
      removeDupletsId(
          getTYPINGSNameWildcard(hd nwl) ⌢
          getTYPINGSNameWildcardList(tl nwl))
    end,

getTYPINGSNameWildcard : NameOrWildcard → Id*
getTYPINGSNameWildcard(nw) ≡
    case nw of
      Name(var) → ⟨id(var)⟩,
      Wildcard → ⟨⟩
    end,

getTYPINGSTypeExpr : TypeExpr → Id*
getTYPINGSTypeExpr(te) ≡
    case te of
      TypeLiteral(tn) → ⟨⟩,
      TypeName(id) → ⟨id⟩,
      TypeExprProduct(tep) →
        getTYPINGSTypeExprList(tep),
      TypeExprSet(tes) →
        case tes of
          FiniteSetTypeExpr(fse) →
            getTYPINGSTypeExpr(fse),
          InfiniteSetTypeExpr(ise) →
            getTYPINGSTypeExpr(ise)
        end,
      TypeExprList(les) →
        case les of
          FiniteListTypeExpr(fle) →
            getTYPINGSTypeExpr(fle),
          InfiniteListTypeExpr(ile) →
            getTYPINGSTypeExpr(ile)
        end,
      TypeExprMap(tem) →
        case tem of
          FiniteMapTypeExpr(tedom, terange) →
            removeDupletsId(
                getTYPINGSTypeExpr(tedom) ⌢
                getTYPINGSTypeExpr(terange)),
          InfiniteMapTypeExpr(tedom, terange) →
            removeDupletsId(
                getTYPINGSTypeExpr(tedom) ⌢
                getTYPINGSTypeExpr(terange))
```

```
            end,
        FunctionTypeExpr(arg, fa, res) →
            removeDupletsId(
                getTYPINGSTypeExpr(arg) ⌢
                getTYPINGSTypeExpr(type_expr(res))),
        SubtypeExpr(st, ve) →
            getTYPINGSTypeExpr(type_expr(st)),
        BracketedTypeExpr(bte) → getTYPINGSTypeExpr(bte)
    end,

getTYPINGSTypeExprList : TypeExpr* → Id*
getTYPINGSTypeExprList(tel) ≡
    if tel = ⟨⟩ then emptyId()
    else
        removeDupletsId(
            getTYPINGSTypeExpr(hd tel) ⌢
            getTYPINGSTypeExprList(tl tel))
    end,

getTypeDecl : Decl* → TypeDef*
getTypeDecl(dl) ≡
    if dl = ⟨⟩ then ⟨⟩
    else
        case hd dl of
            TypeDecl(vdl) → vdl ⌢ getTypeDecl(tl dl),
            _ → getTypeDecl(tl dl)
        end
    end,

getIds : TypeDef* → Id*
getIds(tdl) ≡
    if tdl = ⟨⟩ then emptyId()
    else
        case hd tdl of
            SortDef(id) → makeIdList(id) ⌢ getIds(tl tdl),
            VariantDef(id, vl) →
                makeIdList(id) ⌢ getIds(tl tdl),
            ShortRecordDef(id, ckl) →
                makeIdList(id) ⌢ getIds(tl tdl),
            AbbreviationDef(id, te) →
                makeIdList(id) ⌢ getIds(tl tdl)
        end
    end,
```

```
expandTYPINGSMap :
   Id* × TYPINGS × TRANS → TYPINGS
expandTYPINGSMap(idl, typings, trans) ≡
   if idl = ⟨⟩ then typings
   else
      expandTYPINGSMap(
          tl idl,
          overrideTYPINGS(
               typings,
               mk_TYPINGS(
                   ⟨mk_TYPINGSMapEntrance(
                           hd idl,
                           removeNonTOI(
                              getIdList(hd idl, ⟨⟩, typings),
                              domainTRANS(trans))⟩))), trans)
   end,

removeNonTOI : Id* × Id* → Id*
removeNonTOI(idl, ids) ≡
   if idl = ⟨⟩ then emptyId()
   else
      if not(isinId(hd idl, ids))
      then removeNonTOI(tl idl, ids)
      else ⟨hd idl⟩ ^ removeNonTOI(tl idl, ids)
      end
   end,

getIdList : Id × Id* × TYPINGS → Id*
getIdList(id, ids, typings) ≡
   getIdListIdList(
       getMapValueTYPINGS(typings, id), addId(id, ids),
       typings),

getIdListIdList :
   Id* × Id* × TYPINGS → Id*
getIdListIdList(idl, ids, typings) ≡
   if idl = ⟨⟩ then emptyId()
   else
      if not(isinId(hd idl, ids))
      then
         ⟨hd idl⟩ ^ getIdList(hd idl, ids, typings) ^
         getIdListIdList(
             tl idl, addId(hd idl, ids), typings)
      else
```

```
            removeDupletsId(
                ⟨hd idl⟩ ⌢
                getIdListIdList(tl idl, ids, typings))
        end
    end,

/***** TYPINGS map *****/

overrideTYPINGS : TYPINGS × TYPINGS → TYPINGS
overrideTYPINGS(t1, t2) ≡
    if getTYPINGSMapEntranceList(t1) = ⟨⟩ then t2
    else
        if
            isinId(
                getTypeIdTYPINGS(
                    hd getTYPINGSMapEntranceList(t1)),
                domainTYPINGS(t2))
        then
            overrideTYPINGS(
                mk_TYPINGS(tl getTYPINGSMapEntranceList(t1)),
                t2)
        else
            overrideTYPINGS(
                mk_TYPINGS(tl getTYPINGSMapEntranceList(t1)),
                mk_TYPINGS(
                    getTYPINGSMapEntranceList(t2) ⌢
                    ⟨hd getTYPINGSMapEntranceList(t1)⟩))
        end
    end,

getMapValueTYPINGS : TYPINGS × Id → Id*
getMapValueTYPINGS(typings, id) ≡
    if
        getTextId(
            getTypeIdTYPINGS(
                hd getTYPINGSMapEntranceList(typings))) =
            getTextId(id)
    then
        getTypeListTYPINGS(
            hd getTYPINGSMapEntranceList(typings))
    else
        getMapValueTYPINGS(
            mk_TYPINGS(
                tl getTYPINGSMapEntranceList(typings)), id)
```

```
    end,

domainTYPINGS : TYPINGS → Id*
domainTYPINGS(typings) ≡
    if getTYPINGSMapEntranceList(typings) = ⟨⟩
    then ⟨⟩
    else
        ⟨getTypeIdTYPINGS(
                hd getTYPINGSMapEntranceList(typings))⟩ ⌢
        domainTYPINGS(
            mk_TYPINGS(
                tl getTYPINGSMapEntranceList(typings)))
    end,

getTypeIdTYPINGS : TYPINGSMapEntrance → Id
getTypeIdTYPINGS(te) ≡ type_id(te),

getTypeListTYPINGS : TYPINGSMapEntrance → Id*
getTypeListTYPINGS(te) ≡ id_list(te),

getTYPINGSMapEntranceList :
    TYPINGS → TYPINGSMapEntrance*
getTYPINGSMapEntranceList(typings) ≡ map(typings),

/***** TYPINGS end *****/

/***** FUNC *****/

establishFuncMap : Decl* × TRANS → FUNC
establishFuncMap(dl, trans) ≡
    expandFuncMap(
        getValueDecl(dl),
        makeFuncMap(
            getValueDecl(dl), mk_FUNC(⟨⟩), trans), trans),

getValueDecl : Decl* → ValueDef*
getValueDecl(dl) ≡
    if dl = ⟨⟩ then ⟨⟩
    else
        case hd dl of
            ValueDecl(vdl) → vdl ⌢ getValueDecl(tl dl),
            _ → getValueDecl(tl dl)
        end
    end,
```

```
makeFuncMap : ValueDef* × FUNC × TRANS → FUNC
makeFuncMap(vdl, func, trans) ≡
   if vdl = ⟨⟩ then func
   else
      makeFuncMap(
          tl vdl,
          overrideFUNC(
               func, getMapEntrance(hd vdl, trans)), trans)
   end,

getMapEntrance : ValueDef × TRANS → FUNC
getMapEntrance(vd, trans) ≡
   case vd of
      ExplicitValueDef(st, ve) →
         if
            includesTRANSName(
                Known(type_expr(st)), domainTRANS(trans))
         then
            case binding(st) of
               IdBinding(id) →
                  mk_FUNC(
                     ⟨mk_FUNCMapEntrance(
                           id,
                           mk_FuncSpec(
                              FunctionTypeExpr(
                                 TypeLiteral(UNIT),
                                 TOTAL_FUNCTION_ARROW,
                                 mk_ResultDesc(
                                    NoReadAccessMode,
                                    NoWriteAccessMode,
                                    type_expr(st))), ve,
                              ⟨⟩, ⟨⟩))⟩),
               Make_ProductBinding(pb) → mk_FUNC(⟨⟩)
            end
         else mk_FUNC(⟨⟩)
         end,
      ExplicitFunctionDef(st, ffa, ve, precond) →
         case binding(st) of
            IdBinding(id) →
               mk_FUNC(
                  ⟨mk_FUNCMapEntrance(
                        id,
                        mk_FuncSpec(
```

```
                                type_expr(st), ve, ⟨⟩, ⟨⟩))⟩)
          end
      end,

  expandFuncMap : ValueDef* × FUNC × TRANS → FUNC
  expandFuncMap(vdl, func, trans) ≡
      if vdl = ⟨⟩ then func
      else
          expandFuncMap(
              tl vdl,
              overrideFUNC(
                  func, expandMapEntrance(hd vdl, func, trans)),
              trans)
      end,

  expandMapEntrance : ValueDef × FUNC × TRANS → FUNC
  expandMapEntrance(vd, func, trans) ≡
      case vd of
          ExplicitValueDef(st, ve) →
              if
                  includesTRANSName(
                      Known(type_expr(st)), domainTRANS(trans))
              then
                  case binding(st) of
                      IdBinding(id) →
                          mk_FUNC(
                              ⟨mk_FUNCMapEntrance(
                                      id,
                                      mk_FuncSpec(
                                          FunctionTypeExpr(
                                              TypeLiteral(UNIT),
                                              TOTAL_FUNCTION_ARROW,
                                              mk_ResultDesc(
                                                  NoReadAccessMode,
                                                  NoWriteAccessMode,
                                                  type_expr(st))), ve,
                                          ⟨⟩,
                                          removeDuplets(
                                              getAccess(
                                                  type_expr(st), trans) ⌢
                                              accessList(
                                                  getAccessGen(
                                                      ⟨id⟩, ve,
                                                      Known(type_expr(st)),
```

```
                                        func, trans)))))⟩),
                    Make_ProductBinding(pb) → mk_FUNC(⟨⟩)
                end
            else mk_FUNC(⟨⟩)
            end,
        ExplicitFunctionDef(st, ffa, ve, precond) →
            case binding(st) of
                IdBinding(id) →
                    case type_expr(st) of
                        FunctionTypeExpr(arg, fa, res) →
                            mk_FUNC(
                                ⟨mk_FUNCMapEntrance(
                                    id,
                                    mk_FuncSpec(
                                        type_expr(st), ve,
                                        removeDuplets(
                                            getAccess(arg, trans) ^
                                            accessList(
                                                getAccessObs(
                                                    ⟨id⟩, ve,
                                                    Known(type_expr(res)),
                                                    func, trans)) ^
                                            accessList(
                                                getAccessObsOptPreCondition(
                                                    ⟨id⟩, precond,
                                                    Known(
                                                        TypeLiteral(BOOL)),
                                                    func, trans))),
                                        removeDuplets(
                                            getAccess(
                                                type_expr(res), trans) ^
                                            accessList(
                                                getAccessGen(
                                                    ⟨id⟩, ve,
                                                    Known(type_expr(res)),
                                                    func, trans))))))⟩)
                    end
            end
    end,

/***** FUNC map *****/

domainFUNC : FUNC → Id*
domainFUNC(func) ≡
```

```
    if getFUNCMapEntranceList(func) = ⟨⟩ then ⟨⟩
    else
       ⟨getFirstFUNC(hd getFUNCMapEntranceList(func))⟩ ^
       domainFUNC(
           mk_FUNC(tl getFUNCMapEntranceList(func)))
    end,

getMapValueFUNC : FUNC × Id → FuncSpec
getMapValueFUNC(func, id) ≡
    if
       getTextIdFUNC(
           getFirstFUNC(hd getFUNCMapEntranceList(func))) =
          getTextIdFUNC(id)
    then getSecondFUNC(hd getFUNCMapEntranceList(func))
    else
       getMapValueFUNC(
           mk_FUNC(tl getFUNCMapEntranceList(func)), id)
    end,

overrideFUNC : FUNC × FUNC → FUNC
overrideFUNC(f1, f2) ≡
    if getFUNCMapEntranceList(f1) = ⟨⟩ then f2
    else
       if
          isinId(
              getFirstFUNC(hd getFUNCMapEntranceList(f1)),
              domainFUNC(f2))
       then
          overrideFUNC(
              mk_FUNC(tl getFUNCMapEntranceList(f1)), f2)
       else
          overrideFUNC(
              mk_FUNC(tl getFUNCMapEntranceList(f1)),
              mk_FUNC(
                   getFUNCMapEntranceList(f2) ^
                   ⟨hd getFUNCMapEntranceList(f1)⟩))
       end
    end,

getFirstFUNC : FUNCMapEntrance → Id
getFirstFUNC(me) ≡ first(me),

getSecondFUNC : FUNCMapEntrance → FuncSpec
getSecondFUNC(me) ≡ second(me),
```

```
getTextIdFUNC : Id → Text
getTextIdFUNC(id) ≡ getText(id),

getFUNCMapEntranceList : FUNC → FUNCMapEntrance*
getFUNCMapEntranceList(func) ≡ map(func),

/∗**** FUNC end ****∗/

/∗**** ENV ****∗/

updateENV : ENV × ExpType × TRANS → ENV
updateENV(env, et, trans) ≡
   case et of
      Known(te) →
         case te of
            TypeName(tn) →
               if isinId(tn, domainTRANS(trans))
               then
                  /∗Type of interest.∗/
                  removeENV(env, tn)
               else
                  /∗Not type of interest.∗/
                  env
               end,
            _ → env
         end,
      Unknown → env
   end,

setEnv : Access* × ENV → ENV
setEnv(al, env) ≡
   if al = ⟨⟩ then env
   else
      case hd al of
         AccessValueOrVariableName(vn) →
            setEnv(tl al, removeENV(env, id(vn))),
         _ → setEnv(tl al, env)
      end
   end,

makeENV : LetBinding × TypeExpr × TRANS → ENV
makeENV(lb, te, trans) ≡
   case lb of
```

```
        MakeBinding(b) → makeENVBinding(b, te, trans),
        _ → mk_ENV(⟨⟩)
    end,

makeENVBinding : Binding × TypeExpr × TRANS → ENV
makeENVBinding(b, te, trans) ≡
    case b of
        IdBinding(idb) →
            case te of
                TypeName(tn) →
                    if isinId(tn, domainTRANS(trans))
                    then mk_ENV(⟨mk_ENVMapEntrance(tn, b)⟩)
                    else mk_ENV(⟨⟩)
                    end,
                _ → mk_ENV(⟨⟩)
            end,
        Make_ProductBinding(pb) →
            makeENVBindingList(
                binding_list(pb), typeExprToList(te), trans)
    end,

makeENVBindingList :
    Binding* × TypeExpr* × TRANS → ENV
makeENVBindingList(bl, tel, trans) ≡
    if bl = ⟨⟩ then mk_ENV(⟨⟩)
    else
        overrideENV(
            makeENVBinding(hd bl, hd tel, trans),
            makeENVBindingList(tl bl, tl tel, trans))
    end,

/***** ENV map *****/

domainENV : ENV → Id*
domainENV(env) ≡
    if getENVMapEntranceList(env) = ⟨⟩ then ⟨⟩
    else
        ⟨getFirstENV(hd getENVMapEntranceList(env))⟩ ^
        domainENV(mk_ENV(tl getENVMapEntranceList(env)))
    end,

rangeENV : ENV → Binding*
rangeENV(env) ≡
    if getENVMapEntranceList(env) = ⟨⟩ then ⟨⟩
```

```
    else
      ⟨getSecondENV(hd getENVMapEntranceList(env))⟩ ^
      rangeENV(mk_ENV(tl getENVMapEntranceList(env)))
    end,

  overrideENV : ENV × ENV → ENV
  overrideENV(e1, e2) ≡
    if getENVMapEntranceList(e1) = ⟨⟩ then e2
    else
      if
        isinId(
          getFirstENV(hd getENVMapEntranceList(e1)),
          domainENV(e2))
      then
        overrideENV(
          mk_ENV(tl getENVMapEntranceList(e1)), e2)
      else
        overrideENV(
          mk_ENV(tl getENVMapEntranceList(e1)),
          mk_ENV(
            getENVMapEntranceList(e2) ^
            ⟨hd getENVMapEntranceList(e1)⟩))
      end
    end,

  removeENV : ENV × Id → ENV
  removeENV(env, id) ≡
    if getENVMapEntranceList(env) = ⟨⟩ then env
    else
      if
        getTextId(
          getFirstENV(hd getENVMapEntranceList(env))) =
         getTextId(id)
      then
        removeENV(
          mk_ENV(tl getENVMapEntranceList(env)), id)
      else
        mk_ENV(
          ⟨getHeadENV(env)⟩ ^
          getENVMapEntranceList(
            removeENV(
              mk_ENV(tl getENVMapEntranceList(env)),
              id)))
      end
```

```
    end,

getFirstENV : ENVMapEntrance → Id
getFirstENV(me) ≡ first(me),

getSecondENV : ENVMapEntrance → Binding
getSecondENV(me) ≡ second(me),

getENVMapEntranceList : ENV → ENVMapEntrance*
getENVMapEntranceList(env) ≡ map(env),

getHeadENV : ENV → ENVMapEntrance
getHeadENV(env) ≡ hd getENVMapEntranceList(env),

/***** ENV end *****/

/***** TYPES *****/

makeTYPES : LetBinding × ExpType × TRANS → TYPES
makeTYPES(lb, et, trans) ≡
    case lb of
        MakeBinding(b) → makeTYPESBinding(b, et, trans),
        _ → (mk_TYPES(⟨⟩))
    end,

makeTYPESBinding :
    Binding × ExpType × TRANS → TYPES
makeTYPESBinding(b, et, trans) ≡
    case b of
        IdBinding(id) →
            mk_TYPES(⟨mk_TYPESMapEntrance(b, et)⟩),
        Make_ProductBinding(pb) →
            makeTYPESBindingList(
                binding_list(pb),
                expTypeToExpTypeList(
                    removeBrackets(et),
                    lengthBinding(binding_list(pb))), trans)
    end,

makeTYPESBindingList :
    Binding* × ExpType* × TRANS → TYPES
makeTYPESBindingList(bl, etl, trans) ≡
    if bl = ⟨⟩ then mk_TYPES(⟨⟩)
    else
```

```
      overrideTYPES(
          makeTYPESBinding(hd bl, hd etl, trans),
          makeTYPESBindingList(tl bl, tl etl, trans))
   end,

makeTYPESMap : Binding* × TypeExpr → TYPES
makeTYPESMap(bl, te) ≡
   if bl = ⟨⟩ then mk_TYPES(⟨⟩)
   else
      case hd bl of
         IdBinding(id) →
            overrideTYPES(
                mk_TYPES(
                     ⟨mk_TYPESMapEntrance(hd bl, Known(te))⟩
                ), makeTYPESMap(tl bl, te)),
         Make_ProductBinding(bil) →
            overrideTYPES(
                makeTYPESMap(binding_list(bil), te),
                makeTYPESMap(tl bl, te))
      end
   end,

alterTYPESMap : TYPES × LetDef* → TYPES
alterTYPESMap(types, ldl) ≡
   if ldl = ⟨⟩ then types
   else
      case binding(hd ldl) of
         MakeBinding(b) →
            case b of
               IdBinding(id) →
                  alterTYPESMap(
                       overrideTYPES(
                             types,
                             mk_TYPES(
                                ⟨mk_TYPESMapEntrance(b, Unknown)
                                ⟩)), tl ldl),
               Make_ProductBinding(bl) →
                  alterTYPESMap(
                       alterTYPESMapList(
                             types, binding_list(bl)), tl ldl)
            end
      end
   end,
```

```
alterTYPESMapList : TYPES × Binding* → TYPES
alterTYPESMapList(types, bl) ≡
   if lengthBinding(bl) = 0 then types
   else
      case hd bl of
         IdBinding(id) →
            alterTYPESMapList(
                overrideTYPES(
                     types,
                     mk_TYPES(
                         ⟨mk_TYPESMapEntrance(hd bl, Unknown)
                         ⟩)), tl bl),
         Make_ProductBinding(bil) →
            alterTYPESMapList(
                alterTYPESMapList(types, binding_list(bil)),
                tl bl)
      end
   end,

/∗**** TYPES Map ****∗/

domainTYPES : TYPES → Binding*
domainTYPES(types) ≡
   if getTYPESMapEntranceList(types) = ⟨⟩ then ⟨⟩
   else
      ⟨getFirstTYPES(hd getTYPESMapEntranceList(types))
      ⟩ ⌢
      domainTYPES(
          mk_TYPES(tl getTYPESMapEntranceList(types)))
   end,

getMapValueTYPES : TYPES × Binding → ExpType
getMapValueTYPES(types, b) ≡
   if
      getTextBindingTYPES(
          getFirstTYPES(hd getTYPESMapEntranceList(types))
      ) = getTextBindingTYPES(b)
   then
      getSecondTYPES(hd getTYPESMapEntranceList(types))
   else
      getMapValueTYPES(
          mk_TYPES(tl getTYPESMapEntranceList(types)), b)
   end,
```

```
overrideTYPES : TYPES × TYPES → TYPES
overrideTYPES(t1, t2) ≡
  if getTYPESMapEntranceList(t1) = ⟨⟩ then t2
  else
    if
      isinTYPES(
        getFirstTYPES(hd getTYPESMapEntranceList(t1)),
        domainTYPES(t2))
    then
      overrideTYPES(
        mk_TYPES(tl getTYPESMapEntranceList(t1)), t2)
    else
      overrideTYPES(
        mk_TYPES(tl getTYPESMapEntranceList(t1)),
        mk_TYPES(
          getTYPESMapEntranceList(t2) ^
          ⟨hd getTYPESMapEntranceList(t1)⟩))
    end
  end,

isinTYPES : Binding × Binding* → Bool
isinTYPES(b, bl) ≡
  if bl = ⟨⟩ then false
  else
    if
      getTextBindingTYPES(b) =
        getTextBindingTYPES(hd bl)
    then true
    else isinTYPES(b, tl bl)
    end
  end,

getFirstTYPES : TYPESMapEntrance → Binding
getFirstTYPES(me) ≡ first(me),

getSecondTYPES : TYPESMapEntrance → ExpType
getSecondTYPES(me) ≡ second(me),

getTextBindingTYPES : Binding → Text
getTextBindingTYPES(b) ≡
  case b of
    IdBinding(id) → getText(id)
  end,
```

getTYPESMapEntranceList :
  TYPES → TYPESMapEntrance*
getTYPESMapEntranceList(types) ≡ map(types),

getHeadTYPES : TYPES → TYPESMapEntrance
getHeadTYPES(types) ≡
  hd getTYPESMapEntranceList(types),

/***** TYPES end *****/

/***** Id *****/

removeDupletsId : Id* → Id*
removeDupletsId(il) ≡
  if lengthId(il) = lengthId(elementsId(il)) **then** il
  else
    if isinId(**hd** il, elementsId(**tl** il))
    then removeDupletsId(**tl** il)
    else ⟨**hd** il⟩ ⌢ removeDupletsId(**tl** il)
    end
  end,

makeId : **Int** → Id
makeId(no) ≡ mk_Id("pa_" ⌢ intToText(no)),

intToText : **Int** → **Text**
intToText(no) ≡
  case no **of**
    0 → "0",
    1 → "1",
    2 → "2",
    3 → "3",
    4 → "4",
    5 → "5",
    6 → "6",
    7 → "7",
    8 → "8",
    9 → "9"
  end,

/***** Id end *****/

/***** Set Logic *****/

addId : Id × Id* → Id*
addId(id, idl) ≡
  if isinId(id, idl) **then** idl **else** idl ⌢ ⟨id⟩ **end**,

unionId : Id* × Id* → Id*
unionId(il1, il2) ≡
  if il1 = ⟨⟩ **then** il2
  else unionId(**tl** il1, addId(**hd** il1, il2))
  end,

isinId : Id × Id* → **Bool**
isinId(id, idl) ≡
  if idl = ⟨⟩ **then** **false**
  else
    if getHeadTextId(idl) = getText(id) **then** **true**
    else isinId(id, **tl** idl)
    end
  end,

getHeadTextId : Id* → **Text**
getHeadTextId(idlist) ≡ getText(**hd** idlist),

lengthId : Id* → **Int**
lengthId(idl) ≡
  if idl = ⟨⟩ **then** 0 **else** 1 + lengthId(**tl** idl) **end**,

lengthVE : ValueExpr* → **Int**
lengthVE(vel) ≡
  if vel = ⟨⟩ **then** 0 **else** 1 + lengthVE(**tl** vel) **end**,

concatVE :
  ValueExpr × ValueExpr* → ValueExpr*
concatVE(ve, vel) ≡ ⟨ve⟩ ⌢ vel,

lengthTE : TypeExpr* → **Int**
lengthTE(tel) ≡
  if tel = ⟨⟩ **then** 0 **else** 1 + lengthTE(**tl** tel) **end**,

lengthET : ExpType* → **Int**
lengthET(etl) ≡
  if etl = ⟨⟩ **then** 0 **else** 1 + lengthET(**tl** etl) **end**,

lengthLetDefList : LetDef* → **Int**
lengthLetDefList(ldl) ≡

```
    if ldl = ⟨⟩ then 0
    else 1 + lengthLetDefList(tl ldl)
    end,

/***** Logic *****/

and : Bool × Bool → Bool
and(b1, b2) ≡ if b1 then b2 else false end,

or : Bool × Bool → Bool
or(b1, b2) ≡ if b1 then true else b2 end,

not : Bool → Bool
not(b) ≡ if b then false else true end,

/***** Logic end *****/

/***** HELPERS *****/

notInTYPINGS : Id × TYPINGS × TRANS → Bool
notInTYPINGS(id, typings, trans) ≡
    if isinId(id, domainTRANS(trans))
    then
        not(
            isinId(
                id,
                elementsId(getMapValueTYPINGS(typings, id))))
    else true
    end,

getValueExpr : VE_TYPES → ValueExpr
getValueExpr(vetypes) ≡ valueExpr(vetypes),

makeLetDefList : LetDef → LetDef*
makeLetDefList(ld) ≡ ⟨ld⟩,

getPattern : Pattern → Pattern
getPattern(p) ≡ p,

equals : Int × Int → Bool
equals(i1, i2) ≡ i1 = i2,

greaterEqual : Int × Int × Int → Bool
greaterEqual(i1, i2, counter) ≡
```

```
    if counter = 10 then false
    else
       if i1 = i2 then true
       else greaterEqual(i1, i2 + 1, counter + 1)
       end
    end,

minusOne : Int → Int
minusOne(i) ≡
    case i of
       1 → 0,
       2 → 1,
       3 → 2,
       4 → 3,
       5 → 4,
       6 → 5,
       7 → 6,
       8 → 7,
       9 → 8,
       10 → 9
    end,

equalsText : Text × Text → Bool
equalsText(t1, t2) ≡ t1 = t2,

emptyAccess : Unit → Access*
emptyAccess() ≡
    tl ⟨AccessValueOrVariableName(
                mk_ValueOrVariableName(mk_Id("DUMMY")))⟩,

isinAccess : Access × Access* → Bool
isinAccess(a, al) ≡
    if al = ⟨⟩ then false
    else
       if getHeadTextAccess(al) = getTextAccess(a)
       then true
       else isinAccess(a, tl al)
       end
    end,

getHeadTextAccess : Access* → Text
getHeadTextAccess(al) ≡ getTextAccess(hd al),

getTextAccess : Access → Text
```

```
getTextAccess(a) ≡
  case a of
    AccessValueOrVariableName(vn) → getText(id(vn))
  end,

elementsAccess : Access* → Access*
elementsAccess(al) ≡
  if al = ⟨⟩ then al
  else
    if
      isinAccess(getHeadAccess(al), getTailAccess(al))
    then elementsAccess(getTailAccess(al))
    else
      ⟨getHeadAccess(al)⟩ ^
      elementsAccess(getTailAccess(al))
    end
  end,

lengthAccess : Access* → Int
lengthAccess(al) ≡
  if al = ⟨⟩ then 0
  else 1 + lengthAccess(getTailAccess(al))
  end,

getHeadAccess : Access* → Access
getHeadAccess(al) ≡ hd al,

getTailAccess : Access* → Access*
getTailAccess(al) ≡ tl al,

getHeadTextBinding : Binding* → Text
getHeadTextBinding(bl) ≡ getTextBinding(hd bl),

lengthBinding : Binding* → Int
lengthBinding(bl) ≡
  if bl = ⟨⟩ then 0 else 1 + lengthBinding(tl bl) end,

makeIdList : Id → Id*
makeIdList(id) ≡ emptyId() ^ ⟨id⟩,

getValueExprList : ValueExprPair → ValueExpr*
getValueExprList(ve) ≡
  ⟨getFirstFromPair(ve), getSecondFromPair(ve)⟩,
```

```
getFirstFromPair : ValueExprPair → ValueExpr
getFirstFromPair(vep) ≡ first(vep),

getSecondFromPair : ValueExprPair → ValueExpr
getSecondFromPair(vep) ≡ second(vep),

getExpType : ExpType → ExpType
getExpType(et) ≡ et,

getAccessIdSet : AccessResult → Id*
getAccessIdSet(ar) ≡ accessIdSet(ar),

ifExprToList : IfExpr → ValueExpr*
ifExprToList(ie) ≡
    ⟨getCondition(ie)⟩ ^ elsifToList(elsif_list(ie)) ^
    ⟨getElseCase(ie)⟩,

getCondition : IfExpr → ValueExpr
getCondition(ie) ≡ condition(ie),

getElseCase : IfExpr → ValueExpr
getElseCase(ie) ≡ else_case(ie),

emptyVE : Unit → ValueExpr*
emptyVE() ≡
    tl ⟨Make_ValueLiteral(ValueLiteralInteger("1"))⟩,

getElsifCondition : Elsif → ValueExpr
getElsifCondition(ei) ≡ condition(ei),

getElsifCase : Elsif → ValueExpr
getElsifCase(ei) ≡ elsif_case(ei),

elementsId : Id* → Id*
elementsId(idl) ≡
    if idl = ⟨⟩ then idl
    else
        if isinId(getHeadId(idl), getTailId(idl))
        then elementsId(getTailId(idl))
        else
            ⟨getHeadId(idl)⟩ ^ elementsId(getTailId(idl))
        end
    end,
```

interId : $Id^* \times Id^* \rightarrow Id^*$
interId(idl1, idl2) $\equiv$
  if idl1 = $\langle\rangle$ **then** idl1
  else
    if isinId(**hd** idl1, idl2)
    then $\langle$**hd** idl1$\rangle$ $\frown$ interId(**tl** idl1, idl2)
    else interId(**tl** idl1, idl2)
    end
  end,

getHeadId : $Id^* \rightarrow Id$
getHeadId(idl) $\equiv$ **hd** idl,

getTailId : $Id^* \rightarrow Id^*$
getTailId(idl) $\equiv$ **tl** idl,

emptyId : **Unit** $\rightarrow Id^*$
emptyId() $\equiv$ **tl** $\langle$mk_Id(″DUMMY″)$\rangle$,

emptyList : $Binding^* \rightarrow$ **Bool**
emptyList(bl) $\equiv$ bl = $\langle\rangle$,

getTextId : Id $\rightarrow$ **Text**
getTextId(id) $\equiv$ getText(id)

end

Appendix E

ANTLR Grammar

This appendix contains the ANTLR grammar.

---

rsltoastrsl.g

```
header {
        package translator.syntacticanalyzer;
        import translator.lib.*;
        import translator.rslast.*;
        import translator.rsllib.*;
        import java.util.*;
        }

options {
        language="Java";
        }

class RSLParser extends Parser;

options {
        k=2;
        }

tokens {
        SCHEME          = "scheme";
        CLASS           = "class";
        EXTEND          = "extend";
        WITH            = "with";
        TYPE            = "type";
        VALUE           = "value";
        TEST_CASE       = "test_case";
        IS              = "is";
        PRE             = "pre";
```

```
    END         = "end";
    IF          = "if";
    THEN        = "then";
    ELSIF       = "elsif";
    ELSE        = "else";
    CASE        = "case";
    OF          = "of";
    LET         = "let";
    IN          = "in";

    LIST        = "list";
    INFLIST     = "inflist";
    SET         = "set";
    INFSET      = "infset";
    BOOLEAN     = "Bool";
    INT         = "Int";
    NAT         = "Nat";
    REAL        = "Real";
    CHAR        = "Char";
    TEXT        = "Text";
    UNIT        = "Unit";

    FALSE       = "false";
    TRUE        = "true";

    ABS         = "abs";
    INTCAST     = "int";
    REALCAST    = "real";
    LEN         = "len";
    INDS        = "inds";
    ELEMS       = "elems";
    HD          = "hd";
    TL          = "tl";
    MEMBER      = "isin";
    UNION       = "union";
    INTER       = "inter";
    CARD        = "card";
    DOM         = "dom";
    RNG         = "rng";

    WRITE   = "write";
    READ    = "read";

    CHAOS       = "chaos";
    }

rslast returns [RSLAst rslast] {
    rslast = null;
    LibModule lm = null;
```

```
}
:       lm = lib_module
        {rslast = new RSLAst(lm);}
;

lib_module returns [LibModule lm] {
        lm = null;
        SchemeDef sd = null;
        Id identifier = null;
        RSLListDefault<Id> contextList =
        new RSLListDefault<Id>();
}
:        (
                identifier = id
                {contextList.getList().add(identifier);}
                (
                        COMMA identifier = id
                        {contextList.getList().add(identifier);}
                )*
         )?
         sd = scheme_def
         {lm = new LibModule(contextList, sd);}
;

scheme_def returns [SchemeDef sd] {
         sd = null;
         ClassExpr ce = null;
         Id identifier = null;
}
:        SCHEME identifier = id EQUAL ce = class_expr
         {sd = new SchemeDef(identifier, ce);}
;

class_expr returns [ClassExpr ce] {
         ce = null;
}
:        ce = basic_class_expr
         | ce = extending_class_expr
         | ce = scheme_instantiation
;

basic_class_expr returns [BasicClassExpr be]
{
         be = null;
         RSLListDefault<Decl> decl_list =
         new RSLListDefault<Decl>(); Decl d;
}
:        CLASS
         (
```

```
                d = decl
                {decl_list.getList().add(d);}
        )*
        END
        {be = new BasicClassExpr(decl_list);}
;

extending_class_expr returns [ExtendingClassExpr ece] {
        ece = null;
        ClassExpr ce1 = null;
        ClassExpr ce2 = null;
}
:       EXTEND ce1 = class_expr WITH ce2 = class_expr
        {ece = new ExtendingClassExpr(ce1, ce2);}
;

scheme_instantiation returns [SchemeInstantiation si] {
        si = null;
        Id identifier = null;
}
:       identifier = id
        {si = new SchemeInstantiation(identifier);}
;


decl returns [Decl d] {
        d = null;
}
:       VALUE d = value_decl
        | TYPE d = type_decl
        | TEST_CASE d = test_decl
;

//Type Declaration
type_decl returns [TypeDecl td] {
        td = null;
        RSLListDefault<TypeDef>
                        type_def_list =
                        new RSLListDefault<TypeDef>();
        TypeDef def;
}
:       def = type_def
        {type_def_list.getList().add(def);}
        (
                COMMA def = type_def
                {type_def_list.getList().add(def);}
        )*
        {td = new TypeDecl(type_def_list);}
;
```

```
type_def returns [TypeDef td] {
        td = null;
        Id identifier = null;
}
:       td = sort_def
        | td = variant_def
        | td = short_record_def
        | identifier = id
          EQUAL td = abbreviation_def[identifier]
      /*| td = union_def[identifier])*/
;

sort_def returns [SortDef sd] {
        sd = null;
        Id identifier = null;
}
:       identifier = id
        {sd = new SortDef(identifier);}
;

variant_def returns [VariantDef vd] {
        vd = null;
        Id identifier = null;
        Id identifier2 = null;
        Constructor constructor = null;
        RSLListDefault<Variant>
                        variant_list =
                        new RSLListDefault<Variant>();
        RSLListDefault<ComponentKind> componentKindList =
                                      null;
}
:       identifier = id EQUALEQUAL
        identifier2 = id
        {constructor = new Constructor(identifier2);}
        (
                LPAREN
                componentKindList = component_kind_list
                RPAREN
        )?
        {
                if(componentKindList == null) {
                variant_list.getList().add(
                new Make_Constructor(constructor));
                }
                else {
                variant_list.getList().add(
                new RecordVariant(
                constructor,
```

```
                componentKindList
                ));
                componentKindList = null;
                }
        }
        (
                BAR identifier2 = id
                {constructor = new Constructor(identifier2);}
                (
                        LPAREN
                        componentKindList = component_kind_list
                        RPAREN
                )?
                {
                        if(componentKindList == null) {
                        variant_list.getList().add(
                        new Make_Constructor(constructor));
                        }
                        else {
                        variant_list.getList().add(
                        new RecordVariant(
                            constructor,
                            componentKindList
                        ));
                        componentKindList = null;
                        }
                }
        )*
        {vd = new VariantDef(identifier, variant_list);}
;

union_def[Id identifier] returns [UnionDef ud] {
        ud = null;
        NameOrWildcard nw = null;
        RSLListDefault<NameOrWildcard>
                nameOrWildcardList =
                new RSLListDefault<NameOrWildcard>();
}
:       nw = name_or_wildcard
        {nameOrWildcardList.getList().add(nw);}
        (
                BAR nw = name_or_wildcard
                {nameOrWildcardList.getList().add(nw);}
        )*
        {ud = new UnionDef(identifier, nameOrWildcardList);}
;

name_or_wildcard returns [NameOrWildcard now] {
        Id var = null;
```

```
        now = null;
}
:       var = qualified_id
        {now = new Name(new ValueOrVariableName(var));}
        | UNDERSCORE {now = new Wildcard();}
;


short_record_def returns [ShortRecordDef srd] {
        srd = null;
        Id identifier = null;
        RSLListDefault<ComponentKind> componentKindString = null;
}
:       identifier = id COLON COLON
        componentKindString = component_kind_string
        {srd =
             new ShortRecordDef(identifier, componentKindString);}
;

component_kind_list returns
[RSLListDefault<ComponentKind> componentKindList] {
        componentKindList = new RSLListDefault<ComponentKind>();
        Id identifier1 = null;
        Id identifier2 = null;
        TypeExpr typeExpr = null;
        TypeExpr typeExpr2 = null;
}
:       (identifier1 = id COLON)?
        typeExpr = type_expr
        (LT TOTALARROW identifier2 = id)?
        {
                componentKindList.getList().add(
                new ComponentKind(
                    (identifier1 != null ?
                    new Destructor(identifier1) :
                    new NoDestructor()),
                    typeExpr,
                    (identifier2 != null ?
                    new Reconstructor(identifier2) :
                    new NoReconstructor())));
                identifier1 = null;
                identifier2 = null;
        }
        (
            COMMA
            (identifier1 = id COLON)?
            typeExpr2 = type_expr
            (LT TOTALARROW identifier2 = id)?
            {
```

```
                    componentKindList.getList().add(
                    new ComponentKind(
                        (identifier1 != null ?
                        new Destructor(identifier1) :
                        new NoDestructor()),
                        typeExpr2,
                        (identifier2 != null ?
                        new Reconstructor(identifier2) :
                        new NoReconstructor())));
                    identifier1 = null;
                    identifier2 = null;
                }
        )*
;

component_kind_string returns
[RSLListDefault<ComponentKind> componentKindString] {
        componentKindString =
                new RSLListDefault<ComponentKind>();
        Id identifier1 = null;
        Id identifier2 = null;
        TypeExpr typeExpr = null;
}
:       (
                (identifier1 = id COLON)?
                typeExpr = type_expr
                (LT TOTALARROW identifier2 = id)?
                {
                componentKindString.getList().add(
                new ComponentKind(
                    new Destructor(identifier1),
                    typeExpr,
                    identifier2 != null ?
                    new Reconstructor(identifier2) :
                    new NoReconstructor()));
                    identifier1 = null;
                    identifier2 = null;
                }
        )+
;

abbreviation_def[Id identifier] returns [AbbreviationDef ad] {
        ad = null;
        TypeExpr typeExpr = null;
}
:       typeExpr = type_expr
        {ad = new AbbreviationDef(identifier, typeExpr);}
;
```

```
//Value Declaration
value_decl returns [ValueDecl vd] {
        vd = null;
        RSLListDefault<ValueDef>
                        value_def_list =
                        new RSLListDefault<ValueDef>();
        ValueDef def;
}
:       def = value_def
        {value_def_list.getList().add(def);}
        (
                COMMA def = value_def
                {value_def_list.getList().add(def);}
        )*
        {vd = new ValueDecl(value_def_list);}
;

value_def returns [ValueDef vd] {
        vd = null;
        SingleTyping st = null;
}
:       st = single_typing
        (
                vd = explicit_function_def[st]
                |
                vd = explicit_value_def[st]
        )
;

explicit_value_def[SingleTyping st] returns [ExplicitValueDef evd] {
        evd = null;
        ValueExpr ve = null;
}
:       EQUAL ve = value_expr
        {evd = new ExplicitValueDef(st, ve);}
;

explicit_function_def[SingleTyping st]
returns [ExplicitFunctionDef efd] {
        efd = null;
        IdApplication ia = null;
        ValueExpr ve = null;
        ValueExpr cond = null;
        OptionalPreCondition ocond = null;
}
:       ia = id_application IS ve = value_expr
        (PRE cond = value_expr)?
        {
                if(cond == null) {
```

```
                        ocond = new NoPreCondition();
                }
                else {
                        ocond = new PreCondition(cond);}
        }
        {efd = new ExplicitFunctionDef(st, ia, ve, ocond);}
;

id_application returns [IdApplication ia] {
        ia = null;
        Id identifier = null;
        FormalFunctionParameter ffp = null;
}
:       identifier = id LPAREN
        ffp = formal_function_parameter
        RPAREN
        {ia =  new IdApplication(identifier, ffp);}
;

formal_function_parameter returns [FormalFunctionParameter ffp] {
        ffp = null;
        RSLListDefault<Binding> binding_list =
                                new RSLListDefault<Binding>();
        Binding b = null;
}
:       (
                b = binding
                {binding_list.getList().add(b);}
                (
                        COMMA b = binding
                        {binding_list.getList().add(b);}
                )*
        )?
        {ffp = new FormalFunctionParameter(binding_list);}
;

single_typing returns [SingleTyping st] {
        st = null;
        Binding b = null;
        TypeExpr te = null;
}
:        b = binding COLON te = type_expr
         {st = new SingleTyping(b, te);}
;

typing returns [Typing t] {
        t = null;
        Binding b = null;
        RSLListDefault<Binding> binding_list =
```

```
                              new RSLListDefault<Binding>();
        TypeExpr te = null;
}
:
        b = binding
        (
                COLON te = type_expr
                {t = new Make_SingleTyping(
                        new SingleTyping(b, te));}
                |
                COMMA
                {binding_list.getList().add(b);}
                b = binding
                {binding_list.getList().add(b);}
                (
                        COMMA b = binding
                        {binding_list.getList().add(b);}
                )*
                COLON te = type_expr
                {t = new Make_MultipleTyping(
                        new MultipleTyping(binding_list, te));}
        )
;

binding returns [Binding b] {
        b = null;
        Binding b1 = null;
        Id identifier = null;
        RSLListDefault<Binding> binding_list =
                              new RSLListDefault<Binding>();
}
:
        (
                b1 = single_binding
                {b = b1;}
        |
                b1 = prod_binding
                {b = b1;}
        )
;

single_binding returns [Binding b] {
        b = null;
        Id identifier = null;
}
:       identifier = id
        {b = new IdBinding(identifier);}
;
```

```
prod_binding returns [Binding b] {
        b = null;
        Binding b1 = null;
        RSLListDefault<Binding> binding_list =
                                new RSLListDefault<Binding>();
}
:       LPAREN b1 = binding
        {binding_list.getList().add(b1);}
        (
                COMMA b1 = binding
                {binding_list.getList().add(b1);}
        )*
        RPAREN
        {b = new Make_ProductBinding(
           new ProductBinding(binding_list));}
;

//Test Declarations
test_decl returns [TestDecl td] {
        td = null;
        RSLListDefault<TestDef> test_def_list =
                                new RSLListDefault<TestDef>();
        TestDef def;
}
:       def = test_def {test_def_list.getList().add(def);}
        (
                COMMA def = test_def
                {test_def_list.getList().add(def);}
        )*
        {td = new TestDecl(test_def_list);}
;

test_def returns [TestCase td] {
         td = null;
         Id identifier = null;
         ValueExpr ve = null;
}
:       LBRACKET identifier = id RBRACKET ve = value_expr
        {td = new TestCase(identifier, ve);}
;

//Type Expression
type_expr returns [TypeExpr te] {
        te = null;
        FunctionArrow fa = null;
        TypeExpr te2 = null;
        TypeExpr te3 = null;
        ResultDesc rd = null;}
:       te2 = type_expr_pr2
```

```
        (
                (
                        TOTALARROW
                        {fa = new TOTAL_FUNCTION_ARROW();}
                |
                        PARTIAL TOTALARROW
                        {fa = new PARTIAL_FUNCTION_ARROW();}
                )
                rd = result_desc
                {te = new FunctionTypeExpr(te2, fa, rd);}
        )?
        {if(te == null) te = te2;}
;

type_expr_pr2 returns [TypeExpr te] {
        te = null;
        TypeExpr te2 = null;
        TypeExpr te3 = null;
        RSLListDefault<TypeExpr> typeExprList =
                                new RSLListDefault<TypeExpr>();
}
:       te2 = type_expr_pr1
        {typeExprList.getList().add(te2);}
        (
                CROSS te3 = type_expr_pr1
                {typeExprList.getList().add(te3);}
        )*
        {
            if(te3 != null) {
                    te = new TypeExprProduct(typeExprList);
            }
            else {
                    te = te2;
            }
        }
;

type_expr_pr1 returns [TypeExpr te] {
        te = null;
        TypeExpr te2 = null;
        TypeExpr te3 = null;
        SingleTyping st = null;
        ValueExpr ve = null;
}
:       te2 = type_expr_primary
        (
                MINUS
                (
                        LIST
```

```
                        {te = new TypeExprList(
                            new FiniteListTypeExpr(te2));}
                |
                        INFLIST
                        {te = new TypeExprList(
                            new InfiniteListTypeExpr(te2));}
                |
                        SET
                        {te = new TypeExprSet(
                            new FiniteSetTypeExpr(te2));}
                |
                        INFSET
                        {te = new TypeExprSet(
                            new InfiniteSetTypeExpr(te2));}
                |
                        MAP te3 = type_expr_primary
                        {te = new TypeExprMap(
                            new FiniteMapTypeExpr(te2, te3));}
                )
        )?
        {if(te == null) te = te2;}
        |
        LSET BAR st = single_typing FORWHICH
        ve = value_expr BAR RSET
        {te = new SubtypeExpr(st, ve);}
;

result_desc returns [ResultDesc rd] {
        rd = null;
        OptionalReadAccessDesc orad = null;
        OptionalWriteAccessDesc owad = null;
        TypeExpr te = null;
}
:       (READ orad = read_access_desc)?
        (WRITE owad = write_access_desc)?
        te = type_expr_pr2
        {if(orad == null) orad = new NoReadAccessMode();}
        {if(owad == null) owad = new NoWriteAccessMode();}
        {rd = new ResultDesc(orad, owad, te);}
;

read_access_desc returns [OptionalReadAccessDesc orad] {
        orad = null;
        Access a = null;
        RSLListDefault<Access> al =
                new RSLListDefault<Access>();
}
:       a = access {al.getList().add(a);}
        (
```

```
                COMMA a = access
                {al.getList().add(a);}
        )*
        {orad = new ReadAccessDesc(al);}
;

write_access_desc returns [OptionalWriteAccessDesc owad] {
        owad = null;
        Access a = null;
        RSLListDefault<Access> al =
                new RSLListDefault<Access>();
}
:       a = access {al.getList().add(a);}
        (
                COMMA a = access
                {al.getList().add(a);}
        )*
        {owad = new WriteAccessDesc(al);}
;

access returns [Access a] {
       a = null;
       Id ai = null;
}
:       ai = qualified_id
        {a = new AccessValueOrVariableName(
                 new ValueOrVariableName(ai));}
;

type_expr_primary returns [TypeExpr te] {
        te = null;
        TypeExpr te2 = null;
        Id identifier = null;
}
:               te = type_literal
        |
                LPAREN te2 = type_expr RPAREN
                {te = new BracketedTypeExpr(te2);}
        |
                identifier = id
                {te = new TypeName(identifier);}
;

type_literal returns [TypeLiteral tl] {
        tl = null;
}
:       INT {tl = new TypeLiteral(new INT());}
        | REAL
          {tl = new TypeLiteral(new REAL());}
```

```
          | NAT
            {tl = new TypeLiteral(new NAT());}
          | BOOLEAN
            {tl = new TypeLiteral(new BOOL());}
          | CHAR
            {tl = new TypeLiteral(new CHAR());}
          | TEXT
            {tl = new TypeLiteral(new TEXT());}
          | UNIT
            {tl = new TypeLiteral(new UNIT());}
;

//Value Expressions
value_expr returns [ValueExpr ve] {
          ve = null;
}
:         ve = infix_expr_pr9
;

value_expr_pair returns [ValueExprPair vep] {
          vep = null;
          ValueExpr ve1 = null;
          ValueExpr ve2 = null;
}
:         ve1 = value_expr MAPSTO ve2 = value_expr
          {vep = new ValueExprPair(ve1, ve2);}
;

infix_expr_pr9 returns [ValueExpr ve] {
          ve = null;
          ValueExpr ve1 = null;
          InfixOperator rio = null;
          ValueExpr ve2 = null;
          ValueExpr ve3 = null;}
:         ve1 = infix_expr_pr8
          (
                  rio = infix_op_pr9 ve2 = infix_expr_pr8
                  {ve3 = ve1;
                   ve1 = new ValueInfixExpr(ve3, rio, ve2);}
          )*
          {ve = ve1;}
;

infix_expr_pr8 returns [ValueExpr ve] {
          ve = null;
          ValueExpr ve1 = null;
          InfixOperator rio = null;
          ValueExpr ve2 = null;
          ValueExpr ve3 = null;}
```

```
:       ve1 = infix_expr_pr7
        (
                rio = infix_op_pr8 ve2 = infix_expr_pr7
                {ve3 = ve1;
                 ve1 = new ValueInfixExpr(ve3, rio, ve2);}
        )*
        {ve = ve1;}
;

infix_expr_pr7 returns [ValueExpr ve] {
        ve = null;
        ValueExpr ve1 = null;
        InfixOperator rio = null;
        ValueExpr ve2 = null;
        ValueExpr ve3 = null;}
:       ve1 = infix_expr_pr6
        (
                rio = infix_op_pr7 ve2 = infix_expr_pr6
                {ve3 = ve1;
                 ve1 = new ValueInfixExpr(ve3, rio, ve2);}
        )*
        {ve = ve1;}
;

infix_expr_pr6 returns [ValueExpr ve] {
        ve = null;
        ValueExpr ve1 = null;
        InfixOperator rio = null;
        ValueExpr ve2 = null;
        ValueExpr ve3 = null;}
:       ve1 = infix_expr_pr5
        (
                rio = infix_op_pr6 ve2 = infix_expr_pr5
                {ve3 = ve1;
                 ve1 = new ValueInfixExpr(ve3, rio, ve2);}
        )*
        {ve = ve1;}
;

infix_expr_pr5 returns [ValueExpr ve] {
        ve = null;
        ValueExpr ve1 = null;
        InfixOperator rio = null;
        ValueExpr ve2 = null;
        ValueExpr ve3 = null;
}
:       ve1 = infix_expr_pr4
        (
                rio = infix_op_pr5 ve2 = infix_expr_pr4
```

```
                {ve3 = ve1;
                 ve1 = new ValueInfixExpr(ve3, rio, ve2);}
        )*
        {ve = ve1;}
;

infix_expr_pr4 returns [ValueExpr ve] {
        ve =  null;
        ValueExpr ve1 = null;
        InfixOperator rio = null;
        ValueExpr ve2 = null;
        ValueExpr ve3 = null;}
:       ve1 = infix_expr_pr3
        (
                rio = infix_op_pr4 ve2 = infix_expr_pr3
                {ve3 = ve1;
                 ve1 = new ValueInfixExpr(ve3, rio, ve2);}
        )*
        {ve = ve1;}
;

infix_expr_pr3 returns [ValueExpr ve] {
        ve =  null;
        ValueExpr ve1 = null;
        InfixOperator rio = null;
        ValueExpr ve2 = null;
        ValueExpr ve3 = null;}
:       ve1 = disamb_expr
        (
                rio = infix_op_pr3 ve2 = disamb_expr
                {ve3 = ve1;
                 ve1 = new ValueInfixExpr(ve3, rio, ve2);}
        )*
        {ve = ve1;}
;

disamb_expr returns [ValueExpr ve] {
        ve = null;
        ValueExpr ve1 = null;
        TypeExpr te = null;
}
:       ve1 = prefix_expr
        (COLON te = type_expr)?
        {
                if(te != null) {
                      ve = new DisambiguationExpr(ve1, te);
                }
                else {
                      ve = ve1;}
```

```
                }
;

prefix_expr returns [ValueExpr ve] {
        ve = null;
        PrefixOperator po = null;
        ValueExpr pe = null;}
:                 po = prefix_op pe = prefix_expr
                  {ve = new ValuePrefixExpr(po, pe);}
        |
                  ve = primary_value_expr
;

primary_value_expr returns [ValueExpr ve] {
        ve = null;
        ValueExpr condition = null;
        ValueExpr condition2 = null;
        RSLListDefault<Elsif> elsif_branch_list =
                new RSLListDefault<Elsif>();
        RSLListDefault<ValueExpr> optional_value_expr_list =
                new RSLListDefault<ValueExpr>();
        RSLListDefault<ValueExprPair>
                optional_value_expr_pair_list =
                        new RSLListDefault<ValueExprPair>();
        RSLListDefault<ValueExpr> product_value_expr_list =
                new RSLListDefault<ValueExpr>();
        RSLListDefault<CaseBranch> caseBranchList =
                new RSLListDefault<CaseBranch>();
        RSLListDefault<Typing> typing_list =
                new RSLListDefault<Typing>();
        ValueExpr ve1 = null;
        ValueExpr ve2 = null;
        ValueExpr ve3 = null;
        ValueExprPair vep1 = null;
        ValueExprPair vep2 = null;
        SetExpr se = null;
        ListExpr le = null;
        MapExpr me = null;
        Binding b = null;
        OptionalRestriction or = null;
        Typing ty = null;
        Pattern p = null;
        LetDef ld = null;
        RSLListDefault<LetDef> letDefList =
        new RSLListDefault<LetDef>();
        ValueLiteral vl = null;
        BasicExpr be = null;
}
:       LPAREN ve1 = value_expr
```

```
(
        RPAREN
        {ve = new BracketedExpr(ve1);}
|
        COMMA ve2 = value_expr
        {product_value_expr_list.getList().add(ve1);}
        {product_value_expr_list.getList().add(ve2);}
        (
             COMMA ve1 = value_expr
             {product_value_expr_list.getList().add(ve1);}
        )*
        RPAREN
        {ve = new ProductExpr(product_value_expr_list);}
)
|
IF condition = value_expr THEN ve1 = value_expr
(
        ELSIF condition2 = value_expr
        THEN ve2 = value_expr
        {elsif_branch_list.getList().add(new Elsif(
                                              condition2,
                                              ve2));}
)*
(ELSE ve2 = value_expr)?
END
{ve = new Make_IfExpr(
          new IfExpr(
          condition,
          ve1,
          elsif_branch_list,
          ve2));}
|
vl = value_literal {ve = new Make_ValueLiteral(vl);}
|
be = basic_expr {ve = new Make_BasicExpr(be);}
|
ve1 = name
LPAREN
(
        ve2 = value_expr
        {optional_value_expr_list.getList().add(ve2);}
        (
            COMMA
            ve2 = value_expr
            {optional_value_expr_list.getList().add(ve2);}
        )*
)?
RPAREN
{ve = new ApplicationExpr(
```

```
            ve1 ,
            optional_value_expr_list ) ;}
    |
    ve = name
    |
    LSET
    (
          ve1 = value_expr
          (
              DOT DOT ve2 = value_expr RSET
              {se = new RangedSetExpr(ve1 , ve2 ) ;}
          |
              {optional_value_expr_list . getList () . add(ve1 ) ;}
              (
                  COMMA ve1 = value_expr
                  {optional_value_expr_list . getList () . add(ve1 ) ;}
              )*
              RSET
              {se = new EnumeratedSetExpr(
                    new ValueExprList(
                         optional_value_expr_list ) ) ;}
          |
              BAR
              (ty = typing {typing_list . getList () . add(ty ) ;})+
              (FORWHICH ve2 = value_expr )?
              RSET
              {if(ve2 == null) {
                  or = new NoRestriction () ;}
                else {
                  or = new Restriction(ve2 ) ;
                }}
                {se =
                   new ComprehendedSetExpr(ve1 ,
                                    typing_list , or ) ;}
          )
          |
              RSET
              {se =
                  new EnumeratedSetExpr(
                      new NoValueExprList () ) ;}
    )
    {ve = new Make_SetExpr(se ) ;}
    |
    LLIST
    (
          ve1 = value_expr
          (
          DOT DOT ve2 = value_expr RLIST
          {le = new RangedListExpr(ve1 , ve2 ) ;}
```

```
            |
            {optional_value_expr_list.getList().add(ve1);}
                (
                    COMMA ve1 = value_expr
                    {optional_value_expr_list.getList().add(ve1);}
                )*
            RLIST
            {le =
                new EnumeratedListExpr(
                    new ValueExprList(
                        optional_value_expr_list));}
            |
            BAR b = binding IN ve2 = value_expr
            (FORWHICH ve3 = value_expr)?
            RLIST
            {if(ve3 == null) {
                    or = new NoRestriction();}
             else {
                  or = new Restriction(ve3);
             }}
             {le = new ComprehendedListExpr(ve1, b, ve2, or);}
             )
        |
        RLIST
        {le = new EnumeratedListExpr(new NoValueExprList());}
        )
        {ve = new Make_ListExpr(le);}
        |
        LBRACKET
        (
        vep1 = value_expr_pair
             (
             {optional_value_expr_pair_list.getList().add(vep1);}
                 (
                 COMMA vep1 = value_expr_pair
                 {optional_value_expr_pair_list.getList().add(vep1);}
                 )*
             RBRACKET
             {me =
                 new EnumeratedMapExpr(
                     new ValueExprPairList(
                         optional_value_expr_pair_list));}
             |
             BAR
             (ty = typing {typing_list.getList().add(ty);})+
             (FORWHICH ve2 = value_expr)?
             RBRACKET
             {if(ve2 == null) {
                     or = new NoRestriction();}
```

```
              else {
                      or = new Restriction(ve2);
             }}
             {me =
                 new ComprehendedMapExpr(vep1, typing_list, or);}
             )
             |
             RBRACKET
             {me =
                 new EnumeratedMapExpr(new NoValueExprPairList());}
        )
        {ve = new Make_MapExpr(me);}
        |
        CASE ve1 = value_expr OF
        p = pattern TOTALARROW ve2 = value_expr
        {caseBranchList.getList().add(new CaseBranch(p, ve2));}
        (
            COMMA p = pattern TOTALARROW ve2 = value_expr
            {caseBranchList.getList().add(new CaseBranch(p, ve2));}
        )*
        END {ve = new CaseExpr(ve1, caseBranchList);}
        |
        LET ld = let_def
        {letDefList.getList().add(ld);}
        (
                COMMA ld = let_def
                {letDefList.getList().add(ld);}
        )*
        IN ve1 = value_expr
        END {ve = new LetExpr(letDefList, ve1);}
;

let_def returns [LetDef ld] {
        ld = null;
        LetBinding b = null;
        ValueExpr ve = null;
}
:       b = let_binding EQUAL ve = value_expr
        {ld = new LetDef(b, ve);}
;

let_binding returns [LetBinding lb] {
        lb = null;
        Binding b = null;
        Id identifier = null;
        Make_ValueOrVariableName vovn = null;
        Pattern p = null;
        Pattern ip = null;
        RSLListDefault<Pattern> patternList =
```

```
                              new RSLListDefault<Pattern>();
        RSLListDefault<Pattern> innerPatternList =
                              new RSLListDefault<Pattern>();
        ListPattern lp = null;
        Binding b1 = null;
        RSLListDefault<Binding> binding_list =
                              new RSLListDefault<Binding>();
}
:       identifier = id
        (
                {lb = new MakeBinding(new IdBinding(identifier));}
        |
                LPAREN
                p = pattern
                {patternList.getList().add(p);}
                (
                        COMMA p = pattern
                        {patternList.getList().add(p);}
                )*
                RPAREN
                {lb = new MakeRecordPattern(
                           new ValueOrVariableName(identifier),
                           patternList);}
        )
        |
        LPAREN b1 = binding
        {binding_list.getList().add(b1);}
        (
                COMMA b1 = binding
                {binding_list.getList().add(b1);}
        )*
        RPAREN
        {lb =
            new MakeBinding(
                new Make_ProductBinding(
                    new ProductBinding(binding_list)));}
        |
        LLIST
        (
                ip = pattern
                {innerPatternList.getList().add(ip);}
                (
                        COMMA ip = pattern
                        {innerPatternList.getList().add(ip);}
                )*
                RLIST
                {lp =
                    new Make_EnumeratedListPattern(
                           new EnumeratedListPattern(
```

```
                              new InnerPatternList(
                                  innerPatternList)));}
                |
                RLIST
                {lp = new Make_EnumeratedListPattern(
                          new EnumeratedListPattern(
                              new NoInnerPattern()));}
        )
        (
                HAT ip = pattern
                {lp = new ConcatenatedListPattern(
                          new EnumeratedListPattern(
                              new InnerPatternList(
                                  innerPatternList)),
                          ip);}
        )?
        {lb = new MakeListPatternLet(lp);}
;


value_literal returns [ValueLiteral vl] {
        vl = null;
}
:       vl = int_lit
        | vl = bool_lit
        | vl = real_lit
        | vl = text_lit
        | vl = char_lit
;

basic_expr returns[BasicExpr be] {
        be = null;
}
:       CHAOS {be = new CHAOS();}
;

name returns [Make_ValueOrVariableName vn] {
        vn = null;
        Id qi = null;
}
:       qi = qualified_id
        {vn = new Make_ValueOrVariableName(
                  new ValueOrVariableName(qi));}
;
infix_op_pr9 returns [InfixOperator rio] {
        rio = null;
}
:  IMPLIES {rio = new IMPLIES();}
;
```

```
infix_op_pr8 returns [InfixOperator rio] {
        rio = null;
}
:       OR {rio = new OR();}
;

infix_op_pr7 returns [InfixOperator rio] {
        rio = null;
}
:       AND {rio = new AND();}
;

infix_op_pr6 returns [InfixOperator rio] {
        rio = null;
}
:       EQUAL {rio = new EQUAL();}
        | NOTEQUAL {rio = new NOTEQUAL();}
        | GT {rio = new GT();}
        | LT {rio = new LT();}
        | GT EQUAL {rio = new GTE();}
        | LT EQUAL {rio = new LTE();}
        | PROPSUBSET EQUAL {rio = new SUBSET();}
        | PROPSUBSET {rio = new PROPSUBSET();}
        | REVPROPSUBSET EQUAL {rio = new REVSUBSET();}
        | REVPROPSUBSET {rio = new REVPROPSUBSET();}
        | MEMBER {rio = new MEMBER();}
        | CURLYDASH MEMBER {rio = new NOTMEMBER();}
;

infix_op_pr5 returns [InfixOperator rio] {
        rio = null;
}
:       PLUS {rio = new PLUS();}
        | MINUS {rio = new MINUS();}
        | BACKSLASH {rio = new BACKSLASH();}
        | HAT {rio = new HAT();}
        | UNION {rio = new UNION();}
        | OVERRIDE {rio = new OVERRIDE();}
;

infix_op_pr4 returns [InfixOperator rio] {
        rio = null;
}
:       STAR {rio = new STAR();}
        | SLASH {rio = new SLASH();}
        | INTER {rio = new INTER();}
        | COMPOSITION {rio = new COMPOSITION();}
;
```

```
infix_op_pr3 returns [InfixOperator rio] {
        rio = null;
}
:       EXP {rio = new EXP();}
;

prefix_op returns [PrefixOperator rpo] {
        rpo = null;
}
:       ABS {rpo = new ABS();}
        | INTCAST {rpo = new INTCAST();}
        | REALCAST {rpo = new REALCAST();}
        | LEN {rpo = new LEN();}
        | INDS {rpo = new INDS();}
        | ELEMS {rpo = new ELEMS();}
        | HD {rpo = new HD();}
        | TL {rpo = new TL();}
        | CURLYDASH {rpo = new NOT();}
        | MINUS {rpo = new PREFIXMINUS();}
        | PLUS {rpo = new PREFIXPLUS();}
        | CARD {rpo = new CARD();}
        | DOM {rpo = new DOM();}
        | RNG {rpo = new RNG();}
;

qualified_id returns [Id qi] {
        qi = null;
}
:       qi = id
;

id returns [Id i] {
        i = null;
}
:       a:IDENT {i = new Id(a.getText());}
;

pattern returns[Pattern p] {
        p = null;
        Make_ValueOrVariableName vovn = null;
        Make_ValueOrVariableName vovn2 = null;
        Make_ValueOrVariableName vovn3 = null;
        Pattern ip = null;
        ValueLiteral vl = null;
        RSLListDefault<Pattern> innerPatternList =
                                new RSLListDefault<Pattern>();
        ListPattern lp = null;
}
```

```
:       vl = value_literal
        {p = new ValueLiteralPattern(vl);}
        |
        vovn = name
        {p = new NamePattern(
                vovn.value_or_variable_name().id(),
                new NoOptionalId());}
        |
        vovn = name LPAREN ip = pattern
        {innerPatternList.getList().add(ip);}
        (
                COMMA ip = pattern
                {innerPatternList.getList().add(ip);}
        )*
        RPAREN
        {p = new RecordPattern(
                vovn.value_or_variable_name(),
                innerPatternList);}
        |
        UNDERSCORE {p = new WildcardPattern();}
        |
        LPAREN ip = pattern
        {innerPatternList.getList().add(ip);}
        (
                COMMA ip = pattern
                {innerPatternList.getList().add(ip);}
        )*
        RPAREN
        {p = new ProductPattern(innerPatternList);}
        |
        LLIST
        (
                ip = pattern
                {innerPatternList.getList().add(ip);}
                (
                        COMMA ip = pattern
                        {innerPatternList.getList().add(ip);}
                )*
                RLIST
                {lp = new Make_EnumeratedListPattern(
                        new EnumeratedListPattern(
                            new InnerPatternList(
                                innerPatternList)));}
                |
                RLIST
                {lp = new Make_EnumeratedListPattern(
                        new EnumeratedListPattern(
                            new NoInnerPattern()));}
        )
```

```
        (
                HAT ip = pattern
                {lp = new ConcatenatedListPattern(
                        new EnumeratedListPattern(
                            new InnerPatternList(
                                innerPatternList)), ip);}
        )?
        {p = new MakeListPattern(lp);}
;

int_lit returns [ValueLiteralInteger i] {
        i = null;
}
:       a:INTEGER_LITERAL
        {i = new ValueLiteralInteger(a.getText());}
;

real_lit returns [ValueLiteralReal r] {
        r = null;
}
:       a:REAL_LITERAL
        {r = new ValueLiteralReal(a.getText());}
;

bool_lit returns [ValueLiteralBool b] {
        b = null;
}
:       FALSE {b = new ValueLiteralBool("false");}
        | TRUE  {b = new ValueLiteralBool("true");}
;

text_lit returns [ValueLiteralText t] {
        t = null;
}
:       a:TEXT_LITERAL
        {t = new ValueLiteralText(a.getText());}
;

char_lit returns [ValueLiteralChar c] {
        c = null;
}
:       a:CHAR_LITERAL
        {c = new ValueLiteralChar(a.getText());}
;

class RSLLexer extends Lexer;
options {
        charVocabulary='\u0000'..'\u007F'; //Allow only ascii
        k=2;
```

```
        testLiterals=false;
}

COMMA           : ",";
COLON           : ":";
CROSS           : "><";
TOTALARROW      : "->";
FORWHICH        : ":-";
PARTIAL         : "-~";
INFMAP          : "~m->";
MAP             : "m->";
MAPSTO          : "+>";

LPAREN          : "(";
RPAREN          : ")";
LBRACKET        : "[";
RBRACKET        : "]";
LSET            : "{";
RSET            : "}";
PLUS            : "+";
MINUS           : "-";
STAR            : "*";
EXP             : "**";
SLASH           : "/";
BACKSLASH       : "\\";
HAT             : "^";
EQUAL           : "=";
EQUALEQUAL      : "==";
NOTEQUAL        : "~=";
LT              : "<";
GT              : ">";
LLIST           : "<.";
RLIST           : ".>";
CURLYDASH       : "~";
BAR             : "|";
UNDERSCORE      : "_";
PROPSUBSET      : "<<";
REVPROPSUBSET   : ">>";
OVERRIDE        : "!!";
COMPOSITION     : "#";
OR              : "\\/";
AND             : "/\\";
IMPLIES         : "=>";

/*DOT is matched by rule for integers and reals.*/
//DOT           : ".";

CHAR_LITERAL    : '\''! (ESC|~('\''|'\\')) '\''!;
```

```
TEXT_LITERAL    :  '"'! (ESC|~('"'|'\\'))* '"'!;

protected
ESC
:       '\\'
(       'r'
        | 'n'
        | 't'
        | 'a'
        | 'b'
        | 'f'
        | 'v'
        | '?'
        | '\\'
        | '\''
        | '"'
        | OCT_DIGIT
(
options {
        warnWhenFollowAmbig = false;
}
:       OCT_DIGIT
(
options {
        warnWhenFollowAmbig = false;}
:       OCT_DIGIT
)?
)?
        | 'x'
(
options {
        greedy=true;
}
:
        '0'..'9'
        | 'a'..'f'
        | 'A'..'F'
)+
)
;

protected
OCT_DIGIT
:    '0'..'7'
;

// a numeric literal
INTEGER_LITERAL {
        boolean isDecimal=false; Token t= null;
```

```
}
:    '.' {_ttype = DOT;}
|
(
'0'
// special case for just '0'
|
('1'..'9') ('0'..'9')*
// non-zero decimal
)
(    ('.' ('0'..'9')+) =>
('.' ('0'..'9')+)
{_ttype = REAL_LITERAL;}
)?
;

IDENT
options {testLiterals=true;}
:         ('a'..'z'|'A'..'Z')
('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
;

WS
:    (    ' '
|    '\r' '\n' {newline();}
|    '\n' {newline();}
|    '\t'
)
{$setType(Token.SKIP);}
;

// multiple-line comments
ML_COMMENT
:         "/*"
(
/*
'\r' '\n' can be matched in one alternative or by matching
'\r' in one iteration and '\n' in another.  I am trying to
handle any flavor of newline that comes in, but the language
that allows both "\r\n" and "\r" and "\n" to all be valid
newline is ambiguous.  Consequently, the resulting grammar
must be ambiguous.  I'm shutting this warning off.
*/
options {
          generateAmbigWarnings=false;
}
:
{ LA(2)!='/' }? '*'
|         '\r' '\n'                    {newline();}
```

```
|       '\r'                    {newline();}
|       '\n'                    {newline();}
|       ~('*'|'\n'|'\r')
)*
"*/"
{$setType(Token.SKIP);}
;
```

Appendix F

Source Code

This appendix contains the hand written source code of the transformer in Java.

F.1 RSLRunner

RSLRunner.java

```
package translator;

import translator.lib.*;
import translator.rsllib.*;
import translator.rslast.*;
import translator.syntacticanalyzer.*;
import translator.rsltransformer.*;

import java.util.*;
import java.io.*;
import java.text.*;

public class RSLRunner {
    public static RSLRunner instance;
    private static RSLList<TRANSMapEntrance> variables;

    public void transform(String filename, String newFilename,
                          boolean writeFiles) {
        FileReader fr = null;
        try {
            fr = new FileReader(filename + ".rsl");
        }
        catch(FileNotFoundException e) {
            System.out.println("File "
```

```
                         + filename
                         + ".rsl not found!");
        System.exit(1);
    }
    try {

        System.out.println("Starting: " + filename);
        System.out.println("Starting parsing");

        RSLLexer lexer = new RSLLexer(fr);
        RSLParser parser = new RSLParser(lexer);
        RSLAst arslast = parser.rslast();

        if(arslast == null) {
            System.out.println("Tree not created!");
            System.exit(1);
        }

        System.out.println("Ending parsing\n");

        try {
            StringRSLAstVisitor arslAstVisitor =
                new StringRSLAstVisitor();
            arslast.accept(arslAstVisitor);
        }
        catch(NullPointerException npe) {
            System.out.println("Tree not created "
                               + "correct!\n"
                               + "The RSL "
                               + "specification "
                               + "is not "
                               + "within the "
                               + "subset of RSL");
            System.exit(1);
        }

        File file = new File(filename);

        try {
            TRResult result =
                TransformerRSL1.
                TRRSLAst(arslast,
                         newFilename,
                         new TRANS(variables));


            if(result instanceof RSLast_transformable) {

                System.out.println("Transformable");
```

```
            RSLast_transformable resultast =
                (RSLast_transformable)result;
            RSLAst irslast = resultast.result();

            StringRSLAstVisitor irslAstVisitor =
                new StringRSLAstVisitor();
            irslast.accept(irslAstVisitor);

            System.out.println("RSL:\n"
                                   + irslAstVisitor.
                                   getResult());
            System.out.println("Ending: " + filename);

            try {
                if(file.getParent() != null) {
                    newFilename = file.getParent()
                        + file.separator
                        + newFilename;
                }
                FileWriter fw =
                    new FileWriter(newFilename
                                   + ".rsl", false);
                fw.write("/*"
                         + DateFormat.
                         getDateTimeInstance(DateFormat.
                                             LONG,
                                             DateFormat.
                                             LONG,
                                             Locale.UK).
                         format(new Date())
                         + "*/\n");
                fw.write(irslAstVisitor.getResult().
                         toString());
                fw.flush();
                fw.close();
            }
            catch (IOException ioe) {
                ioe.printStackTrace();
                System.exit(1);
            }
        }
        else {
            System.out.println("The file: "
                                   + filename
                                   + ".rsl cannot "
                                   + "be transformed");
        }
    }
```

```
            catch(NullPointerException npe) {
                System.out.println("The RSL specification "
                                   + "is not "
                                   + "within the subset "
                                   + "of RSL");
                System.exit(1);
            }
        }

        catch(Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }

    public static void main(String[] args) {
        instance = null;
        try {
            if(args.length == 0 || args.length == 1) {
                System.out.
                    println("USAGE: translator.RSLRunner "
                            + "FILE_WITHOUT_EXTENSION "
                            + "NEW_SPECIFICATION_NAME "
                            + "[variable:type]*");
                System.exit(1);
            }
            else {
                instance = new RSLRunner();
                if(args.length > 1) {
                    variables =
                        new RSLListDefault
                        <TRANSMapEntrance>();
                    String[] types =
                        new String[args.length - 2];
                    String[] variableNames =
                        new String[args.length - 2];

                    for(int i = 2; i < args.length; i++) {
                        String typeVariable = args[i];
                        Id varId =
                            new Id(typeVariable.
                                   substring(0,
                                             typeVariable.
                                             indexOf(':')));
                        Id typeId =
                            new Id(typeVariable.
                                   substring(typeVariable.
                                             indexOf(':')+1));
                        variables.getList().
```

```
                                   add(new TRANSMapEntrance(typeId,
                                                          varId));
                    }
                }
                else {
                    variables = null;
                }
            }
            instance.transform(args[0], args[1], true);
        }
        catch(Exception e) {
            System.out.println("USAGE: translator.RSLRunner "
                               + "FILE_WITHOUT_EXTENSION "
                               + "NEW_SPECIFICATION_NAME "
                               + "[variable:type]*");
        }
    }
}
```

F.2 Visitor Modules

RSLElement.java

```
package translator.lib;

import translator.rslast.*;
import translator.*;

public abstract class RSLElement {

    public abstract void accept(RSLAstVisitor visitor);
}
```

RSLAstVisitor.java

```
package translator.lib;

import translator.rslast.*;

public abstract class RSLAstVisitor {

    public void visitRSLAst(RSLAst rslast) {
        rslast.libmodule().accept(this);
    }

    public void visitLibModule(LibModule module) {
        for (Id id : module.context_list().getList()) {
            id.accept(this);
        }
        module.schemedef().accept(this);
    }

    public void visitSchemeDef(SchemeDef scheme) {
        scheme.id().accept(this);
        scheme.class_expr().accept(this);
    }

    public void visitBasicClassExpr(BasicClassExpr basicClassExpr) {
        for (Decl decl : basicClassExpr.declaration_list().getList()) {
            decl.accept(this);
        }
    }
```

```
public void visitExtendingClassExpr(ExtendingClassExpr
                                    extendingClassExpr) {
    extendingClassExpr.base_class().accept(this);
    extendingClassExpr.extension_class().accept(this);
}

public void visitSchemeInstantiation(SchemeInstantiation
                                     schemeInstantiation) {
    schemeInstantiation.id().accept(this);
}

/*Type Declarations */
public void visitTypeDecl(TypeDecl typeDecl) {
    for (TypeDef typeDef : typeDecl.type_def_list().getList()) {
        typeDef.accept(this);
    }
}

public void visitSortDef(SortDef sortDef) {
    sortDef.sd_id().accept(this);
}

public void visitVariantDef(VariantDef variantDef) {
    variantDef.id().accept(this);
    for (Variant variant : variantDef.variant_list().getList()) {
        variant.accept(this);
    }
}

public void visitUnionDef(UnionDef unionDef) {
    unionDef.ud_id().accept(this);
    for (NameOrWildcard nameOrWildcard :
            unionDef.name_or_wildcard_list().getList()) {
        nameOrWildcard.accept(this);
    }
}

public void visitShortRecordDef(ShortRecordDef shortRecordDef) {
    shortRecordDef.srd_id().accept(this);
    for (ComponentKind componentKind :
            shortRecordDef.component_kind_string().getList()) {
        componentKind.accept(this);
    }
}

public void visitAbbreviationDef(AbbreviationDef abbreviationDef) {
    abbreviationDef.abbr_id().accept(this);
    abbreviationDef.type_expr().accept(this);
}
```

```
    public void visitMake_Constructor(Make_Constructor constructor) {
        constructor.constructor().accept(this);
    }

    public void visitRecordVariant(RecordVariant recordVariant) {
        recordVariant.record_constructor().accept(this);
        for (ComponentKind componentKind :
                recordVariant.component_kind_list().getList()) {
            componentKind.accept(this);
        }
    }

    public void visitConstructor(Constructor constructor) {
        constructor.id().accept(this);
    }

    public void visitComponentKind(ComponentKind componentKind) {
        if (componentKind.optional_destructor()
            instanceof Destructor) {
            componentKind.optional_destructor().accept(this);
        }
        componentKind.type_expr().accept(this);
        if (componentKind.optional_reconstructor()
            instanceof Reconstructor) {
            componentKind.optional_reconstructor().accept(this);
        }
    }

    public void visitDestructor(Destructor destructor) {
        destructor.id().accept(this);
    }

    public void visitNoDestructor(NoDestructor noDestructor) {}

    public void visitReconstructor(Reconstructor reconstructor) {
        reconstructor.id().accept(this);
    }

    public void visitNoReconstructor(NoReconstructor noReconstructor)
        {}

    public void visitName(Name name) {
        name.var().accept(this);
    }

    public void visitWildcard(Wildcard wildcard) {}

    /*Value Declarations*/
```

```
public void visitValueDecl(ValueDecl valueDecl) {
    for (ValueDef valueDef :
            valueDecl.value_def_list().getList()) {
        valueDef.accept(this);
    }
}

public void visitExplicitValueDef(ExplicitValueDef explicitValueDef
    ) {
    explicitValueDef.single_typing().accept(this);
    explicitValueDef.value_expr().accept(this);
}

public void visitExplicitFunctionDef(ExplicitFunctionDef
                                     explicitFunctionDef) {
    explicitFunctionDef.fun_single_typing().accept(this);
    explicitFunctionDef.formal_function_application().accept(this);
    explicitFunctionDef.fun_value_expr().accept(this);
    explicitFunctionDef.pre_cond().accept(this);
}

public void visitMake_SingleTyping(Make_SingleTyping singleTyping)
    {
    singleTyping.single_typing().accept(this);
}

public void visitMake_MultipleTyping(Make_MultipleTyping
                                     multipleTyping) {
    multipleTyping.multiple_typing().accept(this);
}

public void visitSingleTyping(SingleTyping singleTyping) {
    singleTyping.binding().accept(this);
    singleTyping.type_expr().accept(this);
}

public void visitMultipleTyping(MultipleTyping multipleTyping) {
    for (Binding binding : multipleTyping.binding_list().getList())
        {
        binding.accept(this);
    }
    multipleTyping.type_expr().accept(this);
}

public void visitIdApplication(IdApplication idApplication) {
    idApplication.id().accept(this);
    idApplication.formal_function_parameter().accept(this);
}
```

```
public void visitFormalFunctionParameter(FormalFunctionParameter
                                            formalFunctionParameter) {
    for (Binding binding :
             formalFunctionParameter.binding_list().getList()) {
        binding.accept(this);
    }
}

public void visitPreCondition(PreCondition preCondition) {
    preCondition.cond().accept(this);
}

public void visitNoPreCondition(NoPreCondition noPreCondition) {}

public void visitIdBinding(IdBinding binding) {
    binding.id().accept(this);
}

public void visitMake_ProductBinding(Make_ProductBinding binding) {
    binding.prod_binding().accept(this);
}

public void visitProductBinding(ProductBinding prod_binding) {
    for (Binding binding : prod_binding.binding_list().getList()) {
        binding.accept(this);
    }
}

/*Variable Declarations*/
public void visitVariableDecl(VariableDecl variableDecl) {
    for (VariableDef variableDef :
             variableDecl.variable_def_list().getList()) {
        variableDef.accept(this);
    }
}

public void visitSingleVariableDef(SingleVariableDef
   singleVariableDef) {
    singleVariableDef.id().accept(this);
    singleVariableDef.type_expr().accept(this);
    singleVariableDef.optional_initialisation().accept(this);
}

public void visitMultipleVariableDef(MultipleVariableDef
                                      multipleVariableDef) {
    for (Id id : multipleVariableDef.id_list().getList()) {
        id.accept(this);
    }
    multipleVariableDef.m_type_expr().accept(this);
```

```
}

public void visitNoInitialisation(NoInitialisation
                                  noInitialisation) {}

public void visitInitialisation(Initialisation initialisation) {
    initialisation.value_expr().accept(this);
}

/*Test Declarations*/
public void visitTestDecl(TestDecl testDecl) {
    for (TestDef testDef : testDecl.test_def_list().getList()) {
        testDef.accept(this);
    }
}

public void visitTestCase(TestCase testCase) {
    testCase.id().accept(this);
    testCase.value_expr().accept(this);
}

/*Type Expression*/
public void visitTypeLiteral(TypeLiteral typeLiteral) {
    typeLiteral.type_literal().accept(this);
}

public void visitTypeName(TypeName typeName) {
    typeName.id().accept(this);
}

public void visitTypeExprProduct(TypeExprProduct typeExprProduct) {
    for (TypeExpr te : typeExprProduct.component_list().getList())
        {
        te.accept(this);
    }
}

public void visitTypeExprSet(TypeExprSet typeExprSet) {
    typeExprSet.type_expr_set().accept(this);
}

public void visitTypeExprList(TypeExprList typeExprList) {
    typeExprList.type_expr_list().accept(this);
}

public void visitTypeExprMap(TypeExprMap
                             typeExprMap) {
    typeExprMap.type_expr_map().accept(this);
}
```

```
public void visitFunctionTypeExpr(FunctionTypeExpr
                                  functionTypeExpr) {
    functionTypeExpr.type_expr_argument().accept(this);
    functionTypeExpr.function_arrow().accept(this);
    functionTypeExpr.type_expr_result().accept(this);
}

public void visitSubtypeExpr(SubtypeExpr subtypeExpr) {
    subtypeExpr.single_typing().accept(this);
    subtypeExpr.restriction().accept(this);
}

public void visitBracketedTypeExpr(BracketedTypeExpr
                                   bracketedTypeExpr) {
    bracketedTypeExpr.b_type_expr().accept(this);
}

public void visitUNIT(UNIT rsl_unit) {}

public void visitINT(INT rsl_int) {}

public void visitNAT(NAT rsl_nat) {}

public void visitREAL(REAL rsl_real) {}

public void visitBOOL(BOOL rsl_bool) {}

public void visitCHAR(CHAR rsl_char) {}

public void visitTEXT(TEXT rsl_text) {}

public void visitFiniteSetTypeExpr(FiniteSetTypeExpr
                                   finiteSetTypeExpr) {
    finiteSetTypeExpr.type_expr().accept(this);
}

public void visitInfiniteSetTypeExpr(InfiniteSetTypeExpr
                                     infiniteSetTypeExpr) {
    infiniteSetTypeExpr.i_type_expr().accept(this);
}

public void visitFiniteListTypeExpr(FiniteListTypeExpr
                                    finiteListTypeExpr) {
    finiteListTypeExpr.type_expr().accept(this);
}

public void visitInfiniteListTypeExpr(InfiniteListTypeExpr
                                      infiniteListTypeExpr) {
```

```
    infiniteListTypeExpr.i_type_expr().accept(this);
}

public void visitFiniteMapTypeExpr(FiniteMapTypeExpr
                                   finiteMapTypeExpr) {
    finiteMapTypeExpr.type_expr_dom().accept(this);
    finiteMapTypeExpr.type_expr_range().accept(this);
}

public void visitInfiniteMapTypeExpr(InfiniteMapTypeExpr
                                     infiniteMapTypeExpr) {
    infiniteMapTypeExpr.i_type_expr_dom().accept(this);
    infiniteMapTypeExpr.i_type_expr_range().accept(this);
}



public void visitTOTAL_FUNCTION_ARROW(TOTAL_FUNCTION_ARROW
                                      totalFunctionArrow) {}

public void visitPARTIAL_FUNCTION_ARROW(PARTIAL_FUNCTION_ARROW
                                        partialFunctionArrow) {}

public void visitResultDesc(ResultDesc resultDesc) {
    resultDesc.read_access_desc().accept(this);
    resultDesc.write_access_desc().accept(this);
    resultDesc.type_expr().accept(this);
}

public void visitReadAccessDesc(ReadAccessDesc readAccessDesc) {

    for(Access ac : readAccessDesc.access_list().getList()) {
        ac.accept(this);
    }
}

public void visitNoReadAccessMode(NoReadAccessMode
                                  noReadAccessMode) {}

public void visitWriteAccessDesc(WriteAccessDesc
                                 writeAccessDesc) {
    for(Access ac : writeAccessDesc.access_list().getList()) {
        ac.accept(this);
    }
}

public void visitNoWriteAccessMode(NoWriteAccessMode
                                   noWriteAccessMode) {}
```

```
public void visitAccessValueOrVariableName(
    AccessValueOrVariableName
                                              valueOrVariableName) {
    valueOrVariableName.variable_name().accept(this);
}

/*Value Expression*/
public void visitMake_ValueLiteral(Make_ValueLiteral valueLiteral)
    {
    valueLiteral.value_literal().accept(this);
}

public void visitMake_ValueOrVariableName(Make_ValueOrVariableName
                                              valueOrVariableName) {
    valueOrVariableName.value_or_variable_name().accept(this);
}

public void visitMake_BasicExpr(Make_BasicExpr basicExpr) {
    basicExpr.basic_expr().accept(this);
}

public void visitProductExpr(ProductExpr productExpr) {
    for(ValueExpr value_expr :
            productExpr.value_expr_list().getList()) {
        value_expr.accept(this);
    }
}

public void visitMake_SetExpr(Make_SetExpr setExpr) {
    setExpr.set_expr().accept(this);
}

public void visitMake_ListExpr(Make_ListExpr listExpr) {
    listExpr.list_expr().accept(this);
}

public void visitMake_MapExpr(Make_MapExpr mapExpr) {
    mapExpr.map_expr().accept(this);
}

public void visitApplicationExpr(ApplicationExpr applicationExpr) {
    applicationExpr.value_expr().accept(this);
    for (ValueExpr ve :
            applicationExpr.appl_value_expr_list().getList()) {
        ve.accept(this);
    }
}

public void visitDisambiguationExpr(DisambiguationExpr disExpr) {
```

```
    disExpr.dis_value_expr().accept(this);
    disExpr.dis_type_expr().accept(this);
}

public void visitBracketedExpr(BracketedExpr bracketedExpr) {
    bracketedExpr.bracketed_expr().accept(this);
}

public void visitValueInfixExpr(ValueInfixExpr valueInfixExpr) {
    valueInfixExpr.left().accept(this);
    valueInfixExpr.op().accept(this);
    valueInfixExpr.right().accept(this);
}

public void visitValuePrefixExpr(ValuePrefixExpr valuePrefixExpr) {
    valuePrefixExpr.op().accept(this);
    valuePrefixExpr.operand().accept(this);
}

public void visitAssignExpr(AssignExpr assignExpr) {
    assignExpr.id().accept(this);
    assignExpr.assign_value_expr().accept(this);
}

public void visitSequencingExpr(SequencingExpr seqExpr) {
    seqExpr.first().accept(this);
    seqExpr.second().accept(this);
}

public void visitLetExpr(LetExpr letExpr) {
    for (LetDef letDef : letExpr.let_def_list().getList()) {
        letDef.accept(this);
    }
    letExpr.let_value_expr().accept(this);
}

public void visitMake_IfExpr(Make_IfExpr makeIfExpr) {
    makeIfExpr.if_expr().accept(this);
}

public void visitCaseExpr(CaseExpr caseExpr) {
    caseExpr.condition().accept(this);
    for (CaseBranch cb : caseExpr.case_branch_list().getList()) {
        cb.accept(this);
    }
}

public void visitValueLiteralInteger(ValueLiteralInteger
                            valueLiteralInteger) {}
```

```
public void visitValueLiteralReal(ValueLiteralReal
                                  valueLiteralReal) {}

public void visitValueLiteralBool(ValueLiteralBool
                                  valueLiteralBool) {}

public void visitValueLiteralChar(ValueLiteralChar
                                  valueLiteralChar) {}

public void visitValueLiteralText(ValueLiteralText
                                  valueLiteralText) {}

public void visitValueOrVariableName(ValueOrVariableName
                                     valueOrVariableName) {
    valueOrVariableName.id().accept(this);
}

public void visitCHAOS(CHAOS chaos) {}

public void visitRangedSetExpr(RangedSetExpr rangedSetExpr) {
    rangedSetExpr.first().accept(this);
    rangedSetExpr.second().accept(this);
}

public void visitEnumeratedSetExpr(EnumeratedSetExpr
                                   enumeratedSetExpr) {
    enumeratedSetExpr.value_expr_list().accept(this);
}

public void visitComprehendedSetExpr(ComprehendedSetExpr
                                     compSetExpr) {
    compSetExpr.value_expr().accept(this);
    for(Typing ty : compSetExpr.typing_list().getList()) {
        ty.accept(this);
    }
    compSetExpr.restriction().accept(this);
}

public void visitRangedListExpr(RangedListExpr rangedListExpr) {
    rangedListExpr.first().accept(this);
    rangedListExpr.second().accept(this);
}

public void visitEnumeratedListExpr(EnumeratedListExpr
                                    enumeratedListExpr) {
    enumeratedListExpr.value_expr_list().accept(this);
}
```

```
public void visitComprehendedListExpr(ComprehendedListExpr
                                        compListExpr) {
    compListExpr.value_expr().accept(this);
    compListExpr.binding().accept(this);
    compListExpr.in_value_expr().accept(this);
    compListExpr.restriction().accept(this);
}

public void visitEnumeratedMapExpr(EnumeratedMapExpr
                                    enumeratedMapExpr) {
    enumeratedMapExpr.value_expr_pair_list().accept(this);
}

public void visitComprehendedMapExpr(ComprehendedMapExpr
                                      compMapExpr) {
    compMapExpr.value_expr_pair().accept(this);
    for(Typing ty : compMapExpr.typing_list().getList()) {
        ty.accept(this);
    }
    compMapExpr.restriction().accept(this);
}

public void visitValueExprList(ValueExprList vel) {
    for(ValueExpr ve : vel.value_expr_list().getList()) {
        ve.accept(this);
    }
}

public void visitNoValueExprList(NoValueExprList nvel) {}

public void visitRestriction(Restriction restriction){
    restriction.value_expr().accept(this);
}

public void visitNoRestriction(NoRestriction nrestriction) {}

public void visitValueExprPair(ValueExprPair vep) {
    vep.first().accept(this);
    vep.second().accept(this);
}

public void visitValueExprPairList(ValueExprPairList vepl) {
    for(ValueExprPair vep : vepl.pair_list().getList()) {
        vep.accept(this);
    }
}

public void visitNoValueExprPairList(NoValueExprPairList
                                      nvel) {}
```

```
public void visitPLUS(PLUS plus) {}

public void visitMINUS(MINUS minus) {}

public void visitEQUAL(EQUAL equal) {}

public void visitNOTEQUAL(NOTEQUAL notequal) {}

public void visitLT(LT lt) {}

public void visitGT(GT gt) {}

public void visitLTE(LTE lte) {}

public void visitGTE(GTE gte) {}

public void visitHAT(HAT hat) {}

public void visitSTAR(STAR star) {}

public void visitSLASH(SLASH slash) {}

public void visitBACKSLASH(BACKSLASH backslash) {}

public void visitEXP(EXP exp) {}

public void visitSUBSET(SUBSET subset) {}

public void visitPROPSUBSET(PROPSUBSET propsubset) {}

public void visitREVSUBSET(REVSUBSET revsubset) {}

public void visitREVPROPSUBSET(REVPROPSUBSET revpropsubset) {}

public void visitMEMBER(MEMBER member) {}

public void visitNOTMEMBER(NOTMEMBER notmember) {}

public void visitUNION(UNION union) {}

public void visitINTER(INTER inter) {}

public void visitOVERRIDE(OVERRIDE override) {}

public void visitCOMPOSITION(COMPOSITION composition) {}

public void visitIMPLIES(IMPLIES implies) {}
```

```
public void visitOR(OR or) {}

public void visitAND(AND and) {}

public void visitABS(ABS abs) {}

public void visitINTCAST(INTCAST intcast) {}

public void visitREALCAST(REALCAST realcast) {}

public void visitLEN(LEN len) {}

public void visitINDS(INDS inds) {}

public void visitELEMS(ELEMS elems) {}

public void visitHD(HD hd) {}

public void visitTL(TL tl) {}

public void visitNOT(NOT not) {}

public void visitPREFIXMINUS(PREFIXMINUS prefixminus) {}

public void visitPREFIXPLUS(PREFIXPLUS prefixplus) {}

public void visitCARD(CARD card) {}

public void visitDOM(DOM dom) {}

public void visitRNG(RNG rng) {}

public void visitLetDef(LetDef letDef) {
    letDef.binding().accept(this);
    letDef.value_expr().accept(this);
}

public void visitMakeBinding(MakeBinding binding) {
    binding.binding().accept(this);
}

public void visitMakeRecordPattern(MakeRecordPattern
                                    recordPattern) {
    recordPattern.value_or_variable_name().accept(this);
    for (Pattern pattern :
            recordPattern.inner_pattern_list().getList()) {
        pattern.accept(this);
    }
}
```

```
public void visitMakeListPatternLet(MakeListPatternLet
                                    listPattern) {
    listPattern.list_pattern().accept(this);
}

public void visitMakeListPattern(MakeListPattern listPattern) {
    listPattern.list_pattern().accept(this);
}

public void visitMake_EnumeratedListPattern(
    Make_EnumeratedListPattern
                                            enumListPattern) {
    enumListPattern.enum_list_pattern().accept(this);
}

public void visitConcatenatedListPattern(ConcatenatedListPattern
                                         concatListPattern) {
    concatListPattern.c_enum_list_pattern().accept(this);
    concatListPattern.c_inner_pattern().accept(this);
}

public void visitEnumeratedListPattern(EnumeratedListPattern
                                       enumListPattern) {
    enumListPattern.inner_pattern().accept(this);
}

public void visitInnerPatternList(InnerPatternList
                                  innerPatternList) {
    for(Pattern pattern : innerPatternList.pattern_list().getList()
        ) {
        pattern.accept(this);
    }
}

public void visitNoInnerPattern(NoInnerPattern noInnerPattern) {}

public void visitValueLiteralPattern(ValueLiteralPattern
                                     pattern) {
    pattern.value_literal().accept(this);
}

public void visitNamePattern(NamePattern pattern) {
    pattern.id().accept(this);
}

public void visitWildcardPattern(WildcardPattern pattern) {}

public void visitProductPattern(ProductPattern pattern) {
```

```
        for(Pattern p : pattern.p_inner_pattern_list().getList()) {
            p.accept(this);
        }
    }

    public void visitRecordPattern(RecordPattern pattern) {
        pattern.value_or_variable_name().accept(this);
        for(Pattern p : pattern.inner_pattern_list().getList()) {
            p.accept(this);
        }
    }

    public void visitIfExpr(IfExpr ifExpr) {
        ifExpr.condition().accept(this);
        ifExpr.if_case().accept(this);
        for (Elsif elsif : ifExpr.elsif_list().getList()) {
            elsif.accept(this);
        }
        ifExpr.else_case().accept(this);
    }

    public void visitElsif(Elsif elsif) {
        elsif.condition().accept(this);
        elsif.elsif_case().accept(this);
    }

    public void visitCaseBranch(CaseBranch caseBranch) {
        caseBranch.pattern().accept(this);
        caseBranch.value_expr().accept(this);
    }

    /*Common*/
    public void visitMake_Id(Make_Id id) {
        id.id().accept(this);
    }

    public void visitNoOptionalId(NoOptionalId noId) {}

    public void visitId(Id id) {}

    public void visitTrans(Trans trans) {
        trans.type_id().accept(this);
        trans.var_id().accept(this);
    }
}
```

StringRSLAstVisitor.java

```
package translator.lib;

import translator.rslast.*;

public class StringRSLAstVisitor
    extends RSLAstVisitor {
    private StringBuffer result;

    public StringRSLAstVisitor() {
        this.result = new StringBuffer();
    }

    public String getResult() {
        return result.toString();
    }

    public void visitRSLAst(RSLAst rslast) {
        rslast.libmodule().accept(this);
    }

    public void visitLibModule(LibModule module) {
        for (Id id : module.context_list().getList()) {
            id.accept(this);
            result.append(", ");
        }
        if (module.context_list().len() > 0) {
            result.delete(result.length() - 2, result.length());
        }
        result.append("\n");
        module.schemedef().accept(this);
    }

    public void visitSchemeDef(SchemeDef scheme) {
        result.append("scheme ");
        scheme.id().accept(this);
        result.append(" = ");
        scheme.class_expr().accept(this);
    }

    public void visitBasicClassExpr(BasicClassExpr basicClassExpr) {
        result.append("class\n");
        for (Decl decl :
                basicClassExpr.declaration_list().getList()) {
            decl.accept(this);
        }
        result.append("end");
```

```
}

public void visitExtendingClassExpr(ExtendingClassExpr
                                    extendingClassExpr) {
    result.append(" extend ");
    extendingClassExpr.base_class().accept(this);
    result.append(" with ");
    extendingClassExpr.extension_class().accept(this);
}

public void visitSchemeInstantiation(SchemeInstantiation
                                     schemeInstantiation) {
    schemeInstantiation.id().accept(this);
}

/*Type Declarations */
public void visitTypeDecl(TypeDecl typeDecl) {
    result.append("type\n");
    for (TypeDef typeDef : typeDecl.type_def_list().getList()) {
        typeDef.accept(this);
        result.append(",\n");
    }
    if (typeDecl.type_def_list().len() > 0) {
        result.delete(result.length() - 2, result.length());
    }
    result.append("\n");
}

public void visitSortDef(SortDef sortDef) {
    sortDef.sd_id().accept(this);
}

public void visitVariantDef(VariantDef variantDef) {
    variantDef.id().accept(this);
    result.append(" == ");
    for (Variant variant : variantDef.variant_list().getList()) {
        variant.accept(this);
        result.append(" | ");
    }
    if (variantDef.variant_list().len() > 0) {
        result.delete(result.length() - 3, result.length());
    }
}

public void visitUnionDef(UnionDef unionDef) {
    unionDef.ud_id().accept(this);
    result.append(" = ");
    for (NameOrWildcard nameOrWildcard :
            unionDef.name_or_wildcard_list().getList()) {
```

```
            nameOrWildcard.accept(this);
            result.append(" | ");
        }
        if (unionDef.name_or_wildcard_list().len() > 0) {
            result.delete(result.length() - 3, result.length());
        }
    }

    public void visitShortRecordDef(ShortRecordDef shortRecordDef) {
        shortRecordDef.srd_id().accept(this);
        result.append(" :: ");

        for (ComponentKind componentKind :
                shortRecordDef.component_kind_string().getList()) {
            componentKind.accept(this);
            result.append(" ");
        }
    }

    public void visitAbbreviationDef(AbbreviationDef abbreviationDef) {
        abbreviationDef.abbr_id().accept(this);
        result.append(" = ");
        abbreviationDef.type_expr().accept(this);
    }

    public void visitMake_Constructor(Make_Constructor constructor) {
        constructor.constructor().accept(this);
    }

    public void visitRecordVariant(RecordVariant recordVariant) {
        recordVariant.record_constructor().accept(this);
        result.append("(");
        for (ComponentKind componentKind :
                recordVariant.component_kind_list().getList()) {
            componentKind.accept(this);
            result.append(", ");
        }
        if (recordVariant.component_kind_list().len() > 0) {
            result.delete(result.length() - 2, result.length());
        }
        result.append(")");
    }

    public void visitConstructor(Constructor constructor) {
        constructor.id().accept(this);
    }

    public void visitComponentKind(ComponentKind componentKind) {
        componentKind.optional_destructor().accept(this);
```

```
        componentKind.type_expr().accept(this);
        if (componentKind.optional_reconstructor()
            instanceof Reconstructor) {
            result.append(" <-> ");
            componentKind.optional_reconstructor().accept(this);
        }
    }

    public void visitDestructor(Destructor destructor) {
        destructor.id().accept(this);
        result.append(" : ");
    }

    public void visitNoDestructor(NoDestructor noDestructor) {}

    public void visitReconstructor(Reconstructor reconstructor) {
        reconstructor.id().accept(this);
    }

    public void visitNoReconstructor(NoReconstructor
                                     noReconstructor) {}

    public void visitName(Name name) {
        name.var().accept(this);
    }

    public void visitWildcard(Wildcard wildcard) {
        result.append("_");
    }

    /*Value Declarations*/
    public void visitValueDecl(ValueDecl valueDecl) {
        result.append("value\n");
        for (ValueDef valueDef :
                valueDecl.value_def_list().getList()) {
            valueDef.accept(this);
            result.append(",\n");
        }
        if (valueDecl.value_def_list().len() > 0) {
            result.delete(result.length() - 2, result.length());
        }
        result.append("\n");
    }

    public void visitExplicitValueDef(ExplicitValueDef
                                      explicitValueDef) {
        explicitValueDef.single_typing().accept(this);
        result.append(" = ");
        explicitValueDef.value_expr().accept(this);
```

```
}

public void visitExplicitFunctionDef(ExplicitFunctionDef
                                     explicitFunctionDef) {
    explicitFunctionDef.fun_single_typing().accept(this);
    result.append("\n");
    explicitFunctionDef.formal_function_application().accept(this);
    result.append(" is ");
    explicitFunctionDef.fun_value_expr().accept(this);
    explicitFunctionDef.pre_cond().accept(this);
}

public void visitMake_SingleTyping(Make_SingleTyping
                                   singleTyping) {
    singleTyping.single_typing().accept(this);
}

public void visitMake_MultipleTyping(Make_MultipleTyping
                                     multipleTyping) {
    multipleTyping.multiple_typing().accept(this);
}

public void visitSingleTyping(SingleTyping singleTyping) {
    singleTyping.binding().accept(this);
    result.append(" : ");
    singleTyping.type_expr().accept(this);
}

public void visitMultipleTyping(MultipleTyping multipleTyping) {
    for (Binding binding :
            multipleTyping.binding_list().getList()) {
        binding.accept(this);
        result.append(", ");
    }
    if (multipleTyping.binding_list().len() > 0) {
        result.delete(result.length() - 2, result.length());
    }
    result.append(" : ");
    multipleTyping.type_expr().accept(this);
}

public void visitIdApplication(IdApplication idApplication) {
    idApplication.id().accept(this);
    result.append("(");
    idApplication.formal_function_parameter().accept(this);
    result.append(")");
}

public void visitFormalFunctionParameter(FormalFunctionParameter
```

```
                                        formalFunctionParameter) {
    for (Binding binding :
            formalFunctionParameter.binding_list().getList()) {
        binding.accept(this);
        result.append(", ");
    }
    if (formalFunctionParameter.binding_list().len() > 0) {
        result.delete(result.length() - 2, result.length());
    }
}

public void visitPreCondition(PreCondition preCondition) {
    result.append("\npre ");
    preCondition.cond().accept(this);
}

public void visitNoPreCondition(NoPreCondition noPreCondition) {}

public void visitIdBinding(IdBinding binding) {
    binding.id().accept(this);
}

public void visitMake_ProductBinding(Make_ProductBinding binding) {
    binding.prod_binding().accept(this);
}

public void visitProductBinding(ProductBinding prodBinding) {
    result.append("(");
    for (Binding binding : prodBinding.binding_list().getList()) {
        binding.accept(this);
        result.append(", ");
    }
    if (prodBinding.binding_list().len() > 0) {
        result.delete(result.length() - 2, result.length());
    }
    result.append(")");
}

/*Variable Declarations*/
public void visitVariableDecl(VariableDecl variableDecl) {
    result.append("variable\n");
    for (VariableDef variableDef :
            variableDecl.variable_def_list().getList()) {
        variableDef.accept(this);
        result.append(",\n");
    }
    if (variableDecl.variable_def_list().len() > 0) {
        result.delete(result.length() - 2, result.length());
    }
```

```
        result.append("\n");
    }

    public void visitSingleVariableDef(SingleVariableDef
                                    singleVariableDef) {
        singleVariableDef.id().accept(this);
        result.append(" : ");
        singleVariableDef.type_expr().accept(this);
        singleVariableDef.optional_initialisation().accept(this);
    }

    public void visitMultipleVariableDef(MultipleVariableDef
                                      multipleVariableDef) {
        for (Id id : multipleVariableDef.id_list().getList()) {
            id.accept(this);
            result.append(", ");
        }
        result.append(" : ");
        multipleVariableDef.m_type_expr().accept(this);
    }

    public void visitNoInitialisation(NoInitialisation
                                   noInitialisation) {}

    public void visitInitialisation(Initialisation initialisation) {
        result.append(" := ");
        initialisation.value_expr().accept(this);
    }

    /*Test Declarations*/
    public void visitTestDecl(TestDecl testDecl) {
        for (TestDef testDef : testDecl.test_def_list().getList()) {
            testDef.accept(this);
            result.append(",\n");
        }
        if (testDecl.test_def_list().len() > 0) {
            result.delete(result.length() - 2, result.length());
        }
        result.append("\n");
    }

    public void visitTestCase(TestCase testCase) {
        result.append("[");
        testCase.id().accept(this);
        result.append("] ");
        testCase.value_expr().accept(this);
    }

    /*Type Expression*/
```

```
public void visitTypeLiteral(TypeLiteral typeLiteral) {
    typeLiteral.type_literal().accept(this);
}

public void visitTypeName(TypeName typeName) {
    typeName.id().accept(this);
}

public void visitTypeExprProduct(TypeExprProduct
                                 typeExprProduct) {
    for (TypeExpr te :
            typeExprProduct.component_list().getList()) {
        te.accept(this);
        result.append(" >< ");
    }
    if (typeExprProduct.component_list().len() > 0) {
        result.delete(result.length() - 4, result.length());
    }
}

public void visitTypeExprSet(TypeExprSet typeExprSet) {
    typeExprSet.type_expr_set().accept(this);
}

public void visitTypeExprList(TypeExprList typeExprList) {
    typeExprList.type_expr_list().accept(this);
}

public void visitTypeExprMap(TypeExprMap
                             typeExprMap) {
    typeExprMap.type_expr_map().accept(this);
}

public void visitFunctionTypeExpr(FunctionTypeExpr
                                  functionTypeExpr) {
    functionTypeExpr.type_expr_argument().accept(this);
    result.append(" ");
    functionTypeExpr.function_arrow().accept(this);
    result.append(" ");
    functionTypeExpr.type_expr_result().accept(this);
}

public void visitSubtypeExpr(SubtypeExpr subtypeExpr) {
    result.append("{| ");
    subtypeExpr.single_typing().accept(this);
    result.append(" :- ");
    subtypeExpr.restriction().accept(this);
    result.append(" |}");
}
```

```
public void visitBracketedTypeExpr(BracketedTypeExpr
                                   bracketedExpr) {
    result.append("(");
    bracketedExpr.b_type_expr().accept(this);
    result.append(")");
}

public void visitUNIT(UNIT rsl_unit) {
    result.append("Unit");
}

public void visitINT(INT tm_rsl_int) {
    result.append("Int");
}

public void visitNAT(NAT rsl_nat) {
    result.append("Nat");
}

public void visitREAL(REAL rsl_real) {
    result.append("Real");
}

public void visitBOOL(BOOL rsl_bool) {
    result.append("Bool");
}

public void visitCHAR(CHAR rsl_char) {
    result.append("Char");
}

public void visitTEXT(TEXT rsl_text) {
    result.append("Text");
}

public void visitFiniteSetTypeExpr(FiniteSetTypeExpr
                                   finiteSetTypeExpr) {
    finiteSetTypeExpr.type_expr().accept(this);
    result.append("-set");
}

public void visitInfiniteSetTypeExpr(InfiniteSetTypeExpr
                                     infiniteSetTypeExpr) {
    infiniteSetTypeExpr.i_type_expr().accept(this);
    result.append("-infset");
}

public void visitFiniteListTypeExpr(FiniteListTypeExpr
```

```
                                    finiteListTypeExpr) {
        finiteListTypeExpr.type_expr().accept(this);
        result.append("-list");
    }

    public void visitInfiniteListTypeExpr(InfiniteListTypeExpr
                                    infiniteListTypeExpr) {
        infiniteListTypeExpr.i_type_expr().accept(this);
        result.append("-inflist");
    }

    public void visitFiniteMapTypeExpr(FiniteMapTypeExpr
                                    finiteMapTypeExpr) {
        finiteMapTypeExpr.type_expr_dom().accept(this);
        result.append(" -m-> ");
        finiteMapTypeExpr.type_expr_range().accept(this);
    }

    public void visitInfiniteMapTypeExpr(InfiniteMapTypeExpr
                                    infiniteMapTypeExpr) {
        infiniteMapTypeExpr.i_type_expr_dom().accept(this);
        result.append(" -~m-> ");
        infiniteMapTypeExpr.i_type_expr_range().accept(this);
    }

    public void visitTOTAL_FUNCTION_ARROW(TOTAL_FUNCTION_ARROW
                                    totalFunctionArrow) {
        result.append("->");
    }

    public void visitPARTIAL_FUNCTION_ARROW(PARTIAL_FUNCTION_ARROW
                                    partialFunctionArrow) {
        result.append("-~->");
    }

    public void visitResultDesc(ResultDesc resultDesc) {
        resultDesc.read_access_desc().accept(this);
        resultDesc.write_access_desc().accept(this);
        resultDesc.type_expr().accept(this);
    }

    public void visitReadAccessDesc(ReadAccessDesc readAccessDesc) {
        result.append("read ");
        for(Access ac : readAccessDesc.access_list().getList()) {
            ac.accept(this);
            result.append(", ");
        }
        if (readAccessDesc.access_list().len() > 0) {
            result.delete(result.length() - 2, result.length() - 1);
```

```
    }
}

public void visitNoReadAccessMode(NoReadAccessMode
                                  noReadAccessMode) {}

public void visitWriteAccessDesc(WriteAccessDesc
                                 writeAccessDesc) {
    result.append("write ");
    for(Access ac : writeAccessDesc.access_list().getList()) {
        ac.accept(this);
        result.append(", ");
    }
    if (writeAccessDesc.access_list().len() > 0) {
        result.delete(result.length() - 2, result.length() - 1);
    }
}

public void visitNoWriteAccessMode(NoWriteAccessMode
                                   noWriteAccessMode) {}

public void visitAccessValueOrVariableName(
   AccessValueOrVariableName
                                          valueOrVariableName) {
    valueOrVariableName.variable_name().accept(this);
}

/*Value Expression*/
public void visitMake_ValueLiteral(Make_ValueLiteral valueLiteral)
    {
    valueLiteral.value_literal().accept(this);
}

public void visitMake_ValueOrVariableName(Make_ValueOrVariableName
                                          valueOrVariableName) {
    valueOrVariableName.value_or_variable_name().accept(this);
}

public void visitMake_BasicExpr(Make_BasicExpr basicExpr) {
    basicExpr.basic_expr().accept(this);
}

public void visitProductExpr(ProductExpr productExpr) {
    result.append("(");
    for(ValueExpr value_expr :
            productExpr.value_expr_list().getList()) {
        value_expr.accept(this);
        result.append(", ");
    }
```

```
        if (productExpr.value_expr_list().len() > 0) {
            result.delete(result.length() - 2, result.length());
        }
        result.append(")");
    }

    public void visitMake_SetExpr(Make_SetExpr setExpr) {
        setExpr.set_expr().accept(this);
    }

    public void visitMake_ListExpr(Make_ListExpr makeListExpr) {
        makeListExpr.list_expr().accept(this);
    }

    public void visitMake_MapExpr(Make_MapExpr mapExpr) {
        mapExpr.map_expr().accept(this);
    }

    public void visitApplicationExpr(ApplicationExpr applicationExpr) {
        applicationExpr.value_expr().accept(this);
        result.append("(");
        for (ValueExpr ve :
                applicationExpr.appl_value_expr_list().getList()) {
            ve.accept(this);
            result.append(", ");
        }
        if (applicationExpr.appl_value_expr_list().len() > 0) {
            result.delete(result.length() - 2, result.length());
        }
        result.append(")");
    }

    public void visitDisambiguationExpr(DisambiguationExpr disExpr) {
        disExpr.dis_value_expr().accept(this);
        result.append(" : " );
        disExpr.dis_type_expr().accept(this);
    }

    public void visitBracketedExpr(BracketedExpr bracketedExpr) {
        result.append("(");
        bracketedExpr.bracketed_expr().accept(this);
        result.append(")");
    }

    public void visitValueInfixExpr(ValueInfixExpr valueInfixExpr) {
        valueInfixExpr.left().accept(this);
        result.append(" ");
        valueInfixExpr.op().accept(this);
        result.append(" ");
```

```
        valueInfixExpr.right().accept(this);
    }

    public void visitValuePrefixExpr(ValuePrefixExpr
                                     valuePrefixExpr) {
        valuePrefixExpr.op().accept(this);
        valuePrefixExpr.operand().accept(this);
    }

    public void visitAssignExpr(AssignExpr assignExpr) {
        assignExpr.id().accept(this);
        result.append(" := ");
        assignExpr.assign_value_expr().accept(this);
    }

    public void visitSequencingExpr(SequencingExpr seqExpr) {
        seqExpr.first().accept(this);
        result.append(" ; ");
        seqExpr.second().accept(this);
    }

    public void visitLetExpr(LetExpr letExpr) {
        result.append("let ");
        for (LetDef letDef : letExpr.let_def_list().getList()) {
            letDef.accept(this);
            result.append(", ");
        }
        if (letExpr.let_def_list().len() > 0) {
            result.delete(result.length() - 2, result.length());
        }
        result.append(" in ");
        letExpr.let_value_expr().accept(this);
        result.append(" end");
    }

    public void visitMake_IfExpr(Make_IfExpr makeIfExpr) {
        makeIfExpr.if_expr().accept(this);
    }

    public void visitCaseExpr(CaseExpr caseExpr) {
        result.append("case ");
        caseExpr.condition().accept(this);
        result.append(" of\n");
        for (CaseBranch cb : caseExpr.case_branch_list().getList()) {
            cb.accept(this);
            result.append(",\n");
        }
        if (caseExpr.case_branch_list().len() > 0) {
            result.delete(result.length() - 2, result.length());
```

```
        }
        result.append("\n");
        result.append("end");
    }

    public void visitValueLiteralInteger(ValueLiteralInteger
                                         valueLiteralInteger) {
        result.append(valueLiteralInteger.getTextInteger());
    }

    public void visitValueLiteralReal(ValueLiteralReal
                                      valueLiteralReal) {
        result.append(valueLiteralReal.getTextReal());
    }

    public void visitValueLiteralBool(ValueLiteralBool
                                      valueLiteralBool) {
        result.append(valueLiteralBool.getTextBool());
    }

    public void visitValueLiteralChar(ValueLiteralChar
                                      valueLiteralChar) {
        result.append("'");
        result.append(valueLiteralChar.getTextChar());
        result.append("'");
    }

    public void visitValueLiteralText(ValueLiteralText
                                      valueLiteralText) {
        result.append("\"");
        result.append(valueLiteralText.getTextText());
        result.append("\"");
    }

    public void visitValueOrVariableName(ValueOrVariableName
                                         valueOrVariableName) {
        valueOrVariableName.id().accept(this);
    }

    public void visitCHAOS(CHAOS chaos) {
        result.append("chaos");
    }

    public void visitRangedSetExpr(RangedSetExpr rangedSetExpr) {
        result.append("{");
        rangedSetExpr.first().accept(this);
        result.append(" .. ");
        rangedSetExpr.second().accept(this);
        result.append("}");
```

```
    }

    public void visitEnumeratedSetExpr(EnumeratedSetExpr
                                        enumeratedSetExpr) {
        result.append("{");
        enumeratedSetExpr.value_expr_list().accept(this);
        result.append("}");
    }

    public void visitComprehendedSetExpr(ComprehendedSetExpr
                                          compSetExpr) {
        result.append("{");
        compSetExpr.value_expr().accept(this);
        result.append(" | ");
        for(Typing ty : compSetExpr.typing_list().getList()) {
            ty.accept(this);
            result.append(", ");
        }
        if (compSetExpr.typing_list().len() > 0) {
            result.delete(result.length() - 2, result.length());
        }
        compSetExpr.restriction().accept(this);
        result.append("}");
    }

    public void visitRangedListExpr(RangedListExpr rangedListExpr) {
        result.append("<.");
        rangedListExpr.first().accept(this);
        result.append(" .. ");
        rangedListExpr.second().accept(this);
        result.append(".>");
    }

    public void visitEnumeratedListExpr(EnumeratedListExpr
                                         enumeratedListExpr) {
        result.append("<.");
        enumeratedListExpr.value_expr_list().accept(this);
        result.append(".>");
    }

    public void visitComprehendedListExpr(ComprehendedListExpr
                                           compListExpr) {
        result.append("<.");
        compListExpr.value_expr().accept(this);
        result.append(" | ");
        compListExpr.binding().accept(this);
        result.append(" in ");
        compListExpr.in_value_expr().accept(this);
        compListExpr.restriction().accept(this);
```

```
        result.append(".>");
    }

    public void visitEnumeratedMapExpr(EnumeratedMapExpr
                                        enumeratedMapExpr) {
        result.append("[");
        enumeratedMapExpr.value_expr_pair_list().accept(this);
        result.append("]");
    }

    public void visitComprehendedMapExpr(ComprehendedMapExpr
                                          compMapExpr) {
        result.append("[");
        compMapExpr.value_expr_pair().accept(this);
        result.append(" | ");
        for(Typing ty : compMapExpr.typing_list().getList()) {
            ty.accept(this);
            result.append(", ");
        }
        if(compMapExpr.typing_list().len() > 0) {
            result.delete(result.length() - 2, result.length());
        }
        compMapExpr.restriction().accept(this);
        result.append("]");
    }

    public void visitValueExprList(ValueExprList vel) {
        for(ValueExpr ve : vel.value_expr_list().getList()) {
            ve.accept(this);
            result.append(", ");
        }
        if(vel.value_expr_list().len() > 0) {
            result.delete(result.length() - 2, result.length());
        }
    }

    public void visitNoValueExprList(NoValueExprList nvel) {}

    public void visitRestriction(Restriction restriction){
        result.append(" :- ");
        restriction.value_expr().accept(this);
    }

    public void visitNoRestriction(NoRestriction nrestriction) {}

    public void visitValueExprPair(ValueExprPair vep) {
        vep.first().accept(this);
        result.append(" +> ");
        vep.second().accept(this);
```

```
    }

    public void visitValueExprPairList(ValueExprPairList vepl) {
        for(ValueExprPair vep : vepl.pair_list().getList()) {
            vep.accept(this);
            result.append(", ");
        }
        if(vepl.pair_list().len() > 0) {
            result.delete(result.length() - 2, result.length());
        }
    }

    public void visitNoValueExprPairList(NoValueExprPairList nvel) {}

    public void visitPLUS(PLUS rsl_plus) {
        result.append("+");
    }

    public void visitMINUS(MINUS rsl_minus) {
        result.append("-");
    }

    public void visitEQUAL(EQUAL rsl_equal) {
        result.append("=");
    }

    public void visitNOTEQUAL(NOTEQUAL rsl_notequal) {
        result.append("~=");
    }

    public void visitLT(LT rsl_lt) {
        result.append("<");
    }

    public void visitGT(GT rsl_gt) {
        result.append(">");
    }

    public void visitLTE(LTE rsl_lte) {
        result.append("<=");
    }

    public void visitGTE(GTE rsl_gte) {
        result.append(">=");
    }

    public void visitHAT(HAT rsl_hat) {
        result.append("^");
    }
```

```
public void visitSTAR(STAR rsl_star) {
    result.append("*");
}

public void visitSLASH(SLASH rsl_slash) {
    result.append("/");
}

public void visitBACKSLASH(BACKSLASH rsl_backslash) {
    result.append("\\");
}

public void visitEXP(EXP rsl_exp) {
    result.append("**");
}

public void visitSUBSET(SUBSET rsl_subset) {
    result.append("<<=");
}

public void visitPROPSUBSET(PROPSUBSET rsl_propsubset) {
    result.append("<<");
}

public void visitREVSUBSET(REVSUBSET rsl_revsubset) {
    result.append(">>=");
}

public void visitREVPROPSUBSET(REVPROPSUBSET rsl_revpropsubset) {
    result.append(">>");
}

public void visitMEMBER(MEMBER rsl_member) {
    result.append("isin");
}

public void visitNOTMEMBER(NOTMEMBER rsl_notMEMBER) {
    result.append("~isin");
}

public void visitUNION(UNION rsl_union) {
    result.append("union");
}

public void visitINTER(INTER rsl_inter) {
    result.append("inter");
}
```

```
    public void visitOVERRIDE(OVERRIDE rsl_override) {
        result.append("!!");
    }

    public void visitCOMPOSITION(COMPOSITION rsl_composition) {
        result.append("#");
    }

    public void visitIMPLIES(IMPLIES implies) {
        result.append("=>");
    }

    public void visitOR(OR or) {
        result.append(" \\/ ");
    }

    public void visitAND(AND and) {
        result.append(" /\\ ");
    }

    public void visitABS(ABS rsl_abs) {
        result.append("abs ");
    }

    public void visitINTCAST(INTCAST rsl_intcast) {
        result.append("int ");
    }

    public void visitREALCAST(REALCAST rsl_realcast) {
        result.append("real ");
    }

    public void visitLEN(LEN rsl_len) {
        result.append("len ");
    }

    public void visitINDS(INDS rsl_inds) {
        result.append("inds ");
    }

    public void visitELEMS(ELEMS rsl_elems) {
        result.append("elems ");
    }

    public void visitHD(HD rsl_hd) {
        result.append("hd ");
    }

    public void visitTL(TL rsl_tl) {
```

```
        result.append("tl ");
    }

    public void visitNOT(NOT rsl_not) {
        result.append("~");
    }

    public void visitPREFIXMINUS(PREFIXMINUS rsl_prefixminus) {
        result.append("-");
    }

    public void visitPREFIXPLUS(PREFIXPLUS prefixplus) {
        result.append("+");
    }

    public void visitCARD(CARD rsl_card) {
        result.append("card ");
    }

    public void visitDOM(DOM rsl_dom) {
        result.append("dom ");
    }

    public void visitRNG(RNG rsl_rng) {
        result.append("rng ");
    }

    public void visitLetDef(LetDef letDef) {
        letDef.binding().accept(this);
        result.append(" = " );
        letDef.value_expr().accept(this);
    }

    public void visitMakeBinding(MakeBinding binding) {
        binding.binding().accept(this);
    }

    public void visitMakeRecordPattern(MakeRecordPattern recordPattern)
        {
        recordPattern.value_or_variable_name().accept(this);
        result.append("(");
        for (Pattern pattern :
                recordPattern.inner_pattern_list().getList()) {
            pattern.accept(this);
            result.append(", " );
        }
        if (recordPattern.inner_pattern_list().len() > 0) {
            result.delete(result.length() - 2, result.length());
        }
```

```
        result.append(")");
    }

    public void visitMakeListPatternLet(MakeListPatternLet listPattern)
        {
        listPattern.list_pattern().accept(this);
    }

    public void visitMakeListPattern(MakeListPattern listPattern) {
        listPattern.list_pattern().accept(this);
    }

    public void visitMake_EnumeratedListPattern(
        Make_EnumeratedListPattern
                                        enumListPattern) {
        enumListPattern.enum_list_pattern().accept(this);
    }

    public void visitConcatenatedListPattern(ConcatenatedListPattern
                                        concatListPattern) {
        concatListPattern.c_enum_list_pattern().accept(this);
        result.append(" ^ ");
        concatListPattern.c_inner_pattern().accept(this);
    }

    public void visitEnumeratedListPattern(EnumeratedListPattern
                                        enumListPattern) {
        result.append("<.");
        enumListPattern.inner_pattern().accept(this);
        result.append(".>");
    }

    public void visitInnerPatternList(InnerPatternList
                                innerPatternList) {
        for(Pattern pattern :
                innerPatternList.pattern_list().getList()) {
            pattern.accept(this);
            result.append(", ");
        }
        if (innerPatternList.pattern_list().len() > 0) {
            result.delete(result.length() - 2, result.length());
        }
    }

    public void visitNoInnerPattern(NoInnerPattern noInnerPattern) {}

    public void visitValueLiteralPattern(ValueLiteralPattern
                                    pattern) {
        pattern.value_literal().accept(this);
```

```
    }

    public void visitNamePattern(NamePattern pattern) {
        pattern.id().accept(this);
    }

    public void visitWildcardPattern(WildcardPattern pattern) {
        result.append("_");
    }

    public void visitProductPattern(ProductPattern pattern) {
        result.append("(");
        for(Pattern p : pattern.p_inner_pattern_list().getList()) {
            p.accept(this);
            result.append(", ");
        }
        if (pattern.p_inner_pattern_list().len() > 0) {
            result.delete(result.length() - 2, result.length());
        }
        result.append(")");
    }

    public void visitRecordPattern(RecordPattern pattern) {
        pattern.value_or_variable_name().accept(this);
        result.append("(");
        for(Pattern p : pattern.inner_pattern_list().getList()) {
            p.accept(this);
            result.append(", ");
        }
        if (pattern.inner_pattern_list().len() > 0) {
            result.delete(result.length() - 2, result.length());
        }
        result.append(")");
    }

    public void visitIfExpr(IfExpr ifExpr) {
        result.append("if ");
        ifExpr.condition().accept(this);
        result.append(" then ");
        ifExpr.if_case().accept(this);
        for (Elsif elsif : ifExpr.elsif_list().getList()) {
            elsif.accept(this);
        }
        result.append(" else ");
        ifExpr.else_case().accept(this);
        result.append(" end");
    }

    public void visitElsif(Elsif elsif) {
```

```
        result.append(" elsif ");
        elsif.condition().accept(this);
        result.append(" then ");
        elsif.elsif_case().accept(this);
    }

    public void visitCaseBranch(CaseBranch caseBranch) {
        caseBranch.pattern().accept(this);
        result.append(" -> ");
        caseBranch.value_expr().accept(this);
    }

    /*Common*/
    public void visitMake_Id(Make_Id id) {
        id.id().accept(this);
    }

    public void visitNoOptionalId(NoOptionalId noId) {}

    public void visitId(Id id) {
        result.append(id.getText());
    }

    public void visitTrans(Trans trans) {
        result.append(trans.type_id());
        result.append(trans.var_id());
    }
}
```

Appendix G

Test Results

This appendix contains the results of the tests performed on the transformer.

G.1 Grey Box Test of the Program

An overview of the results of the grey box test of the program can be found in Table G.1.

The tested applicative specifications and their corresponding imperative specifications if transformable can be found on the enclosed CD-ROM. In order to be able to navigate in these specifications, the test case TC_♯ corresponds to the applicative specification A_TC_♯ and the imperative specification I_TC_♯ on the CD-ROM in the `test_files` directory.

Most of the test cases test more than the functionality mentioned in the test case description. The descriptions only summarize the intent of the test cases, that is the reason why they were constructed. An example is that all test cases test scheme definitions but this is only mentioned in one test case description, namely the one for which this was the actual intension.

Test Case	Description	Types of Interest	Result
Not transformable specifications			
Type declarations			
TC_NTTD1	Type of interest is a sort	t : T	OK
TC_NTTD2	Type of interest recursively defined variant definition, explicit	t : T	OK
TC_NTTD3	Type of interest recursively defined variant definition with destructor, explicit	t : T	OK

continued on next page

continued from previous page

Test Case	Description	Types of Interest	Result
TC_NTTD4	Type of interest recursively defined variant definition with reconstructor, explicit	t : T	OK
TC_NTTD5	Type of interest recursively defined variant definition, implicit	t : T	OK
TC_NTTD6	Type of interest recursively defined short record definition, explicit	t : T	OK
TC_NTTD7	Type of interest recursively defined short record definition, implicit	t : T	OK
TC_NTTD8	Type of interest recursively defined abbreviation definition, explicit	t : T	OK
TC_NTTD9	Type of interest recursively defined abbreviation definition, implicit, through abbreviation definition	t : T	OK
TC_NTTD10	Type of interest recursively defined abbreviation definition, implicit, through short record definition	t : T	OK
TC_NTTD11	Type of interest recursively defined abbreviation definition, implicit, through variant definition	t : T	OK
TC_NTTD12	Type of interest recursively defined abbreviation definition, implicit, third layer	t : T	OK
TC_NTTD13	Union definition	t : T	OK
TC_NTTD14	Type of interest in set collection	t : T	OK
TC_NTTD15	Type of interest in list collection	t : T	OK
TC_NTTD16	Type of interest in map collection, domain	t : T	OK
TC_NTTD17	Type of interest in map collection, range	t : T	OK

continued on next page

continued from previous page

Test Case	Description	Types of Interest	Result
TC_NTTD18	Function type expression in type definition	t : T	OK
TC_NTTD19	Subtype expression where type of interest is part of the single typings	t : T	OK
TC_NTTD20	Infinite map expression	t : T	OK
	Value declarations		
TC_NTVD1	Type of interest in set collection	t : T	OK
TC_NTVD2	Type of interest in list collection	t : T	OK
TC_NTVD3	Type of interest in map collection, domain	t : T	OK
TC_NTVD4	Type of interest in map collection, range	t : T	OK
TC_NTVD5	Higher order function	t : T	OK
TC_NTVD6	Type expression of single typing not a function type expression	t : T	OK
TC_NTVD7	Too few parameters	t : T	OK
TC_NTVD8	A product binding having fewer components than the product expression in a let binding	t : T	OK
TC_NTVD9	Implicit write before read of type of interest	t : T	OK
TC_NTVD10	Implicit write before read of type of interest in product	t : T	OK
TC_NTVD11	Explicit write before read of type of interest in product	t : T	OK
TC_NTVD12	Explicit write before read of type of interest in case expression	t : T	OK
TC_NTVD13	Explicit write before read of type of interest in if expression	t : T	OK

continued on next page

continued from previous page

Test Case	Description	Types of Interest	Result
TC_NTVD14	Type of interest out of scope	t : T	OK
TC_NTVD15	Quantified expression	t : T	OK
TC_NTVD16	Pattern in case expression constant of a type of interest	t : T	OK
TC_NTVD17	Pattern in case expression value name of the type of interest	t : T	OK
TC_NTVD18	Pre-condition violating transformability rules	t : T	OK
Transformable specifications			
TC_T1	Scheme definition, basic class expression, introduction of a variable, explicit function definition not mentioning the type of interest	t : T	OK
TC_T2	Introduction of more variables	t : T, s : S	OK
TC_T3	Observer with only T as parameter	t : T	OK
TC_T4	Observer with more parameters	t : T	OK
TC_T5	Proper T-generator, value name	t : T	OK
TC_T6	Generator with more return values, value literal, value name	t : T	OK
TC_T7	Combined observer and generator, value name	t : T	OK
TC_T8	Single typing of the form id : (T) $\times$ T $\rightarrow$ ...	t : T	OK
TC_T9	Abbreviation type of interest which is exploited in formal function application	t : T	OK
TC_T10	Value literal of the type of interest	t : T	OK

continued on next page

continued from previous page

Test Case	Description	Types of Interest	Result
TC_T11	Value name of the type of interest	t : T	OK
TC_T12	Basic expression	t : T	OK
TC_T13	Product expression of the type of interest	t : T	OK
TC_T14	Product expression containing type of interest	t : T	OK
TC_T15	Product expression containing proper T-generator	t : T	OK
TC_T16	Ranged set expression	t : T	OK
TC_T17	Enumerated set expression	t : T	OK
TC_T18	Comprehended set expression	t : T	OK
TC_T19	Comprehended set expression containing proper T-generator and implicit observer	t : T	OK
TC_T20	Ranged list expression	t : T	OK
TC_T21	Enumerated list expression	t : T	OK
TC_T22	Comprehended list expression	t : T	OK
TC_T23	Enumerated map expression	t : T	OK
TC_T24	Comprehended map expression	t : T	OK
TC_T25	Function application	t : T	OK
TC_T26	Function application with type of interest as parameter	t : T	OK
TC_T27	Function application with T-generator as parameter	t : T	OK
TC_T28	List application with type of interest as parameter	t : T	OK
TC_T29	List application with proper T-generator as parameter	t : T	OK
TC_T30	List application where the list is of the type of interest	t : T	OK

continued on next page

continued from previous page			
Test Case	**Description**	**Types of Interest**	**Result**
TC_T31	Map application with type of interest as parameter	t : T	OK
TC_T32	Map application with proper T-generator as parameter	t : T	OK
TC_T33	Map application where the map is of the type of interest	t : T	OK
TC_T34	Bracketed expression containing the type of interest	t : T	OK
TC_T35	Bracketed expression of the type of interest	t : T	OK
TC_T36	Infix expression of the type of interest	t : T	OK
TC_T37	Infix expression with one of the operands as type of interest	t : T	OK
TC_T38	Infix expression with implicit observer	t : T	OK
TC_T39	Prefix expression of the type of interest	t : T	OK
TC_T40	Prefix expression with the operand as type of interest	t : T	OK
TC_T41	Prefix expression with implicit observer	t : T	OK
TC_T42	Let expression where the value expression does not contain the type of interest	t : T	OK
TC_T43	Let expression where the value expression is of the type of interest	t : T	OK
TC_T44	Let expression where the value expression is a product in which the type of interest occurs	t : T	OK
TC_T45	Let expression of the type of interest	t : T	OK
TC_T46	If expression with condition as proper T-generator	t : T	OK
continued on next page			

continued from previous page

Test Case	**Description**	**Types of Interest**	**Result**
TC_T47	If expression of the type of interest	t : T	OK
TC_T48	If expression comparing the type of interest	t : T	OK
TC_T49	Case expression with the value expression cased over being a proper T-generator, value literal pattern , name pattern, wildcard pattern	t : T	OK
TC_T50	Case expression of the type of interest, product pattern	t : T	OK
TC_T51	Case expression, record pattern	t : T	OK
TC_T52	Case expression, enumerated list pattern, concatenated list pattern	t : T	OK
TC_T53	Assignment expression due to implicit generator	t : T	OK
TC_T54	Sequencing expression due to T-generator in connection with implicit observer	t : T	OK
TC_T55	Formal function application with product expression containing type of interest	t : T	OK
TC_T56	Formal function application with product expression and type of interest outside product expression	t : T	OK
TC_T57	Two types of interest	t : T, s : S	OK
TC_T58	Product expression containing type of interest inside product expression, expected type is carried on	t : T	OK
TC_T59	Access description carried on	t : T	OK
TC_T60	Explicit value definition	t : T	OK

continued on next page

continued from previous page			
Test Case	**Description**	**Types of Interest**	**Result**
TC_T61	Explicit value definition not of the type of interest	t : T	OK
TC_T62	Pre-condition involving the type of interest	t : T	OK

Table G.1: Grey box test of the program

G.2 Black Box Test of the Control Module

An overview of the results of the black box test of the control module can be found in Table G.2.

Test Case	**Description**	**Result**
Correct input		
1	Zero types of interest	OK
2	One type of interest	OK
3	Two types of interest	OK
Incorrect input		
4	Transformation on non existing file	OK
5	No name id of imperative scheme given	OK
6	Wrong format of variable list	OK

Table G.2: Black box test of the control module