Analyzing Action Semantics

Kasper Svendsen

Kongens Lyngby 2007
IMM-BSc-2007-13

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Summary

Programmers usually assume that when they compile a program, the behavior of the generated code corresponds to the semantics assigned to the program by the programming language. However, writing a correct compiler, that generates reasonably efficient code, is a difficult and expensive task. A lot of research has therefore focussed on automatically generating complete compilers from formal specifications of languages. In some cases, even with an accompanying proof of correctness of the generated compiler.

In this thesis we develop a tool for analyzing semantic descriptions of programming langauges, specified in a subset of the semantic description language, Action Semantics. The purpose of the tool is to function as a component of a compiler generator.

The tool implements two analyses: The first analysis is a type and termination analysis, which annotates semantic descriptions with type information about the values, that programs written in the described language can produce. The second analysis analyzes the use of bindings in the languages specification, to generate a bindings analysis for the source language.

Resumé

Programmører antager normalt, at når de compiler et program, så opfører den generede kode sig i overensstemmelse med den semantik, det pågældende program tildeles af programmeringssproget. Men da det er svært og dyrt at skrive en korrekt compiler der genererer hurtig kode, har en del forskning fokuseret på automatisk compiler-generering fra formelle specificationer af sprog. I nogle tilfælde, med et tilhørende bevis for korrektheden af den generede compiler.

I denne rapport udvikler vi et værktøj til at analysere action semantics beskrivelser af programmeringssprog, udtrykt i en begrænset udgave af det semantiske beskrivelsessprog: Action Semantics. Meningen er at værktøjet skal indgå som en komponent i en compiler-generator.

Værktøjet implementerer to analyser: Den første analyse er en kombineret type og terminerings-analyse. Den annoterer sprogspecifikationer med information om typen på værdierne, som programmer skrevet i det beskrevne sprog kan give. Den anden analyse analyserer brugen af bindinger i sprogspecifikationer, hvorfra den genererer en bindingsanalyse til det beskrevne sprog.

Preface

This thesis was prepared at Informatics Mathematical Modelling, the Technical University of Denmark in partial fulfillment of the requirements for acquiring a B.Sc. degree in engineering.

Lyngby, July 2007

Kasper Svendsen

Contents

Summary i

Resumé iii

Preface v

1 Introduction 1

1.1 Our work 2

1.2 Thesis organization 3

1.3 Typographical conventions 4

2 Background 5

2.1 Action Semantics 5

2.2 Action Notation 6

2.3 Action Semantic Descriptions 10

3 Analyzing Action Semantic Descriptions **15**

3.1 Type and termination analysis 15

3.2 Binding analysis 39

4 Discussion **59**

5 Implementation **63**

6 Conclusion **65**

A Semantics of While and Action Notation **67**

A.1 The While langauge 67

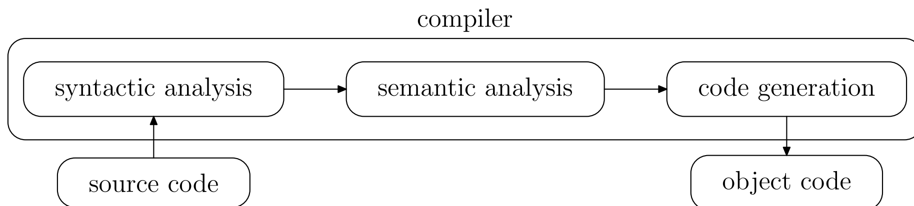
A.2 The Action Notation language 68

B Action Semantic Description for the While language **73**

Chapter 1

Introduction

The traditional way of writing a compiler is to structure the compiler into a series of phases, each taking as input the output from the previous phase. The figure below illustrates this with a simple compiler composed of three phases. The first phase, syntactic analysis, translates the source program into an intermediate representation. The second phase performs a semantical analysis on the intermediate representation. The third phase translates the intermediate representation into object code.



In most compilers, the syntactic analysis is generated automatically from a formal specification of the language's syntax, and the remaining phases are written manually. Compiler generators are programs that take a formal specification of the language's syntax and semantics and automatically generates all the phases of the compiler.

Manually writing a compiler is a difficult and expensive job; a lot of research has therefore focused on automatic compiler generation. Despite this, automatically generated compilers are typically inefficient compared to hand-written compilers and/or generate less efficient code than hand-written compilers. As a result, no automatically generated compiler has yet seen widespread use. However, even though automatically generated compilers are inefficient compared to hand-written compilers, it is certainly possible to imagine situations were they could be useful:

- *Language standardization and research*: Being able to quickly realize a language specification in the form of a slow prototype compiler would allow for a much more exploratory based approach to programming language research.
- *High-assurance software*: For critical high-assurance software, a provably correct compiler that generates slow object code is preferable to a hand-written unverified compiler which generates faster object code. For example, the C source-code of the flight control software for the Airbus A340 has been proven to be free of run-time errors using static analysis tools. However, as the verification was performed on the C source code, a buggy compiler could still introduce run-time errors into the flight control software object code.

1.1 Our work

Our work is best described by comparing it to some of the previous Action Semantics based compiler generators. Figure 1.1 illustrates the idea of the ACTRESS [3] and OASIS [14] compiler generators:[1] they are composed of two programs, one which takes the language specification, $\mathcal{L}_{spec}$, of the language $\mathcal{L}$ and generates a program $\mathcal{L}_{actioneer}$, that takes programs in the language $\mathcal{L}$, denoted by $\mathcal{P}_{\mathcal{L}}$, as input and gives the program's denotation, $\mathcal{P}_{action}$, as output. Combined with a generic action compiler, this is a full compiler generator.

Since the same generic action compiler component is used in every generated compiler, it has to be able to handle actions generated from any language specification, and preferably be able to generate efficient code for at least some of these languages. To achieve this, the action compilers used in ACTRESS and OASIS both perform a series of analyses and optimizations. However, since the action compiler is independent of the source language, it might have to perform

[1]The OASIS compiler generator differs slightly from this figure, but the underlying idea is the same.

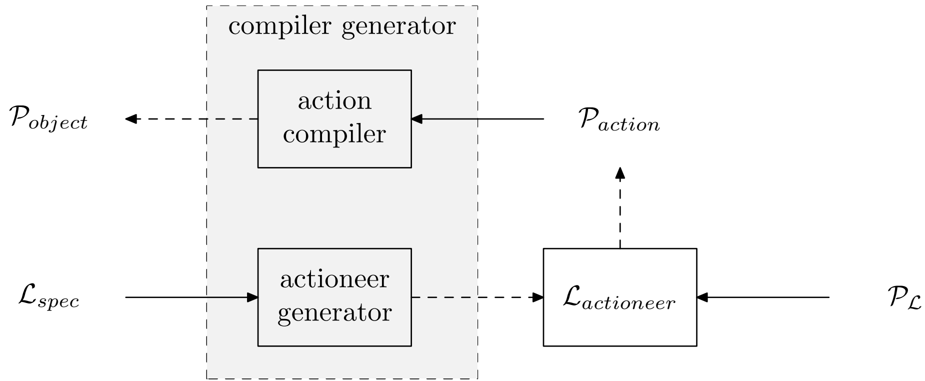


Figure 1.1: The structure of the ACTRESS and OASIS compiler generators.

a lot of extra work analyzing actions, to determine something that is apparent from the language specification. As a concrete example, imagine an imperative language with statically scoped variables and procedures. Instead of analyzing every single action to determine this property, it would be preferable to analyze the language specification just once, when the compiler is generated.

Figure 1.2 illustrates how a compiler generator based on this idea could look. The main component is the analysis generator, which takes as input a language specification and an analysis specification, performs the given analysis on the language and generates an analyzer and optimizer customized to the given language, based on the results of the analysis of the language specification.

In this thesis, we will focus on the analysis generator component of Figure 1.2.

1.2 Thesis organization

The following chapter contains a short introduction to the limited version of action notation that we have chosen to work with. In the third chapter, two analyses for analyzing action semantic descriptions are developed: a type and termination analysis and an analysis and accompanying algorithm for generating a reaching bindings analysis for the source language.

The fourth chapter discusses the limitations of our analyses and possible solu-

tions. The fifth chapter gives a short overview of the implementation of the analyses.

1.3 Typographical conventions

Throughout this document actions have been typeset in a bold italic font, e.g., ***provide*** 42, WHILE statements in typewriter font, e.g., `if` $s > 0$ `then` $s := 0$, and syntactic categories in a bold font, e.g., **Action**.

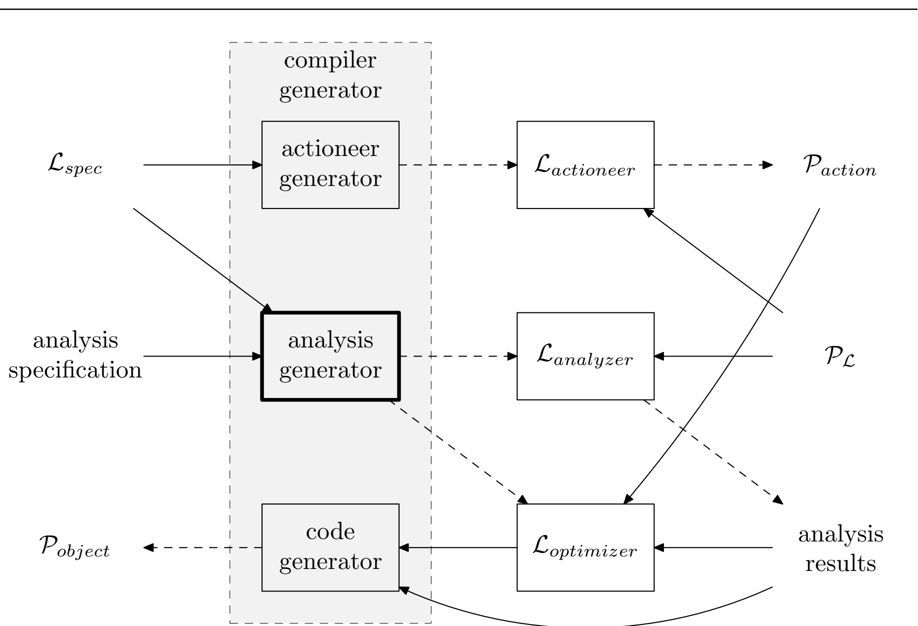


Figure 1.2: The structure of a compiler generator based on the idea of generating a customized analyzer and optimizer from the language specification.

Chapter 2

Background

This chapter introduces the parts of action semantics and action notation that is relevant for this thesis. The reader is assumed to be familiar with dataflow analysis [11] and basic lattice and type theory [5, 12].

2.1 Action Semantics

Action semantics [8] is a framework for formally specifying the semantics of programming languages. It was developed in the early 1990's as a more pragmatic alternative to the existing semantic formalisms such as operational, denotational, and axiomatic semantics. It was designed to be able to describe the semantics of realistic programming languages and to allow for greater re-usability between language specifications [6].

Many of the existing semantic formalisms were less than ideal for specifying the semantics of real-life languages: Both operational and denotational semantics descriptions were plagued with problems of expressiveness, modularity and re-usability [9, 12]. Language extensions and changes, for example, often required changes to be made throughout the entire language specification.

Action semantics combines many of the concepts of operational and denotational

semantics. An action semantics description (abbreviated ASD) can be seen as a form of denotational description, with actions as denotations, where the meaning of a program is the meaning of the denotation of the program (i.e., the action for the given program). The language used for specifying actions is called Action Notation (abbreviated AN). To date, two versions of Action Notation have been specified by the authors of Action Semantics, Action Notation 1 (AN-1) in 1992 and Action Notation 2 (AN-2) in 2000. AN-1 defined the semantics of actions using the formalisms of Unified Algebras and Structural Operational Semantics (SOS). AN-2 was developed as a smaller and simpler version of AN-1, defined in Modular Structural Operational Semantics (abbreviated MSOS) without the use of Unified Algebras.

Over the last 15 years quite a few compiler generators have been written using Action Semantics and AN-1 as the specification language [e.g.., 3, 14], and at least one compiler generator with Action Semantics and AN-2 as the specification language [13]. The work in this thesis will focus on Action Semantics and AN-2. Since the syntax and semantics of AN-2 has not yet been finalized, the work will be based on the current version of AN-2, version 0.7.5.

2.2 Action Notation

The basic concept of Action Notation is that of an action. Superficially, actions resemble expressions in an impure functional programming language: Action Notation defines a number of basic actions and action combinators for combining smaller actions into larger actions. Actions may be executed (*performed* in AN terminology). When performed, an action can terminate *normally*, *exceptionally*, *failingly*, or not terminate at all. On normal and exceptional termination, actions *produce* data. Actions that terminate normally are said to *give* data and actions that terminate exceptionally are said to *raise* data. Unlike expressions, actions also take data as input, when performed. Actions also differ from most existing languages in that bindings are first-class entities, which can be passed around and manipulated like any other piece of data.

The actions of AN-2 are divided into the following five facets:

- *Flow facet*: contains actions for controlling the control and data flow.
- *Declarative facet*: contains actions for manipulating bindings.
- *Reflective facet*: contains actions for working with actions as data.
- *Imperative facet*: contains actions for manipulating the store.

- *Interactive facet*: contains actions for inter-process communication.

For the purposes of this thesis only a subset of the actions defined by Action Notation will be used. This subset includes most of the flow, declarative and imperative facets. The subset was chosen to be expressive enough to allow a reasonable specification of the semantics of simple imperative languages, such as the WHILE language, which will be the main case-study throughout this thesis. The abstract syntax of the subset of AN-2 that will be the subject of thesis is shown in Figure 2.3, along with a very short description of what the different actions do. To avoid writing too many parentheses, we will follow the AN-2 precedence convention, where all infix actions are assigned the same precedence and associate to the left, and all prefix actions are assigned the same precedence, such that infix actions have a lower precedence than prefix actions.

As a short introduction to AN-2, we discuss a few more AN concepts, and briefly describe the operational behavior of the actions used in example shown in Figure 2.1, by showing how they are performed. For a more thorough introduction to AN-2, see [7, 10].

As mentioned previously, actions always take data when they are performed, and may produce data. *Data* is arbitrarily sized tuples of datums. *Datums* are primitive values, i.e., natural numbers, truth-values, bindings, cells, and tokens. It is possible to extend AN-2 with extra primitive values and associated operations, using Data Notation. As the basic AN data types are sufficient to describe the semantics of the WHILE language, we will assume an unextended version of AN.

Figure 2.1 shows a simple action, which counts up from one to five, eventually giving the value five, when performed. The list below contains an informal description of the operational behavior of each of the actions used in the example action.

- ***provide*** d: Always terminates normally, giving the data d. It ignores the data it was given. Note that the syntax for semantic values is overloaded; the datum d is equivalent to the 1-tuple (d).
- ***copy***: Always terminates normally, giving the data it was given.
- A_1 ***then*** A_2, A_1 ***and*** A_2, A_1 ***exceptionally*** A_2: The ***then***, ***and***, and ***exceptionally*** actions control the control and data flow. When performed, they all perform the action A_1 with the given data and bindings. Depending on how A_1 terminates, the actions either perform A_2 or propagate the value given by A_1. The table below describes when A_2 is performed and what data it is given, for each of the three actions:

Action A	A_2 is performed if	A_2 is performed with
then	A_1 terminated normally	the data given by A_1
and	A_1 terminated normally	the original data
exceptionally	A_1 terminated exceptionally	the original data

If A_2 terminates exceptionally or failingly, all three actions propagate the exception or failure. However, if A_2 terminates normally, ***then*** and ***exceptionally*** simply propagate the value given by A_2, whereas ***and*** gives the concatenation of the value given by A_1 and A_2 as its value.

In A_1 c A_2 where c is an action combinator, A_1 is referred to as the first *sub-action* and A_2 as the second sub-action.

- ***give*** *op*: Performs the operation *op* on the given data, terminating normally with the result of the operation if the operation is defined and terminating exceptionally with no data if it is not.
- ***check*** *pred*: Checks whether the predicate *pred* holds for the given data, terminating normally with a 0-tuple if it holds and terminating exceptionally with a 0-tuple if it does not.
- ***unfolding*** A, ***unfold***: The ***unfolding*** and ***unfold*** actions allow for self-reference. The action ***unfolding*** A is performed by executing the action A, such that whenever ***unfold*** is encountered in A, the action A is performed in place of ***unfold***, with the data and bindings given to ***unfold***. In case of nested ***unfolding***s, the innermost ***unfolding*** is used in place of ***unfold***.

```
    provide 1
then
    unfolding (
            provide 5 and copy then check = then provide 5
        exceptionally
            provide 1 and copy then give + then unfold
    )
```

Figure 2.1: An example of an action, which, when performed, counts up from one to five and terminates normally giving the value five.

To understand the behavior of the example action below, we start by looking at the ***unfolding*** action. The first sub-action of the ***exceptionally*** action in the

body of the ***unfolding*** action, checks whether the identity predicate holds between the natural number five and the data it is given. If the predicate holds then it gives the natural number five and if it does not hold it terminates exceptionally. On exceptional termination, the second sub-action of the ***exceptionally*** action is performed with the original data given to the ***unfolding*** action. The second sub-action performs an addition operation on the natural number 1 and the data it is given. If the data it is given is not a natural number, the sub-action will terminate exceptionally raising a 0-tuple and this will be propagated to the top-level. If the data it is given is a natural number, the body of the ***unfolding*** is performed again, with the result of the addition operation. As the ***unfolding*** action is first performed with the natural number 1, the action counts up from 1 to five and terminates normally giving the natural number five.

The actions of AN-2 are further divided into two levels, called Kernel AN-2 and Full AN-2, depending on how the semantics of the actions have been defined. Kernel AN-2 consists of all the actions that have been defined using Modular SOS, while all the actions of Full AN-2 have been defined by reduction to Kernel AN-2. Full AN-2 can thus be seen as a layer of syntactic sugar, which defines a number of macros for various useful Kernel AN-2 actions. Figure 2.2 defines the simplified macros of Kernel AN-2 used in this thesis.

- ***given*** A = ***give the data and*** A ***then check*** = ***exceptionally fail***
- ***give the*** s ***bound to*** D =
  give current bindings and provide D ***then give bound***
  then give the s

Figure 2.2: The Full AN-2 action abbreviations used in this thesis.

In the subset of action notation that will be considered in this thesis, only the ***unfolding*** and ***unfold*** actions belong to Full AN-2. ***unfolding*** A reduces to an action which performs the action A, after having bound the action A to a special token *unf*. ***unfold*** reduces to an action which causes the action bound to the special token *unf* to be performed, using the data and bindings given to ***unfold***. Both abbreviations thus makes use of the reflective facet of AN-2, which is not part of the subset considered in this thesis, and have therefore been moved to the Kernel AN-2 level instead and defined without the use of the reflective facet.

Appendix A.2 contains a natural semantics (NS) specification of the subset of AN-2 that we will be analyzing. Besides the above mentioned differences, the behavior of the A_1 ***and*** A_2 action has also been changed slightly: Instead of performing A_1 and A_2 in parallel, A_1 is always performed before A_2 in A_1 ***then*** A_2.

The reasons that we have rewritten the specification from Modular SOS to NS are:

- *Interpreter*: To familiarize myself with AN, I originally started out by writing an interpreter for AN. A NS specification is immediately translatable into an interpreter.
- *Proofs*: Originally, I wanted to prove that one or more of the type systems presented in this thesis was sound, using the proof assistant Coq [2, 4]. The natural semantics specification was easily formalizable in Coq as an inductively defined predicate, whereas the Modular SOS specification was not easily formalizable. Unfortunately, with more than 30 inference rules, just proving that the inference system was deterministic took me several hours, so I chose to spend time on developing more analyses instead of attempting to formalize and prove a type system sound.

The evaluation judgments, which have the following form,

$$(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A \rightarrow (\delta',\ \mu')$$

says that action A produces the data δ' and store μ' when given the data δ and performed with the bindings ξ, store μ and within the enclosing unfolding a. When A is not within an unfolding, a is set to "$-$". The labels *normal d*, *exceptional d* and *failed* are used to represented normal data, exceptional data, and failure, respectively.

2.3 Action Semantic Descriptions

The action semantic descriptions that we will be considering in this thesis, consists of three modules:

- *abstract syntax*: contains a context-free grammar defining the abstract syntax of the source language.
- *semantic entities*: contains a definition of the data types used by the source language in terms of action notation data types and defines which action notation data types are storable and bindable.
- *semantic functions*: contains a semantic function for each non-terminal in the abstract syntax module, mapping strings derived from the given

$$A, A_1, A_2 \in \textbf{Action}, \quad D \in \textbf{Data}, \quad O \in \textbf{DataOp}, \quad P \in \textbf{DataPred}$$

$$u, u_i \in \textbf{Datum}, \quad n \geq 0, \qquad S \in \textbf{Sort}$$

$Action$ ::=	***provide*** D	gives constant data
	copy	copies given data
	A_1 ***then*** A_2	composition
	A_1 ***and*** A_2	composition
	raise	raises an exception
	A_1 ***exceptionally*** A_2	exceptional composition
	give O	performs an operation on data
	check P	checks that data satisfies predicate
	fail	fails
	A_1 ***otherwise*** A_2	alternative
	give current bindings	gives current bindings as data
	A_1 ***hence*** A_2	scopes bindings
	create	allocates a cell
	update	stores data in cell
	inspect	reads data from cell
	unfold	performs current unfolded action
	unfolding A	allows for self-reference
$Data$ ::=	$u \mid (u_1, ..., u_n)$	datum singleton and tuple
$Datum$ ::=	$n \mid b \mid t$	natural number, boolean and token
$DataOp$ ::=	$+ \mid - \mid *$	arithmetic operations
	binding	singleton binding
	overriding	merge bindings
	bound	reference bindings
	the S	projection
$DataPred$ ::=	$= \mid >$	identity and greater than predicate
$Sort$ ::=	***cell*** | ***data*** | ***nat***	
	bindings | ***bool***	

Figure 2.3: Action syntax.

non-terminal to actions. Each semantic function is defined as a set of semantic productions, one for each rule of the given non-terminal, mapping the syntax elements of the right-hand-side of the given rule to actions. The right-hand-side of semantic productions will be referred to as its denotation. A semantic production has the form:

$$m[\![p]\!] = A$$

where m is the identifier of the semantic function being defined, p is the arguments of the production, and A an action. The syntax of the action A is the syntax given in Figure 2.3, extended with semantic function calls.

In our implementation, the arguments, p, is a list of semantic variables and strings. It does not support the use regular expressions to define the parameters of semantic productions. Semantic function calls are further limited to one argument. However, both of these restrictions are purely restrictions in the implementation and neither of the two analyses depend on these restrictions.

The following is an excerpt of the action semantic description of the WHILE language, from Appendix B. The two semantic productions are part of the definition of the semantic function, *evala*, for arithmetic expressions.

- $evala[\![N]\!] =$ ***provide*** N
- $evala[\![A_1 + A_2]\!] = evala[\![A_1]\!]$ ***and*** $evala[\![A_2]\!]$ ***then give*** $+$

The semantic functions module also defines a mapping from identifiers to abstract syntax non-terminals (either built-in non-terminals such as ident and numeral or non-terminals defined in the abstract syntax module), this mapping will be referred to as the variable environment and represented with the symbol Γ_{ident}. For the above example, N, A_1 and A_2 are variables ranging over numerals and arithmetical expressions, respectively. All the examples of semantic productions given in this report should be interpreted using the variable environment of the WHILE language, as defined in Appendix B, unless otherwise stated.

A program's denotation is the action that the appropriate semantic function gives when applied to the given program. The denotation of the program $1 + 2$, with the above language definition, is thus ***provide*** 1 ***and provide*** 2 ***then give*** $+$.

The details of the syntax of action semantic specifications are not very interesting, so we will ignore them and instead describe a few auxiliary functions to extract the necessary information from an action semantic specification.

- The *funcs* function takes an action semantic description and gives the set of identifiers of the semantic functions defined in the specification.

$$funcs : \mathbf{ASD} \to \mathcal{P}(\mathbf{FuncID})$$

- The *prods* function takes an action semantic description and a semantic function identifier and gives a set containing a 2-tuple for every semantic production defining that semantic function, where the first component is the arguments to the semantic production and the second component is the semantic production's denotation.

$$prods : \mathbf{ASD} \to \mathbf{FuncID} \to \mathcal{P}(\mathbf{FuncParams} \times \mathbf{Action})$$

- The *calls* function takes an action and gives a set of 2-tuples, one for each semantic function call, where the first component is the semantic function identifier and the second the arguments to the call.

$$calls : \mathbf{Action} \to \mathcal{P}(\mathbf{FuncID} \times \mathbf{FuncParams})$$

So, using the action semantics description of WHILE as an example, we have:

$$funcs(\mathcal{L}_w) = \{evala, evalb, exec\}$$

$$prods(\mathcal{L}_w, evala) = \{(I, \textbf{\textit{give the cell bound to}}\ I\ \textbf{\textit{then inspect}}), (N, \textbf{\textit{provide}}\ N), ...\}$$

$$calls(evala[\![AE_1]\!]\ \textbf{\textit{and}}\ evala[\![AE_2]\!]\ \textbf{\textit{then give}} +) = \{(evala, AE_1), (evala, AE_2)\}$$

where $\mathcal{L}_w$ denotes the action semantics description of the WHILE language.

Several of the concepts, analyses, and problems discussed in this report are accompanied with small example languages in the form of a few semantic productions. Most of the example languages are extensions of the WHILE language and the semantic productions should be seen as an addition to the specification given in Appendix B. The languages that are not extensions of WHILE only define the details necessary to illustrate the point of the example. Some of the semantic productions given in the examples do not contain actions to handle program failure correctly (i.e., on program failure they might terminate exceptionally, but the exceptional value could be caught, instead of crashing the program), to avoid too much unimportant complexity in the examples.

Chapter 3

Analyzing Action Semantic Descriptions

This chapter describes our work on analyzing action semantic descriptions. In Section 3.1 we develop a type and termination analysis by combining ideas from dataflow analysis [11] with the Cartesian-Product type inference algorithm [1]. In Section 3.2 we develop a binding analysis and an algorithm that generates a reaching bindings analysis based on the results of the binding analysis. Interestingly, while the generated reaching bindings analysis works on the source program, the analysis results are about the source program's denotation.

3.1 Type and termination analysis

The first analysis is a combined type inference and termination analysis for action semantic descriptions. The analysis annotates the denotation of all semantic productions, and their sub-actions, with an over-approximation of the set of types of all the values that all possible instantiations of the given action might give, when performed. The analysis further annotates all denotations and their sub-actions with information about whether they can terminate exceptionally and/or failingly or neither.

While all actions in AN are well-defined, they can still terminate failingly or exceptionally, on, say, type errors. Since actions that terminate exceptionally raise data that can be trapped and processed at a later stage, the type analysis thus has to keep track of raised data anyway. A type analysis can thus trivially be extended to a termination analysis as well, by keeping track of which actions that can terminate failingly.

In the context of this project, results from this analysis will primarily be used as input to the latter analyses. Listed below are a few other potential uses for this analysis, that are outside the scope of this project.

- *CFGs*: The results of the termination analysis are very useful for constructing more precise control-flow graphs of a given program's denotation, which is important for the accuracy of data-flow analyses.
- *Debugging*: This analysis, along with the latter analysis, can be helpful in debugging action semantic descriptions. If the analysis results are wrong, it might indicate an error in the action semantic description. Three errors were revealed in the action semantic description for the WHILE language, by the analyses developed.

This section is divided into three sub-sections. In Section 3.1.1 we describe our method for analyzing entire action semantics descriptions, without looking at how individual semantic productions are analyzed, and formalize it in the framework of data-flow analysis. In Section 3.1.2 and 3.1.3 we introduce two type-inference algorithms for individual semantic productions. Both type inference algorithms are sound, however, only the second is guaranteed to terminate.

3.1.1 Iterative type analysis

Assuming that we are only interested in languages for which there exists an infinite number of valid programs, which is certainly a reasonable assumption, it is obviously impossible to simply generate all possible instantiations of all semantic productions and run a type inference algorithm on them. However, we can approximate the types of all possible instantiations by simply considering all possible "type instantiations". That is, we run the type inference algorithm once for each possible combination of types that the semantic functions called in the given semantic production might give when performed. Obviously, this is only an improvement compared to analyzing all possible instantiations, if the number of "type instantiations" is not infinite as well. For now, we will assume that the number of "type instantiations" is finite, but we will return to this problem.

An action A is said to be *context neutral*, if, when performed with the same bindings and store, if it terminates, it always gives the same data and store, irrespective of the data it is performed with, and if it does not terminate, that it always "behaves" the same, irrespective of the data it is performed with. A semantic production is said to be context neutral if all possible instantiations of its denotation are context neutral. As an example, ***provide*** 42 is context neutral because it always terminates normally giving the 1-tuple (42), while ***provide*** 42 ***and copy*** is not context neutral as it always terminates normally giving the 2-tuple $(42, d)$, where d is the data the action was performed with.

For this analysis we restrict ourselves to ASDs in which all semantic productions are context neutral, as this allows us to analyze semantic function calls in the denotation of semantic productions independently of the context the semantic function call appears. This allows us to treat the semantic function calls to m as equivalent in ***provide*** 42 ***then*** $m[\![a]\!]$ and ***provide*** 42 ***and provide*** 17 ***then*** $m[\![a]\!]$, independently of the fact that it is called with the 1-tuple (42) in the first action and the 2-tuple $(42, 17)$ in the second action. This restriction greatly reduces the complexity of the type and termination analysis, and besides a few problems with semantic production reuse, we have not encountered any problems with expressiveness because of this restriction. Chapter 4 contains a discussion of the reuse problem and an idea for a type analysis without this restriction.

We further introduce the following restrictions on the use of ***unfold***s and ***unfolding***s.

- All ***unfold*** actions must be enclosed in an ***unfolding*** A action.
- All ***unfold*** actions must be tail-recursive.
- All ***unfolding*** A actions must be context neutral.

The first restriction is introduced to ensure that it is always possible to determine, from the language specification alone, what ***unfolding*** A action a given ***unfold*** action refers to, which is not always possible without this restriction, as illustrated below. The two remaining restrictions are not strictly necessary, however, with these restrictions it is simpler to infer the types of ***unfolding*** and ***unfold*** actions. The consequences of these restrictions on ***unfolding*** and ***unfold*** actions and how to modify the analysis to get rid of these restrictions is also discussed below.

Consider the following non-higher-order functional language that only supports anonymous functions (i.e.., no named functions as our subset of AN does not allow actions to be bound to tokens), such as the following lambda-calculus inspired toy language:

- $fun[\![AE]\!] = evala[\![AE]\!]$
- $fun[\![BE]\!] = evalb[\![BE]\!]$
- $fun[\![(Y\ E)]\!] =$
  give current bindings and ((***give the data bound to*** "$_rec$")
  and $fun[\![E]\!]$ ***then give binding***)
  then give overriding hence unfold
- $fun[\![$`if` BE `then` E_1 `else` $E_2]\!] =$
  $evalb[\![BE]\!]$ ***then*** (
  (***given*** $true$ ***then*** $fun[\![E_1]\!]$) ***otherwise***
  (***given*** $false$ ***then*** $fun[\![E_2]\!]$))
- $fun[\![(\lambda\ I\ .\ E_1)\ E_2]\!] =$
  give current bindings and ((***provide*** I ***and*** $fun[\![E_2]\!]$
  then give binding) ***and*** (***provide*** "$_rec$" ***and***
  provide I ***then give binding***)
  then give overriding) ***then give overriding***
  hence unfolding ($fun[\![E_1]\!]$)
- $fun[\![$`let` $I\ =\ E_1$ `in` $E_2]\!] =$
  give current bindings and (***provide*** I ***and*** $fun[\![E_1]\!]$
  then give binding) ***then give overriding***
  hence $fun[\![E_2]\!]$

AE and BE denote arithmetic and boolean expressions, and E, E_1, and E_2 expression terms. The syntax for arithmetic and boolean expressions is borrowed from the WHILE language, along with the *evala* and *evalb* semantic functions.

The interesting feature of this language is that it supports anonymous recursion, via the $(Y\ E)$ expression. A lambda abstraction, $(\lambda\ I\ .\ E_1)\ E_2$, is evaluated by evaluating E_1 with the value of E_2 bound to I. $(Y\ E)$ causes the body of the lambda abstraction currently being evaluated to be evaluated with the value of E bound to its argument identifier, I. The following little example program uses anonymous recursion to calculate the factorial of 10:

$$(\lambda\ x\ .\ \texttt{if}\ x = 1\ \texttt{then}\ 1\ \texttt{else}\ \texttt{let}\ y = (Y\ (x-1))\ \texttt{in}\ y * x)\ 10$$

Since the ***unfold*** action in the denotation of the semantic production for $(Y\ E)$ is not within an ***unfolding*** action, we cannot determine which ***unfolding*** action

the ***unfold*** action refers to, from looking at the language specification alone. Instead, we could either perform the analysis on concrete programs instead of languages, or we could attempt to somehow "intelligently" determine that the ***unfold*** always refers to the action ***unfolding*** $(fun[\![E_1]\!])$, where E_1 is the body of the inner-most lambda abstraction enclosing the $(Y\ E)$ expression.

The restriction that ***unfold*** actions must be tail-recursive and ***unfolding*** actions context-neutral simplifies the type inference of ***unfolding*** actions: Since the behavior of the ***unfolding*** is independent of the type of the data it is given, we can determine the types of context neutral ***unfolding*** actions in a single pass, by inferring the types of the body of the unfolding once. Whereas for non-context neutral ***unfolding*** actions we might have to iterate until a fix-point is reached, as the types of the data produced by the body of the unfolding might depend on the type of the data it is given, as in the following action:

> ***provide*** 1 ***then unfolding*** (
> (***give the nat then provide*** *true* ***then unfold***) ***exceptionally***
> (***give the bool then create***))

Since ***unfold*** actions must be tail-recursive, we further know that the types of the body of the ***unfolding*** is the union of the types of the non-recursive branches of the ***unfolding*** (that is, ***unfold*** actions cannot give data of types not given by one or more of the non-recursive branches of the body of the enclosing ***unfolding***).

Since semantic productions can further be recursive and mutually recursive, the set of types of each semantic production is calculated iteratively, starting with those that do not call any semantic functions, until a fix-point is reached. To represent the dependencies between semantic productions, we introduce the concept of a language construct graph (LCG) in Definition 3.1. Each semantic function and semantic production in a given action semantic description is represented as a node in its LCG. For each semantic function used in a given semantic production, there is an edge from the function node to the production node. For each production node there is an edge to its function node. A slightly simplified LCG of the action semantic description of the WHILE language is shown in Figure 3.1.

Definition 3.1 *A language construct graph (LCG) for an action semantic de-*

scription $\mathcal{L}$ is a directed graph $G = (V, E)$, where

$$V = \mathit{funcs}(\mathcal{L}) \cup \{(id, p, a) \mid id \in \mathit{funcs}(\mathcal{L}) \wedge (p, a) \in \mathit{prods}(\mathcal{L})(id)\}$$

$$\begin{aligned} E = \{((id, p, a), id) \mid id \in V \wedge (id, p, a) \in V\} \\ \cup \{(id, (id', p, a)) \mid id \in V \wedge (id', p, a) \in V \wedge \exists (m, a') \in \mathit{calls}(a) : id = m\} \end{aligned}$$

The nodes that are members of $\mathit{funcs}(\mathcal{L})$ are called function nodes *and the remaining nodes are called* production nodes*.*

We can now define the type analysis as in Figure 3.2, in terms of a language construct graph. The analysis uses the lattice $(S, \leq)$, where S is the set,

$$\{E \in \mathcal{P}(\mathbf{FuncID}_* \times \mathcal{P}(\mathbf{Type})) \mid \forall (m, T) \in E : \neg(\exists (m', T') \in E : m = m' \wedge T \neq T')\}$$

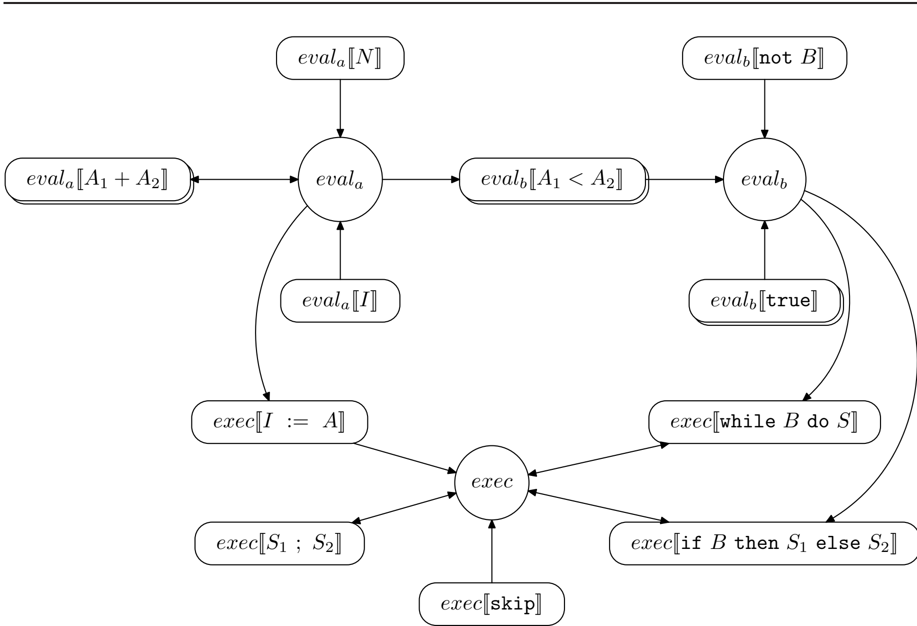


Figure 3.1: Simplified Language Construct Graph for the action semantic description of the WHILE language. Circles represent rule nodes; rounded boxes represent production nodes.

and $\leq$ is the partial order,

$$L_1 \leq L_2 \quad \text{iff} \quad \forall (r,T) \in L_1 : ((\exists T' : (r,T') \in L_2) \wedge (\forall (r,T') \in L_2 : T \subseteq T'))$$

FuncID$_*$ is the set of semantic function identifiers used in the language specification that we are analyzing, and **Type** is the set of type terms used by the type inference algorithms introduced in the following two sub-sections. This set is defined in Figures 3.5 and 3.6 for the first and second type inference algorithm, respectively. The reason we use the set S instead of $\mathcal{P}(\textbf{FuncID}_* \times \mathcal{P}(\textbf{Type}))$, is that $\leq$ is not a partial order for the set $\mathcal{P}(\textbf{FuncID}_* \times \mathcal{P}(\textbf{Type}))$.

Neither of the resulting lattices satisfy the Ascending Chain condition, as they both contain infinite ascending chains, such as the ones given below:

$$\{(a, \{exn((int))\})\} \leq \{(a, \{exn((int)), exn(exn((int)))\})\} \leq \cdots$$

$$\{(a, \{exn([int])\})\} \leq \{(a, \{exn([int]), exn(exn([int]))\})\} \leq \cdots$$

where a is a semantic function identifier from the language specification being analyzed.

However, for the second type and termination analysis, the set of elements from **Type** that the type inference algorithm is actually able to infer is finite, as argued in Section 3.1.3, and since the **FuncID**$_*$ set is obviously finite, the set of lattice elements that is actually used, is also finite. The "effective" lattice used by the second type and termination analysis thus satisfies the Ascending Chain condition.

Let $A = \{S_i\}_{i \in I}$ be a family of elements of S and x be the lattice element,

$$\{(m, \bigcup_{i \in I} \{typs(m, S_i)\}) \mid m \in funcs(A)\}$$

where

$$funcs(S) = \{m \mid U \in S \wedge \exists T : (m,T) \in U\}$$

$$typs(m,U) = \begin{cases} T & \text{if } (m,T) \in U \\ \emptyset & \text{otherwise} \end{cases}$$

Clearly, x is an upper bound for A, by the definition of $\leq$ and x. Let y be an upper bound for A. By the definition of $\leq$, for every tuple (m,T) in one of the S_is, y must contain a single tuple (m,T') and T' must satisfy $T \subseteq T'$. If (m,T) is a member of x, for some T, then at least one of the S_is must contain a tuple (m,T') for some T' and thus y must contain a tuple (m,T'') for some

T''. For all the tuples (m, T) in x, if $t \in T$ then there exists a tuple (m, T') in one of the S_is such that $t \in T'$, y must thus also contain a tuple (m, T'') such that $t \in T''$. By the definition of $\leq$, we thus have that $x \leq y$ and x is thus a least upper bound for $\{S_i\}_{i \in I}$. Since the S_is were arbitrary, every subset of S has a least upper bound and the lattice $(S, \leq)$ is complete, with the least upper bound operator:

$$\bigsqcup A = \{(m, \bigcup_{S \in A} \{typs(m, S)\}) \mid m \in funcs(A)\}$$

For function nodes the TTA_{entry} and TTA_{exit} functions, defined in Figure 3.2, gives the set of types of the values that all possible instantiations of the semantic productions belonging to the given semantic function can produce. For production nodes the TTA_{entry} function gives a set of 2-tuples, one for each semantic function called in the semantic production associated with the given node, where the first component is the function identifier and the second component the current type analysis information about the given semantic function, i.e., the set of types of the values that all possible instantiations of the semantic productions belonging to the given semantic function can produce. For production nodes the TTA_{exit} function gives a single 1-tuple, where the first component is the function identifier of the semantic function that the semantic production associated with the given node belongs to and the second component the current analysis information about the given semantic production, i.e., the set of types of the values that all possible instantiations of the given semantic production can produce.

The *type* function, defined in Figure 3.3, is used to calculate the set of types of the values that all possible instantiations a given semantic production can produce, given the current type analysis information about the semantic functions called in the denotation of the semantic production. From the current analysis information, it generates all possible type environments and invokes the type inference algorithm with each of these type environments, giving the union of the sets of types produced by the type inference algorithm. If the current analysis information, i.e., the set of types, for a semantic function/production is empty, we currently know of no possible instantiations. Thus, if the current set of types for any of the semantic functions called by the denotation of a semantic production is empty, we cannot construct a type environment with which to invoke the type inference algorithm, hence the two cases in the definition of the TTA_{exit} function for production nodes. The analysis results for specifications in which the denotation of all semantic productions call a semantic function will thus be an empty set of types for every semantic production, which is correct in the sense that there exists no possible instantiations of finite length of any of its semantic productions.

From the definition of $\leq$ we see that if $L_1 \leq L_2$ then all the type environments that can be generated from L_1 can also be generated from L_2. We thus have that if $L_1 \leq L_2$ then $type(L_1, a) \subseteq type(L_2, a)$ for all actions a, provided L_1 covers a, as defined by the binary *covers* predicate in Table 3.2.

Let L_1 and L_2 denote two elements of S such that $L_1 \leq L_2$ and let p denote a production node for a semantic production with the denotation a. Let L'_i denote the value of $TTA_{exit}(p)$ calculated using L_i as the value of $TTA_{entry}(p)$. If L_1 covers a then clearly L_2 also covers a, since $L_1 \leq L_2$. If L_1 does not cover a then $L'_1 = (id, \emptyset)$ and clearly $L'_1 \leq L'_2$ by the definition of $\leq$. TTA_{exit} is thus a monotone function.

Since the lattice used is complete, the "effective" lattice used for the second type inference algorithm satisfies the Ascending Chain condition and the transfer function is monotone, we can use a work-list algorithm to iteratively calculate a least fixed point of the TTA_{entry}/TTA_{exit} equations, and the iteration is

---

$$\begin{aligned} init &: \textbf{ASD} \rightarrow \mathcal{P}(\textbf{FuncID} \times \textbf{FuncParams} \times \textbf{Action}) \\ TTA_{entry}, TTA_{exit} &: (\textbf{FuncID} \cup (\textbf{FuncID} \times \textbf{FuncParams} \times \textbf{Action})) \\ &\quad \rightarrow \mathcal{P}(\textbf{FuncID} \times \mathcal{P}(\textbf{Type})) \end{aligned}$$

$$\begin{aligned} init(\mathcal{L}) = \{(m, p, a) \mid\; & m \in funcs(\mathcal{L}) \\ & \wedge (p, a) \in prods(\mathcal{L})(m) \wedge calls(a) = \emptyset\} \end{aligned}$$

$$\begin{aligned} TTA_{entry}(id) &= \bigsqcup \{TTA_{exit}((id, p, a)) \mid (id, p, a) \in V\} \\ TTA_{entry}((id', p, a)) &= \bigsqcup \{TTA_{exit}(id) \mid id \in V \wedge (id, (id', p, a)) \in E\} \end{aligned}$$

$$\begin{aligned} TTA_{exit}(id) &= TTA_{entry}(id) \\ TTA_{exit}((id, p, a)) &= \begin{cases} (id, type(T, a)) & \text{if } T \textit{ covers } a \\ (id, \emptyset) & \text{otherwise} \end{cases} \end{aligned}$$

$$\text{where } T = TTA_{entry}((id, p, a)).$$

$$T \textit{ covers } a \quad \text{iff} \quad \forall (m, a') \in calls(a) : \exists T' : T' \neq \emptyset \wedge (m, T') \in T$$

Figure 3.2: Type and termination analysis.

$$
\begin{aligned}
type &: \mathcal{P}(\mathbf{FuncID} \times \mathcal{P}(\mathbf{Type})) \times \mathbf{Action} \rightarrow \mathcal{P}(\mathbf{Type}) \\
type' &: \mathcal{P}(\mathbf{FuncID} \times \mathbf{Type}) \rightarrow \mathcal{P}(\mathbf{FuncID} \times \mathbf{FuncParams}) \\
&\quad \rightarrow \mathcal{P}(\mathbf{FuncID} \times \mathcal{P}(\mathbf{Type})) \rightarrow \mathbf{Action} \rightarrow \mathcal{P}(\mathbf{Type})
\end{aligned}
$$

$$
type(entry, a) = type'([])(calls(a))(entry)(a)
$$

$$
type'(\Gamma)(\emptyset)(entry)(a) = \begin{cases} T & \text{if } \Gamma,\ -\ \vdash_T\ a\ :\ T \\ undef & \text{otherwise} \end{cases}
$$

$$
\begin{aligned}
type'(\Gamma)(calls)(entry)(a) = \bigcup \{ & type'(\Gamma[m \mapsto t])(calls \setminus (m, a'))(entry)(a) \\
& \mid (m, a') \in calls \wedge (m, T) \in entry \wedge t \in T \}
\end{aligned}
$$

Figure 3.3: A function for inferring the types of a semantic production.

guaranteed to terminate (for the second type and termination analysis).

Figure 3.4 contains a simplified version of the worklist algorithm used to actually calculate the least fixed point of the TTA_{entry}/TTA_{exit} equations for a given specification $\mathcal{L}$ with the LCG (V, E). Here TTA_{entry} is a table mapping vertices to the current type analysis information about the vertex and TTA_{exit} the function defined in Figure 3.2. The worklist, W, which is represented as a set of edges, is used to keep track of what remains to be computed. Each edge, (l, l'), in the worklist indicates that the entry information for node l has changed requiring the entry information for l' to be recomputed. Initially, every edge is added to the worklist, then the algorithm starts selecting edges, (l, l'), from the worklist at random, recalculating the exit information of l using the current entry information. If the newly calculated exit information for l is greater than the current entry information for l', the entry information of l' is updated and all of l''s outgoing edges are added to the worklist.

The worklist algorithm presented above obviously is not very efficient, because of the random selection of edges from the worklist. The worklist algorithm we have implemented is slightly more intelligent: it divides the LCG into strongly connected components (SCCs), constructs a dependency graph for these SCCs, sorts the vertices of the dependency graph in topological order and processes each SCC in this order. Each SCC is processed using an algorithm very similar to the one given above, the only difference being that the only edges added to

```
1  for each (l, l') ∈ E
2          do W ← W ∪ {(l, l')}
3  while W ≠ ∅
4          do select an edge (l, l') from W
5              W ← W \ {(l, l')}
6              if TTA_exit(l) ⋠ TTA_entry(l')
7                  then TTA_entry(l') ← TTA_entry(l') ⊔ TTA_exit(l)
8                      for all (l', l'') ∈ E
9                          do W ← W ∪ {(l', l'')}
```

Figure 3.4: The worklist algorithm.

the worklist are edges between vertices both members of the given SCC.

3.1.2 Type system 1

In this sub-section, the first type inference algorithm is introduced. This algorithm has the unfortunate property, that when used with the type and termination analysis, the type and termination analysis does not always terminate. In Section 3.1.3 we introduce a less precise version of this algorithm, which always terminates.

The type inference algorithm is shown as an inference system in Tables 3.2 and 3.3. The type terms used in the algorithm are given as a grammar in Figure 3.5. τ_d generates types for datums and τ generates types for data (i.e.., datum tuples). The datum types are self-explanatory. The type of the n-tuple $(d_1, ..., d_n)$ is $(t_1, ..., t_n)$, where t_i is the type of datum d_i.

To enforce the context neutrality restriction on the denotation of semantic productions and any unfoldings it might contain, with the type inference algorithm, we introduce the "$-$" type for data that must not be able to affect the performance of the given action. The type of the data given to A is set to "-" when inferring the types of the body of ***unfolding*** A actions and when A is the denotation of a semantic production.

A typing judgment of the form:

$$\Gamma,\ -\ \vdash_T\ A\ :\ T$$

says that action A gives or produces types T in the type environment Γ. A type environment is a finite mapping from semantic function identifiers (**FuncID**) to a type (**Type**).

Most of the inference rules are straightforward, however, the T1-UNF1 rule deserves a few comments: Since we require that ***unfolding*** actions must be context-neutral and ***unfold*** actions tail-recursive, ***unfold*** actions can never give values of types other than those given by the enclosing ***unfolding*** action's non-recursive branches. We can thus safely assign ***unfold*** actions the set of types $\emptyset$, as they do not contribute anything themselves to the types of the enclosing ***unfolding*** action. After the analysis has been completed, all ***unfold***s are annotated with the types of the enclosing ***unfolding***.

Consider the WHILE language extended with the semantic production below, which defines a new language construct, (A_1, A_2), that gives the concatenation of the data given by A_1 and A_2, where A_1 and A_2 are arithmetic expressions. The type produced by the semantic production for the arithmetic expression N, where N is a numeral, is $\{(int)\}$, independent of the type environment used. At some point the type and termination analysis will thus attempt to infer the type of (A_1, A_2), with a type environment of $[evala[\![A_1]\!] \mapsto (int), evala[\![A_1]\!] \mapsto (int)]$ and infer the set of types $\{(int, int)\}$. At some point it will thus attempt to infer the type of (A_1, A_2), with a type environment of $[evala[\![A_1]\!] \mapsto (int, int), evala[\![A_2]\!] \mapsto (int)]$, inferring the set of types $\{(int, int, int)\}$, and so on, never terminating.

- $evala[\![(A_1, A_2)]\!] = evala[\![A_1]\!]$ ***and*** $evala[\![A_2]\!]$

$$
\begin{array}{lll}
\tau ::= & (\tau_d, ..., \tau_d) \mid - & \text{tuple and ...} \\
 & exn(\tau) \mid fail & \text{exceptional values and failure} \\
 & & \\
\tau_d ::= & nat \mid bool & \text{types for natural numbers and truth-values} \\
 & bindings \mid token \mid cell & \text{types for bindings, tokens and cells}
\end{array}
$$

$$
normal, failed, fail, exp : \mathcal{P}(\mathbf{Type}) \rightarrow \mathcal{P}(\mathbf{Type})
$$

$$
\begin{aligned}
fail(T) &= \{fail\} \cap T \\
exp(T) &= \{exn(\tau) \mid exn(\tau) \in T\} \\
failed(T) &= fail(T) \cup exp(T) \\
normal(T) &= T \setminus failed(T)
\end{aligned}
$$

Figure 3.5: Grammar of types for the first type and termination analysis.

$n \in \mathbf{Nat}, \quad b \in \mathbf{Bool}, \quad t \in \mathbf{Token}, \quad d_i \in \mathbf{Datum}$

(TD1-Nat)	$\vdash\ n\ :\ nat$
(TD1-Bool)	$\vdash\ b\ :\ bool$
(TD1-Token)	$\vdash\ t\ :\ token$
(TD1-Num)	$\dfrac{(id, \{numeral\}) \in \Gamma_{ident}}{\vdash\ id\ :\ nat}$
(TD1-Ident)	$\dfrac{(id, \{ident\}) \in \Gamma_{ident}}{\vdash\ id\ :\ token}$
(TD1-Tuple)	$\dfrac{\forall i\ \vdash\ \Gamma\ :\ d_i t_i}{\vdash\ (d_1, ..., d_n)\ :\ (t_1, ..., t_n)}$

Table 3.2: Type rules for action data.

$$d \in \mathbf{Data}, \quad A \in \mathbf{Action}$$

(T1-Provide)
$$\frac{\vdash\ \Gamma\ :\ dt}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{provide}}\ d\ :\ \{t\}}$$

(T1-Copy)
$$\frac{T \neq -}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{copy}}\ :\ T}$$

(T1-Then)
$$\frac{\begin{array}{c}\Gamma,\ T\ \vdash_T\ A_1\ :\ T',\\ \forall t \in normal(T') :\ \Gamma,\ t\ \vdash_T\ A_2\ :\ T_t,\\ T'' = \cup_{t \in normal(T')} T_t \cup failed(T')\end{array}}{\Gamma,\ T\ \vdash_T\ A_1\ \textbf{\textit{then}}\ A_2\ :\ T''}$$

(T1-And)
$$\frac{\begin{array}{c}\Gamma,\ T\ \vdash_T\ A_1\ :\ T',\quad \Gamma,\ T\ \vdash_T\ A_2\ :\ T'',\\ T''' = normal(T') \times normal(T'') \cup failed(T' \cup T'')\end{array}}{\Gamma,\ T\ \vdash_T\ A_1\ \textbf{\textit{and}}\ A_2\ :\ T'''}$$

(T1-Raise)
$$\Gamma,\ T\ \vdash_T\ \textbf{\textit{raise}}\ :\ \{exn(T)\}$$

(T1-Excep)
$$\frac{\begin{array}{c}\Gamma,\ T\ \vdash_T\ A_1\ :\ T',\\ \forall exn(t) \in T' :\ \Gamma,\ t\ \vdash_T\ A_2\ :\ T_t,\\ T'' = \cup_{exn(t) \in T'} T_t \cup normal(T') \cup fail(T')\end{array}}{\Gamma,\ T\ \vdash_T\ A_1\ \textbf{\textit{exceptionally}}\ A_2\ :\ T''}$$

(T1-Arith1)
$$\frac{T \in \{+, -, *\}, \quad T = (nat, nat)}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{give}}\ @\ :\ \{nat\}}$$

(T1-Arith2)
$$\frac{T \in \{+, -, *\}, \quad T \neq (nat, nat)}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{give}}\ @\ :\ \{exn(())\}}$$

(T1-Bind1)
$$\frac{T = (token, d) \wedge d \in bindable}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{give binding}}\ :\ \{(bindings)\}}$$

(T1-Bind2)
$$\frac{T \neq (token, d) \vee d \notin bindable}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{give binding}}\ :\ \{exn(())\}}$$

Table 3.3: The first type system. Continues on the following page.

(T1-BOUND1) $$\frac{T = (bindings, token)}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{give bound}}\ :\ bindable}$$

(T1-BOUND2) $$\frac{T \neq (bindings, token)}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{give bound}}\ :\ \{exn(())\}}$$

(T1-OVER1) $$\frac{T = (bindings\ b, bindings\ b')}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{give overriding}}\ :\ \{(bindings)\}}$$

(T1-OVER2) $$\frac{T \neq (bindings\ b, bindings\ b')}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{give overriding}}\ :\ \{exn(())\}}$$

(T1-PROJ1) $$\frac{T = (cell)}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{give the cell}}\ :\ \{(cell)\}}$$

(T1-PROJ2) $$\frac{T \neq (cell)}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{give the cell}}\ :\ \{exn(())\}}$$

(T1-PROJ3) $$\frac{T = (bindings)}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{give the bindings}}\ :\ \{(bindings)\}}$$

(T1-PROJ4) $$\frac{T \neq (bindings)}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{give the bindings}}\ :\ \{exn(())\}}$$

(T1-PROJ5) $$\frac{T = (nat)}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{give the nat}}\ :\ \{(nat)\}}$$

(T1-PROJ6) $$\frac{T \neq (nat)}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{give the nat}}\ :\ \{exn(())\}}$$

(T1-PROJ7) $$\Gamma,\ T\ \vdash_T\ \textbf{\textit{give the data}}\ :\ T$$

(T1-CHECK) $$\Gamma,\ T\ \vdash_T\ \textbf{\textit{check}}\ pred\ :\ \{(), exn(())\}$$

(T1-FAIL) $$\Gamma,\ T\ \vdash_T\ \textbf{\textit{fail}}\ :\ \{fail\}$$

Table 3.3: The first type system. Continues on the following page.

(T1-OTHER1)
$$\frac{\Gamma,\ T\ \vdash_T\ A_1\ :\ T',\quad \Gamma,\ T\ \vdash_T\ A_2\ :\ T'',\quad fail \in T',\quad T''' = (T' \setminus \{fail\}) \cup T''}{\Gamma,\ T\ \vdash_T\ A_1\ \textbf{\textit{otherwise}}\ A_2\ :\ T'''}$$

(T1-OTHER2)
$$\frac{\Gamma,\ T\ \vdash_T\ A_1\ :\ T',\quad fail \notin T'}{\Gamma,\ T\ \vdash_T\ A_1\ \textbf{\textit{otherwise}}\ A_2\ :\ T'}$$

(T1-CURBIN)
$$\Gamma,\ T\ \vdash_T\ \textbf{\textit{give current bindings}}\ :\ \{(bindings)\}$$

(T1-HENCE1)
$$\frac{\Gamma,\ T\ \vdash_T\ A_1\ :\ T',\quad T' = \{(bindings)\},\quad \Gamma,\ ()\ \vdash_T\ A_2\ :\ T''}{\Gamma,\ T\ \vdash_T\ A_1\ \textbf{\textit{hence}}\ A_2\ :\ T''}$$

(T1-HENCE2)
$$\frac{\Gamma,\ T\ \vdash_T\ A_1\ :\ T',\quad \Gamma,\ ()\ \vdash_T\ A_2\ :\ T'',\quad bindings \in T',\quad T' \neq \{(bindings)\},\quad T''' = T'' \cup \{exn(())\}}{\Gamma,\ T\ \vdash_T\ A_1\ \textbf{\textit{hence}}\ A_2\ :\ T'''}$$

(T1-HENCE3)
$$\frac{\Gamma,\ T\ \vdash_T\ A_1\ :\ T',\quad (bindings) \notin T'}{\Gamma,\ T\ \vdash_T\ A_1\ \textbf{\textit{hence}}\ A_2\ :\ \{exn(())\}}$$

(T1-CREATE1)
$$\frac{T \in storable}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{create}}\ :\ \{unit\}}$$

(T1-CREATE2)
$$\frac{T \notin storable}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{create}}\ :\ \{exn(())\}}$$

(T1-UPDATE1)
$$\frac{T = (cell, T_2),\quad T_2 \in storable}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{update}}\ :\ \{unit\}}$$

(T1-UPDATE2)
$$\frac{T = (T_1, T_2),\quad (T_1 \neq cell \vee T_2 \notin storable)}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{update}}\ :\ \{exn(())\}}$$

(T1-INSPECT1)
$$\frac{T = (cell)}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{inspect}}\ :\ storable}$$

Table 3.3: The first type system. Continues on the following page.

(T1-Inspect2) $$\frac{T \neq cell}{\Gamma,\ T\ \vdash_T\ \boldsymbol{inspect}\ :\ \{exn(())\}}$$

(T1-Unf1) $$\Gamma,\ T\ \vdash_T\ [\ \boldsymbol{unfold}\ ]^l\ :\ \emptyset$$

(T1-Unf2) $$\frac{\Gamma,\ -\ \vdash_T\ [\ A\ ]^{l'}\ :\ T'}{\Gamma,\ T\ \vdash_T\ [\ \boldsymbol{unfolding}\ [\ A\ ]^{l'}\ ]^l\ :\ T'}$$

(T1-Func) $$\frac{T' = \Gamma(m[\![args]\!])}{\Gamma,\ T\ \vdash_T\ [\ m[\![args]\!]\ ]^l\ :\ T'}$$

Table 3.3: The first type system.

3.1.3 Type system 2

The termination problem of the previous type inference algorithm is caused by the fact that the type system keeps track of the size of tuples and the type of each component of tuples, meaning the set of types is infinite for semantic productions that can be instantiated to produce tuples of arbitrary size. To solve this problem we introduce a new type grammar, as seen in Figure 3.6, which does not in general keep track of the size of tuples and the type of each component of tuples. Instead we introduce distinct types for *known tuples* and *unknown tuples*. Unknown tuples are tuples whose size we do not know, tuples whose size is greater than n, and tuples for which we do not know the type of one or more of its components. The only thing the type system records about unknown tuples is the set of types its components might be. As the set of datum types is finite, the set of unknown tuple types is also finite (it is the size of the powerset of datum types). Known tuples are tuples whose size is smaller than or equal to n, and for which we know the types of all its components. The set of known tuple types is thus also finite (it is the size of the nth cartesian product of the set of datum types) and since the inference rules given in Table 3.6 never produces a set containing the type term $exn(\tau)$ where τ is $exn(\tau')$ for some τ', the set of elements from **Type** that the type inference algorithm is able to infer is finite.

The presentation given in Figure 3.6 is parameterized over n, and so is our implementation of the algorithm. For the purpose of analyzing the WHILE language, the analysis is precise enough with n set to two.

The type inference algorithm is shown as an inference system in Tables 3.5 and 3.6. Many of the inference rules are equivalent to the corresponding rules of the inference system for the first inference algorithm. These rules have not been reproduced in Tables 3.5 and 3.6.

The syntax for known tuples is $[\tau_d, ..., \tau_d]$ to suggest that it is a list of datum types. The syntax for unknown tuples is $\{\tau_d, ..., \tau_d\}$ to suggest that it is a set of datum types. So, for instance, the definition of concatenation (the @ function in Figure 3.6) of two unknown tuples with sets of datum types T_1 and T_2 should be understood as giving an unknown tuple with the set of datum types $T_1 \cup T_2$.

With the subset of action notation that we are considering in this thesis, tuples of sizes greater than two are not useful, as our subset of action notation only includes actions for storing and retrieving such tuples, but no actions for manipulating them. However, as it does not cost much in precision to design the analysis such that it could be extended with the ***give*** $\#i$ action – which gives the i'th component of the tuple it is given – without requiring any major

changes, we have done so. With the inference rules given in Table 3.6, the sizes of the tuples that are given the unknown tuple type are always greater than two, however, with addition of the ***give*** $\#i$ action, this is no longer true. The inference rules in Table 3.6 therefore do not rely on this assumption, and would still be valid if the ***give*** $\#i$ action was added to our subset of action notation.

For the following analysis we need the ability to refer to the results of this analysis for any action and sub-action analyzed. Since we further need to be able to distinguish between multiple equivalent sub-actions, like say the two ***provide*** 42 actions in ***provide*** 42 ***and provide*** 42, we assume each sub-action has a unique label. $[\ A\ ]^l$ denotes an action A labeled l. Furthermore, in the following section we assume available a function, *types*, mapping labels to the set of types produced by the action with the given label, and a function $on_{sf}(l, B, B')$ giving B if the action labeled l ever terminates normally and B' if it always terminates non-normally and $B \cup B'$ otherwise, and a predicate $can_b(l)$ which is true when the action labeled l can produce bindings and false otherwise.

Table 3.4 shows the types inferred by the type and termination analysis, using the second type system, for each of the semantic productions of the WHILE language. The types inferred are as precise as possible, for the type systems used.

Production	Types inferred
$evala[\![N]\!]$	$\{[nat]\}$
$evala[\![I]\!]$	$\{[nat], exn([])\}$
$evala[\![AE_1 \ @ \ AE_2]\!]$ where $@ \in \{+, -, *\}$	$\{[nat], exn([])\}$
$evalb[\![true]\!], evalb[\![false]\!]$	$\{[bool]\}$
$evalb[\![\texttt{not}\ BE]\!]$	$\{[bool], exn([])$
$evalb[\![AE_1 \ @ \ AE_2]\!]$ where $@ \in \{=, >\}$	$\{[bool], exn([])\}$
$exec[\![I := AE]\!]$	$\{[bindings], exn([])\}$
$exec[\![\texttt{skip}]\!]$	$\{[bindings]\}$
$exec[\![\texttt{if}\ BE\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]$	$\{[bindings], exn([])\}$
$exec[\![\texttt{while}\ BE\ \texttt{do}\ S]\!]$	$\{[bindings], exn([])\}$
$exec[\![S_1; S_2]\!]$	$\{[bindings], exn([])\}$

Table 3.4: The types inferred for the productions of the WHILE language.

$\tau ::= [\tau_d, ..., \tau_d]$	known tuples
$\{\tau_d, ..., \tau_d\} \mid -$	unknown tuples
$exn(\tau) \mid fail$	exceptional values and failure
$\tau_d ::= nat \mid bool$	types for natural numbers and truth-values
$bindings \mid token \mid cell$	types for bindings, tokens and cells

$$tuple_1(T, t, T') = \begin{cases} T' & \text{if } T = [t] \\ T' \cup \{exn([])\} & \text{if } unknown\ T \wedge t \in T \\ \{exn([])\} & \text{otherwise} \end{cases}$$

$$tuple_2(T, t_1, t_2, T') = \begin{cases} T' & \text{if } T = [t_1, t_2] \\ T' \cup \{exn([])\} & \text{if } unknown\ T \wedge t_1 \in T \wedge t_2 \in T \\ \{exn([])\} & \text{otherwise} \end{cases}$$

$$unknown\ T \quad \text{iff} \quad T \text{ is an } unknown\ tuple \text{ type}$$

$$T_1 \times T_2 = \{\tau\ @\ \tau' \mid \tau \in T_1 \wedge \tau' \in T_2\}$$

$$[\tau_1, ..., \tau_i]\ @\ [\tau'_1, ..., \tau'_i] = \begin{cases} [\tau_1, ..., \tau_i, \tau'_1, ..., \tau'_i] & \text{if } i + i' \leq n \\ \{\tau_1, ..., \tau_i, \tau'_1, ..., \tau'_i\} & \text{otherwise} \end{cases}$$

$$[\tau_1, ..., \tau_i]\ @\ \{\tau'_1, ..., \tau'_i\} = \{\tau_1, ..., \tau_i, \tau'_1, ..., \tau'_i\}$$

$$\{\tau_1, ..., \tau_i\}\ @\ [\tau'_1, ..., \tau'_i] = \{\tau_1, ..., \tau_i, \tau'_1, ..., \tau'_i\}$$

$$\{\tau_1, ..., \tau_i\}\ @\ \{\tau'_1, ..., \tau'_i\} = \{\tau_1, ..., \tau_i, \tau'_1, ..., \tau'_i\}$$

Figure 3.6: Grammar of types for the second termination analysis.

$$n \in \mathbf{Nat}, \quad b \in \mathbf{Bool}, \quad t \in \mathbf{Token}, \quad d_i \in \mathbf{Datum}$$

(TD2-NAT) $$\vdash \ n \ : \ nat$$

(TD2-BOOL) $$\vdash \ b \ : \ bool$$

(TD2-TOKEN) $$\vdash \ t \ : \ token$$

(TD2-NUM) $$\frac{(id, \{numeral\}) \in \Gamma_{ident}}{\vdash \ id \ : \ nat}$$

(TD2-IDENT) $$\frac{(id, \{ident\}) \in \Gamma_{ident}}{\vdash \ id \ : \ token}$$

(TD2-TUPLE1) $$\frac{i \leq n, \quad \forall k \in [1, i] \vdash \ d_k \ : \ t_k}{\vdash \ (d_1, ..., d_i) \ : \ [t_1, ..., t_i]}$$

(TD2-TUPLE2) $$\frac{i > n, \quad \forall k \in [1, i] \vdash \ d_i \ : \ t_i,}{\vdash \ (d_1, ..., d_i) \ : \ \{t_1, ..., t_i\}}$$

Table 3.5: Type rules for action data.

$$d \in \mathbf{Data}, \quad A \in \mathbf{Action}$$

(T2-Arith) $$\frac{T \in \{+, -, *\}, \quad T' = tuple_2(T, nat, nat, \{[nat]\})}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{give}}\ @\ :\ T'}$$

(T2-Bind1) $$\frac{T = [token, d] \wedge d \in bindable}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{give binding}}\ :\ \{[bindings]\}}$$

(T2-Bind2) $$\frac{T \neq [token, d] \vee d \notin bindable}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{give binding}}\ :\ \{exn([])\}}$$

(T2-Bound) $$\frac{T' = tuple_2(T, bindings, token, bindable)}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{give bound}}\ :\ T'}$$

(T2-Over) $$\frac{T' = tuple_2(T, bindings, bindings, bindings)}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{give overriding}}\ :\ T'}$$

(T2-Proj1) $$\frac{T' = tuple_1(T, cell, \{[cell]\})}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{give the cell}}\ :\ T'}$$

(T2-Proj2) $$\Gamma,\ T\ \vdash_T\ \textbf{\textit{give the data}}\ :\ T$$

(T2-Proj3) $$\frac{T' = tuple_1(T, bindings, \{[bindings]\})}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{give the bindings}}\ :\ T'}$$

(T2-Proj4) $$\frac{T' = tuple_1(T, nat, \{[nat]\})}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{give the nat}}\ :\ T'}$$

(T2-CurBin) $$\Gamma,\ T\ \vdash_T\ \textbf{\textit{give current bindings}}\ :\ \{[bindings]\}$$

(T2-Create1) $$\frac{T \in storable}{\Gamma,\ T\ \vdash_T\ \textbf{\textit{create}}\ :\ \{[]\}}$$

Table 3.6: The second type system. Continues on the following page.

(T2-CREATE2)
$$\frac{T \notin storable}{\Gamma,\ T\ \vdash_T\ \ \textbf{\textit{create}}\ :\ \{exn([])\}}$$

(T2-UPDATE1)
$$\frac{T = [cell, d] \wedge d \in storable}{\Gamma,\ T\ \vdash_T\ \ \textbf{\textit{update}}\ :\ \{[]\}}$$

(T2-UPDATE2)
$$\frac{T \neq [d_1, d_2] \vee d_1 \neq cell \vee d_2 \notin storable}{\Gamma,\ T\ \vdash_T\ \ \textbf{\textit{update}}\ :\ \{exn([])\}}$$

(T2-INSPECT1)
$$\frac{T' = tuple_1(T, cell, storable)}{\Gamma,\ T\ \vdash_T\ \ \textbf{\textit{inspect}}\ :\ T'}$$

Table 3.6: The second type system.

3.2 Binding analysis

In this section we first develop an analysis which allows us to track how bindings flow inside and between different language constructs, for some languages. Subsequently, we develop an algorithm which takes the results of the binding analysis and generates a reaching bindings analysis for the source language, about a given program's denotation.

To illustrate the analyses themselves and how they are useful, we start with a simple example. Consider the two semantic productions from the action semantic description of WHILE – reproduced in Figure 3.7 – specifying the semantics of statement sequencing and the if construct, respectively. From the type and termination analysis of the previous section we know that statements either terminate normally giving bindings or terminate exceptionally, raising a 0-tuple. For the statement $S_1; S_2$ we can see that first S_1 is executed using the current bindings, then S_2 is executed using the bindings given by S_1, and that the bindings given by $S_1; S_2$ are the bindings given by S_2. Similarly for the if construct, we see that first the boolean expression is evaluated using the current bindings, and then either S_1 or S_2 is evaluated using the current bindings, and that the bindings produced by the if construct is either the bindings produced by S_1 or the bindings produced by S_2.

From these results we can for instance derive the following set equation templates, which allow us to analyze, which tokens might be bound before and after the execution of the two language constructs. The *entry* function specifies the tokens that might be bound before execution and the *exit* function the tokens

- $exec[\![S_1;\ S_2]\!] = exec[\![S_1]\!]$ ***hence*** $exec[\![S_2]\!]$
- $exec[\![$`if` BE `then` S_1 `else` $S_2]\!] =$
  $eval_b[\![BE]\!]$ ***then*** (
  (***give*** $true$ ***then*** $exec[\![S_1]\!]$)
  otherwise $exec[\![S_2]\!]$)

Figure 3.7: The semantic productions of the WHILE language for statement sequences and if statements.

that might be bound after execution:

$$\begin{aligned}
entry(exec[\![S_1]\!]) &= entry(exec[\![S_1;S_2]\!])\\
entry(exec[\![S_2]\!]) &= exit(exec[\![S_1]\!])\\
exit(exec[\![S_1;S_2]\!]) &= exit(exec[\![S_2]\!])
\end{aligned}$$

$$\begin{aligned}
entry(exec[\![BE]\!]) &= entry(exec[\![\texttt{if } BE \texttt{ then } S_1 \texttt{ else } S_2]\!])\\
entry(exec[\![S_1]\!]) &= entry(exec[\![\texttt{if } BE \texttt{ then } S_1 \texttt{ else } S_2]\!])\\
entry(exec[\![S_2]\!]) &= entry(exec[\![\texttt{if } BE \texttt{ then } S_1 \texttt{ else } S_2]\!])\\
exit(exec[\![\texttt{if } BE \texttt{ then } S_1 \texttt{ else } S_2]\!]) &= exit(exec[\![S_1]\!]) \cup exit(exec[\![S_2]\!])
\end{aligned}$$

Since the bindings used to perform S_1 in $S_1;S_2$ are the bindings used to perform $S_1;S_2$, the bindings that might be bound before the execution of S_1 are the bindings that might be bound before the execution of $S_1;S_2$. This is what the first equation expresses. Likewise, the last equation expresses that the bindings that might be bound after the execution of an if statement are the union of the bindings that might be bound after the execution of the if statement's branches.

From these equation templates we could for instance generate an analysis that takes a source program and uses the equation templates derived for the source language to generate a set of reaching bindings equations for the entire program program and solve these equations.

The semantics of the assignment statement, $I := AE$, in While, specifies that if a cell is already bound to the identifier I then that cell is updated with the value of AE, otherwise a new cell is created with the value of AE and bound to I. For the While language the solved reaching bindings equations could thus be used to eliminate the action that checks whether a cell is already bound to I, for those assignment statements where the reaching bindings equations do not show that I might be bound before execution.

3.2.1 Binding flow analysis

The binding flow analysis is a type analysis, with a type system for tracking the structure of the bindings used and produced by semantic productions. Unlike the type and termination analysis presented before, where we were interested in the types produced by all possible instantiations of semantic productions, here we are interested in the bindings used and produced by semantic productions, parametrized by the bindings used and produced by the semantic functions called. We can therefore perform this analysis on each semantic production independently of each other, without having to iterate until a fix-point is reached, as for the type and termination analysis.

To simplify the analysis, we impose a series of restrictions on the action semantic descriptions that we are able to analyze. These restrictions only apply to the semantic productions of those semantic functions where one or more of its semantic productions manipulate bindings (i.e., if some sub-action of its denotation produces bindings).

- Actions must not terminate exceptionally raising bindings and bindings must not be bindable or storable. Without these restrictions (especially the second one), much of the analysis would have to be delayed and performed on concrete programs instead of language specifications. For example, consider the WHILE language extended with the following two semantic productions.

  - *exec*⟦`save` *I*⟧ =
    give current bindings and
    (***provide*** *I* ***and give current bindings***
    then give binding)
    then give overriding
  - *exec*⟦*S* `using` *I*⟧ =
    give the bindings bound to *I* ***hence*** *exec*⟦*S*⟧
    then give current bindings

  The first allows the current bindings to be bound to an identifier and the second allows a statement to be executed using the bindings bound to the given identifier. Since we are looking at each language construct in isolation, we have no choice but to assume that any token could be bound in the bindings bound to *I* in *S* `using` *I*, which would obviously yield a useless analysis. Most of the analysis would have to be performed on concrete programs to be useful for such languages.

  With some extra work, the first restriction could be relaxed to allow the raising of bindings within semantic productions, as long as they are always caught again within the semantic production.

  This first restriction has the unfortunate consequence that we for instance cannot analyze the WHILE language extended with the following two semantic productions, which add the ability to throw and catch a primitive form of exceptions. In general, this restriction means that we are unable to analyze languages that support exceptions via AN's exceptional data and where the scoping rules of the language are such that the current bindings at the point where the exception is thrown are needed where the exception is caught.

- $exec[\![$`try` S_1 `catch` $S_2]\!] =$

  $exec[\![S_1]\!]$ ***exceptionally copy hence*** $exec[\![S_2]\!]$

- $exec[\![$`throw`$]\!] =$ ***give current bindings then raise***

For languages such as WHILE , where variables do not have to be declared before they can be assigned a value, it seems reasonable to expect that variables which are initialized in S_1 of `try` S_1 `catch` S_2 are accessible in S_2. This analysis will not be able to analyze such a language (if its exceptions are implemented using exceptional values), as S_2 has to be performed with the current bindings at the point where the exception was thrown in S_1. However, languages where the current bindings at the point where an exception is thrown are not used for anything (and therefore not passed along as exceptional data) – like, say, a WHILE-like language, without global variables, where all variables are local to a block, declared via a block construct $\{V\ S\}$, where V is a variable declaration – can still be analyzed.

- If $[bindings] \in T$, where T is the type of a semantic function, then the type S of all its semantic productions must satisfy $[bindings] \in S$. If $[bindings] \in T$, where T is the type of a semantic production or unfolding, then T must satisfy $T \subset \{[bindings], fail, exn(_)\}$. All semantic productions must further satisfy the following conditions, where T is the type of the given semantic production:

  - There must not exist n datum types, $t_1, ..., t_n$, where $n > 1$, such that $[t_1, ..., t_n] \in T$, where one or more of the datum types is *bindings*.
  - There must not exist n datum types, $t_1, ..., t_n$, such that $\{t_1, ..., t_n\} \in T$, where one of the datum types is *bindings*.

In effect, this means that the only way bindings are allowed to flow between language constructs, are as 1-tuples, where the one tuple component is a set of bindings.

The analysis could be extended to avoid these restrictions, however, these restrictions simplified the implementation of the analysis and still allowed us to analyze WHILE, so we chose not to spend the time extending the analysis.

One extension is to allow bindings to flow in n-tuples, where n can vary for each semantic function. That is, all semantic productions belonging to the a given semantic function must always produce tuples of size n and such that it is always the same components of the tuple that are sets of bindings. Consider, for instance, the WHILE language extended with the following two semantic productions:

- $exec2[\![I]\!] =$

  provide I ***and give current bindings***
  and give current bindings

- $exec[\![I]\!] = exec2[\![I]\!]$ ***then give*** $\#2$ ***and give*** $\#3$

They are not very useful, but I could not think of any useful language extensions to WHILE to illustrate the point, so these will have to do. The first production takes an identifier I and gives the tuple (I, b, b) where b are the current bindings, and the second production calls the first production with the given identifier I and gives the 2-tuple (d_2, d_3), where d_i is the ith component of the tuple given by the action produced by the semantic function call.

To be able to handle such languages correctly, the type system would have to be extended to keep track of which component of the tuple given by function calls, bindings given by function calls come from. The algorithm that generates the *entry*/*exit* equations could then generate an $exit_i$ equation for each component, i, of the n-tuples that the given semantic production can produce, that is a set of bindings. For the above example we would get the following equations:

$$
\begin{aligned}
exit_2(exec2[\![I]\!]) &= entry(exec2[\![I]\!]) \\
exit_3(exec2[\![I]\!]) &= entry(exec2[\![I]\!]) \\
exit_1(exec[\![I]\!]) &= exit_2(exec2[\![I]\!]) \\
exit_2(exec[\![I]\!]) &= exit_3(exec2[\![I]\!])
\end{aligned}
$$

The above extension could be extended even further, dropping all restrictions, expect that now no semantic productions can have unknown tuple types among its types. Each semantic production would have to be analyzed once for each "type instantiations" of its semantic function calls, however, fix-point iteration would not be needed, as the "type instantiations" could simply be generated from the results of the type and termination analysis. When the generated *entry*/*exit* equation templates are instantiated for the entire program, some equations might refer to undefined $exit_i$ equations – for example, if semantic productions sometimes produce tuples containing bindings and other times produces bindings which does not contain bindings, depending on instantiation – these should simply be set to $\emptyset$. The analysis would still be correct, as each production would have been analyzed once for each possible "type instantiations", also the ones where the type of the semantic function calls did not include any bindings. However, the analysis would probably be very imprecise for some languages.

- Tokens must neither be storable nor bindable. Again, this analysis is too imprecise for languages which do not satisfy this condition. Consider the WHILE language extended with the following two semantic productions:

  - $exec[\![I_1 := I_2]\!] =$

    give current bindings and

    (***provide*** I_1 ***and provide*** I_2 ***give binding***)

    then give overriding

  - $exec[\![I \leftarrow A]\!] =$

    $evala[\![A]\!]$ ***and***

    (***give the data bound to*** I)

    then update

  The first allows a token to be bound to another token, and the second is a slightly modified assignment construct, which updates the cell bound to the token bound to the given token, with the value of the arithmetic expression. Without looking at a concrete program, all that we would be able to say about the bindings produced by the first semantic production, would be that any token could be bound.

- The denotation of semantic productions must not raise or give tokens. Again, the reasoning is that if language constructs can give or raise tokens, we cannot analyze language constructs individually, we have to look at how they are combined.

- Finally, this analysis depends on the results of the type and termination analysis, so the specification also has to satisfy all the restrictions from the type and termination analysis.

The implementation of this analysis verifies that all the above conditions are satisfies by the given specification, before running the analysis.

The type system used for this analysis – which is shown in Table 3.7 – is similar to the type system used for the first type and termination analysis. The main differences are the introduction of an *other* type for the values that we do not care about and a way of tracking the structure of bindings. These differences are discussed below.

- *The bindings* β *type*: The type for bindings, *bindings* β, has been extended with a parameter to specify the structure of the given bindings. The parameter of binding terms should be interpreted as follows:

  - *entry*: Represents the bindings used to perform the given semantic production with.
  - $\{t\}$: A singleton binding of token t.
  - $\beta_1 + \beta_2$: The bindings produced by β_2 overriding β_1.
  - $given(a)$: The bindings given by action a, where a is a semantic function call.
  - $unf_{entry}(a)$: The current unfolding bindings.
  - $unf_{exit}(a, (\beta_1, ..., \beta_n))$: $\beta_1, ..., \beta_n$ are the bindings produced by action a, where a is an unfolding action.

  The structure of the bindings produced by the `if` b `then` S_1 `else` S_2 construct of WHILE would for example be $given(exec[\![S_1]\!])$ and $given(exec[\![S_1]\!])$ and the bindings used to perform $evalb[\![b]\!]$, $exec[\![S_1]\!]$, and $exec[\![S_2]\!]$ would be *entry* (i.e.., the bindings used to perform `if` b `then` S_1 `else` S_2).

- *The other type*: With this analysis we encounter the same problem of tuples that we encountered with the type and termination analysis: sometimes we need to know the length and types of the components of tuples, but at the same time we do not wish to discriminate against languages that support tuples of arbitrary length.

  Since we are only interested in bindings and tokens for this analysis, we start by introducing a new type, *other*, to cover all other values than bindings and tokens. To solve the problem of tuples of arbitrary length, we change the interpretation of the *other* type to also cover an arbitrary number of consecutive components in an n-tuple (including zero), where none of the datums in these components are bindings or tokens. So for instance the type $[nat, nat, token, nat]$ from the first type system would become $(other, other, token, other)$ or $(other, token, other)$. Every time an action is assigned a type, the type is first reduced, so that it contains the minimum number of *other*s necessary to represent the original type. So, for instance, $(other, other, token, other)$ would reduce to $(other, token, other)$.

  This approach gives a reasonable approximation in all but the most pathological cases. Consider for instance the WHILE language extended with the semantic production below. This language construct is obviously broken, as it has no effect; it will always terminate normally giving the current bindings. However, the analysis is not precise enough to notice that the attempt to bind the value given by $evala[\![A]\!]$ to the identifier I will always fail, as the type inferred for the value given to the ***give binding*** action will be $(token, other)$, leaving us unable to conclude that the action will always terminate exceptionally. The analysis will therefore give the results $\{\{I\} + entry, entry\}$ instead of the more accurate $\{entry\}$.

– $exec[\![I \leftarrow A]\!] =$

give current bindings and (***provide*** I ***and*** $evala[\![A]\!]$ ***and provide*** 42 ***then give binding***) ***then give overriding exceptionally give current bindings***

The analysis itself is given as an inference system in Tables 3.9 and 3.10. A typing judgment of the form,

$$T,\ \xi,\ \eta\ \vdash A : (T', m)$$

should be interpreted as follows: Action A produces types T' when given a value of type T and performed using the bindings ξ enclosed in the unfolding η. m is a mapping from sub-actions of A (more specifically semantic function calls and ***unfolding*** actions) to sets of β terms, of the bindings that the given action could be performed with. This map will be referred to as a binding map. The *merge* function, defined in Table 3.7, takes a set of binding maps S and merges them into a new map. In the merged map, action a is mapped to the union of the sets of binding parameter terms those maps in S which contains a mapping for a maps a to.

Many of the inference rules, especially the inference rules for action combinators, are very similar to the inference rules of the previous type inference algorithms, however, a few, most notably the inference rules for unfold and unfolding, and the actions that manipulated bindings, are quite interesting. These inference rules are discussed below.

- B-Bind: The ***give binding*** action terminates exceptionally unless given a 2-tuple consisting of a token and a bindable datum. If ***give binding*** is given a value of type $(other, token, other)$, then we have to assume that it will produce a binding of the given token, as that type covers such values as a 2-tuple where the first component is a token and the second component a bindable datum. In general, the only conditions under which we can be certain that ***give binding*** will not produce a binding, is if the type T of the value given to ***give binding*** does not satisfy the following two conditions:
  – The first component of T that is not *other* must be *token*. The B-Bind rule makes use of a *trim* function to specify this condition. The *trim* function takes a type such as $(other, token, other)$ and removes all *other*s from the tuple, giving $(token)$ for this example.
  – No 2-tuple where one of the components of the type is a token has a 0- or 1-tuple type, so the size of T must be at least 2.

$$a \in \textbf{Action}, \quad t \in \textbf{Token}$$

$\tau ::=$	$(\tau_d, ..., \tau_d)$	tuples
	$exn(\tau) \mid fail$	
$\tau_d ::=$	$other$	
	$bindings(\beta)$	a set of bindings
	$token(t)$	a token
$\beta ::=$	$entry$	entry bindings
	$\{t\}$	singleton binding
	$\beta + \beta$	overriding bindings
	$given(a)$	function call bindings
	$unf_{entry}(a)$	
	$unf_{exit}(a, (\beta, ..., \beta))$	

$$merge(\{m_1, m_2, ..., m_n\})\ a = \begin{cases} \cup_{i \in I}\ m_i(a) & \text{if } I \neq \emptyset \\ undef & \text{otherwise} \end{cases}$$

where $I \subseteq \mathbf{n}$, such that $\forall i \in I : a \in dom(m_i) \wedge \forall i \in \mathbf{n} \setminus I : a \notin dom(m_i)$

$$conv(a, [\tau_1, ..., \tau_n]) = \begin{cases} () & \text{if } n = 0 \\ (bindings(given(a))) & \text{if } n = 1 \wedge \tau_i = bindings \\ (other) & \text{if } \ n > 0 \wedge \forall i : (\tau_i \neq bindings \\ & \qquad \wedge\ \tau_i \neq token) \\ undef & \text{otherwise} \end{cases}$$

$$conv(a, \{\tau_1, ..., \tau_n\}) = \begin{cases} (other) & \text{if } \forall i : \tau_i \neq bindings \wedge \tau_i \neq tokens \\ undef & \text{otherwise} \end{cases}$$

$$conv(a, exn(\tau)) = exn(conv(a, \tau))$$

$$conv(a, fail) = (fail)$$

Table 3.7: Grammar of types for the binding flow analysis.

- B-Unf1 and B-Unf2: As the restrictions that ensured that ***unfold***s do not contribute anything to the type of the enclosing ***unfolding*** also apply to this analysis, we can disregard the types produced by ***unfold***s. However, as the ***unfold*** causes the enclosing ***unfolding*** (η) to be performed with the current bindings, a mapping is added from η to $\{\xi\}$ (the current bindings).

$$n \in \mathbf{Nat}, \quad b \in \mathbf{Bool}, \quad t \in \mathbf{Token}, \quad d_i \in \mathbf{Datum}$$

(BA-Nat) $$\vdash \; n \; : \; other$$

(BA-Bool) $$\vdash \; b \; : \; other$$

(BA-Token) $$\vdash \; t \; : \; token(t)$$

(BA-Num) $$\frac{(id, \{numeral\}) \in \Gamma_{ident}}{\vdash \; id \; : \; other}$$

(BA-Ident) $$\frac{(id, \{ident\}) \in \Gamma_{ident}}{\vdash \; id \; : \; token(id)}$$

(BA-Tuple) $$\frac{\forall i : \vdash \; d_i \; : \; t_i}{\vdash \; (d_1, ..., d_n) \; : \; (t_1, ..., t_n)}$$

Table 3.9: Type rules for action data.

(B-Provide)
$$\frac{\vdash\ d\ :\ t}{T,\ \xi,\ \eta\ \vdash [\ \textbf{\textit{provide}}\ d\ ]^l : (\{t\}, [])}$$

(B-Copy)
$$T,\ \xi,\ \eta\ \vdash [\ \textbf{\textit{copy}}\ ]^l : (\{T\}, [])$$

(B-Then)
$$\frac{\begin{array}{c} T,\ \xi,\ \eta\ \vdash [\ A_1\ ]^{l'} : (T', m'), \\ \forall t \in normal(T') :\ t,\ \xi,\ \eta\ \vdash [\ A_2\ ]^{l''} : (T_t, m_t), \\ T'' = \cup_{t \in normal(T')} T_t \cup failed(T'), \\ m'' = merge(\{m'\} \cup \{m_t \mid t \in normal(T')\}) \end{array}}{T,\ \xi,\ \eta\ \vdash [\ [\ A_1\ ]^{l'}\ \textbf{\textit{then}}\ [\ A_2\ ]^{l''}\ ]^l : (T'', m')}$$

(B-And)
$$\frac{\begin{array}{c} T,\ \xi,\ \eta\ \vdash [\ A_1\ ]^{l'} : (T', m'), \\ T,\ \xi,\ \eta\ \vdash [\ A_2\ ]^{l''} : (T'', m''), \\ T' = normal(T') \times normal(T'') \cup failed(T' \cup T''), \\ m''' = merge(\{m', m''\}) \end{array}}{T,\ \xi,\ \eta\ \vdash [\ [\ A_1\ ]^{l'}\ \textbf{\textit{and}}\ [\ A_2\ ]^{l''}\ ]^l : (T', m''')}$$

(B-Raise)
$$\frac{\begin{array}{c} T = (\tau_1, ..., \tau_n), \\ \forall i \in [1, n] : \neg\exists b : \tau_i = bindings\ b \end{array}}{T,\ \xi,\ \eta\ \vdash [\ \textbf{\textit{raise}}\ ]^l : (\{exn\}, [])}$$

(B-Excep)
$$\frac{\begin{array}{c} T,\ \xi,\ \eta\ \vdash [\ A_1\ ]^{l'} : (T', m'), \\ \forall exn(t) \in T' : t,\ \xi,\ \eta\ \vdash [\ A_2\ ]^{l''} : (T_t, m_t), \\ T'' = normal(T') \cup fail(T') \cup \{T_t \mid exn(t) \in T'\}, \\ m'' = merge(\{m'\} \cup \{m_t \mid exn(t) \in T'\}) \end{array}}{T,\ \xi,\ \eta\ \vdash [\ [\ A_1\ ]^{l'}\ \textbf{\textit{exceptionally}}\ [\ A_2\ ]^{l''}\ ]^l : (T'', m'')}$$

(B-Give)
$$\frac{\begin{array}{c} @ \in \{+, -, *\}, \\ T' = on_{sf}(l, \{(other)\}, \{exn(())\}) \end{array}}{T,\ \xi,\ \eta\ \vdash [\ \textbf{\textit{give}}\ @\ ]^l : (T', [])}$$

(B-Check)
$$\frac{\begin{array}{c} @ \in \{>, =\}, \\ T' = on_{sf}(l, \{T\}, \{exn(())\}) \end{array}}{T,\ \xi,\ \eta\ \vdash [\ \textbf{\textit{check}}\ @\ ]^l : (T', [])}$$

Table 3.10: The binding flow analysis. Continues on the following page.

(B-Proj1) $$T,\ \xi,\ \eta\ \vdash [\ \textbf{\textit{give the data}}\ ]^l : (\{T\}, [])$$

(B-Proj2) $$\frac{@ \in \{\textbf{\textit{nat}}, \textbf{\textit{bindings}}, \textbf{\textit{cell}}\}}{T,\ \xi,\ \eta\ \vdash \textbf{\textit{give the}}\ @ : (\{T, exn([])\}, [])}$$

(B-Bound) $$\frac{T' = on_{sf}(l, \{(other)\}, \{exn(())\})}{T,\ \xi,\ \eta\ \vdash [\ \textbf{\textit{give bound}}\ ]^l : (T', [])}$$

(B-Bind) $$\frac{trim(T) = (token\ t, ...), \quad |T| > 1}{T,\ \xi,\ \eta\ \vdash [\ \textbf{\textit{give binding}}\ ]^l : (\{(bindings\ \{t\})\}, [])}$$

(B-Over) $$\frac{trim(T) = (bindings\ b, bindings\ b')}{T,\ \xi,\ \eta\ \vdash [\ \textbf{\textit{give overriding}}\ ]^l : (\{(bindings\ (b + b'))\}, [])}$$

(B-CurBin) $$T,\ \xi,\ \eta\ \vdash \textbf{\textit{give current bindings}} : (\{(bindings\ \xi)\}, [])$$

(B-Hence1) $$\frac{\begin{array}{c} T,\ \xi,\ \eta\ \vdash [\ A_1\ ]^{l'} : (T', ,) \\ bins(T') \neq \emptyset \wedge bins(T') = T', \\ \forall bindings\ b \in bins(T') : other,\ b,\ \eta\ \vdash [\ A_2\ ]^{l''} : (T_b, ,) \\ T''' = \cup_{t \in bins(T')} T_b \end{array}}{T,\ \xi,\ \eta\ \vdash [\ [\ A_1\ ]^{l'}\ \textbf{\textit{hence}}\ [\ A_2\ ]^{l''}\ ]^l : (T''', [])}$$

(B-Hence2) $$\frac{\begin{array}{c} T,\ \xi,\ \eta\ \vdash [\ A_1\ ]^{l'} : (T', ,) \\ bins(T') = \emptyset \vee bins(T') \neq T', \\ \forall bindings\ b \in bins(T') : other,\ b,\ \eta\ \vdash [\ A_2\ ]^{l''} : (T_b, ,) \\ T''' = \cup_{t \in bins(T')} T_b \cup \{exn(\perp)\} \end{array}}{T,\ \xi,\ \eta\ \vdash [\ [\ A_1\ ]^{l'}\ \textbf{\textit{hence}}\ [\ A_2\ ]^{l''}\ ]^l : (T''', [])}$$

(B-Create) $$\frac{T' = on_{sf}(l, \{(other)\}, \{exn(())\})}{T,\ \xi,\ \eta\ \vdash [\ \textbf{\textit{create}}\ ]^l : (T', [])}$$

(B-Update) $$\frac{T' = on_{sf}(l, \{()\}, \{exn(())\})}{T,\ \xi,\ \eta\ \vdash [\ \textbf{\textit{update}}\ ]^l : (T', [])}$$

Table 3.10: The binding flow analysis. Continues on the following page.

(B-INSPECT)
$$\frac{T' = on_{sf}(l, \{(other)\}, \{exn(())\})}{T,\ \xi,\ \eta\ \vdash [\ \textbf{\textit{inspect}}\ ]^{l} : (T', [])}$$

(B-UNF1)
$$\frac{m' = [\eta \mapsto \xi]}{T,\ \xi,\ \eta\ \vdash [\ \textbf{\textit{unfold}}\ ]^{l} : (\emptyset, m')}$$

(B-UNF2)
$$\frac{\begin{array}{c}\neg can_b(\eta),\\ T,\ unf_{entry}(l),\ [\ \textbf{\textit{unfolding}}\ [\ A\ ]^{l'}\ ]^{l}\ \vdash [\ A\ ]^{l'} : (T', m'),\\ m'' = [[\ \textbf{\textit{unfolding}}\ [\ A\ ]^{l'}\ ]^{l} \mapsto \xi],\\ m''' = merge(\{m', m''\})\end{array}}{T,\ \xi,\ \eta\ \vdash [\ \textbf{\textit{unfolding}}\ [\ A\ ]^{l'}\ ]^{'} : (T', m')}$$

(B-UNF3)
$$\frac{\begin{array}{c}can_b(\eta),\\ T,\ unf_{entry}(l),\ [\ \textbf{\textit{unfolding}}\ [\ A\ ]^{l'}\ ]^{l}\ \vdash [\ A\ ]^{l'} : (T', m'),\\ T'' = \{bindings\ unf_{exit}(l, bins(T'))\} \cup nbins(T'),\\ m'' = [[\ \textbf{\textit{unfolding}}\ [\ A\ ]^{l'}\ ]^{l} \mapsto \xi],\\ m''' = merge(\{m', m''\})\end{array}}{T,\ \xi,\ \eta\ \vdash [\ \textbf{\textit{unfolding}}\ [\ A\ ]^{l'}\ ]^{l} : (T'', m''')}$$

(B-CALL)
$$\frac{\begin{array}{c}T' = \{conv([\ m[\![args]\!]\ ]^{l}, \tau) \mid \tau \in type(l)\},\\ m' = [[\ m[\![args]\!]\ ]^{l} \mapsto \xi]\end{array}}{T,\ \xi,\ \eta\ \vdash [\ m[\![args]\!]\ ]^{l} : (T', m')}$$

Table 3.10: The binding flow analysis.

3.2.2 Reaching bindings

The next step is an algorithm which can take the results of the binding flow analysis and generate a reaching bindings analysis, in the form of a set of equation templates for each language construct. These equation templates express what tokens might be bound after performing the denotation of the given language construct, in terms of the tokens that might have been bound before its performance.

Most of the work is done by the binding flow analysis, its results simply have to be presented as a series of equation templates. Table 3.11 defines a number of functions for generating equation templates. The eqs_{exit} function takes the denotation of the semantic production p under analysis, an action a, where a is a sub-action of p, and a binding type of the binding flow analysis, and generates a set of *exit* equation templates corresponding to the given binding type. Similarly, the eqs_{entry} function generates *entry* equation templates corresponding to the given binding type. Finally, the *eqs* function takes the denotation of a semantic production a gives a set of *entry*/*exit* equation templates.

If the set of types for the given semantic production contains more than one binding type, or one or more of the actions in the bindings map map to a set of types containing more than one binding type, then the set of generated equation templates will contain multiple equations with the same left hand side, which needs to be combined into one equation. Consider for instance the if construct of the WHILE language. The bindings given by $exec[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]$ are either the bindings given by $exec[\![S_1]\!]$ or the bindings given by $exec[\![S_2]\!]$, so the set of equation templates given by the *eqs* function would include the following two:

$$exit(exec[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]) = exit(exec[\![S_1]\!])$$
$$exit(exec[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]) = exit(exec[\![S_2]\!])$$

The set of tokens that might be bound after $exec[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]$ has been performed is thus the union of the tokens that might be bound after $exec[\![S_1]\!]$ has been performed and the tokens that might be bound after $exec[\![S_2]\!]$ has been performed. We can thus rewrite the equation to:

$$exit(exec[\![\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2]\!]) = exit(exec[\![S_1]\!]) \cup exit(exec[\![S_2]\!])$$

Similarly, in general, equation templates with the same left hand side are rewritten to a single equation where the right hand side of the new equation is the union of the right hand side of all the old equations.

The only interesting aspect of the eqs_{exit} and eqs_{entry} functions is how unfoldings that give bindings are handled. Figure 3.8 shows a graphical representation

$$eqs_{exit} : \mathbf{Action} \times \mathbf{Action} \rightarrow \mathbf{Binding} \rightarrow \mathcal{P}(\mathbf{BinEqs})$$

$$\begin{aligned}
eqs_{exit}(a,p)(entry) &= \{exit(a) = entry(p)\} \\
eqs_{exit}(a,p)(given(a')) &= \{exit(a) = exit(a')\} \\
eqs_{exit}(a,p)(\{t\}) &= \{exit(a) = \{t\}\} \\
eqs_{exit}(a,p)(\beta_1 + \beta_2) &= eqs_{exit}(a,p)(\beta_1) \cup eqs_{exit}(a,p)(\beta_2) \\
eqs_{exit}(a,p)(unf_{entry}(a')) &= \{exit(a) = entry(a')\} \\
eqs_{exit}(a,p)(unf_{exit}(a',B)) &= \{exit(a) = exit(a')\} \\
&\quad \cup \{eqs_{exit}(a',p)(\beta) \mid \beta \in B\}
\end{aligned}$$

$$eqs_{entry} : \mathbf{Action} \times \mathbf{Action} \rightarrow \mathbf{Binding} \rightarrow \mathcal{P}(\mathbf{BinEqs})$$

$$\begin{aligned}
eqs_{entry}(a,p)(entry) &= \{entry(a) = entry(p)\} \\
eqs_{entry}(a,p)(given(a')) &= \{entry(a) = exit(a')\} \\
eqs_{entry}(a,p)(\{t\}) &= \{entry(a) = \{t\}\} \\
eqs_{entry}(a,p)(\beta_1 + \beta_2) &= eqs_{entry}(a,p)(\beta_1) \cup eqs_{entry}(a,p)(\beta_2) \\
eqs_{entry}(a,p)(unf_{entry}(a')) &= \{entry(a) = entry(a')\} \\
eqs_{entry}(a,p)(unf_{exit}(a',B)) &= \{entry(a) = exit(a')\} \\
&\quad \cup \{eqs_{exit}(a',p)(\beta) \mid \beta \in B\}
\end{aligned}$$

$$eqs : \mathbf{Action} \rightarrow \mathcal{P}(\mathbf{BinEqs})$$

$$\begin{aligned}
eqs(a) = &\bigcup \{eqs_{exit}(a,a)(\beta) \mid \beta \in B\} \cup \\
&\bigcup \{eqs_{entry}(a',a)(\beta) \mid \beta \in B' \wedge (a',B') \in m\}
\end{aligned}$$

$$\text{where } other,\ entry,\ p\ \vdash p : (B,m)$$

Table 3.11: Functions for generating reaching bindings equation templates.

of the equation templates generated for unfoldings that produce bindings. The set of types for these unfoldings include exactly one binding type of the form $unf_{exit}(a, B)$, where a is the given unfolding and B the set of binding types produced by the unfoldings non-recursive branches. The non-recursive branches arrow represents the *exit* equation templates generated by both eqs_{exit} and eqs_{entry} by recursively calling eqs_{exit} for each of the binding types in B. As mentioned in the previous sub-section, ***unfold***s enclosed in an ***unfolding*** add a mapping from the enclosing ***unfolding*** to the current bindings used to perform the ***unfold*** to the bindings map. The equation templates represented by the recursive branches arrow are thus generated by the eqs_{entry} function, when *entry* equations are generated for mappings by the eqs function.

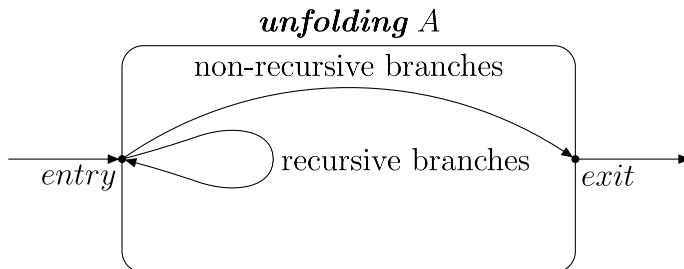


Figure 3.8: Graphical representation of the equations generated for unfoldings in general.

As a simple example, consider the while-construct of the WHILE language. It has the binding types $\{unf_{exit}(unfolding(...), \{unf_{entry}(unfolding(...))\}\}$ and the bindings map:

$$\begin{aligned}[unfolding(...) &\mapsto \{entry, exec[\![S]\!]\},\\ exec[\![S]\!] &\mapsto \{unf_{entry}(unfolding(...))\},\\ evalb[\![b]\!] &\mapsto \{unf_{entry}(unfolding(...))\}]\end{aligned}$$

Figure 3.9 illustrates the equation templates generated based on these binding types and this binding map.

Table 3.12 defines a reaching bindings analysis for the WHILE language, as a standard *kill*/*gen* data-flow analysis over the complete lattice $(\mathcal{P}(Var_*), \subseteq)$, where Var_* is the set of variables used in the program being analyzed. P_* is the program being analyzed, *stm* is a function that takes a label and gives the statement with the given label (we assume statements are uniquely labeled), *flow* is a function that takes a statement and gives the set of edges of the

statements control-flow graph, and *init* is a function that takes a statement, S, and gives the label of the statement that the entry node of the control-flow graph for S represents. The *init* and *flow* functions are defined as you would expect, so we will not go into detail. As an example, for if statements they are defined as follows:

$$init(\texttt{if}\ [\ BE\ ]^l\ \texttt{then}\ S_1\ \texttt{else}\ S_2) = l$$

$$\begin{aligned} flow(\texttt{if}\ [\ BE\ ]^l\ \texttt{then}\ S_1\ \texttt{else}\ S_2) = {} & flow(S_1) \cup flow(S_2) \\ & \cup \{(l, init(S_1)), (l, init(S_2))\} \end{aligned}$$

because the the boolean condition is executed before S_1 and S_2, followed by either S_1 or S_2.

Compared to the automatically generated reaching bindings analysis for the WHILE language, they are equivalent, in the sense that the RD_{entry} and RD_{exit} equations one would obtain by instantiating the equations given in Table 3.12, are exactly the same equations as the equations obtained from instantiating the equation templates from the generated analysis, except that the generated analysis also has *entry* equations for arithmetic and boolean expressions. The only difference is the presentation of the analysis; all of the information about the flow of bindings in the template equations for the while and if statement, and statement sequences is implicitly defined via the *flow* function in Table 3.12.

As a simple example of how to use the generated reaching bindings analysis,

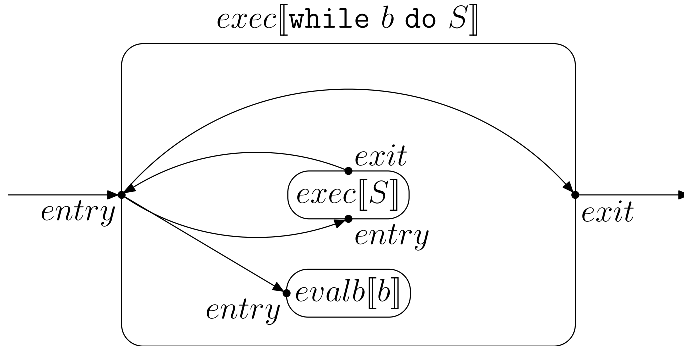


Figure 3.9: Graphical representation of the equations generated for the while-construct of the WHILE language.

consider the following WHILE program,

$$a := 10;\ \texttt{while}\ a > 1\ \texttt{do}\ a := a - 1$$

First, we recursively traverse the abstract syntax tree, instantiating the reaching bindings equation templates associated with the given abstract syntax node, for each abstract syntax node encountered. Then we add the entry equation $entry(m[\![P]\!]) = \emptyset$, where P is the source program and m the semantic function that maps programs of the source language to actions, since no tokens are bound in the initial bindings. Lastly, we determine the least fixed point of the equations. The solved equations are the result of the analysis; the solved *entry* and *exit* functions specify the tokens that might before and after each language construct is executed, respectively.

For the above example we would thus start by instantiating the reaching bindings equation templates for the statement sequence, $S_1; S_2$, where S_1 is $a := 10$ and S_2 is $\texttt{while}\ a > 1\ \texttt{do}\ a := a - 1$, giving the equations,

$$\begin{aligned} entry(exec[\![S_1]\!]) &= entry(exec[\![S_1;\ S_2]\!]) \\ entry(exec[\![S_2]\!]) &= exit(exec[\![S_1]\!]) \\ exit(exec[\![S_1;\ S_2]\!]) &= exit(exec[\![S_2]\!]) \end{aligned}$$

Continuing down the abstract syntax tree of the program, we obtain the follow-

---

$$\begin{aligned} gen_{RB}([\ I := AE\ ]^l) &= \{I\} \\ gen_{RB}(_) &= \emptyset \end{aligned}$$

$$\begin{aligned} RB_{entry}(l) &= \begin{cases} \emptyset & \text{if } l = init(P) \\ \bigcup\{RD_{exit}(l') \mid (l, l') \in flow(P_*)\} & \text{otherwise} \end{cases} \\ RB_{exit}(l) &= RD_{entry}(l) \cup gen_{RB}(stm(l)) \end{aligned}$$

Table 3.12: Reaching bindings analysis for WHILE, presented as a *kill*/*gen* dataflow analysis.

---

ing equations for the whole program:

$$\begin{aligned}
entry(exec[\![S_0]\!]) &= \emptyset \\
entry(exec[\![S_1]\!]) &= entry(exec[\![S_0]\!]) \\
entry(exec[\![S_2]\!]) &= exit(exec[\![S_1]\!]) \\
exit(exec[\![S_0]\!]) &= exit(exec[\![S_2]\!]) \\
entry(evala[\![10]\!]) &= entry(exec[\![S_1]\!]) \\
exit(exec[\![S_1]\!]) &= \{a\} \cup entry(exec[\![S_1]\!]) \\
entry(exec[\![S_5]\!]) &= exit(exec[\![a := a-1]\!]) \cup entry(exec[\![S_2]\!]) \\
entry(evalb[\![a > 1]\!]) &= entry(exec[\![S_5]\!]) \\
entry(exec[\![a := a-1]\!]) &= entry(exec[\![S_5]\!]) \\
exit(exec[\![S_5]\!]) &= entry(exec[\![S_5]\!]) \\
exit(exec[\![S_2]\!]) &= exit(exec[\![S_5]\!]) \\
entry(evala[\![a-1]\!]) &= entry(exec[\![a := a-1]\!]) \\
exit(exec[\![a := a-1]\!]) &= \{a\} \cup entry(exec[\![a := a-1]\!])
\end{aligned}$$

where S_0 is $a := 10$; `while` $a > 1$ `do` $a := a - 1$ and S_5 is ***unfolding*** ($evalb[\![a > 1]\!]$ ***then*** ...).

Solving these equations for the least fixed point, either by hand or using a set constraint solver, we find the following solution:

$$\begin{aligned}
entry(exec[\![S_0]\!]) &= \emptyset \\
entry(exec[\![S_1]\!]) &= \emptyset \\
exit(exec[\![S_1]\!]) &= \{a\} \\
entry(exec[\![S_2]\!]) &= \{a\} \\
entry(evala[\![10]\!]) &= \emptyset \\
entry(exec[\![S_5]\!]) &= \{a\} \\
entry(evalb[\![a > 1]\!]) &= \{a\} \\
entry(exec[\![a := a-1]\!]) &= \{a\} \\
exit(exec[\![S_5]\!]) &= \{a\} \\
exit(exec[\![S_1]\!]) &= \{a\} \\
exit(exec[\![S_2]\!]) &= \{a\} \\
entry(evala[\![a-1]\!]) &= \{a\} \\
exit(exec[\![a := a-1]\!]) &= \{a\}
\end{aligned}$$

which is as one would expect it to be.

Chapter 4

Discussion

In this chapter we discuss a few of the limitations of the analyses developed and discuss possible solutions.

- *Type and termination analysis*: The type and termination analysis imposes two types of restrictions on action semantic descriptions, it imposes context neutrality on semantic productions so that semantic function calls can be analyzed independently of the context in which they appear, and it imposes a few conditions on unfoldings to simplify the analysis of unfoldings.

  One of the annoying consequences of the context neutrality condition is that it prevents useful types of abstraction and reusability in action semantic descriptions. Consider, for instance, the following semantic productions in which the operand applied to two arithmetic expressions has been abstracted out of the *evala* production, instead of having an *evala* production for each operand as in the While specification:

  - $op[\![+]\!] = \boldsymbol{give}\ +$
  - $op[\![-]\!] = \boldsymbol{give}\ -$
  - $op[\![*]\!] = \boldsymbol{give}\ *$
  - $evala[\![A_1 op A_2]\!] = evala[\![A_1]\!]\ \boldsymbol{and}\ evala[\![A_2]\!]\ \boldsymbol{then}\ op[\![op]\!]$

Since the *op* productions do not satisfy the context neutrality condition, the type and termination analysis is unable to analyze this specification. In this particular case, since there is only a finite – and quite small – number of possible instantiations of the *op* production, we could solve it with a pre-processor which replaced the four productions above with three instantiated versions of the *evala* production, one for each operand.

Another solution is of course to design a type and termination analysis without the restriction of context neutrality for semantic productions. Here is an untested idea for such an analysis: We extend the type system such that a type term now has the form $\tau_0 \rightarrow \{\tau_1, ..., \tau_n\}$ where the τ_is are type terms of the second type and termination analysis, with the interpretation that given data of type τ_0 the data it gives has type τ_1 or τ_2, etc. τ_0 is further allowed to be a wildcard, i.e., *, if the type of the data it is given does not affect the performance of the given action. The types of an action is extended to be a 2-tuple where the first component is a set of the type terms described above and the second component is the set of types for the data given if the action is performed with data whose type does not match the τ_0 of any of the types given in the type set.

The types of an action is inferred by traversing it bottom-up, assigning types to the leaves of the tree directly and by merging the types of the sub-actions for action combinators. As a simple example, ***give*** + actions are assigned the types $(\{[int, int] \rightarrow \{[int]\}\}, \{exn([])\})$, and so on for the rest of the leaf actions. Similarly, for A_1 ***and*** A_2, where A_1 and A_2 both has the type $(\{* \rightarrow \{[int]\}\}, \emptyset)$ – as would be the case for arithmetic constants – A_1 ***and*** A_2 would get the type $(\{* \rightarrow \{[int, int]\}\}, \emptyset)$. However, it quickly becomes complex to the types of sub-actions for action combinators. For instance A_1 ***and*** A_2, where A_1 has the types $(\{[bool] \rightarrow \{[int, int], [bool]\}, [int] \rightarrow \{[bool], exn([])\}\}, \{exn([int])\})$ and A_2 the type $(\{[bool] \rightarrow \{[int]\}\}, \{exn([])\})$, is assigned the types $(\{[bool] \rightarrow \{[int, int, int], [bool, int]\}, [int] \rightarrow \{exn([])\}\}, \{exn([int]), exn([])\})$. Semantic function calls are handled the same way as in the original type and termination analysis, the type of function calls is given by the type environment.

- *Bindings analysis*: This analysis impose quite a lot of restrictions on the action semantic descriptions that they analyze. Some of these restrictions are necessary to allow each language construct to be analyzed independently without ending up in situations where for instance all that we can say about a given language construct is that any token might be bound in the bindings that it produces, which is useless as a reaching bindings analysis.

  In general, this problem is hard to avoid without having to resort to performing most of the analysis on concrete programs instead of languages.

A partial solution might be to allow the language designer to provide information about common combinations of language constructs or provide sample programs which could be analyzed to determine common combinations of language constructs. As a simple example, consider the WHILE language extended with the following language constructs:

- $exec[\![$`save` $I]\!] =$
  give current bindings and
  (**_provide_** I **_and give current bindings_**
  then give binding)
  then give overriding
- $exec[\![S$ `using` $I]\!] =$
  give the data bound to I **_hence_** $exec[\![S]\!]$
  then give current bindings

The first allows the current bindings to be bound to an identifier and the second allows a statement to be performed using the bindings bound to the given identifier. As discussed in the binding flow analysis section, we are unable to say anything intelligent about what tokens might be bound in the bindings used to perform S in S `using` I, if we, as we currently do, analyze each language construct independently. However, if these two constructs were commonly combined to form statements, say, of the form `save` $I; S_1$ `using` $I; S_2$ `using` I, we could analyze the denotation of this statement as a whole, which would allow us to correctly determine which bindings that would be used to perform S_1 and S_2.

Chapter 5

Implementation

This chapter briefly introduces the tool developed, which implements the analyses described. The tool is accessed through a web-interface, allowing the user to enter an action semantic description and an abstract syntax tree of a program in the given language. Upon submitting this information, all the analyses are performed on the language. If the language specification satisfied the restrictions for the binding flow analysis, a set of bindings equations is further generated for the given source program, from the results of the reaching bindings analysis. The tool can be accessed online at *http://www.kaspersv.dk/asd*.

The implementation of the analyses follows the presentation given here quite closely; for the inference systems it is to a large extend possible to identify the few lines of code corresponding to each inference rule. Each analysis has been implemented as an OCaml module. The table below identifies each of these modules.

Analysis	Module
Second type and termination analysis	ASDType
Binding flow and reaching bindings analysis	ASDBindings

Besides these modules, the code-base consists of the following modules:

- *ASDSyntax*: Defines the data-types used to represent action semantic descriptions and defines a few auxiliary functions for manipulating these.
- *ASDLexer, ASDParser*: A lexer and parser for action semantic descriptions.
- *ASTLexer, ASDParser*: A lexer and parser for a very simple language for specifying abstract syntax trees.
- *ASDPrinter, Misc*: Defines functions for pretty printing actions, types, abstract syntax trees, etc., and various common auxiliary functions.
- *CFGraph*: Defines an abstract data-structure for directed graphs, along with a few convenient functions used by the work-list algorithm, such as generating a graph with a node for each strongly connected component (SCC) in the original graph and edges for connected SCC.
- *Analysis*: Defines a signature for specifying dataflow analyses and a function for performing a dataflow analysis, using a standard work-list algorithm.
- *ASDCFG*: Defines a data-structure for constructing Language Construct Graphs (LCGs) and a function for generating a LCG from a semantic description.
- *ASD*: Defines a number of functions for performing elementary sanity-checks on ASDs.

The web-interface is also written in OCaml as a CGI module which plugs directly into the Apache web-server. Besides OCaml, the tool has the following external dependencies,

- *Menhir*: A more advanced parser generator for OCaml than the standard ocamlyacc.
- *OcamlGraph*: An OCaml library which defines a number of useful data-structures for graphs, along with lots of useful algorithms, such as traversing the nodes of a graph in topological order and for computing the strongly connected components of a graph.
- *Apache, mod_caml*: A web-server and an OCaml interface for writing CGI scripts for this webserver.
- *Graphviz*: A graph layout program, used to create a graphical representation of the results of the type and termination analysis.

Chapter 6

Conclusion

Previous action semantics based compiler generators [eg., 3, 14] compile programs by first converting them to actions, which are then compiled with a generic action compiler. In this thesis we have explored the idea of analyzing action semantic descriptions, with the purpose of generating language specific action compilers optimized for the given language. Two analyses have been developed, a type and termination analysis, and an analysis and accompanying algorithm for generating a reaching bindings analysis for the source language.

Our approach to analyzing action semantic descriptions is based on the idea of analyzing each language construct independently of the context in which it is used. The two analyses developed using this approach works very well for the While language. However, for both analyses it was necessary to impose restrictions on action semantic descriptions, to ensure that the analyses are able to analyze language constructs independently, without becoming unacceptably imprecise. Unfortunately, some languages of interest do not satisfy these restrictions.

In the case of the While language, the type and termination analysis is as precise as possible, given the type system used, and the generated reaching bindings analysis is equivalent in precision to the reaching bindings data-flow analysis for the While language, derived by hand.

APPENDIX A

Semantics of WHILE and Action Notation

A.1 The WHILE langauge

$$
\begin{aligned}
AExp &::= x \mid n \mid ae_1 + ae_2 \\
BExp &::= \texttt{true} \mid \texttt{false} \mid \texttt{not}\ be \mid ae_1 = ae_2 \\
Stm &::= x \mid \texttt{skip} \mid s_1; s_2 \mid \texttt{if}\ be\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \mid \texttt{while}\ be\ \texttt{do}\ s
\end{aligned}
$$

(WEA-CONST) $$e \vdash_a n \rightarrow n$$

(WEA-ADD) $$\frac{e \vdash_a ae_1 \rightarrow n_1, \quad e \vdash_a ae_2 \rightarrow n_2, \quad n = n_1 + n_2}{e \vdash_a ae_1 + ae_2 \rightarrow n}$$

(WEA-IDENT) $$\frac{e(x) = n}{e \vdash_a x \rightarrow n}$$

(WEB-TRUE) $$e \vdash_b \texttt{true} \rightarrow true$$

(WEB-False) $$e \vdash_b \texttt{false} \rightarrow \mathit{false}$$

(WEB-Eq-T) $$\frac{e \vdash_a ae_1 \rightarrow n_1, \quad e \vdash_a ae_2 \rightarrow n_2, \quad n_1 = n_2}{e \vdash_b ae_1 = ae_2 \rightarrow \mathit{true}}$$

(WEB-Eq-F) $$\frac{e \vdash_a ae_1 \rightarrow n_1, \quad e \vdash_a ae_2 \rightarrow n_2, \quad n_1 \neq n_2}{e \vdash_b ae_1 = ae_2 \rightarrow \mathit{false}}$$

(WE-Assign) $$\frac{e \vdash_a ae \rightarrow n, \quad e' = e[x \rightarrow n]}{e \vdash_w x := ae \rightarrow e'}$$

(WE-Skip) $$e \vdash_w \texttt{skip} \rightarrow e$$

(WE-Seq) $$\frac{e \vdash_w s_1 \rightarrow e', \quad e' \vdash_w s_2 \rightarrow e''}{e \vdash_w s_1;\ s_2 \rightarrow e''}$$

(WE-If-T) $$\frac{e \vdash_b be \rightarrow \mathit{true}, \quad e \vdash_w s_1 \rightarrow e'}{e \vdash_w \texttt{if}\ be\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \rightarrow e'}$$

(WE-If-F) $$\frac{e \vdash_b be \rightarrow \mathit{false}, \quad e \vdash_w s_2 \rightarrow e'}{e \vdash_w \texttt{if}\ be\ \texttt{then}\ s_1\ \texttt{else}\ s_2 \rightarrow e'}$$

(WE-While-T) $$\frac{\begin{array}{c} e \vdash_b be \rightarrow \mathit{true}, \quad e \vdash_w s \rightarrow e', \\ e' \vdash_w \texttt{while}\ be\ \texttt{do}\ s \rightarrow e'' \end{array}}{e \vdash_w \texttt{while}\ be\ \texttt{do}\ s \rightarrow e''}$$

(WE-While-F) $$\frac{e \vdash_b be \rightarrow \mathit{false}}{e \vdash_w \texttt{while}\ be\ \texttt{do}\ s \rightarrow e}$$

A.2 The Action Notation language

The predicates *bindable* and *storable* are true if and only if the given datum is bindable or storable, respectively.

(AE-Const) $$(a,\ \delta,\ \xi,\ \mu) \vdash \boldsymbol{provide}\ d \rightarrow (\mathit{normal}\ d,\ \mu)$$

(AE-Copy) $$(a,\ \delta,\ \xi,\ \mu) \vdash \boldsymbol{copy} \rightarrow (\mathit{normal}\ \delta,\ \mu)$$

(AE-Give1) $$\frac{@ \in \{+,-,*\}, \quad \delta = (int\ n_1, int\ n_2), \quad n = n_1 \ @ \ n_2}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{give}}\ @ \rightarrow (normal\ (int\ n),\ \mu)}$$

(AE-Give2) $$\frac{@ \in \{+,-,*\}, \quad \delta \neq (int\ n_1, int\ n_2)}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{give}}\ @ \rightarrow (exceptional\ (),\ \mu)}$$

(AE-Give3) $$\frac{\delta = (token\ t, d), \quad bindable(d), \quad \xi' = [t \mapsto d]}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{give binding}} \rightarrow (normal\ (bindings\ \xi'),\ \mu)}$$

(AE-Give4) $$\frac{\delta \neq (token\ t, d) \vee \neg bindable(d)}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{give binding}} \rightarrow (exceptional\ (),\ \mu)}$$

(AE-Give5) $$\frac{\delta = (bindings\ b, bindings\ b'), \quad \xi' = b[b']}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{give overriding}} \rightarrow (normal\ \xi',\ \mu)}$$

(AE-Give6) $$\frac{\delta \neq (bindings\ b, bindings\ b')}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{give overriding}} \rightarrow (exceptional\ (),\ \mu)}$$

(AE-Give7) $$\frac{\delta = (bindings\ b, token\ t), \quad d = b(t)}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{give bound}} \rightarrow (normal\ (d),\ \mu)}$$

(AE-Give8) $$\frac{\delta \neq (bindings\ b, token\ t) \vee b(t) = undef}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{give bound}} \rightarrow (exceptional\ (),\ \mu)}$$

(AE-Give9) $$(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{give the data}} \rightarrow (normal\ \delta,\ \mu)$$

(AE-Give10) $$\frac{\delta = (nat\ n)}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{give the nat}} \rightarrow (normal\ \delta,\ \mu)}$$

(AE-Give11) $$\frac{\delta \neq (nat\ n)}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{give the nat}} \rightarrow (exceptional\ (),\ \mu)}$$

(AE-Give12) $$\frac{\delta = (bool\ b)}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{give the bool}} \rightarrow (normal\ \delta,\ \mu)}$$

(AE-Give13) $$\frac{\delta \neq (bool\ b)}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{give the bool}} \rightarrow (exceptional\ (),\ \mu)}$$

(AE-Give14)
$$\frac{\delta = (\mathit{bindings}\ b)}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{give the bindings}} \rightarrow (\mathit{normal}\ \delta,\ \mu)}$$

(AE-Give15)
$$\frac{\delta \neq (\mathit{bindings}\ b)}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{give the bindings}} \rightarrow (\mathit{exceptional}\ (),\ \mu)}$$

(AE-Give16)
$$\frac{\delta = (\mathit{cell}\ c)}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{give the cell}} \rightarrow (\mathit{normal}\ \delta,\ \mu)}$$

(AE-Give17)
$$\frac{\delta \neq (\mathit{cell}\ c)}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{give the cell}} \rightarrow (\mathit{exceptional}\ (),\ \mu)}$$

(AE-Check1)
$$\frac{\delta = (d_1, d_2), \quad d_1 = d_2}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{check}}\ = \rightarrow (\mathit{normal}\ (),\ \mu)}$$

(AE-Check2)
$$\frac{\delta \neq (d_1, d_2) \vee d_1 \neq d_2}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{check}}\ = \rightarrow (\mathit{exceptional}\ (),\ \mu)}$$

(AE-Check3)
$$\frac{\delta = (\mathit{int}\ n_1, \mathit{int}\ n_2), \quad n_1 > n_2}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{check}}\ > \rightarrow (\mathit{normal}\ (),\ \mu)}$$

(AE-Check4)
$$\frac{\delta \neq (\mathit{int}\ n_1, \mathit{int}\ n_2) \vee n_1 \not> n_2}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{check}}\ > \rightarrow (\mathit{exceptional}\ (),\ \mu)}$$

(AE-Then1)
$$\frac{\begin{array}{c}(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1 \rightarrow (\mathit{normal}\ v_1,\ \mu'),\\ (v_1,\ \xi,\ \mu',\ A_2)\ \vdash \rightarrow (\delta'',\ \mu'')\end{array}}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1\ \textbf{\textit{then}}\ A_2 \rightarrow (\delta'',\ \mu'')}$$

(AE-Then2)
$$\frac{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1 \rightarrow (\mathit{exceptional}\ v,\ \mu')}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1\ \textbf{\textit{then}}\ A_2 \rightarrow (\mathit{exceptional}\ v,\ \mu')}$$

(AE-Then3)
$$\frac{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1 \rightarrow (\mathit{failed},\ \mu')}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1\ \textbf{\textit{then}}\ A_2 \rightarrow (\mathit{failed},\ \mu')}$$

(AE-And1)
$$\frac{\begin{array}{c}(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1 \rightarrow (\mathit{normal}\ v_1,\ \mu'),\\ (a,\ v_1,\ \xi,\ \mu')\ \vdash\ A_2 \rightarrow (\mathit{normal}\ v_2,\ \mu'')\end{array}}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1\ \textbf{\textit{and}}\ A_2 \rightarrow (\mathit{normal}\ v_1\ @\ v_2,\ \mu'')}$$

$$\text{(AE-AND2)} \quad \frac{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1 \rightarrow (\mathit{normal}\ v_1,\ \mu'),\quad (a,\ v_1,\ \xi,\ \mu')\ \vdash\ A_2 \rightarrow (\mathit{exceptional}\ v_2,\ \mu'')}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1\ \textbf{\textit{and}}\ A_2 \rightarrow (\mathit{exceptional}\ v_2,\ \mu'')}$$

$$\text{(AE-AND3)} \quad \frac{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1 \rightarrow (\mathit{normal}\ v_1,\ \mu'),\quad (a,\ v_1,\ \xi,\ \mu')\ \vdash\ A_2 \rightarrow (\mathit{failed},\ \mu'')}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1\ \textbf{\textit{and}}\ A_2 \rightarrow (\mathit{failed},\ \mu'')}$$

$$\text{(AE-AND4)} \quad \frac{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1 \rightarrow (\mathit{exceptional}\ v,\ \mu')}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1\ \textbf{\textit{and}}\ A_2 \rightarrow (\mathit{exceptional}\ v,\ \mu')}$$

$$\text{(AE-AND5)} \quad \frac{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1 \rightarrow (\mathit{failed},\ \mu')}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1\ \textbf{\textit{and}}\ A_2 \rightarrow (\mathit{failed},\ \mu')}$$

$$\text{(AE-EXCEP1)} \quad \frac{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1 \rightarrow (\mathit{normal}\ v,\ \mu')}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1\ \textbf{\textit{exceptionally}}\ A_2 \rightarrow (\mathit{normal}\ v,\ \xi')\mu'}$$

$$\text{(AE-EXCEP2)} \quad \frac{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1 \rightarrow (\mathit{exceptional}\ v,\ \mu'),\quad (v,\ \mu',\ A_2,\ \rightarrow)\ \vdash\ (\delta'',\ \mu'')}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1\ \textbf{\textit{exceptionally}}\ A_2 \rightarrow (\delta'',\ \mu'')}$$

$$\text{(AE-EXCEP3)} \quad \frac{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1 \rightarrow (\mathit{failed},\ \mu')}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1\ \textbf{\textit{exceptionally}}\ A_2 \rightarrow (F,\ \mu')}$$

$$\text{(AE-RAISE)} \quad (a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{raise}} \rightarrow (\mathit{exceptional}\ \delta,\ \mu)$$

$$\text{(AE-FAIL)} \quad (a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{fail}} \rightarrow (\mathit{failed},\ \mu)$$

$$\text{(AE-OTHER1)} \quad \frac{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1 \rightarrow (\mathit{normal}\ v,\ \mu')}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1\ \textbf{\textit{otherwise}}\ A_2 \rightarrow (\mathit{normal}\ v,\ \mu')}$$

$$\text{(AE-OTHER2)} \quad \frac{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1 \rightarrow (\mathit{exceptional}\ v,\ \mu')}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1\ \textbf{\textit{otherwise}}\ A_2 \rightarrow (\mathit{exceptional}\ v,\ \mu')}$$

$$\text{(AE-OTHER3)} \quad \frac{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1 \rightarrow (\mathit{failed},\ \mu'),\quad (a,\ \delta,\ \xi,\ \mu')\ \vdash\ A_2 \rightarrow (\delta'',\ \mu'')}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ A_1\ \textbf{\textit{otherwise}}\ A_2 \rightarrow (\delta'',\ \mu'')}$$

(AE-CurBin) $(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{give current bindings}} \rightarrow (normal\ \xi,\ \mu)$

(AE-Cre1) $$\frac{\delta = (d), \quad storable(d), \quad cell\ c, \quad \mu(c) = undef}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{create}} \rightarrow (normal\ c,\ \mu[c \rightarrow d])}$$

(AE-Cre2) $$\frac{\delta \neq (d) \vee \neg storable(d)}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{create}} \rightarrow (exceptional\ (),\ \mu)}$$

(AE-Up1) $$\frac{\delta = (cell\ c, d), \quad storable(d)}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{update}} \rightarrow (normal\ (),\ \mu[c \rightarrow d])}$$

(AE-Up2) $$\frac{\delta \neq (cell\ c, d) \vee \neg storable(d)}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{update}} \rightarrow (exceptional\ (),\ \mu)}$$

(AE-Ins1) $$\frac{\delta = cell\ c, \quad d = \mu(c)}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{inspect}} \rightarrow (normal\ (d),\ \mu)}$$

(AE-Ins2) $$\frac{\delta \neq cell\ c}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{inspect}} \rightarrow (exceptional\ (),\ \mu)}$$

(AE-Unfing) $$\frac{(unfolding\ A,\ \delta,\ \xi,\ \mu)\ \vdash\ A \rightarrow (\delta',\ \mu')}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{unfolding}}\ A \rightarrow (\delta',\ \mu')}$$

(AE-Unfold) $$\frac{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ a \rightarrow (\delta',\ \mu')}{(a,\ \delta,\ \xi,\ \mu)\ \vdash\ \textbf{\textit{unfold}} \rightarrow (\delta',\ \mu')}$$

APPENDIX B

Action Semantic Description for the WHILE language

Abstract Syntax

- $ArithExp ::= Ident \mid Numeral \mid ArithExp + ArithExp$
  $ArithExp - ArithExp \mid ArithExp * ArithExp$
- $BoolExp ::= \texttt{true} \mid \texttt{false} \mid \texttt{not}\ BoolExp$
  $ArithExp = ArithExp \mid ArithExp > ArithExp$
- $Stm ::= Ident\ :=\ ArithExp \mid \texttt{skip} \mid Stm;\ Stm$
  $\texttt{if}\ BoolExp\ \texttt{then}\ Stm\ \texttt{else}\ Stm$
  $\texttt{while}\ BoolExp\ \texttt{do}\ Stm$

Semantic Functions

- $evala[\![_]\!] : ArithExp \rightarrow Action$
- $evalb[\![_]\!] : BoolExp \rightarrow Action$
- $exec[\![_]\!] : Stm \rightarrow Action$

Semantic Entities

- $datum ::= integer \mid boolean$
- $bindable ::= cell$
- $storable ::= integer$

Semantic Equations

$AE, AE_1, AE_2 : ArithExp; BE : BoolExp; I : Ident; N : Numeral; S, S_1, S_2 : Stm;$

Arithmetic Expressions

- $evala[\![I]\!]$ = ***give the cell bound to*** I ***then inspect***
- $evala[\![N]\!]$ = ***provide*** N
- $evala[\![AE_1 + AE_2]\!] = evala[\![AE_1]\!]$ ***and*** $evala[\![AE_2]\!]$ ***then give*** $+$
- $evala[\![AE_1 - AE_2]\!] = evala[\![AE_1]\!]$ ***and*** $evala[\![AE_2]\!]$ ***then give*** $-$
- $evala[\![AE_1 * AE_2]\!] = evala[\![AE_1]\!]$ ***and*** $evala[\![AE_2]\!]$ ***then give*** $*$

Boolean Expressions

- $eval_b[\![\texttt{true}]\!]$ = ***provide*** $true$
- $eval_b[\![\texttt{false}]\!]$ = ***provide*** $false$
- $eval_b[\![\texttt{not}\ BE]\!] = eval_b[\![BE]\!]$ ***then*** ((***given*** $true$ ***then provide*** $false$)
  otherwise provide $true$)
- $evalb[\![AE_1 = AE_2]\!]$ =
  $eval_a[\![AE_1]\!]$ ***and*** $eval_a[\![AE_2]\!]$ ***then*** (***check*** $=$
  then provide $true$ ***exceptionally provide*** $false$)
- $evalb[\![AE_1 > AE_2]\!]$ =
  $eval_a[\![AE_1]\!]$ ***and*** $eval_a[\![AE_2]\!]$ ***then*** (***check*** $>$
  then provide $true$ ***exceptionally provide*** $false$)

Statements

- $exec[\![I\ :=\ AE]\!] =$
  $evala[\![AE]\!]$ ***then*** (
  (***give the cell bound to*** I ***and copy then update***
  then give current bindings)
  exceptionally
  (***give current bindings and*** (***provide*** I ***and*** ($evala[\![AE]\!]$
  then create) ***then give binding***) ***then give overriding***))

- $exec[\![\texttt{skip}]\!] =$ ***give current bindings***

- $exec[\![S_1;\ S_2]\!] = exec[\![S_1]\!]$ ***hence*** $exec[\![S_2]\!]$

- $exec[\![\texttt{if}\ BE\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] =$
  $eval_b[\![BE]\!]$ ***then*** (
  (***give*** $true$ ***then*** $exec[\![S_1]\!]$)
  otherwise $exec[\![S_2]\!]$)

- $exec[\![\texttt{while}\ BE\ \texttt{do}\ S]\!] =$
  unfolding (
  $eval_b[\![BE]\!]$ ***then*** (
  (***given*** $true$ ***then*** $exec[\![S]\!]$ ***hence unfold***)
  otherwise give current bindings))

Bibliography

[1] Ole Agesen. The Cartesian Product Algorithm. In *ECOOP'95 Conference Proceedings*. Springer-Verlag, 1995.

[2] Yves Bertot and Pierre Castéran. *Coq'Art: Interactive Theorem Proving and Program Development*. Texts in Theoretical Computer Science. Springer, 2004.

[3] Deryck F. Brown, Hermano Moura, and David A. Watt. ACTRESS: an Action Semantics Directed Compiler Generator. In R. Heldal, C. K. Holst, and P. L. Wadler, editors, *Functional Programming, Glasgow 1991: Proceedings of the 1991 Workshop, Portree, UK*, pages 51–55. Springer-Verlag, 1992.

[4] The Coq proof assistant website. http://coq.inria.fr/.

[5] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, second edition, 2002.

[6] Kyung-Goo Doh. Action Semantics: A Tool for Developing Programming Languages. Technical report, The University of Aizu, 1993.

[7] Søren B. Lassen, Peter D. Mosses, and David A. Watt. AN-2: Revised Action Notation: Informal Summary. Draft Version 0.7.4, September 2000.

[8] Peter D. Mosses. *Action Semantics*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.

[9] Peter D. Mosses. Theory and Practice of Action Semantics. Technical report, University of Aarhus, 1996.

[10] Peter D. Mosses. AN-2: Revised Action Notation: Syntax and Semantics. Draft Version 0.7.5, March 2001.

[11] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 2005.

[12] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[13] Tijs van der Storm. Implementing Actions. Master's thesis, Universiteit van Amsterdam, 2003.

[14] Peter Ørbæk. OASIS: An Optimizing Action-based Compiler Generator. In Peter Fritzon, editor, *Proceedings of the 1994 Conference on Compiler Construction*, pages 1–15. Springer-Verlag, April 1994.