Master thesis

Redesigning the language for business logic in the Maconomy ERP system and automatic translation of existing code

Author:
Piotr Borowian
(s091441)

Supervisors:
Ekkart Kindler
Christian W. Probst
Martin Gamwell Dawids
Rune Glerup

DTU Informatics
Department of Informatics and Mathematical Modelling

Kongens Lyngby 2012
IMM-M.Sc.-2012-79

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 IMM-M.Sc.-2012-79

Summary

Maconomy Scripting Language (*MSL*) is a Domain Specific Language (*DSL*) for expressing business logic in the *Maconomy ERP System*, developed by Maconomy [1]. For various reasons, which are discussed in Chapter 1, Deltek has decided to investigate the possibility of replacing MSL by Scala as a new business logic development language.

This thesis defines a subset of MSL, called *MSL core*, that comprises the core features of the language. It then introduces a number of extensions to Scala, which altogether make up MScala − a proposed replacement for MSL core.

In the course of this thesis we argue that MScala allows for expressing business logic in Maconomy at the same or higher level of abstraction than MSL core. It means that, in most cases, the same functionality can be expressed in a more succinct and elegant manner in MScala than in MSL, but very rarely, if at all, the other way around. Moreover, we show that Scala is well-suited for embedding domain specific languages. It allows domain specialists (Maconomy business logic programmers for that matter) to define libraries that look and feel like built-in language constructs. This lightweight way of embedding DSLs in Scala makes it much easier to gradually abstract business logic concepts in Maconomy from technical artifacts so that business problems can be solved at the right level of abstraction. Finally, we provide a prototype MSL core to MScala translator along with a precise description of the correspondence between particular MSL core and MScala constructs.

In addition to that, this thesis defines a clear and scalable architecture of a source to source translator, based on state-of-the-art concepts and technologies

like attribute grammars [2] and rewrite rules [3]. The proposed architecture, which the MSL to Scala prototype translator is based on, allows for building composable translation phases out of composable translation rules and further for combining the translation phases to define the actual translation. This architecture is extensible across two axes: it enables adding new source language features as well as new translation phases, which can implement optimizations leading to more idiomatic target code.

To sum it up, this thesis provides a proof-of-concept prototype of an MSL to Scala translator along with a well-grounded rationale of why it would be sensible to migrate the Maconomy MSL code base into Scala.

Acknowledgements

First and foremost, I would like to thank my supervisor Ekkart Kindler for his invaluable guidance, constant support and encouragement for the last two years. We have had many fruitful discussions and your feedback has always been helpful, insightful and right to the point. I would also like to thank Christian W. Probst for co-supervising this thesis and for his valuable feedback on my work.

It has been a true pleasure and honor to work on this thesis with Martin Gamwell Dawids and Rune Glerup – my amazing supervisors at Deltek. The discussions we had have always been insightful, enriching and helped me tremendously to complete this thesis. Thank you very much for your great support in the last days of the project, when I needed it most. Martin deserves special credit for being my personal LaTeX consultant – your help was invaluable, thank you!

I would like to express my gratitude to Anne Hellung-Larsen, my former manager at Deltek, without whom this thesis would not have been possible. Thank you very much for supporting me in all of my initiatives.

I wish to thank Vaidas Karosas and Nathaniel Bo Jensen for taking their time to proof-read this thesis, which significantly reduced the number of typos and unclear statements in it.

Finally, I can never thank enough my parents for their love and constant support during my entire life.

Contents

Translations

Chapter 1

Introduction

This thesis defines MScala – a new language for expressing business logic in the Maconomy ERP system and provides a proof-of-concept prototype of the Maconomy Scripting Language to MScala translator.

In this chapter we describe the motivation for the thesis as well as its goals and objectives. Further, we define the scope of the thesis, its structure and main contributions.

1.1 Motivation

In the late 1980s Maconomy, a Danish software company acquired by Deltek in 2010, started to develop an enterprise resource planning (*ERP*) solution for professional services organizations [1]. To increase the productivity of application programmers as well as to ensure a sound, extensible architecture, the Maconomy Scripting Language (*MSL*) was introduced. MSL is a domain specific language (*DSL*) tailored specifically to the business logic development in the Maconomy system. At its core MSL is a simple procedural language with syntax and semantics similar to Pascal and Ada. It also incorporates some domain specific extensions like custom data types, type-safe database queries and data manipulation statements as well as automatic transaction management.

At the time of its creation MSL gave a real competitive edge to Maconomy. Most notably, it had included language integrated type-safe queries 20 years before they were incorporated into mainstream languages like LINQ in the .NET platform [4]. For the last 20 years, however, the IT industry has been experiencing a vary fast pace of innovation, putting enormous amounts of effort, resources and brainpower into programming languages development, including tools, frameworks, integrated development environments (*IDEs*) etc.

Companies like Deltek, whose main objective is to provide customers with software solutions empowering their businesses as opposed to developing technologies for their own sake, are finding it less and less profitable to develop and maintain their own complex programming languages in-house. Having full control over language development has benefits on its own: the company is independent from any third-party vendors, can evolve the language as needed, extend it, migrate it to new platforms and so on. But these are not the main business reasons for creating and maintaining a DSL in the first place. Most importantly, DSLs make the developers more productive and efficient in bringing new features to the market. Nowadays, however, there are plenty of very powerful and expressive general purpose programming languages available that come along with rich frameworks, libraries and a great tool support. These languages are often open-source and powerful enough to express business logic concepts in an ERP system like Maconomy at a higher level of abstraction than a language like MSL. Another invaluable aspect is a high quality tool support, which can boost the developers' productivity to a whole new level.

One of these modern languages is Scala [5, 6] - a fusion of object oriented and functional paradigms, running on top of the Java Virtual Machine (*JVM*). What makes Scala particularly interesting with regard to replacing an external DSL like MSL is that it is a very good host language for embedding internal DSLs. [7]. It supports shallow embedding in the form of pure library-based DSLs, good examples being actors [8], parser combinators [9] and testing frameworks. Ongoing projects, such as Scala-Virtualized [7] and Scala Macros [10], enable deep embedding of DSLs as well, which is when the DSL implementor has access to an abstract syntax tree (*AST*) representation of a program that is amenable to analysis and optimization.

For these and other reasons Deltek has decided to investigate the possibility of replacing MSL with an internal Scala DSL. This thesis addresses that challenge by providing a proof-of-concept prototype of an MSL to Scala translator. The next section describes the main goals and objectives of the thesis.

1.2 Objectives

The primary objective of this thesis, set right at the beginning of the project, was to investigate whether an internal Scala DSL could make up a good replacement for MSL. The definition of "good" is in this case at least two-fold. First of all, we require Scala to be flexible enough to host a DSL that can express business logic in Maconomy in a more succinct, concise and elegant manner compared to MSL, when new functionality is implemented. Secondly, it should be possible to automatically translate the existing MSL code to the new Scala DSL and furthermore – the target Scala code should be at least as concise and operate on at least the same level of abstraction as the source MSL code. The satisfiability of this requirement does not necessarily follow from the first one, since the conceptual gap between a procedural language like MSL and a hybrid of object oriented and functional paradigms like Scala is rather substantial.

The second objective was to provide a proof-of-concept prototype of an MSL to Scala translator that would be based on a clear and extensible architecture. The architecture should allow for plugging in new transformations that would either extend the number of MSL features covered or lead to more idiomatic Scala code than merely the result of a straightforward translation.

1.3 Scope

Migrating a huge legacy code base to a quite different language is anything but trivial. Semantic Designs, a company that has been developing its own technologies for performing such automatic migrations for more than 15 years and has a number of predefined front and back ends, estimated that a typical migration project involving several people takes them around 9–18 months [11]. Designing a usable DSL is not trivial either and surely consumes some time.

That being said, in order to achieve the described objectives within the given time frame, certain trade-offs had to be made. First of all, we decided to choose a subset of MSL features, hereafter called *MSL core*, that are at the core of the language. Then we designed and implemented a library-based DSL in Scala, hereafter called *MScala*, that is capable of expressing the same concepts as MSL core and is a suitable target language for automatic translation. An MSL core to MScala prototype translator has been implemented, with the focus on defining a clear, extensible and reliable architecture rather than testing thoroughly every single transformation.

1.4 Structure and main contributions

The remainder of this thesis is structured as follows. Chapter 2 defines MSL core and justifies the choice of the language features included.

Chapter 3 elaborates on Scala as a host language for embedding DSLs. It points out, in particular, that a domain specialist knowing Scala can relatively easily implement an internal Scala DSL that looks and feels as if it consisted of native language constructs. Therefore, once the MSL code base is converted to MScala, the Maconomy application programmers will have means to refactor the exsisting code and to gradually raise the level of abstraction of the language constructs used to solve problems in the Maconomy domain.

In Chapter 4 we define MScala, which is Scala along with the minimal number of extensions that enable expressing the same concepts as MSL core. By presenting several examples, we show that the same implementation tasks, common for the Maconomy system, can be expressed in a much more succinct manner in MScala than in MSL.

Chapter 5 describes in detail how to carry out an automatic MSL core to MScala translation. It covers both straightforward translation as well as some optimizations that result in semantically equivalent but more idiomatic Scala code. The proposed translations generally produce Scala code of similar or higher conciseness than the source MSL code. The translations described in this chapter have been implemented in the prototype MSL core to MScala translator, which is delivered with this thesis (see Appendix B).

In Chapter 6 we define a clear and scalable architecture of a source to source translator, based on state-of-the-art concepts and technologies like attribute grammars [2] and rewrite rules [3]. The proposed architecture, which the MSL to Scala prototype translator is based on, allows for building composable translation phases out of composable translation rules and for combining the translation phases to define the actual translation. This architecture is extensible across two axes: it enables adding new source language features as well as new translation phases, which can implement optimizations leading to more idiomatic target code.

In Chapter 7 we evaluate the work that has been done and, based on its results, provide a well-grounded rationale of why it is sensible to migrate MSL code base into Scala, despite possible risks and certain disadvantages.

Finally, Chapter 8 concludes the thesis.

Chapter 2

MSL core

In this chapter we briefly describe MSL and the role it plays in the Maconomy system. Moreover, we define *MSL core* – a subset of MSL comprising the core language constructs, which is a subject for the automatic code translation to MScala. Chapter 5, that specifies the translation, further describes the MSL core constructs in much more detail.

2.1 Maconomy system

Maconomy is an Enterprise Resource Planning solution for professional services organizations, such as consulting and audit agencies, legal services and scientific research institutions. It focuses on supporting business processes in such organizations and, to this end, performs complex data analysis that helps in decision making, strategy setting, keeping track of project progress etc. It targets middle size as well as large companies with thousands of employees.

The Maconomy system must therefore address at least two kinds of challenges. One is handling application business logic, i.e., providing the functionality that customers expect. The other kind has to do with all the architectural and technological challenges in enabling these functionalities for a large number of client applications, working concurrently in a distributed, client/server environment.

To make the development of such a system easier, as well as its architecture more extensible, scalable and maintainable, there is a clear distinction between these two classes of problems in Maconomy. To this end, several domain specific languages have been introduced so that the application programmers focus on solving problems in the business domain, rather than dealing with all of the surrounding technical artifacts. Some of these languages target high level database entities definition, some other UI layout specifications that are then rendered by a generic client. Finally, the Maconomy Scripting Language (*MSL*) is a language tailored specifically to business logic development in Maconomy.

The overall architecture of the Maconomy system is shown in Figure 2.1.

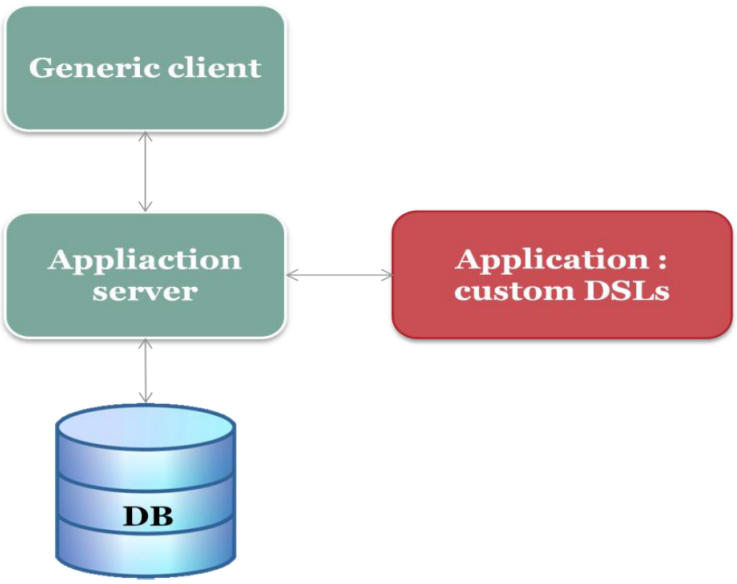

Figure 2.1: Overall architecture of the Maconomy system

2.2 MSL overview

MSL is a procedural, imperative language with syntax and semantics similar to Pascal or Ada. It is statically typed and supports both *reference* and *value types*. All the types in an MSL are value types, unless passed to a function or procedure by reference. Value types are copied upon assignment so that every variable references its own value. In contrast, two variables of reference type

can be bound to the same value in memory, so that when one of these variables changes the value, the change is visible to the other variable as well.

Listing 2.1 shows an example of an MSL function `SubString` that takes a string value (passed by reference for efficiency reasons) and two integer indexes as parameters and returns a substring of the given string specified by the beginning and end indexes.

Listing 2.1: SubString: an example MSL function

```
1  function SubString(var StrPar  : String;
2                         FromPar : Integer;
3                         ToPar   : Integer) : String is
4  var
5    Ivar    : Integer;
6    StrVar  : String;
7  begin
8    Ivar := FromPar;
9    StrVar := "";
10   while  Ivar <= ToPar and Ivar <= StringLength(StrPar) do
11     StrVar := ConcatString(StrVar,Char'Image(StrPar[Ivar]));
12     Ivar   := Ivar + 1;
13   end while;
14   return StrVar;
15 end function;
```

Lines 4–6 specify the variable definition block, as in MSL all the variables must be defined in one place at the beginning of a function or procedure. The `begin` and `end function` keywords specify the statement block, where the actual business logic is implemented. MSL provides a set of standard procedural statements, which are shown in Listing 2.2.

Individual functions can be grouped into modules, that are simply containers (namespaces) for functions and procedures.

2.2.1 Domain specific features of MSL

Besides the standard general purpose constructs of MSL described in the previous paragraph, the language provides some domain specific features as well. This section describes them briefly.

Listing 2.2: MSL statements

```
-- notation
-- [ <x> ]*  => <x> occurs zero or more times
-- [ <x> ]?  => <x> is optional

-- assignment
<variable> := <expr>

-- if then else
if <Boolean expression> then
  <Statement list>
[ elsif <Boolean expression> then
   <Statement list> ]*
[ else
  <Statement list> ]?
end if

-- while
while <Boolean expression> do
  <Statement list>
end while

-- repeat until
repeat
  <Statement list>
until <Boolean expression>

-- call
<Procedure_name> [ ( [ <parameter list> ]? ) ]?

-- return
return <Expression>

-- case statement
case <Expression> of
 [ <value> :  begin <statement list>  end; ]*
end case

-- raise statement
raise

-- break statement
break
```

2.2.1.1 Cursor

One of the most important domain specific extensions to MSL is its support for type-safe database queries and data manipulation statements. An MSL *cursor* is a means of defining type-safe SQL-like queries. Once a query is executed, the defining cursor can be used to access the *result set* of the query, as well as individual *records* comprising the result set.

An example of a simple cursor definition as well as its use can be seen in Listing 2.3. MSL cursors are described in Chapter 4 that, among other things, presents idiomatic MScala solutions to database related problems. Section 5.1.1.5 goes to even more detail in describing MSL cursors, as it defines how their particular features get auto-translated to MScala.

Listing 2.3: MSL cursor as a query definition and a current record

```
1   -- query definition
2   read cursor UniStudent is
3   select all from UniStudent
4   order by Name;
5
6   -- here the query is executed and UniStudent denotes the result set
7   for all UniStudent do
8   -- and here UniStudent denotes the current record
9     if UniStudent.Name = "Piotr" then
10      return true;
11    end if;
12  end for;
```

2.2.1.2 Dialog state machine

The user interface of Maconomy is composed of *dialogs* that are simply forms nd tables presenting data in a uniform manner. Every dialog has an underlying database relation – its primary source of data. Moreover, dialogs provide a set of standard actions, such as creating a new entity (e.g. the dialog *Customers* would create a new customer in the database), updating and deleting it. Besides these standard actions, dialog-specific actions can be defined and plugged into the UI in a uniform way. Figure 2.2 shows an example workspace in Maconomy with a *Fee forecast* dialog opened, where available actions are marked.

The set of standard actions that can be performed on any dialog, as well as

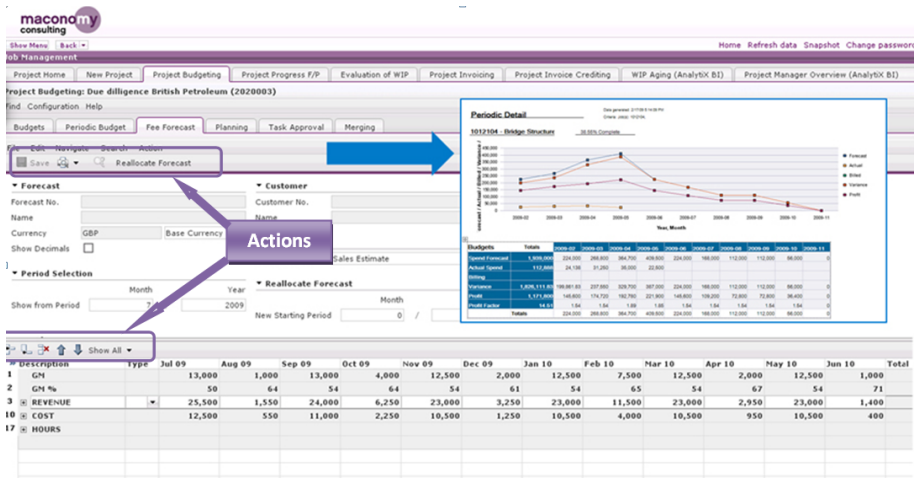

Figure 2.2: Example workspace in Maconomy: project progress reporting to compare with actual budget

the uniform way of plugging in new actions, are handled by the *dialog state machine*. It is a state machine that enforces a valid order of executing actions, e.g., it does not make sense to update an entry before creating it. Moreover, it defines an interface for plugging in new actions. All the actions are declared in declarative DSLs and their semantics, i.e., the business logic – is defined in terms of MSL code. On top of that, the dialog state machine implicitly manages database transactions, so that the application programmers can focus on solving actual problems in the business domain.

2.2.1.3 Domain specific data types

MSL provides a set of data types specific to the Maconomy domain, e.g., `Date`, `Time` , `Amount` and pop-up types that are simply enumerations supported by the Maconomy UI in the form of drop-down lists. Moreover, records – user defined composite data type suitable for grouping data elements together – are available in MSL as structural types.

2.3 Methodology of choosing core MSL features

As explained in the Scope paragraph of Chapter 1, once the thesis project started, it soon became clear that defining an embedded Scala DSL, which could replace the full MSL language, along with implementing an MSL to Scala translator was simply too big of a task for a half year project. Therefore, in order to make the project feasible within the given timeframe, a core subset of MSL had to be chosen.

We established a few criteria that were leading the decision process of what features to include in MSL core. First of all, the chosen features should be essential to the language in a sense that when removed, the language would suffer from some fundamental shortcomings. Moreover, we wanted to include as much domain specific parts of MSL as possible. Finally, the chosen features should make up a representative subset of MSL, meaning that if we can design a Scala embedded DSL expressing these features and carry out an automatic source to source translation, then there should not be any fundamental problems in including the rest of MSL into MSL core and extending the translator to MScala respectively.

To this end, we hosted two workshops with the MSL application developers at Deltek. The first was about identifying key concepts in MSL as well as pointing out its strong points and weaknesses. The second involved abstracting the identified concepts from the concrete MSL syntax to be able to incorporate them in MScala.

2.4 MSL core

After the workshops with the MSL application developers took place, it became evident that not only is the MSL cursor a key concept to the language, but also its strongest and most expressive part. The importance of records following structural subtyping was also highlighted.

In order to incorporate these concepts in MSL core, more fundamental parts of the language had to be included too. As for the type system, most of the types have been included, except for `Date`, `Time`, `Amount` and pop-up types. When it comes to statements, `case` and `break` statements have been left out. MSL core supports defining individual functions and procedures as well as modules. Dialog scripts, which special kinds of MSL scripts are tightly coupled with the dialog state machine, have been left out, since they are more of a library or

framework than an essential language concept.

The full grammar of MSL core is included in Appendix A. Moreover, all the parts of the language are described in detail in Chapter 5, which specifies the MSL core to MScala translations.

2.5 Conclusions

The features included in MSL core cover all the spectrum of the MSL language constructs. The only major part left out is the dialog state machine, which is, however, more of an external framework or library than a native language concept. Hence, if MScala turns out to be a good replacement for MSL core, then it should not be difficult to extend it to cover the full MSL language. It would rather be a matter of time and effort put into more or less straightforward implementation than nontrivial struggles of reconciling two conceptually different languages.

Chapter 3

Scala as a host language for embedded DSLs

In this chapter we briefly introduce Scala, emphasizing the features that make it particularly suitable for hosting embedded DSLs. Moreover, two projects supporting deep embedding of DSLs – Scala-Virtualized and Scala Macros – are briefly described.

3.1 Quick introduction to Scala

Scala is a general purpose programming language that smoothly integrates object oriented and functional paradigms. It is statically typed, but due to local type inference the majority of type signatures do not have to be specified, the most notable exception being types of formal parameters in method signatures.

The object oriented nature of the language manifests itself in that everything in Scala is an object. The usual concepts like classes, access modifies, inheritance, polymorphism are present. Moreover, Scala enables multiple inheritance by mixing in traits [12], which gives a deterministic solution to the diamond problem [13] via trait linearization. Similar to interfaces in Java, traits are used to define object types by specifying the signature of the supported methods.

Unlike Java, Scala allows traits to be partially implemented; i.e. it is possible to define default implementations for some methods. In contrast to classes, traits may not have constructor parameters.

Scala seems to be a more complete and orthogonal object oriented language than Java in a sense that the two main notions of abstraction – parametrization and abstract members – apply to fields, methods and type variables in a uniform way. In Java, on the other hand, only some of the combinations are possible and one could argue that these limitations are fairly arbitrary. Table 3.1 shows which abstraction principles apply to which concepts in both Java and Scala.

	Abstract members	**Parameters**
fields and values	Scala	Java/Scala
methods	Java/Scala	Scala
types	Scala	Java/Scala

Table 3.1: Which constructs can be used as abstract members and parameters in Java and Scala

Besides being a full-blown object oriented language, Scala is a functional language too. It provides a lightweight syntax for defining anonymous functions, supports higher-order functions, allows functions to be nested, and supports currying. Scala's case classes and its built-in support for pattern matching model algebraic types used in many functional programming languages. The following paragraphs present the functional features listed above.

In Scala, functions are first class citizens, i.e., they can be treated as values, passed around as method parameters etc. Scala provides a lightweight syntax for specifying *function literals*, as shown in Listing 3.1:

Listing 3.1: Function literals in Scala

```
1 //anonymous functions (function literals)
2 val intList = List(1,2,3,4)
3 val evenNumbers = intList filter (i => i % 2 == 0)
4 val evenNumbers2 = intList filter (_ % 2 == 0)
```

The variable `intList` is defined as a `val`, which is a final value in Scala, i.e., once assigned – it cannot be changed. Mutable variables are defined in Scala

as vars. The method `filter` defined for `List` expects a function taking `Int` as a parameter and returning `Boolean` – in Scala such a type is denoted by `Int => Boolean`. The passed function literal can either name the argument it takes (e.g. `i => ...`) or use the underscore symbol to denote the first argument that is passed to it.

A *higher order function* is a function that takes other functions as a parameters, e.g., the `filter` method from the previous example. Suppose we want to define a function `applyFun` that takes another function `f` as a parameter and also an argument `arg` that is applicable to the function `f`. Then it returns the result of applying the function `f` to the given argument. The function `applyFun` can be defined as shown in Listing 3.2:

Listing 3.2: Higher order functions in Scala

```
1 //higher order functions with currying
2 def applyFun[T](arg : T) (f : T => T)  = f(arg)
3 //twoArg is a partially applied function of type: (Int => Int) => Int
4 //i.e. it takes a function of type (Int => Int) and returns Int
5 val twoArg = applyFun(2) _
6 //we supply as an argument a multiply by 2 function and get 4 as a result
7 val four = twoArg (_ * 2)
```

This definition uses another functional programming concept – *currying*. It is a technique of transforming a function that takes multiple arguments (or an n-tuple of arguments) in such a way that it can be called as a chain of functions each with a single argument. In our case `applyFun` takes `arg : T` as a parameter and returns a function that takes the function `f : T => T` as a parameter, which returns `T`. The `applyFun` function is given the first argument (partial application) and then, the resulting function `twoArg` is supplied with the remaining argument which is itself a function.

Another important concept in Scala is *pattern matching*. Basically, compound values can be matched against patterns and if the match is successful, extracted to simpler values that made up the compound value. For instance, the result of the `partition` method for `List`, which is a tuple of 2 lists of integers, can be matched against a tuple pattern and the 2 lists will be extracted to `even` and `odd` lists, as shown in Listing 3.3:

Listing 3.3: Simple pattern matching in Scala

```
1 //is extracted into 2 variables: even and odd of type List[Int]
2 val (even, odd) = intList partition(_ % 2 == 0)
```

Pattern matching is supported out of the box for *case classes*, but can be defined for any class by means of *extractors*. Listing 3.4 shows a definition of a few case classes modeling arithmetic expressions. It defines one function: `mulToShift` that, when applied to a multiplication node (`Mul`), tries to turn it into the corresponding left shift node (`Shl`) whenever applicable. For example, a node `Mul(Num(32),Num(7))`, which represents the arithmetic expression $32 * 7$ can be turned into `Shl(Num(7),5)`, which is equivalent to $(7 << 5)$.

Listing 3.4: Case classes and pattern matching in Scala

```
1  sealed trait Exp
2  case class Mul(a : Exp, b : Exp) extends Exp
3  case class Num(a : Int) extends Exp
4  case class Shl(a : Exp, n : Int) extends Exp
5
6  def mulToShift: Mul => Exp = {
7    case Mul(Shl(x,n),Num(y)) if y % 2 == 0
8      => mulToShift(Mul(Shl(x,n+1),Num(y/2)))
9    case Mul(x,Num(y)) if y % 2 == 0
10      => mulToShift(Mul(Shl(x,1),Num(y/2)))
11   case Mul(Num(x),y) => mulToShift(Mul(y,Num(x)))
12   case Mul(x, Num(1)) => x
13   case e => e
14 }
15
16 val e = Mul(Num(32),Num(7))
17 println(mulToShift(e)) //prints Shl(Num(7),5)
```

3.2 DSL hosting capabilities of Scala

When it comes to internal domain specific languages, there is a distinction between *shallow* and *deep* embedding of DSLs [7]. Shallowly embedded DSLs are DSLs defined as pure libraries that, due to the flexibility of the host language syntax, look and feel as if they were providing built-in language constructs. Deep embedding, on the other hand, is when a DSL implementation builds or obtains an internal representation of a domain program (usually some sort of AST), which can first be analyzed, optimized and then executed.

Pure Scala enables both shallow and deep embedding of DSLs, although the latter approach has its limitations, which two ongoing research projects – Scala-

Virtualized [7] and Scala Macros [10] – are trying to address. The two projects are briefly described in sections 3.2.1 and 3.2.2.

Let us take a look at an example of the `repeat .. until` loop, which is not supported by Scala natively but can be easily implemented in a few lines of code, as shown in Listing 3.5

Listing 3.5: Implementation of the repeat .. until loop in Scala

```
1  //repeat until loop definition
2  def repeat(body: => Unit) = new {
3    def until(condition: => Boolean) = {
4      do {
5        body
6      } while (!condition)
7    }
8  }
9
10 //and its use
11 var i = 0
12 repeat{
13   i += 1
14   println(i)
15 } until( i >= 10)
```

The first thing to notice in this example is that the defined `repeat .. until` loop looks no different in terms of syntax than the `do .. while` loop, which is natively supported by Scala, i.e., implemented by the Scala compiler. There is a number of Scala features that enable this piece of code to be implemented:

1. *by-name parameters* – the parameters `body` and `condition` are passed by name, which means that they are evaluated in a lazy manner upon every use in the defining function. In other words, the parameter `body` behaves as if it was a function taking no parameters and being evaluated upon every invocation of it.
2. *infix operator syntax for method calls* – the `until` part of the loop is simply a method call on the anonymous object returned from the `repeat` function. Moreover, method names can be of an almost arbitrary form in Scala (except for predefined keywords), so one can define methods called `+`, `->` or `===`.
3. *curly braces for method calls* - the code passed to the `repeat` function

as a parameter can be enclosed within curly braces instead of standard parenthesis.

Besides the features mentioned above, there is also a number of other features that proved themselves to be very useful in embedding DSLs in Scala.

4. *for expressions (a.k.a. for comprehensions)*
   For expressions consist of three parts : generators, filters and a yield expression, as shown in Listing 3.6. A generator is just a familiar way of

Listing 3.6: For expressions in Scala

```
1 val students = List(Student("Piotr",25),
2                     Student("Anna", 19),
3                     Student("Jens", 10))
4 val adultStudents =  //returns List[Student]
5   for {
6     student <- students   //generator
7     if(student.age >= 18) //filter
8   } yield student         //yield
```

   naming a particular element in a collection and then referring to it while iterating over the collection, like in case of a standard for loop. Filters filter out the elements in a collection that do not meet the specified predicate. Finally, the yield expression just returns a new element for every iteration step that complies with the specified filters.

   For expressions can be seen as a query language for collections of data. The very same query as shown in Listing 3.6 could be expressed in SQL as follows (provided that there is a table `students` with a field `age`):

   ```
   select * from students where age >= 18
   ```

   With multiple generators, filters and a complex yield expression, for expressions make up a very powerful query language. For this reason SqalaQuery [14] – one of the leading Scala database libraries in the market – has adopted the for expression syntax to formulate SQL queries. This is possible, because for expressions are translated by the Scala compiler to a combination of three higher order functions: `map`, `flatMap`, and `withFilter`. Therefore, it suffices to provide an implementation of these three methods for a class in question, e.g `DatabaseTable` class, to be able to query an instance of this class with a for expression.

5. *implicit conversions*
   Whenever a type A is expected but a type B is given instead, the Scala

compiler searches for an implicit conversion definition that could convert B to A (hence it has to be a function of type `B => A`). If such an implicit conversion function is found in the required scope, it is applied. Otherwise a compile error is issued. This mechanism is very powerful – it can be used, for instance, to lift a part of a program to the AST representation of it at runtime. In case of a for expression querying a database table, implicit conversions can be used, e.g., to lift the expression given to a filter as a parameter to an expression tree, which can be used to generate an SQL query instead of simply evaluating the expression.

6. *manifests*
   A method parametrized with a type parameter `T` can request the Scala compiler to provide a runtime descriptor of `T` in the form of `Manifest[T]`. Whenever requested, the manifest is passed as an implicit parameter. The main use of manifests in the context of embedded DSLs is to preserve information necessary for generating efficient specialized code in those cases where polymorphic types are unknown at compile time (e.g. to generate code that is specialized to arrays of primitive type, say, even thought the object program is constructed using generic types).

3.2.1 Scala Virtualized

Scala Virtualized extends the Scala language and compiler by a small number of features that enable combining shallow and deep embeddings of DSLs [7]. First of all, it redefines most control structures (e.g. conditionals, variable definitions, assignments) in terms of method calls, which can be overridden by the DSL implementation to change the meaning of these core language constructs. Moreover, it provides implicit source context which lifts static source information (e.g file names, line numbers) such that it becomes part of the object program.

The project is a branch of the official Scala distribution, but it undergoes the same rigorous testing and quality assurance procedures as the official Scala distribution. It has been successfully used in a number of research projects, e.g., Delite – compiler framework and runtime for parallel embedded DSLs [15] or OptimML– a DSL for machine learning that employs aggressive, domain-specific optimizations resulting in high-performance code [16].

3.2.2 Scala Macros

Scala Macros [10] bring up compile-time meta-programming capabilities to Scala. Basically, macros can be seen as functions taking ASTs, manipulating them and returning possibly rewritten ASTs. Whenever a compiler sees an invocation of a method declared as a macro definition, it calls the implementation of the macro with the arguments being the ASTs that correspond to the arguments of the original invocation. After the macro returns, its result gets inlined into the call site. This all happens at compile-time, giving a DSL implementor the power comparable a compiler plug-in, but much more lightweight in use.

Macros has been included in the official distribution of Scala 2.10 as an experimental feature.

3.3 Conclusions

In order to design a successful DSL, domain expertise seems to be crucial. Building external DSLs is hard – one needs to be an expert in compiler technologies to do so. This kind of knowledge rarely goes together with a particular domain expertise, e.g., in business processes in professional services firms, which makes building accurate DSL operating on the right level of abstraction even harder.

Scala offers a lightweight way of implementing library-based DSLs that look and feel like built-in language features. This way of embedding DSLs proved itself to be sufficient in a number of domains (actors [8], parser combinators [9], testing frameworks [17], language integrated type-safe queries [18, 14], language-processing libraries [19] etc.). If deep embedding of a DSL is needed (i.e. enabling overriding built-in language constructs like assignment operator, if statements or operating on the AST representation of a relevant part of a program), the ongoing projects like Scala-Virtualized or Scala Macros can be utilized.

Once the MSL code base has been converted to Scala and domain specialists have gained confidence in using the language, Scala equips the developers with a very lightweight yet powerful way of defining internal DSLs and hence gradually raising the level of abstraction in expressing business logic in the Maconomy system.

Chapter 4

MScala as a new language for business logic in Maconomy

This chapter highlights the main advantages of introducing Scala as a new language for business logic development in Maconomy. We use the name *MScala* to refer to a DSL embedded in Scala that is the proposed replacement for MSL core. MScala consists of a database library called *Squeryl* [18], which brings language integrated type-safe queries to the language, together with a number of custom types, methods and functions introduced for the sake of automatic code translation from MSL core. All of these extensions are described in detail in Chapter 5.

This chapter starts off by describing productivity gains from using Scala in general, based on some current usability and cognitive psychology research as well as on the availability of commercial quality tools, libraries and frameworks supporting the language. Further, we show, by presenting several examples, how MScala can be employed to solve tasks specific to the Maconomy domain. When compared to the idiomatic MSL solutions, the Scala ones are much more concise, elegant and very often better performance-wise since they enable to reduce the number of database queries that must be executed against the database to solve the task at hand. We finish up by showing how Scala supports building reusable software components on a larger scale by integrating object-oriented and functional paradigms.

4.1 Productivity gains from using Scala

4.1.1 Research in programming style and productivity

Scala is often compared with Java, since it is a very innovative language running on the JVM. Java is a very mature object-oriented language, facilitating generic programming, subtyping and inheritance as well as classes and interfaces. All of these features allow for building extensible software systems made out of reusable components.

That being said,"experience with Scala shows that it typically takes one half to one third of the lines of code to express the same application as Java yet executes at about the same speed. From the research it also appears that Scala better matches the way programmers think, and it seems to be a view supported by experienced Scala programmers to" [20]. In another article [21] Gilles Dubochet claims that it is, on average, 30 % faster to comprehend algorithms that use for-comprehensions and maps, as in Scala, rather than those with the iterative while-loops of Java.

In comparison to Java or Scala, MSL does not support building reusable software components well. The only abstractions allowing for code reuse are functions and procedures. However, in MSL functions and procedures cannot be parametrized with types. MSL has also a rather verbose syntax that further contributes to the increased number of lines of code. Therefore, it is sensible to expect that when implementing new functionality in MScala application programmers should experience at least the same factor of reduction in terms of lines of code as in the case of Java.

4.1.2 Tool support for Scala

Nowadays one of the most significant factors that contribute to programmers' productivity is good tool support, with a first-class Integrated Development Environment being particularly important. Java ecosystem is huge, very mature and by-design everything that is written in Java can be easily used in Scala. Moreover, Typesafe [22], a company created by Martin Odersky (the creator of Scala) to support Scala commercially, is currently putting a lot of effort and resources into improving the Eclipse Scala plug-in, which already gives a commercial quality user experience [23].

4.2 MScala by examples from the Maconomy domain

Oversimplifying things a bit, one could say that Maconomy is a typical database system, where the majority of tasks have to do with reading data from the database, processing collections of data and either displaying it in a form accessible to the user or writing it back to the database. Therefore, this section focuses mainly on processing collections of data as well as working with databases.

4.2.1 Database schema

All of the examples presented in this chapter are based on a simple database of students attending courses at different universities. Figure 4.1 depicts the database schema for this domain.

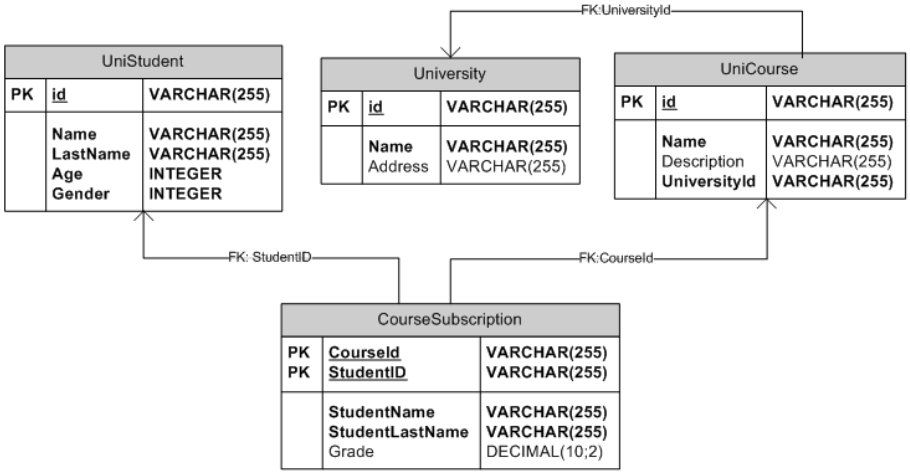

Figure 4.1: Schema of a database of students

4.2.2 Example 1: Operations on collections

As mentioned in Chapter 2, MSL does not support dynamic memory allocation and at the same time does not provide any predefined collections library. The only way of building collections of data, except for using statically allocated

arrays, is therefore to retrieve the data from the database be means of a cursor. Cursors support a substantial part of SQL, but queries are limited to retrieving data from one relation only (no joins).

Suppose we want to display a list of all the students, with a possibility of dividing them into 2 groups: students younger than 26 (e.g. eligible for student discounts) and the ones that are 26 years old or older. To this end, we need to populate three collections with the respective data. Moreover, we would like to find the youngest and the oldest student. Listing 4.1 shows a typical MSL implementation of this functionality. The youngest and the oldest students are found in the `for all` loop to avoid an unnecessary query against the database, which might be expensive in a distributed system.

Listing 4.1: MSL: operations on collections

```
1  youngestAge : Integer := Integer'last;
2  oldestAge : Integer := Integer'first;
3
4  read cursor AllStudents is
5  select all from UniStudent;
6
7  read cursor EligableForDiscounts is
8  select all from UniStudent
9  where Age <= 26;
10
11 read cursor NotEligableForDiscounts is
12 select all from UniStudent
13 where Age > 26;
14
15 CheckFatal(Get(AllStudents));
16 CheckFatal(Get(EligableForDiscounts));
17 CheckFatal(Get(NotEligableForDiscounts));
18 for all AllStudents do
19   youngestAge := MinInteger(youngestAge, AllStudents.Age);
20   oldestAge := MaxInteger(oldestAge, AllStudents.Age);
21 end for;
```

What makes the MSL implementation cumbersome is that for every new collection of data we have to declare a new cursor. MSL provides no means of reusing cursor declarations, which leads to code redundancy. Moreover, every new cursor means a new query executed against the database, which in many situations might introduce a substantial performance overhead in a distributed system like Maconomy. Once we have a list of all the students fetched, there should be no

need to query the database again for students eligible for a discount and those who are not. In MSL, however, this is the only way to obtain a new collection of data. Moreover, since MSL does not support generic programming (functions parametrized with types), the only operation we can perform on a collection of records is to iterate through it by using a built-in `for all` loop. Defining any other generic functions on collections is technically impossible in MSL.

Let us now analyze an idiomatic Scala implementation of the described functionality, which is shown in Listing 4.2:

Listing 4.2: Scala: operations on collections

```
1  def CollectionsTest{
2    val allStudents = from(uniStudentTable)(select(_)).toList
3    val (eligableForDiscounts,notEligableForDiscounts) =
4      allStudents partition (_.Age <= 26)
5    val youngest = allStudents.minBy( _.Age).Age
6    val oldest = allStudents.maxBy(_.Age).Age
7  }
```

The first striking difference is in the conciseness of the 2 implementations; 7 lines of Scala code in contrast to 21 in MSL (excluding blank lines). The Scala version is, moreover, very likely to perform much better, since it avoids executing 2 additional queries against the database. In line 2 we declare a query selecting all the students from the uniStudentsTable and then execute it by calling the `toList` method on it. If we didn't call `toList`, the code would compile too and give the same result, except that it would execute 3 queries against the database instead of one, since in Squeryl a query is executed every time some sort of iteration is performed over it. Line 3 makes use of pattern matching in Scala – it extracts the result of the `partition` method, which is a tuple of 2 lists, into 2 variables. The functions in lines 3–6 (`partition, minBy, maxBy`) are higher order functions – they take other functions as parameters. The underscore character '_' denotes an argument to which the function passed as a parameter should apply; in this case it's a current element of the collection – a `UniStudent` object.

Generally speaking, lines 2–6 owe their conciseness to the following Scala features:

- *higher order functions* (functions that can take other functions as parameters), e.g., the comparison function <= passed as a parameter to the *partition* method

- *generic methods* - methods in Scala can be parameterized with both values and types, which allows for defining generic methods

- *type inference* - whenever a type can be inferred from the context, it does not have to be specified. It results in a much more lightweight syntax, similar to dynamically typed languages like python, yet preserving the compile-time type-safety offered by statically typed languages.

- *in-line variable declarations* - variable declarations can be intermixed in Scala with statements/expressions.

4.2.3 Example 2: SQL joins

MSL cursor queries do not support joins. In other words, an MSL cursor can return a subset of fields of one table only. One can, however, bind 2 cursors together by referencing one of them in the `where` clause of the other. This can be seen as a substitute of an outer join. This workaround, however, can lead to both very verbose and slow code.

Suppose we want to calculate the average of grades for a particular university, which is given as a parameter. Listing 4.3 shows how it can be done in MSL.

Grades are stored in the `CourseSubscription` table, which is bound with the given `University` by means of the `UniCourse` table. Basically, we have to iterate through all the courses belonging to the given `University`, and for each of them store the sum of grades and the number of subscriptions. Once the total sum of grades and the number of course subscriptions belonging to the given `University` are calculated, we can return the average as a simple division of the two values.

The corresponding Scala solution, shown in Listing 4.4, is as straightforward as it can get. It defines one query calculating exactly what we want – the average of grades for all of the courses at the given `University`.

Not only is the Scala version 4 times shorter, but also performs much better, since it executes only one query against the database, as opposed to the MSL version, which executes the number of queries equal to the number of courses at the given `University` plus one, as we need to fetch all the courses first. Moreover, the Scala query fetches only one number from the database, whereas the MSL version fetches potentially a lot of data, which might be very expensive performance-wise.

Listing 4.3: MSL: outer joins substitute

```
function GradeAverage(cursor University : University) : Real is
var
  gradeSum: Real := 0;
  avgCount : Integer  := 0;

  read cursor UniCourse is
  select all from UniCourse
  where UniversityId = University.Id;

  read cursor CourseSubscription is
  select sum(Grade) as GradeSum, count(Grade) as GradeCount from
      ➥ CourseSubscription
  where CourseId = UniCourse.Id;

begin
  for all UniCourse do
    CheckFatal(Get(CourseSubscription));
    if CourseSubscription.GradeCount > 0 then
      gradeSum := gradeSum + CourseSubscription.GradeSum;
      avgCount := avgCount + CourseSubscription.GradeCount;
    end if;
  end for; -- UniCourse
  if avgCount = 0 then
    return 0;
  else
    return gradeSum / avgCount;
  end if;
end function;
```

Listing 4.4: Scala: inner joins

```
def gradeAverage(university : University) : BigDecimal = {
  from(courseSubscriptionTable, uniCourseTable){ (cs,uc) =>
    where(cs.CourseId === uc.id and uc.UniversityId === university.id)
    compute(avg(cs.Grade))
  }.getOrElse(0)
}
```

4.2.4 Example 3: Code reuse in cursors

An MSL cursor can be seen as a definition of a variable of some new type that is valid only in the scope in which it has been defined. In other words, the variable is a singleton instance of the newly defined type. In this respect, such a cursor has quite a schizophrenic nature, as it can denote either a database query or a current record, depending on the context in which it is used. In Listing 4.1, we can see an example of this dual nature. In line 18 the `AllStudents` cursor denotes a query – the `Get` function executes the query against the database. In line 22 and 23 however, the very same identifier, `AllStudents`, denotes a current record in the iteration.

This design decision, although saves a bit of typing needed to declare a separate identifier for a current record, has some profound implications. It does not allow for reusing cursor query declarations – neither is it possible to instantiate a new query of this "type" nor to use the query as a subquery composed into a more complex one. Moreover, when passed as a parameter to a function or procedure, a cursor always denotes a reference to the current record, making it impossible to reuse cursor definitions declared elsewhere. To sum it up, in MSL every database query must be defined (typed or copied) anew, even though there might be a lot of copies of the very same query in other places in the system.

In Scala, on the other hand, not only can one reuse the same query in all parts of the system (e.g. by dependency injection), but also use it as a building block to compose more and more complex queries. These capabilities are shown in Listing 4.5, where subsequent queries are built out of the previously defined ones.

4.3 Building succinct, reusable software components in Scala

4.3.1 Object oriented concepts

Scala is a fully object-oriented language in a sense that all the entities in a Scala program are objects. Object-oriented programming has been so successful for the last decades, because this paradigm is particularly suitable for building software systems out of reusable, loosely coupled components. The title of the famous Design Patterns book [24] develops further as "Elements of Reusable Object-Oriented Software", which is pretty self-descriptive. Concepts like ab-

Listing 4.5: Scala: reusable queries

```
def queryReuse(dtu : University){
    val cursesAtDTU = uniCourseTable.where(uc => uc.UniversityId === dtu.
        ➥ id)
    val courseSubscriptionsAtDTU =
      from(cursesAtDTU, courseSubscriptionTable){ (ucAtDtu, cs) =>
        where(ucAtDtu.id === cs.CourseId)
      select(cs)
      }

    val studentsFromDtu =
      from(courseSubscriptionsAtDTU, uniStudentTable){ (csAtDtu,s) =>
        where(csAtDtu.StudentId === s.id)
        select(s)
      }.distinct

    val maleStudentsFromDtu =
    studentsFromDtu.where( s => s.Gender === MALE)
    val femaleStudentsFromDtu =
    studentsFromDtu.where( s => s.Gender === FEMALE)
  }
```

stract members, polymorphism, type parameters, composition, encapsulation via access modifiers etc. facilitate this reusability and help building extensible software systems.

Scala goes even further than Java in expressing object-oriented abstractions. The two main notions of abstractions in programming languages – abstract members and parametrization – apply in Scala to fields, values methods and types in a uniform way, as described in Section 3.1. Moreover, Scala abstract type members, explicit selftypes, and modular mixin composition bring up new capabilities in building reusable and scalable software components, as described in the "Scalable Component Abstractions" paper by Martin Odersky and Matthias Zenger [12].

To sum it up, Scala provides some very powerful object oriented concepts that, when used wisely, can help structure complex software systems to make them more comprehensible, scalable, extensible and to facilitate as much code reuse as possible.

4.3.2 Functional concepts

The functional aspects of Scala are described in Section 3.1 Summing it up, Scala provides a lightweight syntax for function literals, supports higher-order functions, allows functions to be nested (closures), and supports currying. In addition to that, pattern matching is supported out of the box for case classes as well as for all the classes that define respective extractors.

Each of these features stand out in some particular applications and having them in a toolbox allows for writing very succinct, easily understandable and 'right to the point' code.

4.4 Conclusions

When solving Maconomy specific problems from scratch, idiomatic MScala solutions tend to be much more concise, elegant and often better performance-wise than the corresponding MSL solutions. In the large scale, Scala brings to the table advanced object-oriented concepts like parametrization and abstract members that allow for constructing generic and reusable software components, which is basically infeasible in MSL.

In business terms it means shorter development time as well as a smaller, clearer, more extensible and maintainable code base. In addition to that, a first-class tool support, IDEs and libraries together with great DSLs hosting capabilities of Scala can further boost the MSL developers' productivity to a whole new level, once they switch to Scala.

Chapter 5

Translating MSL core to MScala

Chapter 4 presented some examples of how typical implementation tasks in Maconomy can be expressed in a much more concise way in MScala compared to MSL. When new functionality is to be implemented or the old code to be refactored, Maconomy business logic developers can use the full power of MScala to express the solutions in a clean and elegant manner.

However, what gives Maconomy a competitive edge is the functionality of the current system, which is unique in the market. The 1 million lines of MSL code, which implement all of these features, are a very valuable asset for the Maconomy product and it is simply a business imperative that this code is auto-translated when migrating to a new programming language.

In this chapter we describe in detail how to automatically translate each MSL core language construct into a semantically equivalent MScala construct. All the translations assume to have valid MSL input code. First, a straightforward translation is defined, which results in MScala code that is not only semantically equivalent but also very similarly looking to the source MSL code. Although this might be seen as a benefit, since the auto-translated code will still look familiar to the MSL developers, it also means porting all the weaknesses and limitations of MSL to MScala that, after all, are the reasons to abandon MSL and move to MScala in the first place. Therefore, we further define a number of

transformations that turn this straightforward translations into more idiomatic Scala.

For every MSL language construct that does not have a straightforward equivalent in Scala, we provide a design rationale for the chosen translation, sometimes describing the alternatives that have been rejected. We believe that the contribution of this thesis in terms of automatic translation lies not only in presenting the final solutions, but mostly in describing in detail what kind of considerations and trade-offs were involved when choosing between different alternatives.

5.1 Straightforward translations

5.1.1 Type system

Led by Occam's razor principle, which is sometimes formulates as "plurality should not be posited without necessity", we decided to use Scala built-in types whenever they could make up a good replacement for the corresponding MSL types. Basically, the semantics of *Boolean*, *Char* and *Integer* in MSL is exactly the same as the corresponding *Boolean*, *Char* and *Int* semantics in Scala. The other MSL core types do not have straightforward equivalents in Scala, therefore custom types have been introduced as their replacements. Table 5.1 shows the correspondence between the MSL core and MScala types. The remainder of this section describes each of these custom types translations in detail.

MSL core	**MScala**
Boolean	Boolean
Char	Char
Integer	Int
Real	MReal
Amount	MAmount
String	MString
Array	MArray
Record	Classes with traits
Cursor	MCursor

Table 5.1: Correspondence between MSL core and MScala types

5.1.1.1 Amount and Real

Amount and Real types in MSL are just arbitrary precision signed decimal numbers. MScala provides a wrapper around `BigDecimal` for each of them (`MAmount` and `MReal` respectively), to be able to control their automatic conversion behavior to other types. For instance, it should not be possible to add an amount to a real number as it does not make sense in the Maconomy domain.

5.1.1.2 String

In MSL, strings are mutable sequences of characters. A character in a string can be accessed using 1-based indexing. Since MSL strings are value types, they are copied upon assignment. On the contrary, strings in Scala are 0-based indexed, immutable sequences of characters with reference type semantics of an assignment operation, i.e, an assignment binds a string object to a variable (symbol).

Translation

MSL strings are translated into built-in Scala strings. Since strings in Scala are immutable, there is no need to make a defensive copy upon assignment like in MSL. The mutability aspect of MSL strings as well as the 1-based indexing are solved by providing a rich wrapper around the Scala String class – MString – along with an implicit conversion from String to MString.

Basically, whenever there is a type-safe assignment to a particular character in a string, the MString wrapper provides a method `update1based` that takes 1-based index and the new character and returns a new string, with the char at the given index replaced by the new char. The newly created string is then assigned back to the same string variable, which emulates mutability of strings in MSL.

Similarly, whenever a char at a particular 1-based index is referenced (not in an assignment), MString provides a method `get1based` that takes this index as a parameter and returns the indexed character.

Translation 1 shows an example of a function that performs some string manipulations (it replaces 'S' character by 'M' character).

Table 5.2 formalizes the translation of the operations on strings, as well as some

Listing 5.1: MSL: operations on strings

```
1  function testString() : Boolean is
2  var
3    s : String := "Scala String";
4    i : Integer := 1;
5  begin
6    while i <= s'no_of_elems do
7      if s[i] = "S" then
8        s[i] := "M";
9      end if;
10     i := i + 1;
11   end while;
12   return s = "Mcala Mtring";
13 end function;
```

Listing 5.2: MScala: operations on strings

```
1  def testString(): Boolean = {
2    var s = "Scala String"
3    var i = 1
4    while(i <= s.size) {
5      if(s.get1based(i) == 'S') {
6        s = s.update1based(i,'M')
7      }
8      i = i + 1
9    }
10   return s == "Mcala Mtring"
11 }
```

Translation 1: Operations on strings

of its attributes (`no_of_elems`) and predefined functions (`ConcatString`).

MSL core	MScala
<str_val>[<e1>] := <e2>	<str_val> = <str_val>.update1based(<e1>,<e2>)
<str_val>[<e1>]	<str_val>.get1based(<e1>)
<str_val>'no_of_elems	<str_val>.size
ConcatString(<str_val1>,<str_val2>)	<str_val1> + <str_val2>

Table 5.2: String translations from MSL core to MScala

Rationale

Apart from the presented translation, an alternative solution has been considered. To deal with mutability and 1-based indexing as well as to keep the Scala code operating on strings as close to the MSL source code as possible, one could implement a custom 1-based indexed mutable String class. This would not, however, solve the copy on assignment semantics of MSL, because it is impossible in Scala to override the assignment operator. While translating the

code one could, of course, explicitly copy strings upon assignment as shown in Listing 5.3.

Listing 5.3: Scala: alternative translation of strings

```
1 var str1 = MutableString("hard work")
2 MutableString(6) = 'p'
3 var str2 = str1.copy()
4 str2 = "changed value"
5 println(str1)   //prints "hard pork"
6 println(str2)   //prints "changed value"
```

Such auto-generated code would be, however, hard to maintain and interoperate with, since it requires a user to call a `copy` method whenever a string is reassigned to a variable, but does not enforce it in any way, which is rather error-prone.

Strings are immutable in most of the modern programming languages (Scala, Java, C#, F#, SML just to name a few) and this solution has a lot of advantages. Immutable objects are better for concurrency, hashing and safe sharing (no defensive copying needed). Besides that immutability of strings is crucial for security / safety requirements. For these reasons it has been decided to use standard Scala immutable strings, which is a merit on its own since we are consistent with the host language.

Moreover, the methods `get1based` and `update1based`, offered by the `MString` rich wrapper, encapsulate 1-based indexing in a transparent way and provide meaningful names making it obvious that this is auto-generated code.

5.1.1.3 Array

MSL supports multi-dimensional arrays with custom indexing, e.g., like in the Pascal programming language. That being said, the actual MSL code in the Maconomy system uses almost exclusively one-dimensional arrays, with a few exceptions of two-dimensional ones. For the sake of simplicity and due to time constraints it has been decided to support only one-dimensional arrays in the automatic translation and to emit error messages in case of multi-dimensional arrays, so that they can be handled manually.

Translation

MScala provides *MArray* – a wrapper around a standard Scala Array – that makes the translation of custom indexing clear and transparent. An instance of *MArray*, once instantiated with a range object representing a custom indexing range, is used with original MSL indexes that are then internally shifted to match the underlying 0-based Array instance, whenever an element of the array is accessed.

Translation 2 shows an example of a function that fills out an array with the `newElem` character.

Listing 5.4: MSL: arrays

```
function testArray() : Boolean is
var
    a : Array[1..10] of Integer;
    i : Integer;
    newElem : Integer := 7;
begin
  i := a'first;
  while i <= a'no_of_elems do
      a[i] := newElem;
      i := i + 1;
  end while;
  i := a'first;
  while i <= a'no_of_elems do
      if a[i] <> newElem then
        return false;
      end if;
      i := i + 1;
  end while;
  return true;
end function;
```

Listing 5.5: MScala: arrays

```
def testArray(): Boolean = {
  var a = MArray[Int](1 to 10)
  var i = 0
  var newElem = 7
  i = a.first
  while(i <= a.size) {
    a(i) = newElem
    i = i + 1
  }
  i = a.first
  while(i <= a.size) {
    if(a(i) != newElem) {
      return false
    }
    i = i + 1
  }
  return true
}
```

Translation 2: Operations on arrays

Table 5.3 formalizes the translation of some of the array's attributes (`no_of_elems`, `first`, `last`).

MSL core	MScala
<array_val>'no_of_elems	<array_val>.size
<array_val>'first	<array_val>.first
<array_val>'last	<array_val>.last

Table 5.3: Arrays translations from MSL core to MScala

5.1.1.4 Record types

In MSL, a record is a user defined composite data type suitable for grouping data elements together. It consists of a number of fields, which can be of primitive type only. Records cannot, in particular, have fields that are also records or arrays.

Records are structural types and as such comply with structural subtyping relationships, i.e., a record A is a structural subtype of a record B iff all the fields of B (having the same names and types) can be found in A.

Moreover, records exhibit one more interesting characteristic – a special semantics of an assignment operation. Two records A and B are assignment compatible if the set of fields S of one of them is a subset of the set of fields of the other (again, taking into account the names and types). If compatible, we can assign A to B as well as B to A with the semantics being that the values of the fields in S are simply copied from one record to the other (other fields remain unchanged). So in case of records, an assignment does not bind a new record in memory to a variable - it just copies the values of the common fields.

Listing 5.6 shows a declaration of three records, whose names denote which fields are present in the record.

In the given example, both `AB` and `BC` are structural subtypes of `B`. Moreover, the pairs `AB` and `B` as well as `BC` and `B` are assignment compatible, which means that it is possible to assign one record to the other. Records `AB` and `BC` are neither assignment compatible nor in a structural subtyping relationship.

The type system of Scala is, at its roots, nominal [25], i.e., subtyping relationships are specified explicitly based not only on the presence of the same members, but also on the same semantics of these members. There are two means of specifying nominal subtyping in Scala – class inheritance and trait mixin. Classes and traits are much richer concepts than records in MSL and combined together can model records, structural subtyping and the peculiar semantics of the assignment operation.

Listing 5.6: MSL records: structural subtyping and assignment compatibility

```
1  AB = record
2      A : Integer;
3      B : Integer;
4    end record;
5
6  B = record
7      B : Integer;
8    end record;
9
10 BC = record
11     B : Integer;
12     C : Integer;
13   end record;
```

It is worth mentioning that Scala supports structural subtyping too [25], but a solution facilitating this feature has been rejected in favor of explicit modeling of subtyping relationships via trait mixin. The rationale behind this design decision is given after the chosen translation description.

Translation

The idea behind the proposed translation is to bring to MScala only the subtyping relationships that are actually present in the source program, i.e., to declare that a record A is a subtype of a record B only if A is actually used in the code as a subtype of B. The only places where it can happen in MSL are function calls, in which a record of type B is expected as a formal parameter, but A is given as the actual parameter. In this case, if all the fields of B are present in A, then we can indeed say that A is a subtype of B and that was the intention of the programmer who wrote the code and not merely a coincidence that these two records happen to share the same fields.

So, whenever we encounter a function call where a record A of a type AT is treated as a value of record type BT, where AT is a subtype of BT, we can declare the type AT to mix-in a trait representing the type BT. Listings 5.7 and 5.8 show a translation of an example module using structural subtyping for records.

Listing 5.7: MSL: Records

```
module Records is

type
  Named =
    record
      name : String;
    end record;

  Person =
    record
      name : String;
      age  : Integer;
    end record;

  Employee =
    record
      name : String;
      age  : Integer;
      company : String;
    end record;

  Painting =
    record
      name : String;
      age  : Integer;
    end record;

  procedure setName(
    var n : Named;
    newName : String) is
  begin
    n.name := newName;
  end procedure;

  procedure copyNamedToPerson(
    var n : Named;
    var p : Person) is
  begin
    p := n;
  end procedure;
```

Listing 5.8: MScala: Records

```
object Records {
  trait INamed {
    var name: String
    def update(that : INamed)
  }
  case class Named(
    var name: String = "")
  extends INamed{
    def update(that : INamed) {
    name = that.name
    }
  }
  trait IPerson {
    var name: String
    var age: Int
    def update(that : INamed)
  }
  case class Person(
    var name: String = "",
    var age: Int = 0)
  extends IPerson with INamed {
    def update(that : INamed) {
      name = that.name
    }
  }
  trait IEmployee {
    var name: String
    var age: Int
    var company: String
  }
  case class Employee(
    var name: String = "",
    var age: Int = 0,
    var company: String = "")
  extends IEmployee with IPerson{
    def update(that : INamed) {
      name = that.name
    }
  }
  trait IPainting {
```

```
41
42    procedure resetNamed(
43      var n : Named) is
44    var
45      newNamed : Named;
46    begin
47      n := newNamed;
48    end procedure;
49
50    function testRecords() :
          ➥ Boolean is
51    var
52      p : Person;
53      e : Employee;
54    begin
55      setName(p, "Person");
56      e.age := 11;
57      p.age := 44;
58      e.company := "Deltek";
59      copyNamedToPerson(p, e);
60      resetNamed(p);
61      return (p.name = ""
62              and p.age = 44
63              and e.name = "Person"
64              and e.age = 11
65              and e.company = "
                    ➥ Deltek");
66    end function;
67  end
```

```
41      var name: String
42      var age: Int
43    }
44    case class Painting(
45      var name: String = "",
46      var age: Int = 0)
47    extends IPainting
48
49    def setName(
50      n : INamed,
51      newName : String) {
52      n.name = newName
53    }
54    def copyNamedToPerson(
55      n : INamed,
56      p : IPerson) {
57      p.update(n)
58    }
59    def resetNamed(n : INamed) {
60      var newNamed = Named(Named())
61      n.update(newNamed)
62    }
63    def testRecords(): Boolean = {
64      var p = Person()
65      var e = Employee()
66      setName(p,"Person")
67      e.age = 11
68      p.age = 44
69      e.company = "Deltek"
70      copyNamedToPerson(p,e)

71      resetNamed(p)
72      return p.name == "" &&
73        p.age == 44 &&
74        e.name == "Person" &&
75        e.age == 11 &&
76        e.company == "Deltek"
77    }
78  }
```

In the given example, every record is translated into a Scala trait declaring all

the fields present in the record. Then such a trait can be used as a type of a formal parameter in a function signature. For instance, the `setName` procedure declared in the MSL listing in line 28 takes as a parameter a record n of type `Named`. This procedure is further called in function `testRecords` in line 55 with a record e of type `Employee`. Therefore the Scala class `Employee` extends the trait `INamed`, which introduces the detected subtyping relationship. Similarly, in line 59 the `copyNamedToPerson` procedure expects the parameters of types `Named` and `Person`, but is given a `Person` and an `Employee` respectively. For this reason, the Scala class `Person` mixes in the `INamed` trait and the `Employee` class mixes in the `IPerson` trait.

Assignments to records are handled in a similar manner. For every pair of assignment compatible records `A` and `B`, if there is a an assignment from record `A` to `B`, then it gets translated to an auto-generated `update` method on object `A` that takes object `B` as a parameter. This `update` method simply copies the subset of common fields' values to the receiver's fields. For example, the `copyNamedToPerson` MSL procedure in line 39 from Listing 5.8 assigns a `Named` record to a `Person` record. It gets translated to the `update` method call in line 57 in listing 5.7, which is declared in the `IPerson` trait and implemented by the `Person` class.

Rationale

As mentioned briefly in the previous section, Scala has built-in support for structural subtyping, but there is a certain performance overhead associated with it. Scala runs primarily on the JVM, which does not support structural types natively. To get around this limitation, the Scala compiler uses reflection with caching, which makes method invocations on structural types around 2-7 times slower than on nominal types, depending on the cache hits/misses ratio [25].

Moreover, the type system of Scala is, at its roots, nominal, i.e., subtyping relationships are specified explicitly based not only on the presence of the same members, but also on the same semantics of these members. Such a solution, although arguably more verbose, guarantees much better type safety since it prohibits from treating two types, having incidentally the same members, as subtypes of each other when they are conceptually completely different entities. For instance, let us consider two classes: `Person` and `Painting`, each having the same fields: `name: String` and `age: Integer`. A `Painting` is a structural subtype of a `Person`, therefore, e.g., a method `feed` expecting a structural type `Person` as a parameter could accept an instance of a `Painting` too, although semantically it does not make much sense to feed a painting. For this reason, e.g., the class `Painting` from Listing 5.8 does not mix in the `IPerson` trait, because there is nothing in the code indicating that these entities should be the subtypes of each other. This is a flexible solution, since at any point in time

any auto-translated records, which are in a structural subtyping relationship, can be turned into nominal subtypes simply by mixing in one trait.

In addition to that, there are other drawbacks of using structural types. If a method takes an instance of a structural type as a parameter, then every object having the same members can be passed as a parameter to this method. Once declared in this way, the structural type cannot be constrained in any way by introducing nominal types on top of it, e.g., by specifying some additional inheritance relationships. Hence once introduced to a system, structural types will stay there forever, compromising type safety in the sense explained in the previous paragraph.

That being said, there is also another, simpler way of turning structural subtyping relationships in MSL into nominal ones in Scala, different from the chosen solution. To this end, one could generate a trait for each distinguishable field name and type and then declare records as classes mixing in all the traits for the fields they define. A simplified example of such a solution is shown in listings 5.9 and 5.10. Companion objects are introduced to define a name alias for the type comprising mixed-in traits.

This solution, however, has several drawbacks. First of all, it requires a lot of boilerplate code in the form of trait definitions for every distinct pair of a field name and a corresponding type. Moreover, it truly encodes structural subtyping, bringing to the table all of its weaknesses that have been already discussed.

It is also important to mention that the chosen translation for records has some drawbacks too – most notably its non-locality. In the general case all the code has to be translated at once to resolve the use-sites of structural subtyping and assignments for records. The problem lays in the fact that we cannot translate record definitions separately, because whenever we encounter a use-site of structural subtyping or an assignment for this record, we have to change the translated code for it.

Listing 5.9: MSL: Records v. 2

```
Named =
  record
    name  : String;
  end record;

Person =
  record
    name : String;
    age  : Integer;
  end record;

procedure setName(
  var n : Named;
  newName : String) is
begin
  n.name := newName;
end procedure;

function test() : Boolean is
var
  p : Person;
begin
  setName(p, "name");
  return p.name = "name";
end function;
```

Listing 5.10: MScala: Records v. 2

```
trait StructuralRecord {type T}

trait hasName{var name: String}
trait hasAge{var age : Int}

class Named(var name: String="")
extends hasName

object Named extends
    ➥ StructuralRecord{
  type T = hasName
}

class Person(
  var name : String = "",
  var age : Int = 0 )
extends hasName with hasAge

object Person extends
    ➥ StructuralRecord{
  type T = hasName with hasAge
}

def setName(
  p: Named.T,
  newName:String) {
  p.name = newName
}

def test() = {
  var p = new Person()
  setName(p,"name")
  p.name == "name"
}
```

Translation 3: Records: alternative translation of structural subtyping

A slight modification of the chosen translation can give us the desired locality property. Instead of using trait mixin to model subtyping relationship, we could use an adapter pattern, supported by Scala natively by means of implicits. Translation 4 shows a simplified example of how this can be achieved. In line 25 the MSL procedure `setName` is called with 2 instances of the `Person` record, while it expects instances of `Named`. To model this subtyping relationship the `personToINamed` adapter is introduced, which wraps the `Person` instance and provides the `INamed` interface for it. Similarly the `iNamedToUpdatable` adapter provides an `update` method taking another `INamed` as a parameter, which solves the peculiar assignment semantics for MSL records. This solution, although allows for local translation, is much more verbose that the chosen one. Moreover, there is a certain merit in explicitly stating the subtyping relationships in one place via trait mixin, i.e, it is immediately apparent what are the subtyping relationships in the system. Obviously all the adapters could be placed in one file too, but again – this solution is much more verbose than just enumerating all the traits that are mixed into a class.

5.1.1.5 Cursor

Cursor is a means of expressing type-safe database queries in MSL. It also allows for type-safe data manipulation, i.e., updates, inserts and deletes. As described in Section 4.2.4, cursor has a dual nature: it denotes either a database query definition or a current record of the query's result set, depending on the context. Listing 5.13 shows an example of a cursor definition and its use[1].

Listing 5.13: MSL cursor as a query definition and a current record

```
1   -- query definition
2   read cursor UniStudent is
3   select all from UniStudent
4   order by Name;
5
6   --here UniStudent denotes a query
7   for all UniStudent do
8   -- and here the current record
9     if UniStudent.Name = "Piotr" then
10      return true;
11    end if;
12  end for;
```

[1] All the cursor examples in this section use the database schema introduced in Section 4.2.1

Listing 5.11: MSL: local translation of records

```
type
 Named =
   record
     name  : String;
   end  record;

 Person =
   record
     name : String;
     age  : Integer;
   end record;

procedure setName(
  var n : Named;
  var n2 : Named) is
begin
  n := n2;
end procedure;

function testRecords() :Boolean is
var
  p1 : Person;
  p2 : Person;
begin
  setName(p1, p2);
  return p1.name = p2.name;
end function;
```

Listing 5.12: MScala: local translation of records

```
trait INamed {
  var name : String
}
case class Named(
  var name: String = "")
extends INamed

trait IPerson {
  var name : String
  var age : Int
}
case class Person(
  var name: String = "",
  var age: Int = 0)
extends IPerson

implicit def personToINamed(
   _person : Person) : INamed =
  new INamed{
    def name = _person.name
    def name_=(that : String){
       _person.name = that
    }
  }

implicit def iNamedToUpdatable(
    _named : INamed) =
  new {
  val named = _named
  def update(that : INamed){
    named.name = that.name
    }
  }

def setName(n  : INamed,
            n2 : INamed) {
    n.update(n2)
}
def testRecords(): Boolean = {
    val p1 = Person()
    val p2 = Person(name="p2")
    setName(p1,p2)
    return p1.name == p2.name
}
```

Translation 4: Records: using adapter pattern

Whenever a cursor's field is referenced (like in line 9 of Listing 5.13), the cursor denotes a current record. Otherwise, it denotes a database query.

There are two kinds of cursors – *read* and *readwrite* cursors. A read cursor can only select data from a database, whereas a readwrite cursor can also update, insert and delete data.

Translation

Squeryl [18] is a Scala database library supporting type-safe queries and data manipulation statements. It has been chosen as basis for MScala language integrated queries, as it is considered mature and one of the leading Scala database libraries in the market. Moreover, Squeryl provides query abstractions that are pretty close to MSL cursor, as opposed to the other leading database library – ScalaQuery [14] – that has also been considered as a candidate.

Operations applicable for both read and readwrite cursors

We will start off by describing how to translate operations that can be applied to both read and readwrite cursors. Translation 5 presents a full example utilizing all of these operations. The next paragraphs describe the translations of the individual operations in detail.

MScala defines `MCursor` – a wrapper around the Squeryl's `Query` class, which provides all the operations that can be performed on a cursor in MSL (see line 4 in Listing 5.15). The `Get` function, when applied to a cursor in MSL, executes the query defined by the cursor against the database and sets the current record to be the first record fetched from the database. The `GetNext` function moves the current record pointer to the next record in the result set returned from the database. When applied for the first time to a new cursor, it is semantically equivalent to the `Get` function. Both of these functions return a `Boolean` value indicating whether the operation was successful. The returned boolean value is then, by convention, examined in one of the MSL check procedures ,e.g. `CheckFatal`.

Translation 6, which is an excerpt from Translation 5 (lines 10-20), shows how the MSL functions `Get` and `GetNext` are converted into corresponding MScala methods of the `MSLCursor` class. Moreover, the class offers a `currentRecord` field, which represents the record that the cursor is currently pointing at. In this way `MSLcurosr` distinguishes between the two semantics of cursors in MSL – a query definition and a current record pointer.

Listing 5.14: MSL: read cursor operations

```
function testReadCursorOps() :
    ➥ Boolean is
var
  name : String;
  names : Array[1 .. 2] of String;
  i : Integer := 0;
  read cursor UniStudent is
  select all from UniStudent
  order by Name;
begin
  names[1] := "Anna";
  names[2] := "Jens";
  CheckFatal(Get(UniStudent));
  if UniStudent.Name <> "Anna" then
    return false;
  end if;
  CheckFatal(GetNext(UniStudent));
  if UniStudent.Name <> "Jens" then
    return false;
  end if;
  CheckFatal(Get(UniStudent));
  if UniStudent.Name <> "Anna" then
    return false;
  end if;
  i := 1;
  for all UniStudent do
    if UniStudent.Name <> names[i]
        ➥ then
      return false;
    end if;
    i := i + 1;
  end for;
  CheckFatal(Get(UniStudent));
  i := 1;
  repeat
    if UniStudent.Name <> names[i]
        ➥ then
      return false;
    end if;
    i := i + 1;
  until not GetNext(UniStudent);
  return true;
end function;
```

Listing 5.15: MScala: read cursor operations

```
def testReadCursorOps() :Boolean ={
  var names = MArray[String](1 to
      ➥ 2)
  var i = 0
  val UniStudentQuery = new
      ➥ MSLCursor(
    from(uniStudentTable)( u =>
      select(u)
      orderBy( u.Name)
    )
  )
  names(1) = "Anna"
  names(2) = "Jens"
  checkFatal(UniStudentQuery.get)
  if(UniStudentQuery.currentRecord.
      ➥ Name != "Anna")
    return false
  checkFatal(UniStudentQuery.
      ➥ getNext)
  if(UniStudentQuery.currentRecord.
      ➥ Name != "Jens")
    return false
  checkFatal(UniStudentQuery.get)
  if(UniStudentQuery.currentRecord.
      ➥ Name != "Anna")
    return false
  i = 1
  for(UniStudent<-UniStudentQuery){
    if(UniStudent.Name != names(i))
      return false
    i = i + 1;
  }
  checkFatal(UniStudentQuery.get)
  i = 1
  repeat{
    if(UniStudentQuery.
    currentRecord.Name != names(i))
      return false
    i = i + 1
  } until(!UniStudentQuery.getNext)
  return true
}
```

Translation 5: Operations on read cursor

Listing 5.16: MSL: Get and GetNext

```
1  names[1] := "Anna";
2  names[2] := "Jens";
3  CheckFatal(Get(UniStudent));
4  if UniStudent.Name <> "Anna" then
5    return false;
6  end if;
7  CheckFatal(GetNext(UniStudent));
8  if UniStudent.Name <> "Jens" then
9    return false;
10 end if;
11 CheckFatal(Get(UniStudent));
12 if UniStudent.Name <> "Anna" then
13   return false;
```

Listing 5.17: MScala: Get and GetNext

```
1  names(1) = "Anna"
2  names(2) = "Jens"
3  checkFatal(UniStudentQuery.get)
4  if(UniStudentQuery.currentRecord.
       ➥ Name != "Anna")
5    return false
6  checkFatal(UniStudentQuery.getNext)
7  if(UniStudentQuery.currentRecord.
       ➥ Name != "Jens")
8    return false
9  checkFatal(UniStudentQuery.get)
10 if(UniStudentQuery.currentRecord.
       ➥ Name != "Anna")
11   return false
```

Translation 6: Cursor-related functions: Get and GetNext

MSL provides a `for all` loop for cursors, which first executes the query defined by the cursor against the database and then in every iteration moves the current record pointer to the next record. It gets translated to a for loop in MScala, where for each iteration a separate, locally scoped variable holding the current record is instantiated. See Translation 7 for an example.

Operations on readwrite cursors

MSL provides a way of inserting, deleting and updating data in the database by means of `Put`, `Delete` and `Update` functions respectively. These functions can be called on a readwrite cursor that defines a `select all` query. When calling one of these functions, the cursor parameter denotes the current record.

In MScala, objects that are returned from `select all` queries implement the active record pattern [26], i.e., they provide `put`, `delete` and `update` methods which persist/delete the record in the database. Translation 8 shows an example utilizing `Put` and `Delete` functions to delete all the entries from the `UniStudent` table and to insert 2 new students, named Anna and Jens respectively.

Translation 9 shows an example of updating a `UniStudent` record with `Name = "Anna"` to a new name (`"Piotr"`) and persisting it into the database. When used as a

Listing 5.18: MSL: For all loop

```
1 for all UniStudent do
2   if UniStudent.Name <> names[i]
        ➥ then
3     return false;
4   end if;
5   i := i + 1;
6 end for;
```

Listing 5.19: MScala: For all loop

```
1 for(UniStudent<-UniStudentQuery){
2   if(UniStudent.Name != names(i))
3     return false
4   i = i + 1;
5 }
```

Translation 7: For all iteration over a cursor

Listing 5.20: MSL: Put and Delete functions

```
1  function setUpDb() : Boolean is
2  var
3    readwrite cursor UniStudent is
4      select all from UniStudent;
5  begin
6    for all UniStudent do
7      CheckFatal(Delete(UniStudent));
8      end if;
9    end for;
10   Initialize(UniStudent);
11   UniStudent.Name := "Anna";
12   CheckFatal(Put(UniStudent));
13   Initialize(UniStudent);
14   UniStudent.Name := "Jens";
15   CheckFatal(Put(UniStudent));
16   return true;
17 end function;
```

Listing 5.21: MScala: Put and Delete functions

```
1  def testSetUpDb() : Boolean = {
2    val UniStudentQuery = new
         ➥ MSLCursor(
3     from(uniStudentTable)(select(_))
4    )
5    for (UniStudent <-
         ➥ UniStudentQuery) {
6      checkFatal(UniStudent.delete())
7    }
8    UniStudentQuery.initialize(
         ➥ UniStudent())
9    UniStudentQuery.currentRecord.
         ➥ Name = "Anna"
10   checkFatal(UniStudentQuery.
         ➥ currentRecord.put())
11   UniStudentQuery.initialize(
         ➥ UniStudent())
12   UniStudentQuery.currentRecord.
         ➥ Name = "Jens"
13   checkFatal(UniStudentQuery.
         ➥ currentRecord.put())
14   return true
15 }
```

Translation 8: Put and Delete readwrite cursor functions

parameter to a function, an MSL cursor always denotes a current record. Therefore, in the MScala auto translated code, the `updateStudentName` method takes an instance of the `IUniStudent` trait as a parameter. The trait is present there to model structural subtyping, which MSL cursors are subject to in exactly the same way as described in Section 5.1.1.4 for MSL record types.

Listing 5.22: MSL: cursor updates

```
1  procedure updateStudentName(
2    var student : UniStudent) is
3  begin
4    student.Name := "Piotr";
5  end;
6
7  function testUpdates() : Boolean is
8  var
9    readwrite cursor UniStudent is
10     select all from UniStudent
11     where Name = "Anna";
12 begin
13   CheckFatal(Get(UniStudent));
14   updateStudentName(UniStudent);
15   CheckFatal(Update(UniStudent));
16   return true;
17 end function;
```

Listing 5.23: MScala: cursor updates

```
1  def updateStudentName(
2    student : IUniStudent){
3    student.Name = "Piotr"
4  }
5
6  def testUpdates() : Boolean = {
7    val UniStudentQuery = new
         ➥ MSLCursor(
8      from(uniStudentTable)( u =>
9        where(u.Name === "Anna")
10       select(u)
11     )
12   )
13   checkFatal(UniStudentQuery.get)
14   updateStudentName(UniStudentQuery
         ➥ .currentRecord)
15   checkFatal(UniStudentQuery.
         ➥ currentRecord.update())
16   return true;
17 }
```

Translation 9: Cursor Updates and passing cursors as parameters

Cursor queries

MSL read cursors implement a substantial subset of SQL. The following paragraph describes which SQL features are supported and how they get translated into MScala. As for the MSL cursor limitations, it does not support joins, i.e., a cursor can select data from one database table only.

Translation 10 shows how one can select a number of fields from a database table. It is supported in MScala by means of an anonymous class, which makes the chosen fields available by name.

Listing 5.24: MSL cursor: arbitrary fields selection

```
read cursor StudentName is
  select Name,
         LastName
  from UniStudent;
```

Listing 5.25: MScala : arbitrary fields selection

```
val studentNameQuery = new
    ➥ MSLCursor(
  from(uniStudentTable)( s =>
  select(new {var Name = s.Name;
       var LastName = s.LastName})
  ))
```

Translation 10: Arbitrary fields selection for MSL cursor

An MSL cursor also supports arbitrary `where` clauses as well as the `order by` functionality. Translation 11 shows the corresponding MScala code.

Listing 5.26: MSL cursors: where and order by

```
read cursor MaleStudent is
  select all from UniStudent
  where Gender = 1
  order by Name asc;
```

Listing 5.27: MScala: where and orderBy

```
val maleStudentQuery = new
    ➥ MSLCursor(
  from(uniStudentTable)( s =>
    where(s.Gender === 1)
    select(s)
    orderBy(s.Name asc)
  )
)
```

Translation 11: MSL cursors: where clauses and order by

One of the core SQL functionalities are aggregate functions like `avg`, `max`, `min`, `sum` and `count`, which are all supported by the MSL cursor. Usually, when such an aggregate function is applied, its result is given a name alias through which it can be further referenced. The Squeryl library supports aggregate functions, but it does not allow for giving name aliases to them. To get around this limitation, for every MSL cursor that contains aggregate functions with name aliases, an implicit conversion function in MScala is generated, which acts as an adapter providing the required name aliases.

Translation 12 shows a simple example of a query calculating a grade average for all of the courses. In addition to that, an implicit conversion function is

generated. It can be seen in Listing 5.30 along with an example code of how the query can be used.

Listing 5.28: MSL cursors with aggregate functions

```
read cursor GradeAvarage is
  select avg(Grade) as Value
  from CourseSubscription;
```

Listing 5.29: MScala: cursors with aggregate functions

```
val gradeAvgQuery = new MSLCursor(
  from(courseSubscriptionTable)(
  compute( avg(_.Grade))
  )
)
```

Translation 12: MSL cursors: aggregate functions with name aliases

Listing 5.30: MScala: implicit adapter for name aliases

```
//implicit adapter placed in a special package object
implicit def readCursorCapabilities_GradeAvg_adapter[T]
( result : Measures[Option[T]]) = new {
 def value = result.measures.get
}
//and its use
for(gradeAvarage <- gradeAvgQuery){
  val avg : MReal = gradeAvarage.value // implicit conversion here
  println(avg)
}
```

MSL cursors also allow for utilizing aggregate functions in queries in a more sophisticated way, e.g., together with a `group by` clause. An example of how this can be achieved in MSL and MScala is shown in Translation 13. The corresponding implicit adapter is shown in Listing 5.33, along with an example code of how the query can be used.

5.1.2 Statements

Most of the MSL core statements have their straightforward equivalents in pure Scala, the only exception being the `repeat .. until` loop. MScala, however, provides a custom implementation of this construct, as described in Section 3.2.

Listing 5.31: MSL cursors: aggregate functions and group by

```
read cursor StudentGradeAvg is
  select StudentName,
         StudentLastName,
         avg(Grade) as Value,
         count(Grade) as
              ➥ NumberOfCourses
  from CourseSubscription
  group by StudentName,
       ➥ StudentLastName;
```

Listing 5.32: MScala: aggregate functions and group by

```
val studentGradeAvgQuery = new
    ➥ MSLCursor(
  from(courseSubscriptionTable)(cs
      ➥ =>
  groupBy(cs.StudentName,
          cs.StudentLastName)
  compute( avg(cs.Grade),
           count(cs.Grade))
  )
)
```

Translation 13: MSL cursors: aggregate functions and group by

Listing 5.33: MScala: implicit adapter for name aliases and group by

```
//implicit adapter
implicit def readCursorCapabilities_StudentGradeAvg_adapter[T,U,V,W]
(a : GroupWithMeasures[Product2[T,U],Product2[Option[V],W]]) = new {
   def studentName = a.key._1
   def studentLastName = a.key._2
   def value =  a.measures._1.get
   def numberOfCurses = a.measures._2
}

//and its use
for(studentGradeAvg <- studentGradeAvgQuery){
  println(studentGradeAvg.studentName)
  println(studentGradeAvg.studentLastName)
  println(studentGradeAvg.value)
  println(studentGradeAvg.numberOfCurses)
}
```

Translation 14 establishes the correspondence between MSL core and MScala statements.

Listing 5.34: MSL: statements

```
--assignment
<variable> := <expr>

--if then else
if <Boolean expression> then
  <Statement list>
[ elsif <Boolean expression> then
   <Statement list> ]*
[ else
  <Statement list> ]?
end if



--while
while <Boolean expression> do
  <Statement list>
end while

--repeat until
repeat
  <Statement list>
until <Boolean expression>

--call
<Procedure_name> [ ( [ <parameter
    ➥ list> ]? ) ]?

--return
return <Expression>
```

Listing 5.35: MScala: statements

```
//assignment
<variable > = <expr>

//if then else
if (<Boolean expression>){
  <Statement list>
}
[else if (<Boolean expression>){
    <Statement list>
}]*
[ else {
  <Statement list>
}]?

//while
while (<Boolean expression> ) {
  <Statement list>
}

//repeat until
repeat {
  <Statement list>
} until (<Boolean expression> )

//call
<Procedure_name> [ ( [ <parameter
    ➥ list> ]? ) ]?

//return
return <Expression>
```

Translation 14: Statements

5.1.3 Expressions

Expressions in MSL are characterized by left to right, eager evaluation. The same holds for Scala, unless we use more sophisticated features like lazy vals or

by-name parameters. These features, however, are not utilized for the automatic code translation. Therefore, MSL expressions are simply parsed, then if any of the translation rules described in this chapter applies to any of the operands building the expression, the rule is applied. Finally, the transformed MScala expression tree is pretty-printed according to the algorithm described by Ramsey in "Unparsing expressions with prefix and postfix operators" [27], which takes into account the precedence of operators in Scala, as described in the Scala Language Specification [28].

5.1.4 Passing parameters to functions

5.1.4.1 By-reference parameters

MSL supports passing parameters to functions and procedures both by value and by reference, which is denoted by prefixing the formal parameter with the keyword `var`. In contrast, Scala does not support by-reference parameters, i.e., it is not possible to change a binding to a variable outside of its scope. However, when passing parameters by value in Scala, we pass in fact the address of the parameter (a Scala object) so it is possible to change its state via method calls.

Therefore, in order to emulate passing parameters by reference in Scala, one can wrap it within another object and provide a means of changing the value of the wrapped parameter. The wrapper class used in MScala can be see in Listing 5.36.

Listing 5.36: Scala wrapper for by-reference parameters

```
1 case class MRef [T](var value : T)
```

So, if an MSL variable `v` of type `T` is at least once passed as a parameter to a function `f` that expects a by-reference parameter, the variable must be declared to have a type `MRef[T]` in MScala. Then, in the MScala translation every reference to the variable `v` should be replaced by `v.value`, except when used as an actual parameter to the function expecting a reference type. Naturally, the function `f` must be declared as taking `MRef[T]` in MScala to indicate that it expects by-reference parameter.

Translation 15 shows how the `setString` procedure, expecting the parameter `s` to be passed by reference, gets translated to MScala as well as how it can be

called using by-reference convention.

Listing 5.37: MSL: by-reference parameters

```
1  procedure setString (
2    var s : String;
3    newVal : String) is
4  begin
5    s := newVal;
6  end procedure;
7
8  function testByRef() : Boolean is
9  var
10   str : String := "init";
11   newStr : String := "new string";
12 begin
13  setString(str, newStr);
14  return str = newStr;
15 end function;
```

Listing 5.38: MScala: by-reference parameters

```
1  def setString(s : MRef[String],
2                newVal : String){
3    s.value = newVal
4  }
5
6  def testByRef(): Boolean = {
7    var str = MRef("init")
8    var newStr = "new string"
9    setString(str,newStr)
10   return str.value == newStr
11 }
```

Translation 15: By-reference parameters

5.1.4.2 Parameters as local variables

In MSL, a parameter passed to a function can be seen as a local variable available in the function's scope, i.e., it is possible to assign new values to it. This is different in Scala, since parameters are passed as vals that cannot be assigned new values.

Therefore, whenever in MSL code a parameter passed by value is assigned a new value, a corresponding local variable must be declared in MScala.

Translation 16 shows an example of such a situation. The MSL variable `i` is passed by value and then assigned a new value in line 7 in Listing 5.39. Therefore, it must be declared as a `var` in MScala (Listing 5.40, line 4) and initialized with the $\_i$ parameter, whose name is prefixed by an underscore character to avoid name clash with the newly declared `var`. On the other hand, the $newVal$ parameter is not assigned a new value hence there is no need to declare a `var` for it. The parameter `a` is passed by reference, therefore we can safely assign a new value to it.

Listing 5.39: MSL: parameters

```
1  procedure fillArray (
2    var a  : Array[1..10] of Integer;
3    newVal : Integer;
4    i      : Integer ) is
5  begin
6    --bounds check
7    if i < a'first  or
8       i > a'no_of_elems
9    then
10     i := a'first;
11   end if;
12   while i <= a'no_of_elems do
13     a[i] := newVal;
14     i := i + 1;
15   end while;
16 end procedure;
```

Listing 5.40: MScala: parameters

```
1  def fillArray(a: MRef[MArray[Int]],
2                newVal : Int,
3                _i     : Int){
4    var i = _i
5    // bounds check
6    if(i < a.value.first ||
7       i > a.value.size ) {
8      i = a.value.first
9    }
10   while(i <= a.value.size) {
11     a.value(i) = newVal
12     i = i + 1
13   }
14 }
```

Translation 16: Parameters as local variables

5.2 Optimizations: towards more idiomatic Scala code

Section 5.1 describes how to translate the MSL core constructs in a rather straightforward manner. In this way, we end up with MSL coded in MScala, which is quite often far from how an idiomatic MScala solution to the task at hand would look like. The purpose of this section is to present several optimizations that can make the target MScala code more idiomatic. Chapter 6 further describes how the MSL core to MScala prototype translator supports these optimization refinements of the target code.

5.2.1 Inlining variable declarations

Variables in MSL must be declared at the beginning of a function/procedure. While such an approach simplifies building the compiler for a language, it is not particularly comfortable to work with. When a variable is defined just before its use, it is much easier to find its definition whenever needed. Moreover, it is clear from the code which value the variable is assigned whereas when the variable is

defined at the beginning of a long function, it is easy to overlook references to it and therefore hold wrong assumptions about its state.

Moreover, in Scala `vars` and `vals` must be assigned an initial value explicitly, whereas in MSL this is implicit when an initial value is not specified. It is, however, quite often the case that variables in MSL are initialized explicitly, which makes a literal translation from MSL to Scala rather verbose. Translation 17 shows an example of such a situation. Variables that start with "`yes`" could be easily defined inline, i.e., their definition could be combined with the first assignment to it. On the contrary, the definitions of the variables that start with "no" cannot be combined with the first assignment to it, since they are further referenced in a scope outside of the first assignment.

In order to make the translated code look and feel more like idiomatic Scala, one can combine the definitions of variables with the first assignments to them whenever it is safe to do so. The definition of "safe" is in this case two-fold:

1. The first assignment must be at the same time a first reference to the variable in question (as it appears statically in the code), and
2. Assuming that the declaration is moved to where the first assignment to the variable is, there cannot be any references to the variable in the scopes outside the scope of the first assignment.

Translation 18 shows a result of such an optimization.

5.2.2 Translating non-reassignable vars to vals

Because of its functional roots, immutability is always the first choice in Scala whenever possible. Therefore, it is desired to structure code in a way that only `vals` (equivalent to constants or final values in other programming languages) are used. Variables in MSL are semantically equivalent to vars in Scala in a sense that `vars` are reassignable symbols referring to objects in memory. Hence a straightforward translation would turn all the MSL variables into Scala `vars`. But whenever a variable is assigned a value only once, it can be safely turned into a `val`, which gives compiler guarantees that the variable will not be reassigned a new value in the code.

In lines 2–4 of Listing 5.5 the MSL variables are translated into Scala `vars`. However, the variables `a` and `newElem` are not assigned new values after being initialized, therefore they can be turned into `vals`. Translation 19 shows the result of this optimization.

Listing 5.41: MSL: variable definitions

```
function VariablesRewrites (
  paramToDeclareVar : Integer;
  param: Integer ) : Integer is
var
  yes1 : Integer;
  yes2 : Integer;
  yes3 : Integer;
  yes4 : Integer;
  yes5 : String;
  no1  : Integer;
  no2  : Integer;
  no3  : Integer;
begin
  yes1 := param;
  while yes1 > 1 do
    no1 := yes1;
    yes5 := "111";
    if no1 > no2 then
      yes2 := yes1;
      yes5[1] := "X";
      if 1 > 2 then
        no3 := 1;
        yes3 := 1;
      elsif 2 > 3 then
        yes4 := 4;
      else
        no3 := 2;
        paramToDeclareVar := 7;
      end if;
    end if;
  end while;
  no2 := 10;
  return no1;
end function;
```

Listing 5.42: MScala: variable definitions

```
def VariablesRewrites (
  _paramToDeclareVar : Int,
  param : Int ): Int = {

 var paramToDeclareVar =
     ➥ _paramToDeclareVar
 var yes1 = 0
 var yes2 = 0
 var yes3 = 0
 var yes4 = 0
 var yes5 = ""
 var no1 = 0
 var no2 = 0
 var no3 = 0
 yes1 = param
 while(yes1 > 1) {
   no1 = yes1
   yes5 = "111"
   if(no1 > no2) {
     yes2 = yes1
     yes5 = yes5.update1based(1,'X'
         ➥ )
     if(1 > 2) {
       no3 = 1
       yes3 = 1
     } else if(2 > 3) {
       yes4 = 4
     }
     else {
       no3 = 2
       paramToDeclareVar = 7
     }
   }
 }
 no2 = 10
 return no1
}
```

Translation 17: Straightforward translation of variable definitions with no inlining

Listing 5.43: MSL: inline variables

```
function VariablesRewrites (
  paramToDeclareVar : Integer;
  param: Integer ) : Integer is
var
  yes1 : Integer;
  yes2 : Integer;
  yes3 : Integer;
  yes4 : Integer;
  yes5 : String;
  no1  : Integer;
  no2  : Integer;
  no3  : Integer;
begin
  yes1 := param;
  while yes1 > 1 do
    no1 := yes1;
    yes5 := "111";
    if no1 > no2 then
      yes2 := yes1;
      yes5[1] := "X";
      if 1 > 2 then
        no3 := 1;
        yes3 := 1;
      elsif 2 > 3 then
        yes4 := 4;
      else
        no3 := 2;
        paramToDeclareVar := 7;
      end if;
    end if;
  end while;
  no2 := 10;
  return no1;
end function;
```

Listing 5.44: MScala: inline variables

```
def VariablesRewrites (
  _paramToDeclareVar : Int,
  param : Int ): Int = {
  var no1 = 0
  var no2 = 0
  var no3 = 0
  var yes1 = param
  while(yes1 > 1) {
    no1 = yes1
    var yes5 = "111"
    if(no1 > no2) {
      var yes2 = yes1
      yes5 = yes5.update1based(1,'X
          ➥ ')
      if(1 > 2) {
        no3 = 1
        var yes3 = 1
      } else if(2 > 3) {
        var yes4 = 4
      }
      else {
        no3 = 2
        var paramToDeclareVar = 7
      }
    }
  }
  no2 = 10
  return no1
}
```

Translation 18: Inlining variable definitions

Listing 5.45: MSL: arrays

```
function testArray() : Boolean is
var
    a : Array[1..10] of Integer;
    i : Integer;
    newElem : Integer := 7;
begin
  i := a'first;
  while i <= a'no_of_elems do
      a[i] := newElem;
      i := i + 1;
  end while;
  i := a'first;
  while i <= a'no_of_elems do
      if a[i] <> newElem then
        return false;
      end if;
      i := i + 1;
  end while;
  return true;
end function;
```

Listing 5.46: MScala: arrays

```
def testArray(): Boolean = {
  val a = MArray[Int](1 to 10)
  var i = 0
  val newElem = 7
  i = a.first
  while(i <= a.size) {
    a(i) = newElem
    i = i + 1
  }
  i = a.first
  while(i <= a.size) {
    if(a(i) != newElem) {
      return false
    }
    i = i + 1
  }
  return true
}
```

Translation 19: Operations on arrays – turning non-reassignable vars to vals

5.2.3 Removing unnecessary MRef types

As described in Section 5.1.4.1, whenever an MSL function declares a formal parameter of type `T` to be passed by reference, then the translated formal parameter type in MScala is `MRef[T]`. Such a translation preserves the semantics of MSL, but can lead to Scala code that is more verbose than really needed.

For instance, arrays and records in MSL must be passed by reference whenever used as parameters in functions. This requirement is imposed mostly for efficiency reasons, i.e., it would be simply too expensive to copy entire records or arrays if they were passed by value. In most of the cases though, it is not the intention to change the binding to a variable in the calling code (which is essentially the semantics of by-reference parameters) but to simply pass a pointer to the record/array into the function so that the record's fields or the array's contents can be changed.

This is perfectly aligned with the Scala pass by value semantics as far as objects are concerned. Therefore, if a variable is passed by reference in the MSL code but the receiving function does not reassign a new value to the variable, the formal parameter type in MScala can be simplified from `MRef[T]` to `T`.

Translation 20 shows a non-optimized example of functions/procedures, which expect by-reference parameters, but do not reassign new values to these parameters. The parameter types get translated to `MRef[T]`, although it is unnecessary. Translation 21 shows an optimized version, where the unnecessary `MRef[T]` type wrappers are removed.

5.3 Conclusions

In this chapter we have described how to automatically translate MSL core to MScala. All of these transformations, except for the MSL cursor part, have been implemented in the prototype translator that is delivered with this thesis. The translator has been tested using a variety of different input files, for which it produced correct results. Most of the MSL test functions were written in a way that they would take no parameters and return `true` if a function behaves as expected. Thus, testing these auto-translated functions in MScala was based on calling them and checking whether they still return `true`. As for the original Maconomy MSL code that was a subject to translation, the respective auto-translated MScala code was examined manually to determine its correctness.

Listing 5.47: MSL: parameters

```
function isOne(var i : Integer) :
    ➥ Boolean is
begin
  return i = 1;
end function;

function id(var ix : Integer) :
    ➥ Integer is
begin
  isOne(ix);
  return ix;
end function;

procedure setArray(
  var a : Array[1..10] of Integer;
  newElem : Integer) is
var
  i : Integer;
begin
  i := a'first;
  while i <= a'no_of_elems do
    a[i] := newElem;
    i := i + 1;
  end while;
end procedure;

function testArray() :Boolean is
var
  a : Array[1 .. 10] of Integer;
  i : Integer;
begin
  setArray(a,10);
  i := a'first;
  while i <= a'no_of_elems do
    if a[i] <> 10 then
      return false;
    end if;
    i := i + 1;
  end while;
  return true;
end function;
```

Listing 5.48: MScala: parameters

```
def isOne(i: MRef[Int]): Boolean ={
  return i.value == 1
}

def id(ix : MRef[Int]): Int = {
  isOne(ix)
  return ix.value
}

def setArray (
  a : MRef[MArray[Int]],
  newElem : Int) {
  var i = a.value.first
  while(i <= a.value.size) {
    a.value(i) = newElem
    i = i + 1
  }
}

def testArray(): Boolean = {
  val a = MRef(MArray[Int](1 to 10)
      ➥ )
  setArray(a,10)
  var i = a.value.first
  while(i <= a.value.size) {
    if(a.value(i) != 10) {
      return false
    }
    i = i + 1
  }
  return true
}
```

Translation 20: Straightforward translation of by-reference parameters

Listing 5.49: MSL: parameters

```
function isOne(var i : Integer) :
    ➥ Boolean is
begin
  return i = 1;
end function;

function id(var ix : Integer) :
    ➥ Integer is
begin
  isOne(ix);
  return ix;
end function;

procedure setArray(
  var a : Array[1..10] of Integer;
  newElem : Integer) is
var
  i : Integer;
begin
  i := a'first;
  while i <= a'no_of_elems do
    a[i] := newElem;
    i := i + 1;
  end while;
end procedure;

function testArray() :Boolean is
var
  a : Array[1 .. 10] of Integer;
  i : Integer;
begin
  setArray(a,10);
  i := a'first;
  while i <= a'no_of_elems do
    if a[i] <> 10 then
      return false;
    end if;
    i := i + 1;
  end while;
  return true;
end function;
```

Listing 5.50: MScala: parameters

```
def isOne(i : Int): Boolean = {
  return i == 1
}

def id(ix : Int): Int = {
  isOne(ix)
  return ix
}

def setArray(
  a : MArray[Int],
  newElem : Int) {
  var i = a.first
  while(i <= a.size) {
    a(i) = newElem
    i = i + 1
  }
}

def testArray(): Boolean = {
  val a = MArray[Int](1 to 10)
  setArray(a,10)
  var i = a.first
  while(i <= a.size) {
    if(a(i) != 10) {
      return false
    }
    i = i + 1
  }
  return true
}
```

Translation 21: Optimized translation of by-reference parameters

The optimized translations defined in this chapter usually result in the target MScala code that is at least as concise as the source MSL code. There are rare cases, however, where the reconciliation of substantial differences between the two languages required a fair amount of boilerplate code. For instance, in order to emulate structural subtyping in MSL by nominal subtyping in MScala, classes and cursors mix-in a number of traits that were not present in the source MSL code. This solution is arguably more verbose, but at the same time brings to the table all the benefits of nominal subtyping, which can be considered a good thing. Another example of not necessarily pleasant to read auto-translated boilerplate code is the implicit adapters generated for MSL cursors containing aggregate functions, as shown in Listing 13. This boilerplate code, however, can be placed in some predefined package object so that the developers do not see it unless they are very curious.

It is important to bear in mind that by the help of top-class tooling as well as the flexibility of the language, the auto translated MScala code can be much easier refactored compared to MSL.

Chapter 6

Architecture of the MSL core to MScala translator

A source to source translator can be seen as a special kind of compiler, since it takes a source code in one programming language and outputs semantically equivalent target code in another programming language. In case of traditional compilers, the output language is a low-level language (e.g. byte-code or machine code), whereas translators emit code in a higher-level language.

Compiler construction has been a vary vital and lively research field for the last 50 years, with a lot of focus on algorithms performing certain semantic analysis phases, optimization phases etc. As for the architecture of a compiler, however, there does not seem to be a state-of-the-art way of structuring a compiler. In particular, Lambda the Ultimate – one of the most popular scientific on-line forums that has to do with programming languages design – hosts a very interesting discussion about what is known as the abstract syntax tree (AST) typing problem [29]. The problem is basically about how to structure a compiler in a statically typed language so that the consecutive compiler passes are guaranteed to accept only a valid version of the AST, that has already undergone particular analysis phases and perhaps optimization phases. The solution should be type-safe, i.e., the validity of the input AST for a particular phase should be guaranteed by its type, which denotes that the AST has the required form and all the attributes that are needed for the phase in question.

This problem is obviously valid for source to source translators. In addition to that, there are also other important requirements. A translator should be extensible – i.e it should be easy to define new analysis and transformation phases and integrate them with the existing ones. Moreover, the architecture should be clear and testable. Translation rules should be composable in a sense that it should be possible to define simple and easily understandable translation rules that can be combined together in a clear and transparent way.

The MSL core to MScala translator is built on top of the architecture that meets all of these requirements. It incorporates state-of-the-art concepts in compiler construction such as term rewriting [3], attribute grammars [2] and pretty printing [30]. All of these powerful concepts are implemented in the Kiama language processing library [31] as internal Scala DSLs, which is in itself a strong evidence of how flexible Scala is as a host language for embedding DSLs [19].

6.1 Architecture overview

When building a source to source translator, there are some essential steps that cannot be skipped. First, the source code has to be parsed to obtain an AST. Then, the AST is analyzed, rewritten – sometimes to a completely different intermediate representation (*IR)* – and finally pretty printed.

Figure 6.1 shows a high-level overview of the MSL core to MScala translator's architecture. The MSL source code is parsed by means of Scala parser combinators [9]. The parser outputs an immutable, MSL specific AST, which is further decorated with different *attributes* that store the results of semantic analysis passes like symbol resolving or type checking. Once this information is calculated, it can drive the translation into the MScala AST, which is carried out by means of *rewrite rules* that apply to particular MSL nodes and turn them into corresponding MScala nodes.

Once this step is completed, we obtain an immutable MScala AST that is the result of all the all the straightforward translations described in Section 5.1. If the AST was pretty printed at this stage, we would get the straightforward, non-optimized translation. The architecture, however, supports successive rewriting of the MScala AST in order to obtain optimized, more idiomatic MScala code, as described in Section 5.2. This can be easily achieved by adding new attributes, possibly referencing the existing ones, formulating new rewrite rules and composing new optimization phases from them.

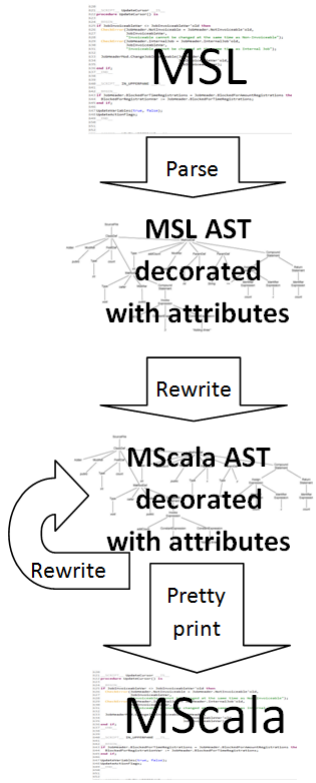

Figure 6.1: Overall architecture of the MSL core to MScala translator

6.2 Attribute grammars

Attribute grammars allow for adding new attributes to immutable AST nodes, essentially solving the AST typing problem described in the beginning of this chapter. Let us consider an example: a symbol resolution phase in a compiler binds symbols to their respective definitions. Once carried out, all the symbols in the program (e.g. variable references, function calls, type constructors) should have references to their respective definitions. Here, the AST typing problem comes into play – how do we add new attributes to the AST in a type-safe manner, so that a transformation that requires an AST after the symbol resolution can express it in its type signature?

One solution is to have multiple versions of the AST – actually, a separate version for every analysis phase performed by the compiler. This is not very

maintainable and can easily lead to a high number of versions of nearly the same thing. On the other hand, we could have just one version of the AST with mutable placeholders for attributes to be computed later on. But in this way we give up type safety, since it is impossible to determine at compile-time whether the required attributes have been computed or not.

The solution that Kiama advocates is to represent attributes as memoised functions, i.e functions the results of which are cached, from AST nodes to attribute values [32]. In their definitions, attributes usually refer to other attributes defined for the neighboring nodes. Attributes defined in this way are lazy – nothing is computed unless requested. The laziness (or equivalently, demand-driven evaluation) takes care of dependencies between attributes, so one does not need to explicitly schedule AST traversals.

Listing 6.1 shows an example of a `subprogramDecl` (subprogram declaration) attribute definition for a `CallStm` (call statement) AST node, which computes a `MslSubprogramDef` (subprogram definition) for the given call statement as a part of symbol resolution.

Listing 6.1: Attribute definition - subprogram declaration for a call stm

```
1  val subprogramDecl : CallStm => MslSubprogramDef =
2      attr {
3        //attributes are defined as partial functions applicable for
4        //particular AST nodes  - in this case a call statement node
5        case call @ CallStm(name, args) =>
6        (call->subprogramLookup(name)).getOrElse(UnknownSubprogramDef)
7      }
8
9  private val subprogramLookup : String => ASTNode => Option[
       ➥ MslSubprogramDef] = {
10     paramAttr{  //we define an attribute with a parameter
11       name => { // the String parameter is assigned to name
12         node => {  //the ASTNode parameter is assigned to node
13           (node->enclosingSubprogram)
14            .parent[ModuleDef]
15            .subprograms.find (_.name == name)
16         }
17       }
18     }
19   }
```

The `subprogramDecl` attribute definition delegates the computation to another attribute – `subprogramLookup` – also defined for the `CallStm` node, which searches

through all the subprogram definitions in the entire module. As already mentioned, these attributes are computed in a lazy manner and then cached in a hash map to avoid potentially expensive recomputations.

It is also trivial to add new attributes to the AST. They can either reference other attributes or be computed independently. This is type-safe, because once we reference an attribute of the required type, its value is guaranteed to be computed on demand and to remain constant.

6.3 Strategy-based term rewriting

Kiama implements the concepts defined in *Stratego* - a successful term rewriting language based around the concept of generic tree traversals [33]. Its semantics [34] is naturally suited to a combinator-style implementation where a strategy is implemented as a function that takes a subject term as argument and returns either a transformed term or a failure indication.

Listing 6.2 presents an example of a function that rewrites simple arithmetic expressions. It first defines 3 rewrite rules – `multToShiftRule`, `simplifyNeg` and `AddToMul`. The `multToShiftRule` rule rewrites multiplications by the power of two into corresponding shift left nodes. It is based on the function already defined in Listing 3.4 in Section 3.1. The other two rules are just partial functions matching particular AST nodes and defining simple rewrites. The individual rules are then combined in line 21 by means of the `+` functional combinator that has the same semantics as the `OR` operator in case of boolean expressions.

`Everywherebu` is a combinator that applies its argument to every sub-term of the subject term in a bottom up fashion – i.e it starts with the leaf nodes and proceeded further up the expression tree. Finally, `rewrite` applies its argument strategy to an expression and, if it succeeds, returns the resulting expression, otherwise it returns the original expression.

The goal of this example was to show how small rewrite rules can be combined together by means of functional combinators to perform more complex transformations. Kiama offers plenty of different predefined combinators that can fit almost every need as far as strategy based rewriting is concerned. If unavailable, it is rather easy to define new custom combinators.

In this way we achieve what we call *composable translation rules*. One can easily plug-in a new translation rule by combining it with the existing ones by means of a combinator with the required semantics. This enables extending the

Listing 6.2: Kiama rewrite rules

```
def simplify : Exp => Exp = {
  def mulToShift: Term => Term = {
    case Mul(Shl(x,n),Num(y)) if y % 2 == 0
      => mulToShift(Mul(Shl(x,n+1),Num(y/2)))
    case Mul(x,Num(y)) if y % 2 == 0
      => mulToShift(Mul(Shl(x,1),Num(y/2)))
    case Mul(Num(x),y) => mulToShift(Mul(y,Num(x)))
    case Mul(x, Num(1)) => x
    case e => e
  }
  //make a rule out of the mulToShift function
  val mulToShiftRule = rulef(mulToShift)
  val simplifyNeg = rule{
    case Neg(Neg(x)) => simplify(x)
  }
  val AddToMul = rule {
    case Add (x, y) if x == y => simplify(Mul(Num(2), x))
  }
  // rewrite everywhere from bottom to up, applying
  // simplifyNeg or AddToMul or mulToShiftRule to every node
  rewrite(everywherebu( simplifyNeg + AddToMul + mulToShiftRule))
}
```

translator with new language features as well as new optimizations.

6.4 MSL core to MScala translator

This section describes how attribute grammars and strategy-based rewriting are utilized in the MSL core to MScala translator.

The translator defines two versions of an abstract syntax tree – MSL and MScala specific, both being fully immutable. The first translation phase transform the MSL AST into MScala AST, which corresponds to the straightforward translations defined in Section 5.1. Each consecutive phase rewrites the immutable MScala AST, incorporating the optimizations described in Section 5.2. This approach is therefore purely functional – every translation phase take an immutable AST as a parameter and returns a rewritten immutable AST.

Typically, in order to perform a translation phase we need to carry out some sort of semantic analysis and compute relevant attributes. Then, the rewrite rules composed into rewrite strategies can reference the defined attributes. When extending the translator, we can either add a new translation phase or hook into the existing ones. Either way, adding new attributes to drive the translation does not take more than simply defining the attributes, possibly by referencing other attributes. Then, we can either rewrite the AST in a new pass, which has no influence on the existing functionality, or hook into an existing pass by combining our rewrite rules with the ones that already make up the pass. In the latter case we have to be cautious though not to interfere with the existing rules.

6.4.1 Architecture of the translator – properties

The proposed architecture, which the MSL to Scala prototype translator is based on, allows for building composable translation phases out of composable translation rules and further for combining the translation phases to define the actual translation. This architecture is extensible across two axes: it enables adding new source language features as well as new translation phases, which can implement optimizations leading to more idiomatic target code.

Moreover, our functional approach to translation combined with attribute grammars essentially solves the AST typing problem. When composing a new translation phase out of individual translation rules, we can specify in a type-safe manner which attributes have to be present (computed) in the AST. Because attributes are just higher order functions, they are computed on demand and their results are cached. Therefore, referencing an attribute in a translation rule is type safe, as its value is guaranteed to be computed on demand.

The possibility of plugging in and out individual translation rules and the entire phases offers great scalability, extensibility and testability. Moreover, such an architecture is very clear and easier to comprehend for humans than doing everything in one pass.

On the other hand, such a flexibility does not come for free. There is a certain performance overhead associated with storing attributes in a hash map rather than directly in the AST nodes as fields. Moreover, doing everything in one go is certainly faster than splitting the translation into consecutive translation phases. We believe, however, that the flexibility, type-safety and extensibility of the proposed architecture are definitely worth this little performance overhead, especially taking into account that one of the most possible scenarios of using the translator in migrating all the MSL code at once.

6.5 Conclusions

Due to the high flexibility and extensibility of the MSL core to MScala translator described in this chapter. it should be rather straightforward to extend it in order to cover the full MSL language. In addition to that, more optimization phases can be easily defined so that the target MScala code is more idiomatic.

Chapter 7

Discussion

As mentioned in the Introduction, the primary objective of this thesis was to investigate whether Scala could be a good replacement for MSL. In this chapter we evaluate the work that has been done, summarize the arguments given in the previous chapters as well as provide some additional considerations to finally draw the conclusion that is is indeed sensible and recommended to migrate the MSL code base into Scala. Not only do we make such a recommendation, but also provide an initial implementation of MScala along with the prototype MSL core to MScala translator, which can be relatively easily extended to cover the full MSL language.

7.1 Why migrate MSL to another existing language

Let us first flip the question that this section seeks to answer and describe what would be the benefits of keeping the Maconomy business logic code base in MSL. Developing and maintaining a programming language in-house gives the company full control over it. Requirements such as adding new features to the language, migrating it to a new platform etc. are feasible and within reach of

the company, provided that it is able to allocate enough resources to make it happen.

MSL is a very restricted language with a relatively small number of features, which makes the project of writing a translator from MSL to another language doable within a sensible time frame. Such a migration is like buying a one-way ticket though, because once the code is migrated to, say, Scala, building a translator from Scala to another language suddenly becomes a huge task, requiring enormous resources and strong expertise in programming languages theory as well as compiler construction.This is certainly a risk that should be taken into account when deciding whether to translate MSL into Scala or not.

That being said, risk management is all about balancing the benefits of a particular action with the risk associated with it. One of the most important parameters when assessing the risk of an action is the likelihood of it to occur as well as its consequences. The following sections summarize the main benefits of migrating the MSL code base into Scala. In essence, this will certainly strengthen Deltek's long term competitive advantage by making the developers faster in bringing new features to the market as well as gradually improving the architecture of the Maconomy system in terms of scalability, maintainability and reusability. Suppose the MSL code has been migrated to Scala and in five years time other programming languages advance to such extend, that in order to keep up with the competitors, the Scala code must be migrated again to one of these new languages. First of all, Scala is one of the most innovative languages today so this is very unlikely to happen. But assuming it did happen, what would be the chances of Maconomy to be ahead of the competitors with the 30 years old MSL on board, if Scala was too obsolete of a language and had to be migrated to something more modern? In this scenario, sticking to MSL would simply mean gradually loosing competitive advantage in favor of more innovative competitors for the next five years, until a point when it might be simply too late to innovate and thus survive. It is also worth pointing out that migrating MSL to Scala really means migrating it to the JVM, thus enabling seamless interoperability with other JVM languages. Therefore, it would be possible to switch business logic development language from Scala to some other JVM language in the future, if such a requirement emerges. What is more, the efforts of enabling Scala for the .NET platform are in a very advanced stage [35] which also opens up a possibility of switching to a .NET language instead.

Let us now come back to the initial question: why migrate MSL to another language? First of all, maintaining a language in-house is expensive. If having this language does not provide any competitive advantage or, even worse, the language is inferior to other open source programming languages available on the market, then from the pure business standpoint it does not make any sense at all to keep it. Moreover, developing a top-quality tool support, such as a

decent IDE is even more expensive than maintaining the language itself and, to be done well, requires a lot of expertise. As we have shown in the course of this thesis, Scala has a number of advantages over MSL. These advantages will be summarized in the following sections. But even if Scala was just as good of a language as MSL, it would pay off to migrate MSL to Scala just to be able to leverage the top-class tool support it offers along with an uncounted number of open source libraries and frameworks in the JVM ecosystem. In addition to that, although Scala is still a fairly new language, it is now gaining momentum and it is certainly easier to find a Scala developer on the market compared to finding a former Maconomy's employee, who happens to know MSL. Currently, every newly hired MSL developer must be trained from scratch in using the language, which is both costly and time consuming.

7.2 Advantages of MScala as a replacement for MSL

As shown in Chapter 4, idiomatic MScala solutions to the typical problems from the Maconomy domain tend to be much more concise and elegant than the corresponding MSL solutions. Moreover, advanced object oriented as well as functional concepts in Scala, explained in Chapters 3 and 4, make it possible to build generic and reusable software components, which is basically infeasible in MSL. Not only will it lead to further reduction in terms of lines of code, but it will also make the code base easier to comprehend, more extensible and easier to maintain.

Chapter 3 emphasizes that Scala provides a very lightweight way of implementing library-based DSLs that look and feel like native language constructs. It gives the application programmers a means of gradually raising the level of abstraction when expressing business logic concepts in the Maconomy system. In this way the domain specialists can easily enrich MScala with new domain specific constructs, which will both make them more efficient in implementing new functionality, as well as help their new colleagues get started much faster and soon become productive programmers.

In addition, Typesafe [22], the company that is commercially supporting Scala, is currently putting a lot of effort into improving the Eclipse plug-in for Scala, which is already a top-quality development environment. Having a decent support for refactoring is of great significance when maintaining and evolving a very large code base like in the case of the existing MSL code.

7.3 MSL core to MScala translator

Not only does this thesis precisely describe how to translate each construct included in MSL core, but it also provides a prototype implementation of the MSL core to MScala translator. As shown in Chapter 5, the target MScala code is in most of the cases at least as concise as the source MSL code. What is more, the implemented translator supports composable translation phases so that new optimization phases leading to more idiomatic (and hence concise) MScala code can easily be plugged in. The translator can also be extended to handle new MSL features so that eventually the full MSL language will be covered. This can be achieved by means of composable translation rules.

7.4 Future work

The problem of migrating the existing MSL code base into Scala has several dimensions to it. They range from business and risk management considerations, through designing a good DSL embedded in Scala, quality assurance, i.e, making sure that the translation produces semantically equivalent target code, up to possible automatic refactoring of the code while performing the translation.

Due to time constraints, this thesis has been mostly focusing on showing that it is indeed possible to translate the core part of MSL into MScala. We developed a prototype of an MSL core to MScala translator, which is extensible both in terms of adding new MSL features to be handled, as well as new optimization phases that turn the target code into more idiomatic Scala. Some efforts have been also dedicated to showing that the typical implementation problems from the Maconomy domain can be solved in MScala in a much more concise and elegant manner compared to MSL. Moreover, we have discussed the features of Scala that support building reusable software components, which – in practice – is impossible in MSL.

We believe that this thesis is a good foundation for migrating the MSL code into Scala. There is, however, still a lot of work to be done to complete the transition. First of all, there is a very important business aspect to it, namely, how to make the transition as smooth as possible to avoid a drastic decrease in the productivity of application developers. This might also have an impact on the requirements for the translator. If a one-time translation is to take place, then the performance requirements towards the translator are not so important. Moreover, the translator can assume to have valid MSL code as input, i.e., the code that has been successfully compiled by the MSL compiler. On the other

hand, if the translator is going to act as a cross-language compiler for a longer period of time, during which the development is kept both in MSL and Scala (i.e. everything is ultimately translated to Scala and compiled down to the Java bytecode) then good performance of the translator is a very significant factor. Moreover, in this scenario the translator must assume additional responsibilities for performing all of the semantic analysis that used to be done by the MSL compiler.

In such a migration project the quality assurance aspect is of critical importance. Our approach of building translation phases out of composable translation rules and of composing translation phases to define the actual translation has a great potential for testability. This area, however, was not given much attention in this thesis and should be explored further. Ideally, one could build a testing framework that would be able to execute an MSL function against some predefined (or auto-generated) data, then execute the corresponding auto-translated MScala function and compare their results, which should be identical.

Finally, perhaps the most obvious and already mentioned area for future work is to extend the translator to cover the full MSL language and to implement more optimizations to increase the quality of the target MScala code.

7.5 Related work

In a narrow sense this thesis is unique as it solves a problem that has not been even tackled before, namely, the problem of translating MSL into Scala. In a broader sense, however, our work was very much focused on two research areas: embedding domain specific languages in Scala as well as compiler construction, with building source to source translators being particularly important.

When it comes to building source to source translators, Terrence Par discusses various ways of going about this task in his book "Language implementation patterns" [36]. He presents three basic strategies that one can take: *syntax-directed*, *rule-based* and *model-driven* translations. Our approach incorporates both rule-based and model-driven strategies.

Stratego/XT [33] is a language and toolset for program transformation, based on the concept of *rewrite rules*. This concept has been incorporated into Kiama – the language processing library that has been used to implement the MSL core to MScala translator.

JastAdd [37] is another popular system for program transformation. It ad-

vertises itself as a metacompilation system for generating language-based tools such as compilers, source code analyzers, and source to source translators. JastAdd defines a declarative object oriented language based on reference attribute grammars, which have also been included in the Kiama library.

When it comes to embedding domain specific languages in Scala, there is a number of very successful examples in both research and industrial applications. The Kiama language processing library provides internal Scala DSLs, which express essentially the same concepts as the external DSLs that they have been inspired by, such as Stratego/XT or JastAdd [19].

Scala itself comes along with two very powerful DSLs – *actors* [8] and *parser combinators* [9]. Moreover, the Squeryl library, which has been used as a basis for MScala cursor implementation, is itself a DSL embedded in Scala [18].

As for examples of application in some different domains, *Delite* [15, 38] is a research project from Stanford University's Pervasive Parallelism Laboratory (PPL). Delite is a compiler framework and runtime for parallel embedded DSLs. To enable the rapid construction of high performance, highly productive DSLs, Delite provides several facilities, such as code generators for Scala, C++ and CUDA, built-in parallel execution patterns, optimizers for parallel code as well as a DSL runtime for heterogeneous hardware. OptiML [39] is a DSL for machine learning based on Delite.

Chapter 8

Conclusion

We have designed and implemented MScala – an internal Scala DSL that is well-suited for expressing business logic in the Maconomy system. Compared to MSL, solutions to the typical implementation problems from the Maconomy domain tend to be much more concise and elegant when expressed in MScala. The new DSL is also suitable for being a target language for an automatic code translation from MSL core. Not only have we shown and described precisely how to carry out the translation, but we have also provided a prototype implementation of the MSL core to MScala translator. The implemented translator is extensible across tho axes: it makes it easy to add new MSL features to be handled as well as to plug in new optimization phases leading to more idiomatic MScala code.

Abstracting from the Maconomy context, we believe that the architecture which the MSL core to MScala translator is based on is a significant contribution in itself. It is easy to reason about due to its functional nature – the consecutive translation phases gradually rewrite an immutable AST. It essentially solves the AST typing problem by utilizing the concept of reference attribute grammars. Moreover, the properties of composable translation rules and translation phases provide great flexibility, testability and extensibility.

These contributions are not only a proof of concept that it is indeed possible to create a good internal Scala DSL for expressing business logic in Maconomy and to achieve a high quality automatic translation. They also comprise a solid

foundation to build on top of, if Deltek decides to carry out the full migration of the Maconomy MSL code base to Scala.

Appendix A

MSL core grammar

Listing A.1 shows the MSL core grammar, expressed in terms of notation used by parser combinators. The notation is essentially equivalent to EBNF.

Listing A.1: MSL core grammar

```
/**
 * Meaning of symbols:
 *  ~       : sequential composition
 *  opt(x)  : x is optional
 *  rep(x)  : repetition of x ( 0 or more times)
 *  a | b   : deterministic choice - a or b
 *
 *  Note that MSL is CASE INSENSITIVE
 */
class MSLParser extends JavaTokenParsers{

   /**********************************************
    *        Modules
    *********************************************/
  def moduleDef =
    "module"~ident~"is"~opt("type"~rep(recordDef~";"))~rep(subprogramDef)~
        ➥ "end"

  def recordDef =
```

```
19      ident~"="~"record"~rep(ident ~ ":"~mslType~";")~"end"~"record"
20
21    /*********************************************
22     *        Subprograms
23     *********************************************/
24    def subprogramDef = procedureDef | functionDef
25
26    def procedureDef =
27      "procedure"~ident~opt("("~opt(formalParamList)~")")~"is"~block~"
            ➥ procedure;"
28
29    def functionDef =
30      "function"~ident~opt("("~opt(formalParamList)~")")~":"~mslType~"is"~
            ➥ block~"function;"
31
32    def formalParamList =
33      repsep(paramDef,";")
34
35    def paramDef =
36      opt("var")~ident~":"~mslType
37
38    def block =
39      declarativePart~statementPart
40
41    def declarativePart  =
42      opt(varDefs)~opt(constDefs)
43
44    def varDefs =
45      "var"~> rep(varDef~";")
46
47    def constDefs  =
48      "const"~> rep(varDef~";")
49
50    def varDef =
51      ident~":"~mslType~opt(":="~literal)
52
53    def constDef =
54      ident<~":"~mslType~":="~>literal
55
56    def mslType = (
57        primitiveType
58      | "array"~>"["~>repsep(wholeNumber~".."~wholeNumber,",")~"]"~"of" ~
            ➥ primitiveType
59      | ident
60    )
61
62    def primitiveType = (
```

```
63        "INTEGER"
64    | "CHAR"
65    | "BOOLEAN"
66    | "AMOUNT"
67    | "REAL"
68    | "STRING"
69    | "VOID"
70    )
71
72    /***********************************************
73     *        Statements
74     ***********************************************/
75    def statementPart =
76      "begin"~stmList~"end"
77
78    def stmList : Parser[Any] =
79      rep(statement~";")
80
81    def statement =
82      returnStm | repeatStm | ifElseStm | whileStm | assignmentStm | callStm
83
84    def assignmentStm =
85      variableRef~":="~expression
86
87    def callStm =
88      ident~opt("("~repsep(expression, ",")~")")
89
90    def returnStm =
91      "return"~expression
92
93    def whileStm =
94      "while"~expression~"do"~stmList~"end while"
95
96    def repeatStm =
97      "repeat"~stmList~"until"~expression
98
99    def ifElseStm =
100     "if"~expression~"then"~stmList~elsifStmsPart~elseStmPart<~"end if"
101
102   def elsifStmsPart =
103     rep("elsif"~expression~"then"~stmList)
104
105   def elseStmPart =
106     opt("else"~ stmList)
107
108   /***********************************************
109    *        Expressions
```

```
110     ********************************************/
111
112   def expression =
113     booleanTerm~rep("or"~booleanTerm)
114
115   def booleanTerm =
116     booleanFactor~rep("and"~booleanFactor)
117
118   def booleanFactor =
119     simpleExpr~rep(
120         "="~simpleExpr
121       | "<>"~simpleExpr
122       | "<="~simpleExpr
123       | ">="~simpleExpr
124       | ">"~simpleExpr
125       | "<"~simpleExpr
126     )
127
128   def simpleExpr =
129     term~rep(("+" | "-")~term)
130
131   def term =
132     factor~rep(("*" | "/" | "mod" | "div")~factor)
133
134   def factor : Parser[Any] = (
135       "-"~factor
136     | "+"~factor
137     | "not"~factor
138     | atom
139   )
140
141   def atom = (
142       "("~expression~")"
143     | literal
144     | typeAttr
145     | ident~"("~repsep(expression, ",")~")"
146     | variableRef
147   )
148
149   def variableRef = (
150       ident~"."~ident
151     | ident~"'"~ident
152     | ident~"["~expression~"]"
153     | ident
154   )
155
156   def typeAttr =
```

```
157      primitiveType~"'"~ident~opt(expression)
158
159    def literal = (
160        floatingPointNumber
161      | wholeNumber
162      | "true".r
163      | "false".r
164      | charLiteral
165      | stringLiteral
166    )
167  }
```

Appendix B

MSL core to MScala prototype translator

The CD, which is delivered along with this thesis, contains the prototype MSL core to MScala translator. Please read the ReadMe.txt file in the top folder of the CD for the detailed instruction on how to use the translator.

Bibliography

[1] "Deltek Maconomy, ERP for professional services organizations." http://www.deltek.com/products/maconomy.aspx/, 2012.

[2] P. Deransart, M. Jourdan, and B. Lorho, *Attribute grammars: definitions, systems and bibliography*. New York, NY, USA: Springer-Verlag New York, Inc., 1988.

[3] E. Visser and Z.-E.-A. Benaissa, "A core language for rewriting," *Electronic Notes in Theoretical Computer Science*, vol. 15, 1998.

[4] Microsoft, "Linq: .net language-integrated query." http://msdn.microsoft.com/en-us/library/bb308959.aspx, 2007.

[5] M. Odersky and al., "An overview of the scala programming language," Tech. Rep. IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[6] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd Edition*. USA: Artima Incorporation, 2nd ed., 2011.

[7] A. Moors, T. Rompf, P. Haller, and M. Odersky, "Scala-virtualized," in *Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation*, PEPM '12, (New York, NY, USA), pp. 117–120, ACM, 2012.

[8] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theor. Comput. Sci.*, vol. 410, no. 2-3, pp. 202–220, 2009.

[9] A. Moors, F. Piessens, and M. Odersky, "Parser combinators in scala," Tech. Rep. CW491, Department of Computer Science, K.U. Leuven, 2008.

[10] E. Burmako, M. Odersky, C. Vogt, S. Zeiger, and A. Moors, "Sip 16: Self-cleaning macros." https://docs.google.com/document/d/1O879Iz-567FzVb8kw6N5OBpei9dnbW0ZaT7-XNSa6Cs/edit?pli=1, 2012.

[11] Semantic Designs, "Legacy software migration." http://www.semdesigns.com/Products/Services/LegacyMigration.html, 2012.

[12] M. Odersky and M. Zenger, "Scalable component abstractions," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, (New York, NY, USA), pp. 41–57, ACM, 2005.

[13] E. Truyen, W. Joosen, B. N. Jørgensen, and P. Verbaeten, "A generalization and solution to the common ancestor dilemma problem in delegation-based object systems," in *Proceedings of the 2004 Dynamic Aspects Workshop*, pp. 103–119, 2004.

[14] ScalaQuery.org, "A type-safe database API for Scala." http://scalaquery.org/, 2012.

[15] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A heterogeneous parallel framework for domain-specific languages," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, (Washington, DC, USA), pp. 89–100, IEEE Computer Society, 2011.

[16] A. Sujeeth, H. Lee, K. Brown, T. Rompf, H. Chafi, M. Wu, A. Atreya, M. Odersky, and K. Olukotun, "Optiml: An implicitly parallel domain-specific language for machine learning," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)* (L. Getoor and T. Scheffer, eds.), ICML '11, (New York, NY, USA), pp. 609–616, ACM, June 2011.

[17] ScalaTest, "Scalatest github website." https://github.com/rickynils/scalacheck, 2012.

[18] Squeryl, "A Scala ORM and DSL for talking with databases with minimum verbosity and maximum type safety." http://squeryl.org/l, 2012.

[19] T. Sloane, "Experiences with Domain-specific Language Embedding in Scala," in *Domain-Specific Program Development* (J. Lawall and L. Réveillère, eds.), (Nashville, United States), p. 7, 2008.

[20] The Scala programming language, "Research: Programming style and productivity." http://www.scala-lang.org/node/3069, 2012.

[21] G. Dubochet, "Computer Code as a Medium for Human Communication: Are Programming Languages Improving?," in *Proceedings of the 21st Working Conference on the Psychology of Programmers Interest Group* (C. Exton and J. Buckley, eds.), (Limerick, Ireland), pp. 174–187, University of Limerick, 2009.

[22] Typesafe, "Typesafe: the company's official website." `http://typesafe.com/`, 2012.

[23] Eclipse IDE, "Scala ide for eclipse – roadmap." `http://scala-ide.org/docs/dev/roadmap.html`, 2012.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[25] G. Dubochet and M. Odersky, "Compiling structural types on the jvm: a comparison of reflective and generative techniques from scala's perspective," in *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, (New York, NY, USA), pp. 34–41, ACM, 2009.

[26] M. Fowler, *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[27] N. Ramsey, "Unparsing expressions with prefix and postfix operators," tech. rep., Charlottesville, VA, USA, 1997.

[28] M. Odersky, "The scala language specification version 2.9," tech. rep., Switzerland, 2011.

[29] Lambda the Ultimate. The Programming Languages Weblog, "The AST typing problem." `http://lambda-the-ultimate.org/node/4170`, 2012.

[30] S. d. Swierstra and O. Chitil, "Linear, bounded, functional pretty-printing," *J. Funct. Program.*, vol. 19, pp. 1–16, Jan. 2009.

[31] Kiama, "A scala library for language processing." `http://code.google.com/p/kiama/`, 2012.

[32] A. M. Sloane, L. C. L. Kats, and E. Visser, "A pure object-oriented embedding of attribute grammars," *Electron. Notes Theor. Comput. Sci.*, vol. 253, pp. 205–219, Sept. 2010.

[33] E. Visser, "Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9," in *Domain-Specific Program Generation* (C. Lengauer *et al.*, eds.), vol. 3016 of *Lecture Notes in Computer Science*, pp. 216–238, Spinger-Verlag, June 2004.

[34] M. Bravenboer, A. van Dam, K. Olmos, and E. Visser, "Program transformation with scoped dynamic rewrite rules," *Fundam. Inf.*, vol. 69, pp. 123–178, July 2005.

[35] T. S. P. Language, "Scala comes to .Net." `http://www.scala-lang.org/node/10299/`, 2011.

[36] T. Parr, *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages*. Pragmatic Bookshelf, 1st ed., 2009.

[37] G. Hedin, "An introductory tutorial on jastadd attribute grammars," in *Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III*, GTTSE'09, (Berlin, Heidelberg), pp. 166–200, Springer-Verlag, 2011.

[38] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun, "Building-blocks for performance oriented dsls," in *DSL* (O. Danvy and C. chieh Shan, eds.), vol. 66 of *EPTCS*, pp. 93–117, 2011.

[39] A. K. Sujeeth, H. Lee, K. J. Brown, T. Rompf, H. Chafi, M. Wu, A. R. Atreya, M. Odersky, and K. Olukotun, "Optiml: An implicitly parallel domain-specific language for machine learning," in *ICML*, pp. 609–616, 2011.