Program abstraction and efficient execution

Ph.D. dissertation
2019.

Boldizsár Németh
http://nboldi.web.elte.hu/
nboldi@elte.hu

Supervisor: Prof. Tejfel Máté, Ph.D.
Eötvös Loránd University, Faculty of Informatics
H-1117 Budapest, Pázmány Péter sétány 1/C.

Ph.D. School of Computer Science
Head of the school: Prof. Erzsébet Csuha-Jarju, Ph.D., D.Sc.

Ph.D. Program Basics and methodology of informatics
Head of the program: Prof. Demetrovicz János, Ph.D., D.Sc.
ACKNOWLEDGEMENT

First of all, I would like to thank my supervisor, Máté Tejfel, for his support during the course of my graduate studies. This dissertation also owes a lot to Gergely Dévai, who gave me the opportunity to take part in several collaborations. Several of my papers were written as parts of the projects I have been working on under his directions. This dissertation could not have been written without the extensive professional support given by both. I believe that their tenacity in approaching research problems has helped to improve my research skills.

I would also like to thank other research collaborators, especially my co-authors who gave me significant help over the course of my studies. Including Zoltán Csörnyei, Tamás Kozsik, Tibor Gregorics, Zoltán Kelemen, Máté Karácsony, Róbert Kitlei, Balázs Gregorics, Dávid János Németh, Gábor Ferenc Kovács, Zoltán Gera, András Dobreff. I am also grateful for the entire staff and PhD students of Department of Information Systems and for all of the members of the wider research community who have been involved with my research.

On a personal note, I would like to thank my parents and extended family for the support throughout my graduate school career. Finally, to my fiancée, Mária, for her patience and for urging me to finish this dissertation.
CONTENTS

1. Introduction .......................................................... 6
  1.1 Introduction to Haskell ........................................... 10

2. Executing abstract programs with limitations on runtime behavior .... 19
  2.1 Introduction ...................................................... 19
  2.2 Tail call optimisation in various languages .................... 20
    2.2.1 TCO in Scala ............................................. 21
    2.2.2 TCO in Haskell ........................................... 22
  2.3 In-language abstractions for stackless programming .............. 23
  2.4 Language design with runtime constraints: the Miller Programming Language ........................................... 25
    2.4.1 Basic language concepts of the Miller Programming Language 26
    2.4.2 Defining control flow with bubbles ....................... 27
    2.4.3 Parameter passing with interfaces ....................... 27
    2.4.4 Limited depth function calls ............................ 30
    2.4.5 Evaluation of expressions using sandbox ................. 31
  2.5 Conclusions ..................................................... 32
  2.6 Contribution ..................................................... 33

3. Design in-language abstractions for handling complex data structures 34
  3.1 Introduction ..................................................... 34
  3.2 Extending Haskell with Effectful Property Abstraction .......... 34
    3.2.1 References ................................................ 36
    3.2.2 Monadic semantics ........................................ 37
    3.2.3 Typing references ....................................... 39
    3.2.4 Reference laws .......................................... 39
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.5</td>
<td>Polymorphic references</td>
<td>41</td>
</tr>
<tr>
<td>3.2.6</td>
<td>Reference operations</td>
<td>42</td>
</tr>
<tr>
<td>3.2.7</td>
<td>Monadic semantics of composed references</td>
<td>43</td>
</tr>
<tr>
<td>3.2.8</td>
<td>IO-based references</td>
<td>46</td>
</tr>
<tr>
<td>3.3</td>
<td>Derivable Partial Locking for Algebraic Data Types</td>
<td>47</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Background</td>
<td>48</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Partial locking</td>
<td>49</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Derivable Partial locking</td>
<td>50</td>
</tr>
<tr>
<td>3.3.4</td>
<td>Shared representation of a data type</td>
<td>50</td>
</tr>
<tr>
<td>3.3.5</td>
<td>Automatic generation of shared references and constructors</td>
<td>53</td>
</tr>
<tr>
<td>3.3.6</td>
<td>Formal definition of transforming types and values to their</td>
<td>55</td>
</tr>
<tr>
<td></td>
<td>shared representation</td>
<td></td>
</tr>
<tr>
<td>3.3.7</td>
<td>Case study</td>
<td>57</td>
</tr>
<tr>
<td>3.4</td>
<td>Conclusions</td>
<td>61</td>
</tr>
<tr>
<td>3.5</td>
<td>Contribution</td>
<td>63</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>64</td>
</tr>
<tr>
<td>4.2</td>
<td>UML Model Execution via Code Generation</td>
<td>65</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Background</td>
<td>66</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Model execution and model compilation</td>
<td>67</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Interpretation, code generation and JIT</td>
<td>68</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Debug symbols and mappings</td>
<td>69</td>
</tr>
<tr>
<td>4.2.5</td>
<td>Using the Java Debug Interface</td>
<td>70</td>
</tr>
<tr>
<td>4.3</td>
<td>Embedding UML models in Java</td>
<td>73</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Automated testing</td>
<td>76</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Interactive debugging</td>
<td>76</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Exporting to UML model</td>
<td>77</td>
</tr>
<tr>
<td>4.4</td>
<td>Defining C Preprocessor Macro Libraries with Functional Programs</td>
<td>79</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Related work</td>
<td>80</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Architecture</td>
<td>82</td>
</tr>
<tr>
<td>4.4.3</td>
<td>The C preprocessor</td>
<td>85</td>
</tr>
<tr>
<td>4.4.4</td>
<td>The generated macro system</td>
<td>87</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

In the recent years, there is an increasing need for rapidly designing high-complexity software systems. These systems are implemented by large groups of programmers and are too big for one person to fully comprehend. It is ineffective to define the behavior of such a complex system using simple operations, like the ones that are available in an imperative programming language. The difference in the complexity level of the implementation tool and the software that should be implemented is too large. This difference in abstraction levels increases the cost of designing, implementing and maintaining such systems. In my thesis I often refer to the “abstraction level” of a programming language, a library or any tool through which the behavior of software can be defined.

In my dissertation I evaluate methods of raising the abstraction level of software development to match the software being developed. I will present different approaches toward this goal.

Programming languages

In the recent decades the greatest improvements of abstraction level in programming came from the development of new programming languages.

Programming languages have a great influence over the software that is produced using them. Different programming languages have different levels of abstraction. The elements of lower-level programming languages are closer to the features of the actual hardware (generally CPU or GPU). This makes them more suited for manual optimization by experts. Higher-level programming languages operate using higher-level concepts, which makes them better at expressing complex parts of the business logic of the application implemented and enables better automatic optimization possibilities.
We can analyse programming languages by many of their properties. We can group them according to the paradigm they support (like imperative, object-oriented, functional or logic), or by features like type system (static or dynamic) or memory model (managed or unmanaged). A large-scale study on 728 projects in 17 different languages suggests [1] that there are significant differences in the frequencies of programming mistakes in different languages.

There is a wide range of programming languages already implemented. As of March, 2018, there are 47 programming languages that have more than 1000 project only in the repositories of GitHub [2]. If one programmer’s needs are not very special, he has a good chance of finding a suitable programming language. In sections 2.2 and 2.3 of my first thesis, I will present a case study of searching for programming languages suitable for implementing stackless programming.

For different problem domains, special programming languages are designed, called Domain-specific languages (DSLs). Examples range from Make [3], a language for programming software builds to VHDL [4] or Verilog, that are hardware description languages. DSLs have three main criteria:

- A special notation used and widely shared by experts in a problem domain.
- The notation makes it significantly easier to express the problem or solution than other, more general descriptions.
- A precise interpretation (semantics).

Please note that DSLs are not necessary Turing-complete or even used to describe computations. Their usability lies in their power to express objects related to their domain, not in their wide usability.

By having a broader view of the programming languages, we can notice that between general purpose and domain specific languages there is a continuum of more or less general languages, that focus on certain areas or concepts. In Section 2.4, I will describe parts of the programming language Miller, a general-purpose programming language, but with some specific restrictions on the generated code.

We have seen how a programming language designed for a specific task can make implementation easier. However, there are major drawbacks of introducing a new
programming language. First, it is a very hard task to create a compiler for the new language. A compiler for a modern programming language contains many components, from lexical and syntactical analyser, to scoping and type checker systems, validation and usually several steps of transformation, code generation and optimization.

It is also hard to introduce a new programming language, because it takes a big effort to learn its usage, especially if it contains unusual language constructs. The benefits of using the new language should be very clear to have any chance of being used. Part of why a programming language is effective for solving problems is community support. Additionally, a modern programming language requires a large number of tools to work efficiently, for example, a sophisticated editor, different code analysis and transformation tools, version control, debugging, etc. The development of these tools also take a big amount of work.

Embedded abstraction

While programming languages are capable of providing a general abstraction level for the programmer, the development of modern software usually depends on multiple artifacts that provide an abstraction level for certain tasks.

In-language abstraction layers can also help managing complexity, while they are easier to create than new languages. An embedded DSL is written in the host programming language, but with higher-level concepts. A library just offers an interface that the programmer can use for a specific task, but the DSL offers a way to describe the whole application in domain-specific way. Another benefit of using embedded DSLs is that the language is familiar for the users but they can build upon the higher abstraction level to design complex systems. Moreover, the programming elements of DSL’s can be mixed with programming elements of the host language to provide both high-level and flexible programming tools.

In my second thesis, in sections 3.2 and 3.3 I present two embedded domain-specific languages to Haskell that are focused on multi-threaded monadic programs with complex mutable program state.

Apart from being easier to implement, embedded languages also inherit most of their tooling from the host languages. In my third thesis, in Section 4.3 I will
describe how we embedded the UML modeling language into the general-purpose host language Java. One of our main motivation was the fine already existing tooling for Java that is much more mature than tooling for modeling languages.

Source-to-source compilation

The process of compiling a programming language, but outputing source code of another programming language, rather than producing machine code directly executable by the computer.

This provides opportunities of raising the abstraction level of a language without the need to re-create the complete language infrastructure. Source-to-source compilation is present in many modern languages, notably script languages like TypeScript or CoffeeScript that are translated to JavaScript as the normal compilation process.

Source-to-source compilation is frequently used when there is some limitation on the programming language, and providing the program in a simpler programming language makes it easier to compile it. For example, the Glasgow Haskell Compiler contains a backend to output C code. By using this backend it is even possible to run Haskell programs on platforms that are not supported by the compiler.

In sections 4.2 and 4.4 I will show two examples of source-to-source compilation. In the former one, the source language is the modeling language UML that we translate to Java to execute efficiently, in the second example the source language is Haskell that we translate to preprocessor macros, to define preprocessor libraries that work on any system where the C toolchain is available.

Structure of the dissertation

My dissertation describes different methods of executing programs that are described in highly abstract concepts. Each method has their own advantages and drawbacks. I show my results in three thesis.

The first thesis of this dissertation – *Executing abstract programs with limitations on runtime behavior* – is centered around finding solutions to hardware restrictions, focusing on stackless programming. I describe how stackless programming is implemented in existing programming languages and how optimizations reduce stack
usage. I present parts of the design of the Miller programming language that is able to run with serious restrictions enforced by the specific platform it runs on. This thesis shows how language design can elevate the abstraction level of source code even when there are strict performance requirements.

In my second thesis – Design in-language abstractions for handling complex data structures – it is shown how programming libraries can be designed to enable higher-level programming in the same programming language. One of the libraries I describe makes parts of complex data structures accessible in effectful computations, another makes the parallelization of a Haskell program to be described by an abstract configuration. These libraries are so general and expressive, that they are a domain-specific extension of the Haskell language. I focus my investigation on how these libraries help programmers writing more abstract and elegant source code.

In my third thesis – Executing UML models and Haskell programs by embedding and cross-translation – I describe the ways how translation between different programming languages and models can make it easier to define complex systems. I map higher-level descriptions of systems to other languages that are easier to execute. In one of the examples we created a Java representation for the UML modeling language. In the other one we translated Haskell programs to preprocessor macros to define preprocessor libraries on a higher abstraction level. I also present an embedding of the complete UML modeling language into the existing Java programming language. The main point of this thesis is to describe how different programming languages can be mapped to each other, to make program execution possible.

1.1 Introduction to Haskell

Since the functional programming language Haskell is used in all 3 of my thesis (illustrating the tail call optimization in Section 2.2.2 as an implementation language for continuation passing style in Section 2.3 as the base language for both of the libraries in Chapter 3 and as the source language in Section 4.4), first I would like to introduce the reader to the concepts and practicalities of the language. However, instead of having an overview of the language, the goal of this section is to describe the language elements that are used in the following chapters.
Haskell was first released in 1990, the latest standard of Haskell is Haskell 2010. Compared to other functional languages (Scala, F# or Erlang) it excels with its purely functional structure and expressive type system.

- Functions are first-class objects in the language. Functions are values which can be used in exactly the same ways as any other sort of value.
- The semantics of Haskell programs is centered around evaluating expressions rather than executing instructions.
- Referential transparency: The same expression results the same value anywhere in the program. This implies that values and data structures are immutable, expressions have no ”side effects” and function calls are deterministic.
- Lazy evaluation: In Haskell, expressions are not evaluated until their results are actually needed.
- Garbage collection: The programmer doesn’t have to track which objects are needed and which are not. Garbage collection automatically collects data that are no longer accessible.

Syntax

In Haskell, one-line comments are marked with --, while block comments are put between {- and -} symbols.

Operators in Haskell are completely customizable. The programmer can define new operators as easily as new definitions with textual identifiers. It is also possible to customize operators to have a given precedence.  

1. Simple types

Logic values are represented by the `Bool` data type with `True` and `False` values. `Char` is a Unicode-based type for characters. Character literals are written between

---

1 I use the Unicode version of some syntactic constructs in this dissertation. Examples: ≤, ≥, ≠, ⇒, ←, ::, λ, ∀, ≫, ≫=.
apostrophes (’c’). String is a list of Char characters. String literals are written between quotes ("abcd").

Integer is an unbounded integer value, while Int is a bounded integer value with a machine-dependent range. Float and Double types are floating-point values that are similar to ones that can be found in other programming languages.

The () datatype is called unit, and has exactly one value (also written as ()). It is usually encountered when a function has no explicit return value.

Defining bindings

A binding associates a name with a value. The type signature \( x :: \text{Integer} \) is optional. Haskell has type inference, so it can calculate the type of a binding if it is not given.

\[
\begin{align*}
x :: & \text{Integer} \\
x = & 3
\end{align*}
\]

Functions can be defined using pattern matching. The factorial function (defined in the following listing) takes an Integer as a parameter and returns a resulting Integer. The input is used for pattern matching. When the function is evaluated it is matched with the pattern 0 first. If it matches, the result is 1. Otherwise it is matched with the variable \( n \), that matches any value. In this case the result will be a recursive call to factorial. The pattern _ is a wildcard pattern that matches anything. It is used when the parameter is ignored.

\[
\begin{align*}
\text{factorial} :: & \text{Integer} \rightarrow \text{Integer} \\
\text{factorial} & 0 = 1 \\
\text{factorial} & n = n \times \text{factorial} \ (n - 1)
\end{align*}
\]

Unlike other programming languages, functions in Haskell can be partial (not defined for all inputs), but using partial functions is considered to be a bad practice and the compiler can warn the programmer for these definitions. Bindings can appear as top-level bindings or local bindings.

Types and data structures

A simple type synonym can be defined using the \text{type} keyword.
1. Introduction

Data types can be defined using constructors like Red, Yellow and Green in the following example.

```haskell
data Color = Red | Yellow | Green
```

The constructors defined can be used in functions as patterns.

```haskell
canCrossRoad :: Color → Bool
canCrossRoad Red = False
canCrossRoad _ = True
```

Each constructor can have one or more properties in it. If they are defined as records, the record names can be used to query, construct or update them.

```haskell
-- define point with x and y coordinates
data Point = Point { x :: Integer, y :: Integer }

g getX :: Point → Integer
getX p = x p -- query the x coordinate of a point

t transposePoint :: Point → Point
t transposePoint p
  = p { x = y p, y = x p } -- update point "p"

origo :: Point
origo = Point { x = 0, y = 0 } -- create a new point
```

Polymorphism

Haskell supports parametric polymorphism. Here is the definition of a polymorphic data structure. The a after the name of the type is called a type variable.

```haskell
data Tree a = Tip | Branch a (Tree a) (Tree a)

t treeDepth :: Tree a → Int
t treeDepth Tip = 0
t treeDepth (Branch elem t1 t2)
  = 1 + (max (treeDepth t1) (treeDepth t2))
```
Haskell also supports type classes. They allow certain types to support certain operations therefore restricting polymorphic behavior.

```haskell
class ToBool t where
toBool :: t -> Bool

Type classes are open, once defined, you can extend a typeclass anywhere in the program by writing instances.

```haskell
instance ToBool String where
toBool "" = False
     toBool _ = True
```

```haskell
instance ToBool Int where
toBool 0 = False
     toBool _ = True
```

When it is required to be able to call a class-function, this fact can be stated with a predicate on the types. With this it is possible to define an “if” construct that does not require the predicate to be boolean:

```haskell
ifG :: (ToBool t) => t -> a -> a -> a
ifG pred thenVal elseVal
    = if toBool pred then thenVal else elseVal
```

Since property abstraction is inherently polymorphic, this language feature will be heavily used in Section 3.2

Creating functions

Anonymous functions can be created in Haskell using the lambda language construct, (shown as \(\lambda a \to a\) in this dissertation).

Composition of functions can be produced with the . operator. The expression \(g . f\) first applies the \(f\) function to its argument, than applies the \(g\) function to the result. The identity function is \(id\).

Local functions can be defined using the where keyword:

```haskell
factorial :: Integer -> Integer
factorial i = facs !! i
    where
        facs = foldl (*) 1 [1..]
```
Higher-order functions

An essential idea of functional programming is that functions are just like any other value, that can be passed as arguments or combined with other functions. A function that takes another function (or several functions) as an argument is called a higher-order function.

An example of a higher-order function is a filter function:

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter predicate (val:rest)
    = if predicate val then val : filter predicate rest
        else filter predicate rest
filter _ [] = []
```

Lists and tuples

Lists in Haskell are homogenous ordered collections. Lists can be created using two constructors. The [] expression creates an empty list, the : operator prepends an element to the beginning of the list. A special list syntax can be used for constructing and pattern matching on fixed-length lists: [1,2,3].

Since Haskell is lazy by default, infinite lists can be constructed if they are not fully evaluated.

Tuples are fixed length structures, whose fields may be of differing types ("heterogeneous"). They are known as product types in programming language theory.

```haskell
x :: (Int, String)
x = (1, "One")
```

Monads

Monads in Haskell provide a way to simulate the side-effects of computation in cases where the computation can be described easier with changing or accumulating values. They are separated from pure calculations by the type system. Most monads can be evaluated to produce a pure value as the result of simulating an stateful computation.
Monads come with two operations. The **return** operation puts a pure value into the monadic context. The bind operation $\gg=$ executes two operations sequentially. The second operation can depend on the value produced by the first. There is a simpler version of the bind operator, $\gg$, that is used when the second operation does not need the result of the first. Monadic operations are essential for understanding Section 3.2.

For monads, Haskell provides a special do notation. Each line is a statement and connected with implicit binds:

```haskell
do x ← [1,2,3]
y ← [1,2,3]
   True ← return (x \neq y)
return (x,y)
```

Common monads include **Maybe** to define computations that can fail, **Either**, for computations that can result in an error value, [] for non-deterministic computations, **State** for stateful computations, **Writer** for computations that record logs, **Reader** that accesses constant information and **IO** for computations that require to access the running environment.

Monad transformers

With monads providing a common way to use such useful general-purpose tools, a natural thing we might want to do is using the capabilities of several monads at once. The monad transformer library provides a way to combine monads in a monad stack. The operations can be lifted to a higher level of the stack by the **lift** function.

The transformer version of common monads can be found with a T prefix. For example we have **MaybeT** or **ListT**. The type of a combined operation could be **MaybeT (StateT Int IO)**, that combines IO operations with statefulness and the possibility of failure.

**IO**

Pure functions are great at representing program logic, but at some point the software needs to make some changes in its environment to have any effect. This could mean working with the file system, sending messages through the network.
or communicating with the user. In Haskell operations that are allowed to make changes in the environment are marked with a special \texttt{IO} type. Since \texttt{IO} is a monad you can use the do-notation to write effectful computation.

\begin{verbatim}
  do ln ← getLine
    putStrLn ("Hello,\ uni2423" ++ ln)
\end{verbatim}

IO based data types

An \texttt{IORef} is a mutable variable that can be used in \texttt{IO} computations. The most essential \texttt{IORef} operations allow the creation of an \texttt{IORef}, as well as reading and writing of the value stored in an \texttt{IORef}.

\begin{verbatim}
newIORef :: a → IO (IORef a)
readIORef :: IORef a → IO a
writeIORef :: IORef a → a → IO ()
\end{verbatim}

An \texttt{MVar} is also a mutable variable type. Unlike \texttt{IORef}, MVars are synchronization primitives. An MVar can be empty or full, threads that are trying to take the value of an empty MVar or trying to put a value into a full MVar will have to wait until the MVar is filled or emptied. The basic operations allow the creation of an MVar as well as taking the value of an MVar or putting a new value into one.

\begin{verbatim}
newEmptyMVar :: IO (MVar a)
takeMVar :: MVar a → IO a
putMVar :: MVar a → a → IO ()
\end{verbatim}

Forcing eager evaluation

By default, Haskell is a lazy functional programming language. This means that Haskell values are only evaluated when needed. However in some cases, lazyness can lead to bad performance, since unevaluated calculations can pile up and take increasing amounts of memory. Lazyness can also hide errors, because when errors are not actually evaluated, they are not reported to the programmer. To alleviate these problems, we can force eager (non-lazy) evaluation. Eager evaluation is used in Section \ref{sec:eager-evaluation}.
The `seq` operation forces the evaluation of its first parameter to weak head normal form (evaluated to the first constructor). This is shown in the following function that defines a strict `foldl` function.

```haskell
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' _ z [] = z
foldl' f z (x:xs) = let z' = f z x
                     in z' `seq` foldl' f z' xs
```

Here the local binding defined by the `let` is important, because by storing `f z x` in one variable, it will not be evaluated twice.

This function does not lead to unevaluated operations taking up memory, and can be applied to very long lists: `foldl' (+) 0 [1..1000000]`.

Type families

Type families are an extension to the Haskell language, and can be enabled with the `TypeFamilies` extension of GHC. It allows the user to create open, extensible associations between types. To define type family you only the name of the family and the type parameters are needed. Type families will be used in Section 3.3 for annotating types.

```haskell
type family AssocElem t
```

To make instances of the type family you have to supply concrete types for the parameters and the resulting type.

```haskell
type instance AssocElem [a] a
type instance AssocElem (Map k v) v
```

Template Haskell

is a language extension and a library for generating and inspecting the Abstract Syntax Tree (AST) of a Haskell program. It can only add new elements to the program, but cannot change the already defined parts. Program fragments can be inspected by looking them up using their names with the `reify` function, but the implementation of functions cannot be seen. To access the implementation, Template Haskell can process parts of the AST by receiving them directly as an argument.
2. EXECUTING ABSTRACT PROGRAMS WITH LIMITATIONS ON RUNTIME BEHAVIOR

In this chapter the author presents his approach for writing high-performance and high-level programs using programming language design.

Thesis I. proves that it is possible to design a programming language that solves severe memory constraints and utilizes memory hierarchies by demonstrating the Miller programming language designed by the author and colleagues. The thesis also demonstrates that the Miller programming language builds on already existing techniques for stackless programming.

2.1 Introduction

The work that is the basis of this thesis is focused on high-performance packet forwarding. Machines designed for packet forwarding have specific architecture for addressing the performance requirements of the software. The specific machines we were working on are programmed in C and a special assembly language. Memory allocation is done manually. The software being developed on the other hand is complex and customizable, not suited for being implemented with so low-level techniques.

Nearly all of our software is modeled with function calls. Function calls are the building blocks of the layers of abstraction that create large and complex software systems. The traditional implementation of function calls is the following: The program has a section of the memory, called stack, dedicated to keeping track of function calls. The caller writes the values of the input parameters into the stack, and writes the address of the instruction following the function call to this memory. Now the callee reads the parameters and execute its own instructions. When finished, it writes the return value to the stack and gives back the control to the address stored
on the stack. The data related to the called function is erased from the stack. After that the caller continues to be executed.

In our scenario, the main problem with this model is that it requires a dedicated memory. The dedicated memory have to be fast to enable quick transitions between functions, so implementing this modell requires to surrender big amounts of fast memory space. Additionally, each function call takes space from this memory, so the execution of the program can run out of stack space. When this happens, the execution of the program cannot continue. This error is called a stack overflow. Since stack overflows cannot be detected at compile-time, they can be a serious problem in critical systems.

The problem of call chains can be solved in several ways. Some programming languages apply the technique of tail call optimization (TCO). It transformes recursive function call to be able to run in constant stack space as a part of the optimization process. The advantage of this solution is that it is non-intrusive, does not require any change in the source code. However it is limited by the implementation of this optimization. This is described in Section 2.2.

Libraries for solving this problem are also available in many programming languages. The most known technique is continuation passing style (CPS). The basic idea of CPS is to define the computation in a way that each step receives the rest of the computation as a function. This is elaborated in Section 2.3.

Finally programming languages themselves can support stackless programming by their language constructs. We will show how the Miller programming language is designed with stackless programming in mind. This is discussed in Section 2.4.

2.2 Tail call optimisation in various languages

Tail call optimisation (TCO) is a common technique for transforming some execution units of the program to operate in constant stack space. The function with the recursive call is transformed into a loop, while preserving the semantics with the appropriate condition.

More precisely, the procedure $E$ is in *tail position* of the enclosing procedure $F$ if $F$ does not have any action to performed after $E$ is returned. This means that
the return value of $E$ is the result of $F$. *Tail call* means a call to expression in tail position, and the recursion is said to be *tail recursion* if the recursive calls are tail calls.

Thus in the case of tail calls there is no need for extra memory to the control information, the result of the procedure $E$ is the result of $F$. Namely, after executing $E$, the control of execution is passed to the process which is the continuation of $F$.

We selected two languages to show their capability of optimizing tail calls. We chose Scala, a functional language running on JVM and the purely functional language Haskell. We chose functional languages because the problem is most relevant to those languages where recursive calls are the only source of iterative behavior. We also wanted to compare different approaches to this problem.

### 2.2.1 TCO in Scala

The Scala compiler implements a limited form of the TCO\[5\]. The problem with the recursive approach is that calls to non-static functions in the JVM are dynamically dispatched. There is a direct and an indirect cost of this, better studied in C++ programs \[6\]. However extensive research also had been done on the resolution of virtual calls in Java programs \[7\].

The system only handles self-recursion, so two functions mutually calling each other will not be transformed into a single cycle. The designers of the compiler introduced this constraint because they didn’t want to cause duplications in the generated code, that could cause the program to slow down.

TCO can only be used on non-overridable functions, because the dynamic method invocation of the JVM prevents further optimisations \[8\].

With the @tailrec annotation, the programmer can ensure that the TCO will be performed by the compiler, otherwise the compilation fails.
Example

\[
\text{factorialAcc}(\text{acc}: \text{Int}, n: \text{Int}): \text{Int} = \{
\text{if } (n \leq 1) \text{ acc}
\text{ else factorialAcc}(n \times \text{acc}, n - 1)
\}
\]

The program is compiled to the same bytecode as:

\[
\text{factorialAcc}(\text{acc}: \text{Int}, n: \text{Int}): \text{Int} = \{
\text{while}(n \leq 1) \{
\text{acc} = n \times \text{acc}
\text{ n} = n - 1
\}
\text{ acc}
\}
\]

So this example shows that we can get the elegant functional solution with no additional costs.

2.2.2 TCO in Haskell

Tail call elimination in Haskell is a little different than in languages with strict execution. It allows not only tail call functions to be executed in constant stack space, but a wider class of functions, that are called \textit{productive} functions [9].

Every function is productive if only contains recursive calls in a data constructor.

For example take the three function definitions below:

\[
\text{infinite_list} = 1 : \text{infinite_list}
\]

\[
\text{infinite_number} = \text{go } 0
\text{ where } \text{go } n = (\text{let } n' = n + 1 \text{ in } n' \text{'seq'} \text{ go } n')
\]

\[
\text{infinite_number'} = 1 + \text{infinite_number'}
\]

The \text{infinite_list} function is productive, because the recursive call is a parameter of the : data constructor, therefore the execution will not result in a stack overflow. And it will generate an infinite list of ones.

The \text{infinite_number} function has a tail call, where the result is accumulated as an argument, and strictly evaluated by the \text{seq} function. If we would allow the lazy execution of the accumulator parameter, the tail call would be eliminated,
but because the parameter is constantly growing, the "out of memory" error would be inevitable as seen in the case of \texttt{infinite\_number}'s. See also the strict folding functions \texttt{foldr'} and \texttt{foldl'} in the documentation of the Data.List module \cite{10}.

Let's see the result of the execution of the three statements above:

\begin{verbatim}
> infinite\_list
[1,1,1,1,1... -- This runs forever, but
  -- does not result in stack overflow.
> infinite\_number
    -- Runs forever, also
    -- in constant stack size
    -- but without producing output.
> infinite\_number'
<interactive>: out of memory
\end{verbatim}

\section{In-language abstractions for stackless programming}

While compilers can alter the generated program code to produce programs that run in constant stack space, these optimizations can only handle simpler control structures, as we saw in Section \ref{section:control-structures}. For more complex systems, the programmer must actively design their system to run in constant memory size. We chose the \texttt{Cont} monad from Haskell as an example.

The \texttt{Cont} monad can be found in the Monad transformer library \cite{11}, in the \texttt{Control.Monad.Cont} module. It is a monad for writing functions with continuation passing style. The definition of the \texttt{Cont} monad looks like the following:

\begin{verbatim}
newtype Cont r a = Cont { runCont :: (a \rightarrow r) \rightarrow r }
\end{verbatim}

The type \texttt{r} is the type of the final value of the computation, \texttt{a} is the actual result of one continuation step. To get the final result after a continuation is run, we have to apply the final continuation, that has the type \texttt{a \rightarrow r}.

Creating the monadic instance for the \texttt{Cont} structure requires us to define how should the semantics of \texttt{return} and \texttt{bind} behave. When returning, the monad simple supplies the returned value to the final continuation. When binding, the execution
of the second statement with the final continuation \((fc)\) is given as the continuation of the first statement.

```haskell
instance Monad (Cont r) where
  return a = Cont (\fc \rightarrow fc a)

  m >>= k = Cont (\fc \rightarrow runCont m (\a \rightarrow runCont (k a) fc))
```

After making a Monad instance for the Cont datatype, we can use it to create a factorial calculation in a simple way. The \texttt{fac} function simply executes \texttt{fac\_cont} with an identity transformation as the final continuation.

The \texttt{fac\_cont} \texttt{n} applies the final continuation with return, or executes \texttt{fac\_cont} \texttt{(n\textminus1)} with the multiplication as a continuation.

```haskell
fac :: Int \rightarrow Int
fac n = runCont (fac\_cont n) id
  where fac\_cont :: Int \rightarrow Cont Int Int
        fac\_cont 0 = return 1
        fac\_cont n = do fprev \leftarrow fac\_cont (n\textminus1)
                        return (fprev * n)
```

For demonstrating the power of our continuation-using factorial function, we also present a naïve recursive implementation.

```haskell
fac\_naive :: Int \rightarrow Int
fac\_naive n = n * fac\_naive (n\textminus1)
```

Then we execute both for a number larger than the maximum stack size, and inspect the results:

```haskell
> fac 1000000
0 -- (Because of arithmetic overflow)
> fac\_naive 1000000
<interactive>: out of memory
```

If we follow the execution of the \texttt{fac\_cont} function we can observe how it can run in constant stack space.
> fac_cont 3
runCont (fac_cont’ 3) id
runCont (fac_cont’ 2)
  (λa1 → runCont (return (a1*3)) id)
runCont (fac_cont’ 1)
  ((λa2 → runCont (return (a2*2)))
   (λa1 → runCont (return (a1*3)) id))
runCont (fac_cont’ 0)
  (λa3 → runCont (return (a3*1))
   ((λa2 → runCont (return (a2*2)))
   (λa1 → runCont (return (a1*3)) id)))
runCont (return (1))
  ((λa2 → runCont (return (1*2)))
   (λa1 → runCont (return (a1*3)) id))
runCont (return (1*2))
  (λa1 → runCont (return (a1*3)) id)
runCont (return (1*2*3)) id
1*2*3
6

Writing programs with true continuation passing style is most convenient in functional programming languages. However these languages are usually not well suited for execution with serious performance and resource constraints.

2.4 Language design with runtime constraints:

the Miller Programming Language

Because of the high-performance requirements of the implemented software, the general consensus of the designers of packet-forwarding systems is that parameters should be stored in registers instead of memory. The number of registers is limited and they are also used for storing temporary results and variables. This makes it necessary to limit the depth of call chains.

Since the domain has many serious runtime constraints it is unlikely that an existing high-level programming language can be sufficient, as we saw the capabilities and limitations of both built-in optimizations and in-language abstractions.
In this part of the thesis we present the aspects of the Miller Programming Language that are related to stackless programming.

2.4.1 Basic language concepts of the Miller Programming Language

The Miller Programming Language is an industry-oriented programming language, focused on the development of performance-critical applications for special hardware while providing a safe abstraction layer.

The simplest execution unit of Miller is the *bubble*. A bubble is a separate compilation unit. *Primitive bubbles* can contain simple sequential code (a basic block), but no control structures such as branches and cycles. At the end of the bubble there is a section where conditions decide which bubble will be executed next. These are the *exit points* of the bubble.

Bubbles can also form a network, interconnected by their exit points. A *graph bubble* embeds a network of bubbles into itself. The nested bubbles can only be used by transferring the control to one of the entry points of the graph bubble (the control cannot be directly given to a nested bubble). If an exit point of an inner bubble is connected to the entry point of another bubble inside it is called a *local exit point*. Otherwise if it’s connected to an exit point of the containing graph bubble it is a *far exit point*. See these concepts illustrated on Figure 2.1.

![Figure 2.1: Miller’s graph bubbles](image)

The graph bubble creates an encapsulation for its inner bubbles, and defines an interface through which they can be accessed. Graph bubbles can also be nested into

---

1 George Armitage Miller (February 3, 1920 - July 22, 2012) was one of the founders of the cognitive psychology field. He also contributed to the birth of psycholinguistics and cognitive science in general. Miller wrote several books and directed the development of WordNet, an online word-linkage database usable by computer programs [12]. We chose his name for our language because of his great contribution for the understanding on the usage of human memory, while our language focuses on the usage of electronic memory.
other graph bubbles. Nested bubbles can use any program element defined in their graph bubbles.

It is easy to see that graph bubbles, bubbles and exit points are equivalent in expressive power to the control structures of structured programming. Branches and loops can be simulated using a network of bubbles.

2.4.2 Defining control flow with bubbles

The general case of defining the transfer of control is to set exit points of the bubbles. This also allows the programmer to create control cycles. We experimented with this mode of control, but found that it is too cumbersome for actual programming.

Non-sequential code (for example branches and cycles) have to be described as a network of bubbles. However, the language does not require the programmer to manually create these bubbles and their connections. An imperative programming interface is presented and the compiler automatically creates the final bubble graph from this representation. This interface contains \textit{if-then} branching, \textit{if-then-else} branching, other special branching operations, and a \textit{do-while} (post-test) loop.

The example on Figure 2.2 shows a typical conversion from imperative frontend to a network of bubbles. It shows how the while cycle is transformed to bubbles in a program that computes the greatest common divisor of two positive integers.

Using bubbles for the transfer of control does not need a stack. The programmer cannot return control to the caller from an execution module, just pass the control to the next execution unit. If call-and-return behaviour is expected, limited depth function calls provide help.

The result of the execution of the bubble body decides through which exit the control flow is passed. To allow programming in continuation passing style, one more language feature is required, that transfers a point of execution from bubble to bubble. This is called parametrisation of the bubble exit points.

2.4.3 Parameter passing with interfaces

The description of each bubble contains the variables that are passed to the next bubble on each exit point. This is called the exit specification. The bubble also
Figure 2.2: Transforming Miller’s imperative syntax into graph bubbles

dDeclares the variables that it uses, this is the entry specification. The interface checker verifies that the exit and entry specifications match.

It is very important to clarify that no copy operations happen, for this is not the traditional way of parameter passing. All variables declared in the exit and entry specification must appear in one of the ancestor bubbles.

The interface check theoretically prevents the program from accessing uninitialised variables. In practise this does not always happen. For example, it cannot be statically proven that a loop cycle running on an array gives all elements a starting value.

Nevertheless, the interface check gives us the same confidence that can be given by inspecting the initial assignment of variables, and does not require any data copy
to be made for parameter passing. In addition, it also enables to create variables with constant values locally, in the scope of one bubble.

Example: factorial function in Miller

We present two approaches to calculate the factorial function. The first approach is a naïve recursive function, using stack. It is presented in a C-like pseudo-code. The second is a stackless approach, with bubbles accessing global variables, and using iterative control structure. It is presented with the pseudo-code version of the language Miller. We give the sequence of evaluation for each implementation.

Please note that, unlike the earlier examples, it is written in procedural and not in functional style.

```c
int32 fac( int32 n ) {
    if( n <= 1 ) {
        return 1;
    } else {
        return n * fac(n-1);
    }
}
```

The next table shows the content of the stack after each step of execution. The ‘n’ columns show the value of the variable \( n \) in the given context. The ret columns show the points where the control is returned after the return call.

<table>
<thead>
<tr>
<th>step #</th>
<th>ret</th>
<th>n</th>
<th>ret</th>
<th>n</th>
<th>ret</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>caller</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>caller</td>
<td>3</td>
<td>fact</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>caller</td>
<td>3</td>
<td>fact</td>
<td>2</td>
<td>fact</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>caller</td>
<td>3</td>
<td>fact</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>caller</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The second part of the example shows the factorial function implemented with stackless bubbles of Miller. Please note that the while cycle is needed because we have to connect the `again` exit point with the entry point of the cycle.
int32 n;
int32 val;

bubble fact(n)
exits out(val) {
    val = 1;
    while(true) { cycle; }
}
bubble fact::cycle(n,val)
exits again(n,val)
far exits out(val) {
    if( n <= 1 ) {
        exit out(val);
    } else {
        exit again(n-1, val*n);
    }
}

The next table shows the evaluation of the \textit{fact}(3) expression in Miller with only two global variables. The two columns represent the values of the corresponding global variables.

<table>
<thead>
<tr>
<th>step</th>
<th>n</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (in fact)</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2 (in cycle)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3 (in cycle)</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4 (in cycle)</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>5 (in out)</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

As can be seen, the expression is evaluated in constant stack size.

2.4.4 Limited depth function calls

There are situations where calling functions and returning the control is highly preferred to low level passing of control (bubbles). The Miller language provides limited function calls in such situations.

Currently it is the programmer’s responsibility to return the control to the caller from the called method. This approach enables the function to be a complete system
of bubbles, and any of them can return, if appropriate.

An important aspect of the functions is that the depth of the call chain is limited. We can calculate it in the following way:

- The bubble that calls no other bubbles have the call depth of zero.
- When a bubble calls other bubbles, its call depth is greater by one than the maximum of the call depths of the called bubbles.

Because the depth of all units must be bounded, the function calls cannot be recursive. This kind of control flow would require a stack to implement.

Thanks to the limitations on the function calls it becomes possible to store all function arguments and return addresses in registers, which is a common practice in performance critical systems. For example, if we limit our call chains to a depth of five, at most five registers would be enough to store the return addresses of the calls.

Of course, if values are passed to the called functions, more registers will be needed. To implement calls inside the bubble system, the compiler creates a calling bubble, which executes the call operation.

2.4.5 Evaluation of expressions using sandbox

In the earlier sections we described how control structures are implemented in Miller. However, it is also important to perform arithmetic calculations efficiently.

The sandbox is a tool for generating instructions to evaluate complex expressions. It has a finite amount of registers and a larger amount of memory locations.

The compiler always optimises the usage of registers and memory locations to minimise the number of temporary variables. This means that the subexpressions to be evaluated first are the subexpressions that need more registers.

The sandbox works according to a simple strategy. As long as there is space the intermediate values are stored on registers, then it puts them on the specified memory areas. For now, this is enough, when it will be necessary, we design an algorithm that takes into account the different types of memory as well.
Example

In this example we present two methods to evaluate a simple expression. The first method uses the stack to evaluate expressions of any size, while the second uses a sandbox with a finite amount of registers to evaluate expressions.

The evaluation of the expression $a \land b$ to the lower part of the $ax$ register with a very simple code generator is based on stack operations.

```
... instructions for the evaluation of $a$
... to the lower part of ax register
push ax
... instructions for the evaluation of $b$
... to the lower part of ax register
pop bx ... (loading the previously stored value of $a$)
and al,bl ... (executing the instruction on
... the lower part of ax and bx registers)
```

The evaluation of $a \land b$ to the lower part of ax register with our sandbox model:

```
... instructions for the evaluation of $a$
... to the lower part of ax register ...
... instructions to acquire a new register $r$
... from the sandbox
... (that may cause moving previously stored data
... from a register to the memory.)
... instructions for the evaluation of $b$
... to the allocated $r$ register
and ax,r
... instructions to release the allocated $r$ register
```

If more registers are needed for calculating the subexpressions then more registers can be acquired. When the sandbox has temporary registers, the allocation of registers doesn’t generate any instructions.

2.5 Conclusions

I investigated the techniques of generating programs with serious limitations on performance and resource usage. We focused on stackless programming as a case study. We analysed three ways of being able to write such programs. The first way was to use optimizations already present in programming languages. The second was to
use libraries and other in-language constructs for expressing the program in a way that satisfies the runtime requirements. The third method was designing a complete programming language that supports the requirements with built-in language constructs.

My approach to designing programs that run in constant stack space is implemented in the imperative Miller language. It replaced function calls with passing of control between bubbles, and function arguments with controlled global variables. This method can be used on the special hardware, which is not equipped with an efficient stack implementation. Our research was motivated by these problems.

It is important to notice that the core concept of the Miller programming language is to be a high-abstraction level language, in spite of the strict performance requirements of the programs that are written in it. To enable this, Miller introduces new programming concepts, like bubbles, that enable the programmer to write highly efficient code in an abstract way. Miller is supposed to replace the existing practice of programming these systems using C and assembly code.

2.6 Contribution

The results presented in this chapter were published in a conference paper [13]. My personal contribution to this study is the inspection of optimizations for tail calls in different languages and the part related to the Miller programming language.

At the time of the publication of that article I was also contributing to the development of the Miller programming language. In that project my main field was code generation and optimization. I created the backend for the code generation to the field-specific assembly language that was used by our industry partner.

My masters’s thesis also contributed to the programming language Miller, it was an extension of the type system to effectively and safely use dynamic data structures[14].
3. DESIGN IN-LANGUAGE ABSTRACTIONS FOR HANDLING COMPLEX DATA STRUCTURES

In the second chapter of the dissertation is focused on finding methods to high-performance and high-level programming by using program elements written in programming languages, especially embedded domain-specific languages. The theses in this chapter state that domain specific languages can support higher-level programming without a change in the used programming language.

**Thesis II.** proves that effectful property abstraction can be implemented as an embedded domain-specific language in Haskell by demonstrating the design of the References DSL in Haskell.

3.1 Introduction

In-language abstraction is easier to implement than improvements to the existing programming languages and easier to apply to a specific problem or environment.

In this thesis I specify the design and implementation of two major libraries for the Haskell functional programming language. One is used for defining property abstractions while the other helps with transforming an existing program to enable parallelism. The libraries are related, property abstraction is used as a way of transforming a program from single-threaded to multi-threaded implementation.

3.2 Extending Haskell with Effectful Property Abstraction

Haskell is a high-level language for developing complex applications. Complex applications have structured inner representations, so it is important to provide ways of handling these data structures. The most frequent operations are querying the
parts of a data structure and updating them. However handling deeply nested data structures can be tedious in Haskell because the lack of direct language support.

Take for example an application about geometric shapes. The shapes are built from points that are in turn built from numbers:

```haskell
data Rectangle { point1 :: Point 
    , point2 :: Point }

data Point { x :: Integer 
    , y :: Integer }
```

Querying the data structure is simple with the help of the record labels. For example the `x . point1` expression takes a rectangle and returns the x coordinate of the first point.

The source code for updating one of the coordinates of a shape is unnecessary long and complicated, because one has to not only set the property but apply the changes on a higher level:

```haskell
rect' = rect { point1 = (point1 rect) 
    { y = y (point1 rect) + 1 } }
```

As an alternative, the user deconstruct the data structures and reconstruct them with a function:

```haskell
modRect (Rectangle (Point x y) b) 
  = Rectangle (Point x (y+1)) b 
  -- but this method gets worse with more attributes 
rect' = modRect rect
```

It gets even longer for data structures with more levels of nested datatypes. The complexity of updating parts of these data structures grows linearly with the depth of the data structure.

Meanwhile in traditional imperative languages, the same operation could be implemented conveniently with a property update:

```haskell
rect.point1.y += 1;
```

It should be noted however that in imperative languages, properties are generally not first-class elements of the language. This means that they cannot be used like objects, cannot be passed to functions as arguments. They are also restricted to
represent simple fields of record or class structures. They also usually don’t allow the application of a function.

The lens library [15] provides a way to access parts of a complex data structure:

\[ \text{rect#point1.y \%~ (+1)} \]

The lens library uses functions to represent properties. It supports optional and multi-valued properties as well as simple fields. The lenses that can be defined by the lens library work by applying a generic function to the data accessed. Because of the implementation of lenses, there is no way to create lenses that use specific monadic operations to read or modify the values they access.

In some cases however, we would like to use properties in a computation that is not pure. There are two cases when this is necessary:

- The data structure in which the accessed value is contained is only available through monadic operations. This is the case with IORef [16] and MVar [17] data structures that are used to represent mutable variables in Haskell. There is no way of defining a lens that performs operations in the IO monad.

- The operation that we want to perform on the value accessed by the property is a monadic operation. With lenses, only pure functions can be applied to the values accessed by lenses.

### 3.2.1 References

To mitigate this lack of support for monadic data structures and operations we introduced references that are effectful property abstractions for Haskell. These structures introduce a high-level abstraction for accessing parts of non-clean data structures and it could be argued that they define a domain specific language inside Haskell.

A reference is a property abstraction. The basic function of a property is to define a relation between parts of the program data. Properties represent a containment relation on the data structure. These relations control the lifetime of the values. For example, the coordinate x is the part of the point. When the point is no longer needed, its coordinate can get disposed as well.
When using references we call the container data structure the context of the reference, and the contained data the value of the reference. The reference represents an accessor to that part of the data and can be used to query or update it.

For example take the reference \(_1\). It can access a value of type \(a\) through a tuple that's first element is of type \(a\). So the context of \(_1\) is a tuple of type \((a, b)\) (the type of the tuple that has an element of type \(a\) at the first position) and the accessed element is of type \(a\). We implemented references using a data type called \texttt{Reference}, that collects the getters, setters and updaters to perform the needed operations.

The simplest definition of the datatype is the following. It is a very strict definition, in the sense that only allows references to access one and only one value in the context.

\[
\text{data Reference } s \ a = \text{Reference}\{ \text{refGet} : s \rightarrow a, \text{refSet} : a \rightarrow s \rightarrow s, \text{refUpdate} : (a \rightarrow a) \rightarrow s \rightarrow s \}
\]

This definition is sufficient to implement the \(_1\) reference described earlier:

\[
_1 :: \text{Reference} (a, b) \ a
_1 = \text{Reference}\{ \text{refGet} = \lambda(a, b) \rightarrow a, \text{refSet} = \lambda x (a, b) \rightarrow (x, b), \text{refUpdate} = \lambda f (a, b) \rightarrow (f a, b) \}
\]

The type parameter \(s\) is the context type, where \(a\) is the type of the referenced object. The \texttt{refUpdate} operation (applying a function to the data), can be expressed through the \texttt{refGet} and \texttt{refSet} operations, but for practical purposes we store it as a separator field.

\subsection{Monadic semantics}

The previous definition of references can be used do create references like \(_1\) where it is certain that the accessed value is present exactly at one instance in the data structure. This corresponds to the classical definition of properties in other languages. However, in real life, most containment relations have no such strict limitations. The exact structure of real data is only revealed at runtime, so it is good to extend our
3. Design in-language abstractions for handling complex data structures

reference relations for data that may not be found, or may be found in multiple instances.

Take for example the elements reference, that accesses all elements of a list. The type of such reference should be Reference [a] a, since it accesses individual elements, but with the previous definition, this is not correct, since then the refGet elements would have the type [a] → a, and would not be able to return all elements of the list.

To enable our references to work with data that may not exist, or may exist in any number of instances we must introduce monads to the representation.

data Reference w r s a
  = Reference
    { refGet     :: s → r a
      , refSet     :: a → s → w s
      , refUpdate  :: (a → w a) → s → w s
    }

With this new definition, the elements reference can be defined:

elements :: Reference Identity [] [a] a
elements = Reference
  { refGet = λls → ls
    , refSet = λx ls → map (λ_ → x) ls
    , refUpdate = λf ls → map f ls
  }

The type parameters w and r are the writer and reader monads. These will affect the semantics of accessing a value. When using the getter on a reference the value will be returned in the reader monad. When writing the value of the reference the updated context will be returned in the writer monad.

For example if the reader monad is Identity, the reference accesses exactly one instance of the accessed data in a given context. If it is Maybe the reference may access one element of the value type, or none at all. If it is [] the reference may access any number of elements from a given context. If it is IO the reference accesses one element, but it can be accessed through IO operations.

The writer monad is usually Identity or IO since we would like to have exactly one result after an update. Typically the reader monad is more complex than the writer monad.
3.2.3 Typing references

When defining references \( r \) and \( w \) we decided on using type variables instead of concrete types. This enables the use of the created reference in more than one kind of computation. The actual types for the monadic context should be decided when the reference is used. This works because Haskell uses type inference instead of type checking, so it is possible to define generic references and only constrain their types when used.

We introduce type synonyms for generic definition of references. \textbf{Property} is the type of references that access one and only one value and can be used in any monadic context. This type is named \textbf{Property} because it implements the same concept as properties in other programming languages.

\textbf{Partial} is the type of a reference that accesses a value that may not exist and can be used in monadic contexts that can represent non-existence. This is specified in the type synonym as the reader monad being ”\texttt{MaybeLike}”. \textbf{Traversal} is the type of a reference that accesses a value that may exist in any number of instances and can be used in monadic contexts that can represent non-determinism. This is specified in the type synonym as the reader monad being ”\texttt{ListLike}”.

\begin{verbatim}
  type Property s a
  -- r and w is any Monad, no special properties are known
  = ∀ w r . (Monad w, Monad r) ⇒ Reference w r s a

  type Partial s a
  = ∀ w r . (Monad w, MonadPlus r, MaybeLike r) ⇒ Reference w r s a

  type Traversal s a
  = ∀ w r . (Monad w, MonadPlus r, ListLike r) ⇒ Reference w r s a
\end{verbatim}

3.2.4 Reference laws

So far the definition of references is very generic since the data structure can be filled with any three functions that conform to the given types. Five properties of references
are defined in this section to make references to conform to the expectations. From the infinite number of values that can be constructed, we only consider references those that satisfy the following laws.

The laws are formulated as equivalences between different monadic expressions, to express that using set or update operations can cause side-effects. The laws apply to these side-effects (≡ represents equivalent monadic action).

The *get-get law*: Get should cause no side-effects, so the get operation is idempotent.

\[
\text{refGet ref } s \gg \text{refGet ref } s \\
\equiv \text{refGet ref } s
\]

The *set-get law*: After setting the element the assigned value can be retrieved. (But the set operation can cause side effects.)

\[
\text{refSet ref } a s \gg \text{refGet ref } \\
\equiv \text{refSet ref } a s \gg \text{return } a
\]

The *get-set law*: Writing the accessed element with the value that is asked does not change it (and cause no side effect).

\[
\text{refGet ref } a \gg \lambda b \rightarrow \text{refSet ref } b s \\
\equiv \text{return } s
\]

The *set-set law*: A write operation overwrites every previous modification.

\[
\text{refSet ref } a s \gg \text{refSet ref } b \\
\equiv \text{refSet ref } b s
\]

The *update law*: Update is equivalent to getting the value then transforming it and then writing it.

\[
\text{refGet ref } a \gg f \gg \lambda b \rightarrow \text{refSet ref } b s \\
\equiv \text{refUpdate ref } f s
\]

As it can be seen, updates could be substituted with get and set operations. However in some cases it is not beneficial to split the update operation into two parts (possible synchronization issues). Moreover, the result of the update is given by the write monad instead of the read monad. This simplifies the usage of update operations.
3.2.5 Polymorphic references

We call a reference polymorphic, if it can change the type of the context by changing the type of the accessed element. Take for example, the `elements` reference. Since it can modify all elements of the list, it sounds logical to be able to transform a `[Int]` into a `[String]` by applying the `show` function on the `elements` reference, similarly to `map show`.

When this is implemented the representation of a reference looks like the following:

```haskell
data Reference w r s t a b = Reference
  { refGet :: s → r a,
    refSet :: b → s → w t,
    refUpdate :: (a → w b) → s → w t
  }
```

Now, additionally to the context type `s` and the element type `a`, there is a modified element type `b` and the modified context type `t`. It is natural that `b` and `t` should be more general than `a` and `s` in the sense that even if `s` is fixed by the object we use the reference for, `b` still can be polymorphic.

The types of the references are also extended with the new type variables to enable polymorph references:

```haskell
type Property s t a b = ∀ w r . (Monad w, Monad r) ⇒ Reference w r s t a b
type Partial s t a b = ∀ w r . (Monad w, MonadPlus r, MaybeLike r) ⇒ Reference w r s t a b
type Traversal s t a b = ∀ w r . (Monad w, MonadPlus r, ListLike r) ⇒ Reference w r s t a b
```

Finally, the type of the `elements` reference, that is used as an example, will be `Traversal [a] [b] a b`, since it can be applied to lists with any type parameter (that is `a`) but the type of the accessed element can still be changed to any other type (that is `b`).
3.2.6 Reference operations

Apart from creating the references themselves, it is also important for the user to have operations on the references. These operations can be used to apply the references to specific tasks or to combine them into more complex references.

3.2.6.1 Operators

Operators are the interface of using references. Operators assign concrete types for references, because by definition references are polymorphic.

Operators can be used to get, set or update the data accessed through the references. Pure operators can only be used when the reader or writer monad is Identity.

The most important operators are listed in Table 3.1.

<table>
<thead>
<tr>
<th>Op.</th>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>s → ref → a</td>
<td>pure getter</td>
</tr>
<tr>
<td>.?</td>
<td>s → ref → r a</td>
<td>monadic getter</td>
</tr>
<tr>
<td>.=</td>
<td>ref → b → s → t</td>
<td>pure setter</td>
</tr>
<tr>
<td>!=</td>
<td>ref → b → s → w t</td>
<td>monadic setter</td>
</tr>
<tr>
<td>-</td>
<td>ref → (a → b) → s → t</td>
<td>pure updater</td>
</tr>
<tr>
<td>!~</td>
<td>ref → (a → m b) → s → w t</td>
<td>monadic updater</td>
</tr>
</tbody>
</table>

In the table ref is a shorthand for Reference w r s t a b

Table 3.1: Reference operators

Some examples of using the operators:

-- Get the first element of a pair (pure getter)

```
fst' :: (a,b) → a
fst' p = p . _1
```

-- Set the x coordinate of a point to zero (pure setter)

```
xTo0 :: Point → Point
xTo0 = x .= 0
```

-- Print the elements of the list (monadic updater)

```
printList :: [a] → IO [a]
printList ls = ls !~ print
```
3.2.6.2 Combinators

One of the most important advantage of references over properties in other languages is that they can be combined together. Since references are first-class values in Haskell, they can be passed as arguments to functions, combined and changed.

The composition combinator, written as \(&\) is the most important reference combinator. It allows to access values deep inside a complex data type. Combination requires that the monadic semantics of the two references can be unified (this is discussed in more details in Section 3.2.7). It also requires that the context of the right-side reference is the same as the element type of the left-side reference.

For example, the just reference can access the maybe-existing value of the Maybe data type. The reference _1 can access the first member of a pair. So by combining the two references as _1 & just the result can read or write the optional value in the first member of the pair.

The reference self, that accesses itself, is a neutral element regarding to composition: \(r \& \text{self} \equiv \text{self} \& r \equiv r\).

Additive combination (\(r_1 \&+& r_2\)) is another interesting combinator. It accesses all elements accessed by \(r_1\) or \(r_2\). It only works when the two reference has the same context and element type, and should be used with caution, because it is easy to create references that access the same element multiple times. For example, take self &+& self that accesses the value twice. In this case any modification that uses the reference will be performed twice.

A simple useful example for the usage of \&+& operator is both = _1 &+& _2, a reference that accesses the first and the second element of a tuple. It can only be applied if the two element have the same type.

Composition and additive combination is distributive:

\(r \& (p \&+& q) \equiv (r \& p) \&+& (r \& q)\),

\((r \&+& p) \& q \equiv (r \& q) \&+& (p \& q)\).

3.2.7 Monadic semantics of composed references

One of the hardest challenges when creating this library was to resolve monadic type parameters for combined references. Take for example the reference we get from combining the accessor of the first element of a tuple _1 with the accessor of the
optional value in a maybe just. The first reference has Identity semantics for both
read and write operations while the second has Maybe semantics for read operations.
Since the combined reference will not always contain the accessed value, it will be
having the same semantics as just.

To calculate the type of the composed references, we need to define a relation
between monads. We can say that m monad is at least as strong than k if any value
in the k monad can be represented in m. That means that there exists a generic
function liftUp :: k a → m a. This relation is named ”as-strong” and is reflexive,
transitive and antisymmetric, a partial ordering on monad types. It is compatible
with the monadic operations (return and bind) of the two monads. The relation
is realized by the type class AsStrong and generalizes the MaybeLike and ListLike
constraints from former examples.

It is trivial to give such a method when k is Identity. The value in Identity
monad can be simply extracted. The liftUp function for the identity monad can
be defined as λ(Identity a) → return a. Identity is the weakest monad, any
monads is as strong as Identity.

The as-strong relationship can also be used to define references. For example,
just is a reference with a reader monad that is at least as strong as Maybe. Meanwhile
_1’s reader monad is at least as strong than Identity so it can be any monad. The
combination _1 & just should have the reader monad of just that is the strongest
of the two.

In other cases the strongest monad will not be one of the combined references, for
example the composition ioref & just have the reader monad MaybeT IO where
the ioref’s reader is IO and just’s reader is Maybe. The reference ioref will be
described in Section [3.2.7]

Finding the monad that is at least as strong than both combined monads should
not be done by the composition operator. In an earlier version of the library monadic
types were computed this way, but the approach resulted a lot of ambiguous types
when combining complex references.

In the current design of the library the reader and writer monads of the references
are given by the context in which they are used. So, if just is used to get a value
in a Maybe operation, its reader will have Maybe semantics. If used in a MaybeT IO
operation, the reader semantics will be \texttt{MaybeT IO}. Any monad can be the context as long as it is as strong as \texttt{Maybe} that is the \textit{minimal reader monad} of \texttt{just}.

The complete ”as-strong” relations between the basic monads is shown on Figure 3.2. Implementing the ”as-strong” relation is quite complicated in Haskell, since the type-class resolution mechanism cannot search according to transitivity rules. Instead of directly coding the transitivity rules with type class instances, we created a type-level shortest-path search and guided the typeclass resolution according to the results of the search. With the latest additions to GHC it would be possible to create a type checker plugin to solve this problem in a more elegant way, but that was not yet available when the library was designed.

Handling unused semantic monads

If the context is used to find the reader and writer semantics of a reference, how can the type checker fix the reader semantics when the reference is not read but written? Conceptually, it could be any monad that is as strong as the minimal reader monad of the reference. So if \texttt{just} is written in a context its reader semantic can be any monad that is as strong as \texttt{Maybe}.

We used a simple monad to solve this issue in a convenient way. We call that the monadic unit (\texttt{MU} for short), but is also known as \texttt{Proxy}. It is defined as \texttt{data MU a = MU}. In practice \texttt{MU} is just a placeholder monad that means that the given operation cannot be performed. If the writer semantics of the reference is \texttt{MU} then the reference is read-only. If the reader semantics of a reference is \texttt{MU} then the reference is write-only.

It is easy to see that there is no way to extract a value from an \texttt{MU} data structure. In this it is the opposite of \texttt{Identity}, from which the value can always be extracted. As \texttt{Identity} is the weakest type in our hierarchy of types, it is easy to see that \texttt{MU} will be the strongest. Any monad can be lifted into \texttt{MU} with an instance like the following: \texttt{instance AsStrong MU m where liftUp _ = MU}
3. Design in-language abstractions for handling complex data structures

![Diagram](image-url)

**Figure 3.2:** "as-strong" relations between basic semantic types

### 3.2.8 IO-based references

Being able to create references for simple types like tuples, `Maybe` or `[]` is necessary, but our motivation for creating references was to handle monadic operations. The currently existing tools, like the lens library cannot give access to values that can only be accessed using monadic operators.

As an example we will provide references to access values using an IO computation. References for other monads can be defined in similar ways. To construct the references, first we have to define their types.

`IOLens` is the type of a reference that accesses its value through operations with side effects. The precise meaning of the type restrictions on references will be described in Section 3.2.7.

```haskell
type IOLens s t a b = ∀ w r . ( Monad w, Monad r, AsStrong w IO, AsStrong r IO ) ⇒ Reference w r s t a b
```

Creating references in a generic way is not easy when using library types that can only be used in certain monads. For example, the operations for `IOLens` provided by the standard library are working in the `IO` monad. However using the `liftUp` function, the operations originally defined for the `IO` monad can be used in any monad as strong as `IO`. Using this technique, we can now define the reference `ioref` that can access the content of an `ioref` using the `IOLens` operations provided by the standard library and `liftUp`. 
3. Design in-language abstractions for handling complex data structures

\[
ioref :: \text{IOLens (IORef a) (IORef a) a a}\\
ioref = \text{Reference}\\
\{ \text{refGet} = \lambda s \rightarrow \text{liftUp (readIORef s)}\\
\quad , \text{refSet}\\
\quad \quad = (\lambda a s \rightarrow \text{liftUp (writeIORef s a)}\\
\quad \quad \quad \gg \text{return s})\\
\quad , \text{refUpdate}\\
\quad \quad = (\lambda f s \rightarrow \text{liftUp (readIORef s)}\\
\quad \quad \quad \gg f\\
\quad \quad \quad \gg (\lambda a \rightarrow \text{liftUp (writeIORef s a)}\\
\quad \quad \quad \gg \text{return s})\\
\}\]

References for other IO-based data structures, like mutable variables (MVars in Haskell) can be defined in the same fashion.

3.3 Derivable Partial Locking for Algebraic Data Types

Thesis III. proves that a method of automatic partial locking with a configurable amount of granularity can be implemented in Haskell while keeping the business logic and data representation abstract. The thesis demonstrates this with the design and implementation of an embedded domain-specific language for automatically making program representation thread-safe.

Programming parallel algorithms is complicated when the representation of business data is mixed with different synchronization primitives that must be used with their own separate operations.

We offer a solution to transform program data into a representation that is thread-safe, in the functional language Haskell. Our goals are to enable the programmer to parallelize an algorithm without having to reimplement parts of the solution, and change how the representation can be used safely by multiple threads.

The solution is composed of generating the type of the shared representation, transforming values into the representation, generating accessors for the shared representation and functions that act like the constructors of the original data type. Generating constructor functions and references helps the programmer to adapt his program for concurrent execution.
To be able to quickly change how multiple threads can work on the data structure, this section describes a method to control the way program data is shared. This information is separated from the program logic. This is called the sharing configuration of the program. The configuration declares how the access to the program state is controlled.

Every type can be configured independently to define how instances of the given type in the program state are shared. In our solution, the synchronization in shared programs is controlled by the datatypes in the program state. The synchronization primitives are parts of the data structure. This way, the actual use of the exclusion primitives can be guaranteed.

The execution of a program using our sharing method is done in three phases, as seen on Figure 3.3.

1. The problem is analyzed and the correct initial state of the program is shared.

2. The problem is solved by a number of threads working on the shared representation.

3. The result state is merged.

If the program runs continuously its state can be inspected any time while it is running, as seen on Figure 3.4.

3.3.1 Background

Hawkins et al. described a system [18] where data structures are synthesised from an abstract relational specification and a decomposition. A relational specification
3. Design in-language abstractions for handling complex data structures

Figure 3.4: Sharing version of a program that runs continuously

is similar to the pure ADT we use and decomposition is similar to our configuration. They applied the theory to concurrency in a subsequent paper [19].

We emphasize controlling the granularity at which the generated parallel computation locks objects. It can have a significant impact on the overall performance. This topic is thoroughly inspected at the context of automatically parallelized programs in a paper by Diniz and Rinard [20].

Our method is practice-oriented, and that was inspired by Simon Marlow’s book on the practical side of parallel programming in Haskell [21]. It has excellent chapters on MVars and Threads, and parallel programming using threads.

Other technologies exist for parallelizing algorithms. The Parallel Java streams [22] and C# PLINQ [23] can be used to parallelize a collection. Each thread is working on a separate part of the collection. Our goal is to parallelize heterogeneous data structures, not just homogeneous collections, and to enable different threads to access the same data in a thread-safe way.

Combining parallelism with partial locking as a method to enable multiple threads to access different parts of a data structure is a well known and thoroughly researched topic in the field of database design [24].

3.3.2 Partial locking

Partial locking is a way to share a data structure and enable working threads to lock parts of it without interfering with each other. This can improve the scalability and performance of the whole program, because it reduces the time each thread spends waiting. The resulting data structure still ensures consistent use by multiple threads.

The shared database supports two kinds of locking: read locks and write locks. Reader threads can access the same part of the data concurrently. A thread cannot
lock a part of the data for writing if another thread reads or writes the data or some part of it at that time. If the system is configured right, each thread will only lock the data part it needs for the computation.

For example, take a list of some type. If different elements of the list should be operated on independently but elements can only locked by one thread, it can be manipulated by partial locking.

### 3.3.3 Derivable Partial locking

Partial locking is a good strategy for making data structures thread-safe. However implementing it manually is tedious and error-prone. Parallelism-related code may be mixed with the business logic of the application which is an example of bad design.

We offer a solution to automatically derive the implementation of partial locking. Using this method the programmer writes a short configuration for partial locking that is a high-level description of the granularity of parallelism for the application. The configuration is completely separated from the business logic of the application.

Our method relies on a shared representation of the business data. To use our method the application logic should be implemented on the shared representation. To make this easy we provide references and constructors as well as automatically generating the shared representation from the user-provided configuration.

### 3.3.4 Shared representation of a data type.

Lets define the following data types for constructing the shared representation.

#### 3.3.4.1 Skeleton representation of Data Structures

Definition 1: Elements of the Skeleton representation

- Mem$_n$ is a simple tuple type for collecting $n$ members of arbitrary types. We defined a function for constructing them.

```
data Mem$_n$ t$_1$ ··· t$_n$ = Mem$_n$ t$_1$ ··· t$_n$
mem$_n$ = Mem$_n$
```

- Alt$_n$ is a union type that can have $n$ states each having a value of possibly
different types. The function $\text{alt}_n^i$ creates the $i$th alternative of the possible $n$.

$$\text{data } \text{Alt}_n^1 \ t_1 \ \cdots \ t_n = \text{Alt}_n^1 \ t_1 \ | \ \cdots \ | \ \text{Alt}_n^n \ t_n$$

$$\text{alt}_n^i = \text{Alt}_n^i$$

- Skeleton is a composition of the Mem$_n$ and Alt$_n$ types.

The Skeleton representation is an alternative representation of ADT types.

### 3.3.4.2 Node types

There are five different node types, each representing a unique way to control accessing the shared representation between threads. As it can be seen on Figure 3.5, they can be ordered by how much parallelism they allow. Of course, more powerful locking mechanisms have a higher cost in terms of computation and memory overhead.

- **Clean** states that the configured data type must not be converted into a shared representation. *Clean* data cannot be accessed by multiple threads. *Clean* data in itself is immutable, but it can be part of a mutable database. *Clean* representation has no additional costs.

- **Noth** provides no extra protection for the data. It is similar to *Clean*, but the parts of the data configured to *Noth* can have a different configuration.

- **Prim** guarantees mutual exclusion for a part of the data. At any time of the execution, only one thread can have access to a part of the database that is configured to *Prim*. Threads trying to access the protected data will queue up, and access the resource in a first-come-first-served (FCFS) order. *Prim* is relatively low-cost, it is implemented by one synchronization primitive.

- **Multi** allows multiple threads to access the data structure if their actions do not interfere. This applies to multiple reader threads or threads updating different parts of the data. However, it does not guarantee fairness. If a thread would like to gain exclusive access to the data, it may have to wait forever, if threads
that can share the data access it frequently. *Multi* is more expensive than *Prim*.

- *Fair* provides a protection that allows non-interfering threads to work simultaneously and guarantees fair FCFS access for all threads. Fair nodes use concurrency primitives to implement a shared lock [25], but their cost is higher than that of *Multi* nodes.

It is clear that a good configuration strategy is needed to reach an optimal performance. Common sense dictates that nodes on the upper level of the representation are configured to high-level types and nodes on the lower level (that appear in higher numbers) are configured to types with low cost. Typically predefined types like *Char*, *String* and *Int* are configured to *Clean*.

We generated references for these nodes to be able to access the data inside them. For example, check the implementation of the prim node. Prim contains a mutable variable containing the representation of the element (parts of which can be also shared, therefore it is processed by the *Distribued* type transformation). The \_prim reference accesses the element inside a Prim node.

```haskell
data Prim a = Prim { \_fromPrim :: MVar (Distribued a) }

\_prim :: Simple IOLens (Prim a) (Distribued a)
\_prim = fromPrim & mvar
```
We also created \texttt{newPrim}, \texttt{newMulti} and \texttt{newFair} generator functions for the node types. Clean and Noth nodes do not need generator functions since they do not contain additional data.

As it was mentioned, each type can have different configurations. If different instances of the given type should be given different protection levels, the original type should be configured to the neutral node type \texttt{Noth} and wrapper data structures can be introduced with a different configuration. There is no default configuration, so each type that can be a part of the shared program state must be configured.

### 3.3.5 Automatic generation of shared references and constructors

Because of the complexity of the shared representation, it would not be a feasible solution to burden the programmer with rewriting his code to work in a thread-safe way. Instead we provided a way to generate constructor functions and references for the shared representation of the data structure, according to the constructors of the original data structure and references as described in Section 3.2.1. The bodies of these references are generated according to the configuration of the types that are part of the representation.

The implementation of this automatic generation of references and constructors is using Template Haskell \cite{26}. We decided to store the configuration as instances of type families, because it can be queried from Template Haskell easily.

Generating shared references

The generated references are composed of two parts. The first part is a tuple reference for the index of the member accessed by the reference. Tuple references are simple means to access the \textit{n}th field of a simple data structure parametrized by the types of it’s members (for example, a pair, triplet, and so on).

The second part is a reference accessing the protected data from a node type. These references will be generated by inspecting the configuration for a given type, and can access the data in a protected, thread-safe way, depending on which kind of protection does the representation offer. These are the \_\texttt{clean}, \_\texttt{noth}, \_\texttt{prim}, \_\texttt{multi} and \_\texttt{fair} references.
For example, given a normal representation of a log message with a string message and a timestamp:

```haskell
data LogMsg = LogMsg { _logMsg :: String , _logDate :: Time }
```

If the user defines a configuration (using the Conf type family) to store String elements without any synchronization and Time elements with a simple exclusive locking, this directs the generated references to contain the same primitives.

```haskell
type instance Conf String = Clean
type instance Conf Time = Prim
```

Calling the generator function `makeSharedReferences` for the `Log` datatype will create the equivalent of the following references. While references generated as described in Section 3.2.1 would have the type `Simple Property Log Time` their shared counterpart work on the IO semantics, and map the relationship to the `Shared` counterpart of the original representation.

```haskell
msg :: Simple IOLens (Shared Log) String
msg = _1 & _clean

loggingDate :: Simple IOLens (Shared Log) (Shared Time)
loggingDate = _2 & _prim
```

Generating constructors

The generated constructors provide a way to build up larger shared data structures from smaller ones. The alternative method is to build up the data structure in its original representation and have to share it after that.

The shared constructor takes the shared arguments, creates the protection primitives for thread-safety and builds up the shared structure of the data to contain these protection primitives.

For example, lets inspect what kind of constructor functions would be generated from a list of log messages, if `[Log]` (list of Logs) is configured to `Multi` and `Log` is configured to `Prim`. The type variable for the element type of the list will be replaced
3. Design in-language abstractions for handling complex data structures

by Log. From [] the %[] operator will be generated, and from (:) the (%:) operator will be created.

Fortunately, the shared constructors can be generated automatically, by using the makeSharedCons function. This generator uses Template Haskell to generate the complex logic of creating the representation. The names of the constructor functions are based on the names of the original constructors. Here the shared representation is also referenced using the type transformer Shared. Take for example, the result of generating the shared constructors for the type [Log]:

( %[] ) :: IO ( Shared [ Log ] )
-- The empty list has no inner structure, the shared
-- version of [] is simply returned.
( %[] ) = return ( Alt 2 1 Mem 0 )

-- For the : operator, the element must be wrapped in
-- a "prim" node, and the rest of the list in a "fair"
-- node. The combine result is returned.
( %: ) :: Shared Log → Shared [ Log ] → IO ( Shared [ Log ] )
( %: ) x xs = do d ← newPrim x
d s ← newMulti xs
 return ( Alt 2 ( Mem 2 d d s ) )

It was an implementation challenge to enable the user to create constructor functions for concrete types, because the original constructors belong to the general type. But it was solved by taking the types of the constructors and transforming them by replacing the type variables with their actual types.

3.3.6 Formal definition of transforming types and values to their shared representation

Let's define a transformation of a type in a general way, according to a given type mapping $C$. We will refer to this transformation with $\text{Trf}_C$.

Definition 2: Generic transformations of types

\[ \text{Trf}_C :: \text{Raw} \rightarrow \text{ExtendedSkeleton}_C \]

Where $\text{Raw}$ is the set of all types, and $\text{ExtendedSkeleton}_C \subseteq \text{Raw}$ is the set
of all types that can be constructed using $\text{Alt}_n$, $\text{Mem}_n$ applying the type-level function $C$. The function $C$ has the type of $\text{Raw} \rightarrow \text{ExtendedSkeleton}_C$.

If $T$ is a scalar type, $\text{Trf}_C T = T$

If $T$ is not a scalar type, then $T$ has $n$ constructors with $r_1$, $r_2$ ... $r_n$ fields:

\[
\text{data } T = \text{Ctr}_1 m^1_1 \cdots m^1_{r^1} | \cdots | \text{Ctr}_n m^n_1 \cdots m^n_{r^n}
\]

In this case

\[
\text{data } \text{Trf}_C T = \text{Alt}^n (\text{Mem}_{r_1} (C m^1_1) \cdots (C m^1_{r^1}))
\]
\[
\cdots (\text{Mem}_{r_n} (C m^n_1) \cdots (C m^n_{r^n}))
\]

Definition 3: Generic transformation of values

We should also define the corresponding transformation of values, according to a given value-level function $c$:

\[
\text{trf}_c :: (t \rightarrow \text{Trf}_C t) \text{ where } c \text{ is } s \rightarrow \text{Trf}_C s
\]

If $t$ is a scalar type, $\text{trf}_c v = v$

Otherwise if $t$ has $n$ different constructors:

\[
\text{trf}_c (\text{Ctr}_i m^i_1 \cdots m^i_{r^i}) = \text{alt}^n_i (\text{mem}_{r_i} (c m^i_1) \cdots (c m^i_{r^i}))
\]

Definition 4: Mappings for sharing

And then we can define the mappings that give sharing as our transformation:

The configuration given by the user (using the $\text{Conf}$ type family) declares how much multi-threading support is needed for the configured types.
3. Design in-language abstractions for handling complex data structures

\[
\text{Sh}_\text{Conf} \ t = t \quad \text{if} \ \text{Conf} \ t = \text{Clean}
\]
\[
\text{Sh}_\text{Conf} \ t = \text{Trf}_\text{Sh}_\text{Conf} \ t \quad \text{if} \ \text{Conf} \ t = \text{Noth}
\]
\[
\text{Sh}_\text{Conf} \ t = \text{Prim} \ (\text{Trf}_\text{Sh}_\text{Conf} \ t) \quad \text{if} \ \text{Conf} \ t = \text{Prim}
\]
\[
\text{Sh}_\text{Conf} \ t = \text{Multi} \ (\text{Trf}_\text{Sh}_\text{Conf} \ t) \quad \text{if} \ \text{Conf} \ t = \text{Multi}
\]
\[
\text{Sh}_\text{Conf} \ t = \text{Fair} \ (\text{Trf}_\text{Sh}_\text{Conf} \ t) \quad \text{if} \ \text{Conf} \ t = \text{Fair}
\]

\[
\text{sh}_\text{Conf} :: t \rightarrow \text{Sh}_\text{Conf} \ t
\]
\[
\text{sh}_\text{Conf} \ v = v \quad \text{if} \ \text{Conf} \ t = \text{Clean}
\]
\[
\text{sh}_\text{Conf} \ v = \text{trf}_\text{sh}_\text{Conf} \ v \quad \text{if} \ \text{Conf} \ t = \text{Noth}
\]
\[
\text{sh}_\text{Conf} \ v = \text{newPrim} \ (\text{trf}_\text{sh}_\text{Conf} \ v) \quad \text{if} \ \text{Conf} \ t = \text{Prim}
\]
\[
\text{sh}_\text{Conf} \ v = \text{newMulti} \ (\text{trf}_\text{sh}_\text{Conf} \ v) \quad \text{if} \ \text{Conf} \ t = \text{Multi}
\]
\[
\text{sh}_\text{Conf} \ v = \text{newFair} \ (\text{trf}_\text{sh}_\text{Conf} \ v) \quad \text{if} \ \text{Conf} \ t = \text{Fair}
\]

Note: use of IO monad and its binding is omitted for the sake of simplicity.

\[\triangle\]

The programmer using our method can invoke the complete transformation of types and values after defining the configuration for partial locking. We will use \texttt{Shared} for the type-level mapping and \texttt{share} for converting normal values to shared ones:

\begin{verbatim}
  type Shared a = Trf_{shConf} a
  share :: a -> Shared a
  share = trf_{shConf}
\end{verbatim}

3.3.7 Case study

We created a case study for our method. We implemented a small application that consists of a number of worker threads, producing log messages and sending them through a channel to a logger thread, or querying log messages, that are sent back on separate channels. In this thesis we present code samples from the case study, but the whole package can be found at the project repository\cite{27}.

This case study application is implemented using our library, so the configuration
that controls the granularity of the parallelization is separated from the representation and the business logic.

We use a simple database representation shown on Listing 3.1 for the case study.

```haskell
data LogDB = LogDB { _criticalErrors :: LogList, _errors :: LogList, _debugInfos :: LogList, _startDate :: Time, _lastLogDate :: Time }
data Log = Log { _loggerThread :: String, _msg :: String, _loggingDate :: Time }
data LogList = LogList { _logListData :: [Log], _logListNum :: Size }
data Size = Size { _sizeInt :: Int }
data Time = Time { _year :: Int, _month :: Int, _day :: Int, _hour :: Int, _minute :: Int, _sec :: Int }

Listing 3.1: Representing the business logic in the case study

An excerpt from the original version of the main loop is on Listing 3.2. On receiving a LogThat request with the log message, it performs 3 updates on the logDB state variable. First, it sets the time of the last logging to the current time, then it adds the log message to the list of messages, and then increments the size of the list by one.

case q of
  LogThat log → do
    time ← getTime
    (lastLogDate != time)
    >>= (debugInfos & logListData !~ (log %:))
    >>= (debugInfos & logListNum & sizeInt !- (+1)))
```
3. Design in-language abstractions for handling complex data structures

logDB

Listing 3.2: Original main loop of the case study’s logger

For the shared version of the case study many different configurations can be written – depending on the granularity of parallelizm required (one of them is shown on Listing 3.3) – to enable multiple threads to work on the same database simultaneously.

```haskell
type instance Conf LogDB = Fair

- type instance Conf LogList = Fair
- type instance Conf [Log] = Fair

- type instance Conf Log = Prim
- type instance Conf Time = Prim
- type instance Conf Size = Prim
- type instance Conf String = Clean
- type instance Conf Int = Clean
- type instance Conf Char = Clean

Listing 3.3: Configuration of the case study representation
```

For the representation, the automatic shared reference generation feature is used, alongside with the shared constructor creation described in Section 3.3.5. Both generators can be called simply, for example `makeSharedReferences ''LogDB` and `makeSharedCons ''LogDB` generate the references and shared constructors for the type `LogDB`. (The '' notation is needed to get the name of the type to invoke the generator functions.)

Compare the modified main loop on Figure 3.4 with the original one on Figure 3.2. We had to wrap the database into a simple wrapper data type and an IORef [16]. The simple wrapper is needed to enable the configuration of the outermost database type. The references for this two data types became the prefixes of our structural references, as it can be seen in the boxed parts of the modified main loop. The code did not have to change in any other way.

```haskell
case q of
  LogThat log →
    time ← getTime
    ioref & wrappedDB & lastLogDate != time
    >=> (ioref & wrappedDB & debugInfos
```
3. Design in-language abstractions for handling complex data structures

& logListData "~ (log %:))
⇒ (ioref & wrappedDB & debugInfos
& logListNum & sizeInt !- (+1)))
logDB

Listing 3.4: Main loop of the case study’s logger

3.3.7.1 Evaluation of the case study

To measure the complexity and performance of the derivable approach to partial locking we created two implementations of the same simple data storage application. In one of the implementations we used the derivable partial locking technique described in this section. In the other implementations we used concurrency primitives manually.

Complexity of the solution

To compare the complexity of manual and derivable approaches to partial locking we compared the number of lines in each implementation.

On Table 3.6 the effective (executable) lines of code are categorized as Representation (Repr. eloc.), application logic (Logic eloc.) and configuration (Conf. eloc.), the latter only present in the derivable solution where there is a configuration present. All lines of code are counted in each implementation for all of the categories.

It is easy to see that partial locking can be implemented in fewer lines than using concurrent data structures manually. However, the relatively low difference is more important, because it is present in the innermost loops of the application that only contain a few lines. As the manual locking example was implemented we needed to write the functions for handling concurrent lists manually. In a more complex application these differences would be far more impactful.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual locking</td>
<td>52</td>
<td>57</td>
<td>0</td>
<td>109</td>
</tr>
<tr>
<td>Partial locking</td>
<td>33</td>
<td>54</td>
<td>9</td>
<td>96</td>
</tr>
</tbody>
</table>

Table 3.6: Effective lines of code in different implementations of the case study
Performance of the solution

To measure the overhead caused by the flexible derivable locking method we measured and compared the performance of the manual and derived parallelization strategies in different scenarios. The amount of read and write operations were different in each scenario. On Figure 3.7 the x axis shows the different scenarios while the performance of the implementations is shown on the y axis. The performance is measured in transactions performed per second.

The measurements were done on a Lenovo T440p laptop, that has an Intel Core i7\textsuperscript{TM} processor with 16GB of DDR3 RAM. Each measurement was performed 20 times and the results was averaged. We can see that all solution that using primitive concurrency operators (Manual locking) does not increase performance more than 25%. Manual locking is implemented by using the concurrency primitives of GHC. Using it results a rather complicated and hard-to-maintain code for representation and usage. The better performance of the edge cases is explained by Haskell’s lazy evaluation. When the application mostly reads data and never writes, the reads will be much simpler and can be performed quicker than in a balanced situation. We did not compare our results to the performance of the single-thread solution since the performance gain depends more on the hardware configuration than on how well multi-threaded software is implemted.

The results show that although the library described in this section gives a higher level of abstraction than the concurrency primitives of GHC, its performance is close to the performance of using the primitives.

3.4 Conclusions

Thesis I. presents a method to implement property abstraction in Haskell with monadic semantics. This method introduces references, property abstractions that are first-class language elements, can be combined and can represent multi-valued properties. References can access data with monadic (for example, IO) operations, like synchronized data, file contents, remote data and databases. This method relies on directly storing getter, setter and updater functions in a data structure.

The references are defined with concrete types for context and referenced element,
but with flexible types for reader and writer monads to be usable in many kind of computations.

In Thesis II., we have shown how complex data structures can be made thread-safe with an abstract configuration. This configuration separates the implementation details of parallelization from the business logic of the application. The section provides ways how single-threaded operations can be automatically transformed to enable multi-threaded execution.

Both libraries are fully implemented and accessible\cite{28} \cite{27}. Both libraries are expressive tools for writing Haskell applications on a more abstract level. References allow properties to be used in a more abstract way while the automatic partial locking enables the programmer to change how the software is running in parallel, without touching the business logic.
3.5 Contribution

The results presented in this chapter were published in two conference papers [29] [30]. On both papers I worked with Zoltán Kelemen, a student of Eötvös Loránd University, Faculty of Informatics. His main contribution was designing and implementing the parallelization primitives and their use for the derivable partial locking.

My idea was to use property abstraction to making programs thread-safe automatically. I designed and implemented references for this use case. Only later did I realize that there are a wide range of applications for the library. Apart from co-designing the method and implementing the library with Zoltán, my personal contribution for derivable partial locking was designing how each part of the parallel representation was automatically generated, including constructors and references that work on the shared representation.
4. EXECUTING UML MODELS AND HASKELL PROGRAMS 
BY EMBEDDING AND CROSS-TRANSLATION

This chapter of my dissertation is focused on providing higher-abstraction level or 
better performance by transforming the source code of one programming language 
into the source code of another. The theses in this chapter state that cross-language 
transcompilation can be used to enable higher-level programming of systems with 
performance restrictions.

Thesis IV. proves that it is possible to represent UML models in Java code and 
to generate Java code from existing UML models. The thesis demonstrates this by 
presenting two tools designed by the author and colleagues that transform UML models 
into Java programs and transform Java code (representing UML models) into UML 
models.

4.1 Introduction

Execution of high-level programming languages and models requires non-trivial compi-
lation and/or interpretation. To achieve good performance usually several transforma-
tion steps are taken. Additionally, there are several causes that are limitations 
on generating executable code.

- For some languages it is more convenient to define the mapping to an already 
established language, than to describe the behavioral semantics of the language 
constructs and to generate code according to the required behavior.

- For other languages, especially modeling formalisms, the execution semantics 
are defined from a theoretical viewpoint, without considering the actual ef-
fectiveness and convenience of running these programs (or models). One of
the possible ways to execute such programs is to map them to languages with established and efficient execution.

- In other cases, the platform’s limited capabilities prevent the execution of the program or model. It may be that the compiler of the language is not available on the platform, or it is not capable of running the programs generated by the compiler.

In this thesis, we compare three approaches for overcoming the limitations of code generation. In our first approach, we map the UML modeling formalism to Java source code to enable efficient code generation through the Java compiler. In our second approach, we take the approach further and actually embed the UML models in the Java language, reusing not only the compiler, but also the already existing development tooling. In our last approach, we map the functional programming language Haskell to C preprocessor macros for compatibility.

## 4.2 UML Model Execution via Code Generation

Executable UML [31] models define both behavior and structure of software. These models can be executed, debugged, and tested independently of the target platforms, providing early validation. Model compilers translate them to efficient, platform-specific target code.

What makes executable modeling unique among modeling approaches is the ability to test and debug the models without translating them to actual program code. These two requirements make two distinct constraints on the method of execution. For automated testing, the model is exercised on a configured set of test cases as part of nightly testing or sanity checks before making a change in the model. For this, the best tooling is a simple command line tester, with good runtime performance. For interactive debugging, the model needs to be run with close interactions with the development environment that provides generic debugging features (breakpoints, stepping, variable view) and model-specific debugging features, such as the animation of state machines.

To aid fast automated test execution and interactive debugging, we developed a method of executing UML models via code generation. We chose Java as the
target language for code generation. This method leverages the existing support for debugging Java programs and makes running automated tests efficient. It is important to note that our goal is executing models for testing purposes, and not to be used in production.

4.2.1 Background

*Foundational UML*, or fUML for short [32], is a standard defining formal execution semantics for a subset of UML. The goal of fUML is to be a basis for defining precise semantics for richer UML subsets (like the PSCS [33] standard). A Java-like textual syntax for fUML, called Alf [34] is also standardized. There are fUML and Alf reference implementations available. The fUML reference implementation is written in Java, and follows closely the formal semantics definition of the standard. The Alf reference implementation is integrated in the Eclipse environment using Xtext for parsing and OCL (Object-Constraint Language) [35] for semantic checks. Alf execution is provided by transformation back to fUML activities and leveraging the fUML implementation. The efficiency of model execution was ignored in these implementations.

The above mentioned reference implementations themselves do not provide model debugging or animation support. Moka [36] is an extension to the Papyrus UML editor [37]. It simulates UML activity diagrams and provides basic debugging support, such as breakpoints on actions. Moka uses the diagrams of Papyrus as graphical front-end for simulation and debugging, and uses a modified version of the fUML reference implementation for the execution logic. Moka also provides an interface which allows the definition of new execution engines.

Moliz [38] is a testing and debugging framework for fUML activities. It defines a test specification language, and extends the fUML reference implementation with debugging and tracing capabilities. The execution traces are used to decide if a given test case passes or fails. Since Moka and Moliz use the fUML reference implementation, the performance limitations discussed earlier also apply to these projects, questioning their scalability in the mass test execution use case.

Topcased [39] is a set of plugins for the Eclipse platform, which is mainly aimed at the implementation of critical embedded systems. It uses Papyrus for model editing,
and adds simulation capabilities. Topcased also provides visual model simulation for
state machines, but no breakpoint support is implemented, neither have we found
any way to use it for automated testing.

4.2.2 Model execution and model compilation

In case of model execution via code generation a natural question arises: What is
the difference between model execution (simulation) via code generation and model
compilation? While they seem to be similar at first sight, they are highly different,
due to their different purposes:

- Model simulation has to follow as many execution paths during intense testing
  as possible, to reveal possible errors. Model compilers have to generate code
  that takes one execution path and is as performant as possible.

- Model simulation has to check if invariants of the model (e.g. multiplicity
  constraints) are kept at runtime. The code generated by a model compiler
  will perform no runtime checks, or just a limited amount, in order to meet the
  performance requirements.

- The platform of a model simulator is the one where models are developed.
  Model compilation targets a given platform, independent of the development
  one, and takes its specifics into account.

- Model simulation has to connect to the debug framework, while model compila-
  tion has no such obligation.

- Model simulation has to be prepared to replay execution traces. Model compi-
  lers have to support logging only.

- Model simulation can be a single threaded emulation of a concurrent model.
  Model compilers have to emit parallel code, if that is required by the target
  platform.

- Model simulation has to provide quick feedback about the correctness of the
  business logic captured in the model. Results of the model compiler are de-
ployed on the target platform, which can be time consuming, and the runtime results may include platform-specific errors.

4.2.3 Interpretation, code generation and JIT

An interpreter stores the internal state of an executed model (object instances, values of their attributes and actual state machine states, message queues of signals etc.) as data. It queries the model to find out the next action to take and changes the model execution state accordingly. Another possibility to achieve model execution is to compile the model to program code, then build and run it. It is also possible to combine interpretation and compilation using just-in-time-compilation (JIT). In this case the model is executed by an interpreter, but frequently executed or critically slow fragments are compiled, built and loaded into the process of the interpreter.

In order to compare the performance of the three discussed options, we have created an experiment with models limited to a single state machine. A predefined number of instances of the state machine is created and a given number of signals are sent to each of them. When a transition is triggered, action code snippets (assignments, conditionals and basic arithmetic) are executed in the transition effect, state entry and exit.

In the compilation cases, state machine logic is implemented via nested switch-case statements. The interpreter uses an event matrix to look up the next state using current state and the received event. In case of JIT, the action code statements are compiled and the state machine logic itself remains interpreted. Two variants of this solution have been implemented: one that generates Java source code and uses the Java compiler to compile it to bytecode, while the other one generates bytecode directly using the Javassist library [40].

Table 4.1 shows execution times in milliseconds of the four different implementations executing the same models with 1000 and 5000 instances and processing 10000 signals per instance. Two different machines/platforms have been used to increase confidence in the experiment: Configuration $C_1$ denotes a HP EliteBook8540w laptop with Intel Core i5 CPU @ 2.53 GHz and 4 GB RAM, running 64 bit Windows 7. $C_2$ is a HP EliteBook 9480m machine with Intel Core i5 CPU @ 2 GHz and 8 GB RAM, also running 64 bit Windows 7.
4. Executing UML models and Haskell programs by embedding and cross-translation

<table>
<thead>
<tr>
<th>Technology</th>
<th>1000 instances</th>
<th>5000 instances</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$C_1$</td>
<td>$C_2$</td>
</tr>
<tr>
<td>Interpreter</td>
<td>11436</td>
<td>7687</td>
</tr>
<tr>
<td>Generated code</td>
<td>496</td>
<td>354</td>
</tr>
<tr>
<td>JIT to Java</td>
<td>7295</td>
<td>4986</td>
</tr>
<tr>
<td>JIT to bytecode</td>
<td>1468</td>
<td>1332</td>
</tr>
</tbody>
</table>

Figure 4.1: Execution times (ms) of interpretation, generated code and JIT

The results show that, in this experiment, interpretation turned out to be 18-24 times slower than running generated code. The different versions of JIT compilation are better, but still 3-14 times slower than generated code. (We have carried out further experiments, with very similar results.) This means that designers of high-performance model execution engines targeting mass test execution have to seriously consider code generation as the technology to build upon.

4.2.4 Debug symbols and mappings

During compilation of Java class files, the compiler inserts the information needed to find the line of source code corresponding to a given instruction. However, interactive model debugging should support highlighting the actual state in running state machines or stepping through them as well as stepping over lines of action code or highlighting them. To provide these debugging features, a mapping between model elements and their generated Java source code is needed.

This problem is partially solved by JSR 45: Debugging Support for Other Languages [41]. It provides a standard way for correlating Java byte code with source code of textual programming languages other than Java. JSR 45 uses a data structure called Source Map (SMAP) to specify the mapping between the lines of code in the original source language (which could be for instance the action language of the modeling system) and the generated Java source code. These source maps are injected into the binary class files using the SourceDebugExtension attribute after their compilation. The Java Debug Interface can then be configured to use the given source mapping while accepting and reporting breakpoint and other location
information during a debug session. A single class file could have multiple attached source mapping information for multiple source languages. Each Source Map has a name, and the debugger will use this identifier to select the appropriate mapping.

Unfortunately, this source mapping facility works only for textual languages. However, for ordinary model elements, like states and transitions which are represented mostly graphically on the user interface, a virtual line mapping can be provided. For non-textual elements, the Source Map will contain virtual line numbers, and another mapping will be used to resolve these into the original UML elements. Like the Source Map, this data can also be created during Java code generation. While it is also possible to store this mapping from virtual line numbers to UML elements directly inside a class file using a custom attribute, the method presented in this thesis serializes the mapping into a separate binary file to be loaded by the debugger itself.

### 4.2.5 Using the Java Debug Interface

The Java Debug Interface (JDI) defines a high-level Java interface to access debugging capabilities of a running Java virtual machine. It is the front-end part of the Java Platform Debugger Architecture (JPDA) \[42\]. Eclipse Java development tools (JDT) also uses this technology to implement its debugger. As JDI supports the inspection and manipulation of the connected virtual machine’s state through a simple API, it is a convenient choice to provide interactive visual debugging for UML models. The API is provided under the `com.sun.jdi` package, bundled with JDK distributions.

JDI provides the following four ways to connect a debugger with a target process:

1. Debugger launches target
2. Debugger attaches to running target
3. Target attaches to running debugger
4. Target launches debugger

The default Java process launching mechanism of Eclipse JDT uses the third option, as it creates a listening connector and then launches the target process with
special command line options. These options are commanding the target virtual machine to connect to the debugger process through a socket. From this point, we can use a VirtualMachine object in the debugger to send commands to and receive events from the target virtual machine. This mechanism of JDT could be entirely reused to start a model executor process. Only the underlying IDebugTarget instance should be replaced to a custom implementation. IDebugTarget is an abstraction defined by the Eclipse Debug Framework (EDF), which is used to control a debugger from the user interface (like stepping, suspending and resuming), and to coordinate the presentation of the data fetched from the target process (for instance threads, stack frames and local variables).

To place a breakpoint in the execution process, the debugger creates a breakpoint event request. This request contains only the location of the given breakpoint. The calculation of this location involves the usage of debug symbols and mappings presented in the previous section. For example, when a breakpoint is placed on the entry of a state in a state machine, the location resolution will follow the next steps:

1. Calculate the fully qualified name of the Java class which contains the code for the given state
2. Fetch the virtual line number of the given state entry
3. Get a reference to the class from the target virtual machine
4. Use the class reference and the virtual line number to get a JDI location

As a Source Map is installed on the generated class, the mapping between virtual line numbers and lines of the generated Java source code is available. It makes it possible to resolve the virtual line number to a bytecode offset inside a method. Before resolving, the debugger can select which Source Map to use from the available ones using the setDefaultStratum method on the VirtualMachine instance.

Models that have to be handled by the model simulator may contain a large number of model elements, therefore they induce a large number of potential breakpoint locations. We have conducted an experiment to determine how the number of breakpoints affects performance. The experiment was run on an Intel Core i7 CPU @ 3.4 GHz and 8 GB RAM, running 64 bit Windows 8.
The test cases in Table 4.2 set individual breakpoints on statements (which, in the experiment, simply increase a counter) on separate lines. The row BPs shows the number of breakpoints; the test cases loop over them Passes times for a total of BP hits breakpoint hits. When the execution of the virtual machine hits a breakpoint, execution is handed over to the testing application.

The tests show that the execution time is quite dependent on the number of breakpoints: for the same number of total breakpoint hits (50,000), the execution time nearly doubles when using 1000 breakpoints instead of only 10. If we increase the number of breakpoints further, the degradation is even more apparent: the average time to hit 100 breakpoints (Run/100 row) doubles between 10,000 and 20,000 breakpoints.

Furthermore, breakpoints have to be set before execution can progress toward them. Setting a low number (less than 1000) breakpoints is almost instantaneous and not shown in the table, however, as the number of breakpoints increase, we see that the cost of simply setting them approaches 10% of passing through all of them.

These results show that a visual model debugger that is using generated code for execution must limit the number breakpoints it uses. Fortunately, the model debugger needs to keep breakpoints only on lines that correspond to model elements visible to the user (for animation) and model breakpoints set by the user. To further optimize performance, the debugging environment can disable breakpoints that are likely to be used again instead of removing them.

We have taken into consideration other methods of communication with the runtime. Because Java technology is used on both sides of the communication, using
virtual memory is not favourable. Sockets and files could be used to transmit information, but the debugger provides a higher-level interface.

Examples of using alternative modes of communication are presented in Section 4.3.2.

4.3 Embedding UML models in Java

Providing a practical toolchain for large scale executable UML modeling is challenging: version control, compare and merge functions, convenient editor, debugging support, high quality diagrams and model compilation need to be provided. On the other hand, the toolchain should be lightweight for scalability, stability and for low tool development costs.

While the development of modeling toolchain is the topic of active research, the methods of designing tools for textual programming languages have been established. To re-use the techniques designed for textual languages to modeling, we defined a textual representation of a subset of the executable UML models. To have a stable base for our executable UML we chose Java as the host language. This also have the benefit of being a widely used programming language and programmers are familiar with the elements of Java tooling. We did not extend or modify our host language in any way, enabling users to write their models using only well-known language constructs and our API. The current implementation is based on Java SE 8, we used the newer syntactic features of the language to provide a convenient, fast and easy-to-read syntax.

As an embedded programming language txtUML uses normal Java syntax, but represent the more abstract UML concepts. By using the modeling language the user can define her program using higher-level abstractions than it could be done in a programming language. UML is a widely used methodology to plan software implementation so implementing the software in the same methodology can make it easier to develop the planned design.

UML and Java have lots of shared artifacts, like classes, objects, methods and attributes, to represent these we are using their natural Java syntax (with some extensions, like using ModelClass as a base class for all UML classes instead of
Object). Other UML artifacts, like signals, associations, state machines (with states, transitions, etc.), ports and components and many others are modeled using Java language elements like classes, methods and attributes, with special base classes and/or annotations.

The following example shows the description of an executable model with two classes, a state machine in one of them, an association between the two classes and a signal that can be sent from one class to the other.

```java
package examples.counter;
import hu.elte.txtuml.api.model.*;

class S extends Signal {}

class Sender extends ModelClass {
    public void emit() {
        Action.send ( new S(),
            this.assoc(S_R.r.class).selectAny());
    }
}

class Receiver extends ModelClass {
    private int count;
    class Init extends Initial {}
    class Accepting extends State {}

    @From(Init.class) @To(Accepting.class)
    class Initialize extends Transition {}

    @From (Accepting.class) @To (Accepting.class)
    @Trigger ( S.class )
    class Accept extends Transition {
        @Override
        public void effect() {
            count++;
        }
    }
}
```
class S_R extends Association {
    class s extends HiddenOne<Sender> {}
    class r extends Many<Receiver> {}
}

To describe the structure of a model, no mutable language constructs (like variables) are used to prevent accidental modification of the model structure at runtime. This approach resulted in the fact that almost all model elements are represented by a Java type (a Java class, in most cases) with a special super type to show the kind of the particular element and also to inherit behavior which becomes important when executing models. To keep code free from string literals referencing model elements by name, we take advantage of Java reflection which let us refer to a type at runtime through its associated `java.lang.Class` object.

Annotations and generics (type arguments) are widely used as well to write static information in models embedded in Java. Annotations are suitable for adding data that is not always required (e.g. the trigger of a transition), explicitly naming properties (e.g. the @From and @To annotations) or containing primitive values (e.g. the @Min and @Max annotations which are used to write custom association end multiplicities; this feature is not presented in the above example). Generics can help to reference types when this information is also required at compile time, like in the case of association ends, as the this.assoc call has to return a collection of the desired type. These type parameters are retrievable at runtime as well because they are set in the declaration of a type and that can be inspected with Java reflection.

From the example, we can see how Java features like inheritance, embedded classes, class literals, generics, annotations can be used to make the representation of UML easier to read and write. Despite these powerful features of Java, some limitations of the language proved extremely hard to overcome. Type erasure, to begin with, deprived us of many possibilities to write things in a simpler way. The lack of value types forced us to use immutable classes which can be inconvenient for the users too, as they also have to manually implement custom value types in an immutable and therefore verbose way. Garbage collection gives us no opportunity to force the deletion of objects from the heap or at least to check whether the user’s code holds any references to them which would be helpful to effectively implement and dynamically validate model object deletion. The parameter passing rules of Java will
make it challenging to implement UML’s out and inout parameter passing modes. However, the greatest limitation seemed to be the single inheritance of Java, which made us unable to introduce multiple inheritance between model classes, which is allowed in UML. The default Java solution for this problem, the usage of interfaces, could not be applied here because Java interfaces are too limited in features to be used instead of classes and it would be very inconvenient for a user to create both the interface and the implementing class for a single model class.

In case of the action code, both our opportunities and requirements proved to be much less than in the case of the model structure. It is simple Java action code with the extension of public and protected methods of API types, most importantly, the class Action, whose static methods implement basic operations of the modeling language, like sending signals, linking associations or deleting model objects.

4.3.1 Automated testing

Apart from being familiar to the programmers, the Java syntax for executable UML is in itself executable. Since the model as described in Java is an executable Java program, testing the model is convenient, requires no exporting. All Java compatible libraries and tools can be used with the models. The test can be called from the command line and can integrate well with testing frameworks.

Executing the model as a Java program is also good for performance, and performance is important for automated testing. The performance measurement from Section 4.2 shows that executing Java source code is faster by a magnitude than using an interpreter to evaluate the same model.

Although creating a user-friendly API sometimes requires slight compromises on runtime performance, our experience so far is that the achieved performance is more than good enough for testing and debugging purposes. To create a performant program executable on the target machine, our framework also provides export for the C++ programming language.

4.3.2 Interactive debugging

The models are Java programs using the txtUML API and runtime library, therefore these can be executed and debugged in any Java development environment. This
usually includes breakpoint support, variable view, and session control functions like step over, step into, step out, resume and stop, that are also necessary features for model debugging.

The model execution runtime library adds two useful features: runtime validation and state machine diagram animation. Runtime validation generates warnings, for example, when multiplicity constraints are violated or signals are dropped. This feedback helps the modelers to find bugs early in the development process, even without generating code and deploying it on a target platform.

The runtime behind txtUML API does have sophisticated tracing capabilities. These are switched off by default when the program is run as a plain Java application. If the extra functionality is switched on, the Eclipse-side plugin makes a connection towards the runtime in order to receive trace information. Data is provided even about individual object states and are fully kept track of.

The trace data is on one hand used to provide the user with sophisticated warnings, and, on the other hand to animate the generated state machine diagrams. This is achieved by the CSS capabilities of Papyrus. Because normal debugging features still work in this mode, breaking or reaching a breakpoint gives the possibility to examine various model states on the paused animation.

### 4.3.3 Exporting to UML model

For visualizing and compiling the models we decided to export them into standard UML model format. The generated UML models are used as an intermediate representation for compilation to other programming languages and they can also be processed by external tools. The export process is currently implemented as a batch operation converting the whole model at once. It parses all Java source files and outputs an EMF-UML2 model. We tried two approaches for extracting information from txtUML models:

- Java reflection and AspectJ: This solution uses standard Java reflection to analyze the structural elements (for example classes, method signatures) of the code. However, Java reflection cannot provide information on method internals. Therefore we experimented with AspectJ to export operations. AspectJ can
inject aspects (additional method calls) to predefined points in the Java code, and these aspects can collect the necessary information to complete the export.

- Parsing: In this case we parse the Java code using JDT [43] and walk through the abstract syntax tree in order to translate the txtUML model to an EMF-UML2 model.

The first option provides an easier access at runtime to the program structure and requires no compiler involvement. However, the second approach provides a more complete and systematic analysis of the UML model. Since txtUML is intended to be used while leveraging the power of a development toolchain, we decided that it should be connected to an integrated development environment (IDE) and can use the compiler that is used by the IDE. Therefore we chose to parse the source code to analyse the model and make use of the more complete comprehension of the program that a compiler can offer.

Another dilemma is about the representation of action code in the UML model. One possibility to encode behavior in UML is using opaque behaviors: These are just strings labeled with the name of the language they are written in. We decided not to use opaque behaviors for two reasons: Polluting the UML model with action code in XtxtUML or JtxtUML syntax would introduce non-standard elements, limiting the compatibility with third party tools. Also, a model compiler would have to parse and type check these opaque behaviors and do reference resolution, which introduces a lot of complexity. Therefore we have chosen the other possibility, namely UML activities. This provides standard and language-independent action code format. On the other hand, it requires nontrivial translation logic both in the exporter module and in the model compiler. It is also a threat that UML activities are extremely verbose, and this might lead to scalability problems in case of large models with much action code (even a project with the moderate size of 100 thousand lines of code can contain millions of subexpressions that will each take a non-trivial amount of memory and disk space).

In this section we saw how embedding UML modeling into Java programming language can help software implementation on a high abstraction level while also providing the user with the same refined tooling capabilities that Java programmers are used to. The complete conclusions will be presented at the end of the chapter.
4. Executing UML models and Haskell programs by embedding and cross-translation

4.4 Defining C Preprocessor Macro Libraries with Functional Programs

*Thesis V* proves that functional programs written in Haskell can be transformed into C preprocessor directives while keeping their behavior. We demonstrate the *hs2cpp* tool designed by the author and colleagues that performs the transformation from Haskell to preprocessor directives.

In the previous two sections we saw how program transformation and embedding can help to define software using a higher abstraction level. In this section we will see how program transformation can solve compatibility problems to achieve wider platform compatibility while keeping the abstraction level.

Preprocessor macros are fundamental elements of the C programming language. Besides being useful to define constants and common code snippets, they also allow developers to extend the capabilities of the language without having to change the compiler itself. Possible usage are serialization, compile-time reflection, optimisation based on extra knowledge about the domain of the application or calculation of complex static data based on the actual compilation options.

While most C developers are familiar with the macro language, its structure and semantics generally does not allow to express programs in a straightforward way or using a higher level of abstraction. These problems are making it difficult to understand and develop program configuration systems and metaprograms.

During macro expansions the preprocessor does not allow to modify any global, mutable information. When every macro has a single definition that does not change during preprocessing (and no `#undef` pragma is used), referential transparency of macro invocations is ensured. These properties are making the macro system very similar to a purely functional language.

Since the preprocessor manipulates token streams, macros can be considered typeless. This shortcoming can easily lead to mistakes when metaprograms containing nested invocation of function-like macros are modified — especially when lists of tokens are simulating data structures.

Only a very few number of semantic checks are done on the metaprograms during processing (e.g. avoiding recursive macro expansions). Although an expansion
fails only when an inappropriate number of arguments are provided, the resulting source code is not guaranteed to be free of syntactic or semantic errors. Because the locations of these errors are often pointing into the preprocessed, not into the original source code, in most cases it is non-trivial to find the error in the metaprogram implementation.

By utilizing the similarity of preprocessor metaprogramming and functional programming, a purely functional language can be used to ease the development and maintenance by modeling macro definitions with functional programs \cite{44}. Our solution is to automatically translate functional programs written in Haskell to C preprocessor macro definitions. It is implemented as a software tool named hs2cpp. This section presents the ideas and applied techniques and the program transformation in a formalized way.

While we considered writing a small DSL to generate macros from, we decided to use Haskell instead. As it can be seen later, some features of Haskell can only be used with certain limitations when generating preprocessor code. Nevertheless, we found strong arguments for using Haskell with GHC as a compiler. For example, existing compiler features (parser, type checker, etc.) can be reused. Haskell is relatively well-known and has a plug-in system that lets compilation steps to be registered, working on the Core representation. This enables the use of Haskell's rich set of language features and extensions without having to define translation rules for them, since they are transformed into simpler constructs by the time the preprocessor code is being generated.

### 4.4.1 Related work

The Boost Preprocessing library \cite{45} provides macros for handling various data structures, including tuples, arrays, lists and sequences. Operations on boolean and integral values, like negation, addition and subtraction in limited ranges are also supported. It also provides emulation of conditional and loop control structures. Some list-processing operations, like filter, map and fold are also supported. This also shows that operations familiar from functional languages can contribute to writing preprocessor macros. Despite its rich features, it still could be hard to develop and debug complex preprocessor metaprograms using this library.
For platforms where not only C, but C++ compilers are also available, there are further options. Earlier researches are showed that C++ templates are Turing-complete [46]. Zoltán Porkoláb describes the connection between functional programs and C++ templates [47]. There is a chapter about embedding Haskell into C++ [48] and the conversion of these embedded functional programs using the internal representation of YHC, the York Haskell Compiler [49]. There are certain similarities between his method and our approach. First the YHC compiler is used to translate Haskell code sections, embedded into C++ code, into Yhc.Core representation. Then it is translated to a language called Lambda, defined by the author. Lambda code is finally translated into template metaprograms.

A different solution for generating C++ template metaprograms could be found in a paper by Ballantyne, Earl and Might [50]. Here the authors used the metaprogramming facilities provided by a functional programming language Racket to support EDSL development in C++.

The Boost library also includes a template metaprogramming library, called MPL [51]. It supports programming with similar constructs and data types as the preprocessor library shown earlier. The interface of this collection resembles the C++ Standard Template Library. Using template metaprograms as the generated language solves many problems our implementation must handle, for example, template metaprograms are allowed to be recursive. However, it is limited to C++ and cannot generate arbitrary code.

Starting from the C++ 11 standard [52], the constexpr specifier allows compile-time usage of variables and functions [53]. The specification supports limiting the call depth of constexpr functions. In case of the solution presented in this thesis, it was a necessity. The wide-spread GCC compiler supports compile-time programming with memoization. However, full constexpr support is available only from GCC version 4.7. This prevents legacy systems with older compilers from taking advantage of this compile-time programming method.

Another approach to support metaprogramming in C is to extend the language itself. Meta-C [54] introduces new programming elements into the language while it remains fully backwards compatible with the C99 standard [55] it is based on. It provides Turing-complete tools for analyzing and manipulating C programs at
compile-time. The language was implemented by its own, custom compiler. According to the project’s website [56], the last release was an alpha version in 2007.

Other macro languages have semantics that are different. For example the M4 preprocessor [57] is more expressive than the C preprocessor. It allows recursion which was a serious problem when specifying the transformations. It also allows the creation of new definitions while evaluating macros. It is, however, not a standard tool available for nearly all platforms like C preprocessor is.

4.4.2 Architecture

Our translation solution is designed as a plug-in for the Glasgow Haskell Compiler (GHC) [58]. Figure 4.3 summarizes the high-level architecture, including the most important data flows. The plug-in installs a Core-to-Core pass that does not change the program being compiled, rather creates C header files containing the generated macro definitions as a side effect. According to this design decision, the transformation is executed on the relatively simple Core syntax rather than the rich Haskell abstract syntax tree. The transformation (as mentioned earlier) takes advantage of the functional nature of the preprocessor macros. This is the reason why it is based on the Core representation instead of the C-- representation that is used by ordinary Haskell backends. For details about GHC’s compilation pipeline see [59].

As the implementation of the translation is a GHC plugin it can be used by invoking GHC with a special flag to indicate the used plugin. The translation can be integrated into any higher-level build solution.

The next two subsections give a brief introduction to the source and target systems of the translation. As both of them are well-described areas, only those aspects are included that are the most important for understanding the translation logic. At the end of this chapter, Section 4.4.4 presents the structure of the generated macro system, and introduces the notations used to define a transformation function.

4.4.2.1 GHC Core

In this section a simple extension for the $\lambda$-calculus is defined (see Table 4.4). We will use this formalism in the description of transformations. This extended $\lambda$-calculus has a very schematic form without any special syntactic extensions. We are not
4. Executing UML models and Haskell programs by embedding and cross-translation

Our solution generates preprocessor macros from GHC Core language that is based on an extended polymorphic lambda calculus. This calculus is named System F₇₉ [60], and it is a practical compiler intermediate language based on System F [61].

The representation of a program in GHC Core consists of multiple parts. For our translation the most important part is the list of bindings. Bindings contain evaluable expressions for the execution of the program. A simple binding contains a unique identifier and an expression. A recursive binding group contains a list of name-expression pairs that can refer each other. The defined transformation only handles type-correct GHC Core representation that is created from a well-typed program.

There are six different constructions in GHC Core that are transformed. These constructions and there corresponding λ-expressions can also be seen in Table 4.4

- A variable contains a unique identifier that references a definition.
- A literal can be a character, string, simple int, 64 bit int, float, etc. GHC Core distinguishes eleven different kinds of literals.
- An abstraction has a unique name bound to the argument of the abstraction,
4. Executing UML models and Haskell programs by embedding and cross-translation 84

<table>
<thead>
<tr>
<th></th>
<th>(\lambda)-calculus</th>
<th>GHC Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>(x)</td>
<td>(\text{Var } x)</td>
</tr>
<tr>
<td>literal</td>
<td>(l)</td>
<td>(\text{Lit } l)</td>
</tr>
<tr>
<td>abstraction</td>
<td>(\lambda x.e)</td>
<td>(\text{Lam } x\ e)</td>
</tr>
<tr>
<td>application</td>
<td>(e_1 e_2)</td>
<td>(\text{App } e_1\ e_2)</td>
</tr>
<tr>
<td>pattern matching</td>
<td>case of (p_1 \rightarrow e_1) : (p_n \rightarrow e_n)</td>
<td>(\text{Case } e\ v\ t) [(m_1,\ldots, m_n)]</td>
</tr>
<tr>
<td>let expression</td>
<td>let (x = e_1) in (e_2)</td>
<td>(\text{Let } (\text{bnd } x\ e_1)\ e_2)</td>
</tr>
<tr>
<td>non-recursive binding</td>
<td>–</td>
<td>(\text{NonRec } n\ e)</td>
</tr>
<tr>
<td>recursive binding</td>
<td>–</td>
<td>(\text{Rec } \text{bnds})</td>
</tr>
</tbody>
</table>

\[
\text{where } m_i = (\text{tag}(p_i), \text{var}(p_i), e_i) \\
\text{bnd } n\ e = \text{NonRec } n\ e \\
\text{tag}(p_i) = \text{DataAlt } \text{ctor} \\
\quad | \text{LitAlt } \text{lit} \\
\quad | \text{DEFAULT}
\]

Table 4.4: \(\lambda\)-calculus and GHC Core

and an expression that can contain the new name.

- A function application contains two expression. The first is the function applied, the second is the argument.

- Pattern matching is represented by the \texttt{Case} constructor. Patterns \(m_1,\ldots, m_n\) are tuples with three elements: the tag of the constructor (\texttt{tag}), the captured variables (\texttt{var}) and the expression that is evaluated when the pattern matches. There are three kinds of matches: algebraic data type matches, literal matches and default matches. Core’s pattern matchings are always complete. If the individual matches cannot cover the whole range of values, a default match will be generated during desugaring. The \(v\) and \(t\) arguments contain GHC-specific information and they will be ignored by the transformation.

- A let expression is composed of a binding \(b\) and an expression \(e\). The binding’s
name can be used in the expression.

**Haskell source code**

\[
\text{add} \ (\text{Succ}\ a)\ b = \text{add}\ a\ (\text{Succ}\ b) \\
\text{add}\ \text{Zero}\ b = b
\]

**GHC Core**

\[
\begin{array}{l}
\text{(Rec add} \\
\text{\quad (Lam ds} \\
\text{\quad \quad (Lam b} \\
\text{\quad \quad \quad (Case (Var ds) wild typ} \\
\text{\quad \quad \quad \quad [ ( (\text{DataAlt Zero}), [], (Var b) )} \\
\text{\quad \quad \quad \quad , ( (\text{DataAlt Succ}), [a] \\
\text{\quad \quad \quad \quad \quad , ( App (App (Var add) (Var a))} \\
\text{\quad \quad \quad \quad \quad \quad (App (Var Succ) (Var b))) ) )} \\
\text{\quad \quad \quad \quad ] ) ) ) )}
\end{array}
\]

**\(\lambda\)-calculus**

\[
\begin{align*}
\text{add} &= \lambda a.\lambda b. \text{case } a \text{ of Succ } a \rightarrow \text{add } a \ (\text{Succ } b) \\
& \quad \text{Zero } \rightarrow b
\end{align*}
\]

*Figure 4.5: Different representations of add*

Figure 4.5 introduces an example represented in Haskell, lambda calculus and GHC Core representation after some syntactic simplification. The example code has two parameters, that are natural values created by constructors \textbf{Succ} and \textbf{Zero}. The return value is the sum of the two parameters.

### 4.4.3 The C preprocessor

As mentioned earlier, the C preprocessor operates by translating a token stream into another one. Although the preprocessor supports many directives, this section considers only file inclusion and macro definitions, because only they were relevant for translating Haskell modules. Description of all supported directives could be found in the C99 standard [55], while [62] specifies the exact algebraic semantics of the preprocessor.
File inclusion is implemented through the \texttt{#include} directive, which allows to insert tokens from the specified file to the point where the directive occurs. With the \texttt{#define} directive, the programmer can assign a token sequence (called body) to a single-token name.

When the preprocessor detects a macro invocation, it triggers an expansion: it replaces the invocation with the tokens from its body. In this section the “⇒” notation will represent macro expansions in the following code listings. Based on their arity, macros can be object-like (having no arguments) and function-like. In the second case, when a macro has parameters, their actual values will be substituted into its body upon expansion. A macro expansion could never result in a new preprocessing directive, so it is impossible to define a macro by expanding an existing one.

To guarantee termination, the preprocessor keeps a list with the names of currently expanding macros. When the preprocessor detects a macro invocation in a list of tokens, it looks up if the expansion list contains the source of the tokens. When such situation is found, the macro invocation will not be expanded:

\texttt{#define REC(x) 1 + REC(x)}
\texttt{REC(2) ⇒ 1 + REC(2)}

As a consequence, real recursion — including mutual — is not allowed so the preprocessor is not Turing-complete. However, recursion can be simulated to a certain depth and branching can be done by the preprocessor. These suggest that the C preprocessor is able to emulate a calculation that is guaranteed to terminate in a given number of steps.

Although recursion is disabled, nested invocations of a macro is supported within its arguments. Before the expansion of a function-like macro, the arguments are scanned, and all possible expansions will be executed before their substitution. This process is called argument prescan. After the resulting tokens are substituted into the body, the scan happens once more, and possible macro invocations will be expanded too. Having this second scan, the argument prescan looks unnecessary. However, it has an important effect: during the prescan, the name of the currently expanding macro is not yet appended to the expansion list. Argument prescan makes preprocessing strict, as each argument will be expanded before it gets substituted. Without this mechanism, the following nested substitutions would not happen:
# define ADD(x, y) ((x) + (y))
ADD(ADD(1, 2), 3)
⇒ ADD((((1) + (2)), 3))
⇒ (((((1) + (2))) + (3)))

There are two special operators in the preprocessor we need to mention. First, the stringification (#) operator creates a string literal from the tokens of a macro argument. It can be used for debugging and code generation. Second, token concatenation (##) could merge two individual tokens into a single one. For example, it makes possible to create macro names from multiple arguments, and expand them.

# define PRINT_(x) printf(#x)
# define CONCAT_(x, y) x ## y
# define ONE 1
# define TWO 2
PRINT_(ONE) ⇒ printf("ONE")
CONCAT_(ONE, TWO) ⇒ ONETWO

As the example shows, argument prescan is not applied to the operands of these operators. For this reason, a second expansion is needed to prescan their arguments:

# define PRINT(x) PRINT_(x)
# define CONCAT(x, y) CONCAT_(x, y)
PRINT(ONE) ⇒ PRINT(1) ⇒ PRINT_(1) ⇒ printf("1")
CONCAT(ONE, TWO) ⇒ CONCAT(1, 2) ⇒ CONCAT_(1, 2) ⇒ 12

As it will be detailed later in Section 4.4.6.8, this mechanism will have an important role in handling pattern matching expressions. It is the only way to branch in the body of a preprocessor macro.

4.4.4 The generated macro system

Haskell modules are compiled into C headers containing preprocessor macros. Any definition that is exported in its containing Haskell module will be a preprocessor macro with the same name. Import declarations between Haskell modules will be translated into preprocessor include directives. All the exported definitions of a Haskell module are transformed into macros that are usable by including the generated header file. From Haskell modules importing each other the translation creates
header files including each other. Like precompiled modules, pregenerated header files can be used by an included module. Existing macro definitions can be used in a new header file by including the generated headers. These similarities enable the use of prewritten macros as imported foreign functions in Haskell code. Invocation to existing macros can be integrated into a new, translated system.

While generating preprocessor pragmas we make sure that they don’t change the global state of the preprocessor. Therefore we avoid name collisions and do not use pragmas that modify pre-defined macros, like the \texttt{#undef} pragma.

The translation cannot be applied to IO computations, because preprocessor macros cannot perform file modification or network activity. Handling exceptional cases that can be done in an IO calculation in Haskell can be performed when using the macro definitions in a C source file.

Macro definitions in this section appear in their original syntax. For demonstration purposes, we added meta-syntax placeholders, surrounded by angle brackets. These templates can later be instantiated with different values to get concrete macro definitions. For example, the following template

\begin{verbatim}
#define ⟨name⟩(x) ((⟨value⟩) + (x))
\end{verbatim}

with substitution of \texttt{name = INCREMENT} and \texttt{value = 1} gives

\begin{verbatim}
#define INCREMENT(x) (1 + (x))
\end{verbatim}

4.4.5 Basic transformations

As we described the source and result of the translation process in the previous subsections we can explore how the Haskell representation is mapped to the preprocessor macro system.

4.4.5.1 Primary Data and Operations

The transformation described in this section generates preprocessor macros that use the \textit{Boost Preprocessing library} [45, 51]. Boost is used as a runtime library for the generated code. It provides a collection of various data structures and algorithms one can rely on: it supports representation and operations for tuples, sequences, boolean
and integral values. For example, the next listing shows three numbers stored in two different data structures.

```c
#define TUPLE_OF_NUMBERS (1, 2, 3)
#define SEQUENCE_OF_NUMBERS (1)(2)(3)
```

A tuple simply stores a fixed number of items, which are accessed by their position in the tuple. Sequences are preprocessor-optimized lists. Almost any kind of data could be stored in the place of numbers, including the possibility of nesting them into each other. As detailed later, these simple data types can be used to represent any kind of compound data, including abstract data types.

All of these primitive types and data structures come with limitations. Numbers are limited to the range of \([0, 256]\), tuples can have a maximal length of 64 elements, while sequences can hold at most 256 elements in the current version of Boost.

### 4.4.5.2 Objects in the Macro System

Three kinds of Haskell objects have to be represented by token lists when the system of generated macros is preprocessed. These are values, thunks and exceptions.

**Values** are data structures that represent a Haskell value. Values can be of primitive types or algebraic data types. For example, the number unboxed 3 is represented with a sequence of two elements, one that identifies it as a value, and one that stores the actual number:

```
(VALUE)(3)
```

**Thunks** are functions that can be partially applied. A thunk stores the name of the macro that must be expanded when all arguments are present, the number of arguments needed and the arguments collected so far. The parts of the thunk can be accessed by the `TH_NAME`, `TH_REQ_ARGS` and `TH_ARGS` macros. `TH_NAME` gets the name of the function, `TH_REQ_ARGS` gets the number of required arguments and `TH_ARGS` gets the values that are already applied to the thunk. As an example, the function \((+3)\) is represented as the following tuple:

```
(THUNK)(PLUS)(2)(((VALUE)(3)))
```

**Exceptions** are errors that can be handled where a macro of the generated macro system is used. The representation of an exception is also a sequence. The first
element identifies the object as an exception, and the second is the exception message:

\[(\text{EXCEPTION})(\text{Exception error message})\]

Exceptions appear in the generated code when the \texttt{error} function would be evaluated in the Haskell program. Common cases of exceptions include pattern matching errors, undefined behaviour (\texttt{undefined} or \texttt{⊥}) and illegal arguments given to a function.

Exceptions are propagated by the generated code when an exception appears as a function inside a function application or when an exception is the subject of pattern matching. In these cases the result of the expression will be the unchanged exception. Exceptions are handled with the \texttt{IS\_EXCEPTION} macro, which decides whether an object is an exception.

### 4.4.5.3 Representation of Haskell Values

To translate functional programs into preprocessor macros it is essential to find the correct substitutions of Haskell data structures. The resulting token list will contain parts that are the direct transformation of the Haskell values that were the parts of the original Haskell binding.

Fortunately, Haskell data structures are built from simple constructs. The transformation needs to represent primitive values and algebraic data types and encode these values respecting the lexical structure of the preprocessed file.

Boolean values are easy to implement. The range of boolean values is limited, therefore all operations can be implemented by a finite system of macros. For example, the logical negation can be implemented by concatenating the argument to a prefix, and generating two versions of the prefixed operation:

```c
#define NOT(b) NOT_ ## b
#define NOT_TRUE FALSE
#define NOT_FALSE TRUE
```

Unboxed integral values can be represented with numeric tokens. Integral operations can be performed by concatenating these tokens to macro names. The same token concatenation based approach can be used as for boolean values. Of course, this means that the range of integral values will be limited to the range of 0..256, a range defined by the Boost library. For example, see the \texttt{INCREMENT} macro on the
2-bit representation of integral values. Incrementing the largest number does not change its value, to prevent errors when `INCREMENT` would be used multiple times on the maximal number.

```c
#define INCREMENT(b) INCREMENT_ ## b
#define INCREMENT_0 1
#define INCREMENT_1 2
#define INCREMENT_2 3
#define INCREMENT_3 3
```

For practical reasons, characters should not be represented individually, only as part of text, as it can be seen in Section 4.4.6.5.

Floating point values cannot be represented properly because of the large number of these values and the lack of precise semantics of the operations. Haskell programs containing floating point values cannot be translated to preprocessor macros. The goal of this transformation is to provide help for writing preprocessor libraries and floating point operations are not often used in those, so we do not consider this a severe limitation.

Algebraic Data Types are the building blocks of all complex types in Haskell. In fact, the Boolean type shown above is also an example of algebraic data type as well as the boxed `Int` and `Char` types that are usually taken as primitive types.

Values of algebraic data types can be created using the constructors of the data type. Each constructor has a number of argument types. A value with an algebraic data type can be represented as a tag and a list of values. The tag identifies the constructor that is used to create the value. The values are the arguments of this constructor. For example the representation of the value `Just 3` is the following:

```
(VALUE)((JUST_TAG , ((VALUE)(3))))
```

### 4.4.6 Advanced constructions

While basic Haskell language constructs like primitive values, tuples, lists, functions and thunks are easy to represent, some language constructs are more problematic. The challenges are caused by the limitations of the target language, and the execution rules of C preprocessor.
4. Executing UML models and Haskell programs by embedding and cross-translation

4.4.6.1 Problem with recursive macros

There are cases when a macro call would be placed inside the body of the same macro. As Section 4.4.3 shows, the inner macro will not be expanded. This problem applies to the helper macros in our system. For example the APPLY macro (that applies a function to an argument and will be defined precisely later) will be called in the outermost expression and also in the definition of APP_BODY when the macros for the following expression will be generated:

Take the simple definition of a function application combinator (similar to the \$ operator in Haskell):\[\text{app } f \ a = f \ a.\]

\[\# \text{define } \text{APP BODY}(f,a) \Rightarrow \text{APPLY}(f,a)\]
\[\# \text{define } \text{APP} = (\text{THUNK})(\text{APP BODY})(2)()\]

Let's look at how the APP macro is expanded when it is applied during the evaluation of the app id x expression.

\[\text{APPLY (APPLY (APP, ID), } \langle x \rangle)\]
\[\Rightarrow \text{APP ((THUNK)(APP BODY)(2)(〈ID〉)), } \langle x \rangle)\]
\[\Rightarrow \text{APPLY (〈ID〉, } \langle x \rangle)\]

Here the APPLY macro should be expanded from a token that was produced by the expansion of the same APPLY macro. If multiple definitions of the macro are created, then \text{APPLY}_1 could be used in the first step and \text{APPLY}_2 in the third.

The transformation automatically detects that the two invocations must use a different version of the APPLY macro. It generates at least as many independent definitions for the macro as needed.

4.4.6.2 Translating recursive functions

The representation of recursive functions are macros that are expanded into a token list that can contain the invocation of the same macro.

\[\varphi' \left( \begin{array}{l}
\text{map } = \lambda f. \lambda l. \text{case } l \text{ of}
\text{nil } \rightarrow \text{nil}
\text{cons } h \ t \rightarrow \text{cons } (f \ h) \ (\text{map } f \ t)
\end{array} \right) = \]
4. Executing UML models and Haskell programs by embedding and cross-translation

#define CASE_nil() nil
#define CASE_cons(f,h,t) \n   APPLY(cons, APPLY(f, h), APPLY(map, f, t))
#define DISPATCH(l,f) CASE_ ## TAG(l) APPEND(f, ARGS(l))
#define map_BODY(f,l) TRY(DISPATCH,l,f)
#define map (THUNK)(map_BODY)(2)()

Here the name of the map macro appears in the body of the CASE_cons macro. The invocation of the CASE_cons macro is expanded from the map macro itself. The helper macro TRY checks the second argument for exceptions and only use it in the application if it did not raise an exception.

The solution to the problem is the same as in the case of the APPLY macro before. The only difference is that in this case the author of the Haskell function have to estimate how many times his function can appear in a call chain (expansion of nested macros). There is a default number, but it can be overridden by the user as it can be seen in Section 4.4.6.4. In the case of the map function, this is simple, as it is maximal length of a list that can be mapped. When the generated macro would be used in a way that exceeds the maximum number of nested applications, an exception will be raised. This approach will also support the use of mutually recursive declarations as long as no higher-order functions are involved.

4.4.6.3 Translating higher-order functions

One problem with higher-order functions is that they can receive themselves as arguments. For example, take the expression map (map f) ls. In the generated code the outer map function will receive the thunk of the partially applied map function map f. When the inner map is applied for the first time, the macro expansion will not happen because the application of the outer map already used the map macro.

Different macros (families of macros) can be generated for every use of the higher-order functions. For mentioned example, map (map f) ls would be translated into: APPLY(map^1, APPLY (map^2, f), ls)

However, there is a better solution: first analyse the representation and check if one instance of the higher-order function can call the other instance. Different macros must only be generated when one function can call the other. This modification
reduces the number of macros generated, but the evaluation will behave in the same way. This method generates false positives and can only be used inside a compilation unit.

The final solution to this problem is to track function usage dynamically. Information about function usage can be passed as an argument between calls, and it can be checked what is the first macro definition that is not used already in the call chain. This will cause a large overhead when evaluating macros of the generated macro system.

There is another problem when using higher-order functions. If they receive another function as an argument it can be evaluated multiple times, even if it is not a recursive function. For example, take the `until` function that applies the given `f` function until a `p` condition is satisfied. Depending on the condition `p`, the macro generated from `f` should be expanded multiple times in a single call chain, which will be prevented by the preprocessor.

These problems are similar to the problem with higher-order functions that receive themselves as arguments, and the possible solutions are the same. If the `f` function is not recursive, but used in a higher order function the user must be able to specify how many times can it be evaluated.

Fortunately, only a few higher order functions are often used in source code analysis and transformation. They are the simple list processing functions (`map`, `fold`), usually used on reasonably small lists.

### 4.4.6.4 Annotating definitions

There are cases when the transformation needs user-defined information to generate macros. For example such information is the maximal call depth of recursive functions. Two ways were evaluated to give such information to the macro generator plug-in.

By using annotation pragmas it is possible to store any value as an annotation for a top-level binding, type constructor or module. For example, annotate the recursive function `add` to be able to run for ten recursive calls. It looks like the following in Haskell code:

```
{-# ANN add (Recursive 10) #-}
```
add :: \(\mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}\)

\[
\begin{align*}
\text{add } Z \ n &= n \\
\text{add } (S \ n) \ m &= \text{add } n \ (S \ m)
\end{align*}
\]

In some cases this approach does not work properly. Core-to-Core transformations can rewrite expressions and create new bindings. For example, in some situations GHC collects mutually recursive functions into a tuple. The original definition only selects a function from this tuple. In this case there is no guarantee that the annotation will be found on the definition that was annotated in the source code.

Another solution to annotate definitions uses type classes to provide information for the transformation of a given definition. The information will be encoded in a type constraint that is always satisfied. This type class of the constraint does not serve any other purpose than to label a definition with arbitrary information.

\[
\begin{align*}
\text{add } :: \text{Recursive } 100 &\Rightarrow \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\
\text{add } Z \ n &= n \\
\text{add } (S \ n) \ m &= \text{add } n \ (S \ m)
\end{align*}
\]

When the add function is transformed, its type can be queried. The context of the type is analysed and constraints meaningful for the translation are collected. The result will be that 100 instances will be generated from the \text{add} function.

In both approaches, \text{Recursive} is defined in hs2cpp. In the first case it is a simple datatype with a numeric field, while in the second case, it is a type class with a numeric parameter.

\[
\begin{align*}
\text{class } \text{Recursive} \ (i :: \text{Nat}) \\
\text{instance } \text{Recursive} \ i
\end{align*}
\]

### 4.4.6.5 Storing textual information

One of the goals of our system is to provide a high-level abstraction for code generation. To support this we need to find a good representation for text. The conventional way to represent text in Haskell are Strings, that are lists of characters. There are two problems with storing textual information in Strings. First, the preprocessor operates on the level of tokens, therefore white space characters cannot be represented as tokens. Second, representing text with the Haskell Strings also comes with a big overhead because the list data structure contains many constructors. From these
constructors a lot of unnecessary preprocessor constructions would be generated, because the text is usually constant or a concatenation.

For these reasons an alternative Haskell type, named TokenList is provided for creating large sections of text. The text represented as TokenList is mapped to simple tokens in the generated preprocessor definitions, without any overhead. The construction of TokenLists is limited in Haskell. Text literals can have TokenList type if they contain a balanced number of parentheses and quotes and do not contain the # character. This constraint will be checked at compile-time. It is important to state that a Haskell String cannot be converted into a TokenList, because it would lose its white space information.

Two new operators are introduced in Haskell, that resemble operators of the preprocessor. Two TokenLists can be concatenated by the # operator. Two tokens can be used to create a new token by the ## operator. These two tokens must form a valid token together, for example "f" ## "()" is not a valid expression and will result in a runtime error.

"a\text{\_}b" # "c\text{\_}d" ⇒ "a\text{\_}b\text{\_}c\text{\_}d"
"a" ## "b" ⇒ "ab"

4.4.6.6 Strictness of the generated macros

As it is known, the evaluation of a Haskell program is lazy by default [64]. It means that an expression is only evaluated when the value of the expression is used. This means that exceptions and non-terminating subexpressions can be found in a terminating expression if they are not necessary to be evaluated.

As it was mentioned in Section 4.4.3, the evaluation of macros is strict, the arguments are expanded first, and then they are inserted into the macro body.

The generated macro system emulates lazy evaluation of Haskell, using token concatenation to prevent the preprocessor from eager macro expansion. This is done implicitly, for example, when translating case expressions. When the translation does not use token concatenation, but laziness is required, it can be archived using the APPLY\_WHEN macro described later.
4.4.6.7 General transformations applied

To be able to easily transform expressions and bindings, the structure of the intermediate representation must be altered.

Lambda-lifting

Local bindings like let $x = expr_1$ in $expr_2$ are eliminated before the transformation begins. They are replaced with top-level bindings, and any implicitly passed data becomes explicitly passed as arguments.

$$f\ p = \text{let } a = p \text{ in } a + 12 \quad \Rightarrow \quad f\ p = (a\ p) + 12$$

Closure conversion

One problem that appears when the bindings are transformed into macro definitions is that the scope of the bindings are changed by the transformation. For example see the transformation of the $\text{const} = \lambda ab. a$ function.

```
#define const (THUNK)(const1)(1)()
#define const1(a) (THUNK)(const2)(1)()
#define const2(b) a
```

The problem is that $a$ is not in scope when the $\text{const}_2$ macro is expanded. This problem is solved by explicitly passing arguments that are in scope in the Haskell program but are out of scope in the generated macro definition. This happens in the case of abstractions and pattern matches. The previous example will be transformed to the following:

```
#define const (THUNK)(const1)(1)()
#define const1(a) (THUNK)(const2)(2)((a))
#define const2(a, b) a
```

Type erasure

A GHC Core Cast expression performs a type coercion on the received expression. Because the Haskell type system is not used during the transformation, Cast
expressions are simply ignored and only the expression that is affected by Cast is transformed.

Because GHC Core representation contains type abstraction and type application, it represents types as expressions with the Type constructor. It can also represent type coercions with the Coercion constructor. Because such constructs are not needed, they are completely ignored while generating code.

4.4.6.8 Mapping of $\lambda$-expressions to Macros

The representation of an expression is a list of tokens that will be placed where the expression is used. Compiling the expression can also generate macro definitions, that will be inserted into the generated file sequentially. The transformation can be formalized as a function where $expr$ is a lambda expression and $macro$ is a list of C tokens, and $macrodef$ is a list of C preprocessor directives:

$$\varphi : expr \rightarrow (macro, macrodef)$$

It is practical to also define the transformation on top-level bindings. A single binding $(bnd)$ can be transformed into a system of macros:

$$\varphi' : bnd \rightarrow macrodef$$

A variable will be transformed into a token containing the unique name of the variable. The unique name prevents multiple variables or definitions with the same name interfering with each other in the generated macro system.

Literals will be transformed to the representation of their value.

Function application will be performed by adding a parameter to the thunk of the applied function. After application if the thunk is satisfied (it received the required number of arguments) the macro stored in the thunk will be expanded. Otherwise the argument is added to arguments collected in the thunk.

$$\varphi(fe) = \text{APPLY}((\varphi(f)), (\varphi(e)))$$

The $\text{APPLY}$ macro is a helper macro that adds a new argument to a (partially applied) function. If the function receives all arguments needed (point (1) in the
following listing), it performs the application by calling the macro that is stored in the thunk (2). Otherwise it reconstructs the thunk with an additional argument added (3). The evaluation must be explicitly delayed until it is needed (4).

```c
#define APPLY(f,a)
(1) IF( EQUAL( SEQ_SIZE(SEQ_ADD(TH_ARGS(f),a)), TH_REQ_ARGS(f)), \ EXPAND( \n (4) APPLY_WHEN( \n NOT_EQUAL( SEQ_SIZE(SEQ_ADD(TH_ARGS(f),a)), \n TH_REQ_ARGS(f)), \n (2) TH_NAME(f), \n SEQ_TO_TUPLE(SEQ_ADD(TH_ARGS(f),a))) ) , \n (3) (THUNK)(TH_NAME(f))(TH_REQ_ARGS(f)) \n (SEQ_ADD(TH_ARGS(f),a)))
```

The macros `EQUAL`, `NOT_EQUAL`, `SEQ_SIZE`, `SEQ_ADD`, `SEQ_TO_TUPLE` and `EXPAND` are Boost macros with their names simplified (omitting the `BOOST_PP_` prefix), their names are self-explanotory, but can be looked up at the Boost library documentation. The `APPLY_WHEN` macro performs a lazy evaluation, expanding the given macro only when the condition holds, otherwise it separates the macro name from the argument list with a `$` symbol to prevent eager macro expansion by the preprocessor. Eager macro expansion would produce preprocessing errors even if the branch that causes the errors is not selected by the branching macro. This is the reason why the same condition appears twice in the definition of `APPLY`, the tokens where the expansion is prevented with the `$` symbol will not be used. ($$ was chosen because it cannot form a macro call with the parentheses)

```c
#define APPLY_WHEN(p,f,args) f IF(p,,$) args
#define IF(p,t,f) CAT(IF_,p)(t,f)
#define IF_1(t,f) t
#define IF_0(t,f) f
```

A lambda abstraction is transformed into a thunk that expands the macro generated from the body of the abstraction when the argument is given.

\[
\varphi(\lambda x. e) = (\text{THUNK})(\langle body\_id \rangle)(1)()
\]

```c
#define ⟨body_id⟩(x) ⟨ϕ(⟨⟩)⟩
```
The list of macros performing pattern matching can be divided into three parts. The first one checks that the object matched on is a value. The second propagates the argument unchanged if it is an exception. Finally, the third concatenates the tag to a unique prefix for branching on different cases. If the matched value has algebraic data type, the tag is explicitly present. If it is a primitive value, the value is used as a tag. The macros generated for cases are based on the expressions for each case.

\[
\varphi \left( \begin{align*}
\text{case } e \text{ of } p_1 \to e_1 \\
& \vdots \\
p_n \to e_n \\
& \to e_{\text{def}}
\end{align*} \right) = \langle \text{try} \rangle(\langle \varphi(e) \rangle)
\]

```c
#define \langle \text{try} \rangle(\langle x \rangle) IF(IS\_EXCEPTION(\langle x \rangle), x, \langle \text{match} \rangle(x))
#define \langle \text{match} \rangle(\langle x \rangle) IF(\ 
 IS\_COVERED(\langle x \rangle), \langle \text{tag}(p_1)\ldots\text{tag}(p_n) \rangle), \ 
  \langle \text{dispatch} \rangle(x), \ 
  APPLY\_WHEN(IS\_COVERED(\langle x \rangle), \langle \text{tag}(p_1)\ldots\text{tag}(p_n) \rangle), \ 
  \langle \text{default} \rangle, ())
#define \langle \text{dispatch} \rangle(\langle x \rangle) \langle \text{prefix} \rangle_\# \# \langle x \rangle \langle \text{ARGS} \rangle(\langle x \rangle)
#define \langle \text{prefix} \rangle_{\langle \text{tag}(p_1) \rangle}(\langle \text{args}(p_1) \rangle) \langle \varphi(e_1) \rangle
...
#define \langle \text{prefix} \rangle_{\langle \text{tag}(p_n) \rangle}(\langle \text{args}(p_n) \rangle) \langle \varphi(e_n) \rangle
#define \langle \text{default} \rangle() \langle \varphi(e_{\text{def}}) \rangle
```

The \text{IS\_COVERED} helper macro decides whether the tag given during macro expansion is explicitly handled by this pattern match or becomes a default case. The \text{TAG} function extracts the tag from the representation of an algebraic data type while the \text{ARGS} extracts its arguments.

### 4.4.6.9 Data types and constructors

In Haskell, data constructors are used to create values of algebraic data types. The translator simply maps constructors without parameters to values. It also creates thunks from constructors that have arguments. After enough parameters are given the representation will be constructed as seen in Section 4.4.5.3. The next listing shows code generated from data constructors \text{Bool} and \text{Pair}. Even when the constructor has no arguments the representation contains a comma to separate the tag.
and the (empty) sequence of arguments.

```haskell
data Bool = True | False
data Pair a b = Pair a b
```

The result of the transformation is a set of macro definitions. The constructors of `Bool` are represented by the `True` and `False` macros. For `Pair` a constructor thunk `Pair_CTOR` will also be generated, so it can be curried.

```c
#define True (VALUE)((True_TAG,))
#define False (VALUE)((False_TAG,))
#define Pair_CTOR(a1,a2) (VALUE)((Pair_TAG,(a1)(a2)))
#define Pair (THUNK)(Pair_CTOR)(2)(
```

### 4.5 Conclusions

The main contribution of Thesis IV. and Thesis V. is to show how programs described in a high-level programming language or modeling formalism can be mapped to lower-level representations that can be executed. In these approaches we had to be aware of the limitations of the target language when designing how different language constructs are mapped from the higher-level language to the lower-level language.

Thesis IV. is aimed for execution and textual representation of executable UML models. The main motivation for this solution is higher runtime performance, required by automated testing and the ability to integrate with the familiar Java development tools. In the first part we design a method to generate Java code for executing the UML models. In the second part we go further and embed UML models into Java. Textual representation of the model enables the re-use of the rich development toolset for the Java programming language. This embedded DSL is available online[65).

Our paper Defining C Preprocessor Macro Libraries with Functional Programs [66] presents more examples and example transformations. In the paper we compare the performance of the preprocessor code generated from Haskell programs to the handwritten preprocessor code.

As a part of Thesis V. I also designed a method to automatically translate Haskell programs into C preprocessor directives. The solution is implemented as a plug-in that integrates into the Glasgow Haskell Compiler[67]. Because the simple Haskell
4. Executing UML models and Haskell programs by embedding and cross-translation

Core representation is used for the transformation, all language features and extensions can be used to generate preprocessor macros.

There are certain similarities between the domain of this problem and the problem that was described in Thesis I. However, here we used an existing programming language to make it easier for the users to generate preprocessor directives. In Thesis I, we defined a new language more suited for the task.

4.6 Contribution

The results presented in this chapter were published in two journal papers [68] [66] and one conference paper [69]. My personal contribution to the studies related to model execution is the design of program transformation from UML to Java and vice versa. I also developed model validation for UML models. To the study related to transforming Haskell programs into preprocessor macros, I was the main designer of the implementation of the library and my main work was in addressing the theoretical and practical problems that arise as the method was applied to the problem.
5. CONCLUSIONS

The dissertation addresses the problem of defining complex software and overcoming the abstraction gap between the developed software and the programming languages used to implement it. In particular we focused on cases where already existing high-level solutions cannot be applied to the problem because constraints on performance, tooling or the execution environment.

The process of my research

When working on the project that inspired my first thesis, our research was focused on developing a standalone programming language that satisfies the severe restrictions on the performance and the resources available to the generated code. It was an industry-motivated scenario, where programmers were using C and a special assembly language prior to our research. One of the main requirements of the program code was to work without using stack to store the call chain. We inspected already existing techniques and optimizations that allowed stackless execution. The results are presented in sections 2.2 and 2.3.

The development of the programming language was successful, and my research group managed to create an abstract way to program the special hardware. The results are presented in Section 2.4. However, we experienced a great resistance from the management and the programmers against using the new programming language. Probably the reason was that the tooling and support for the language was not refined enough. We were disappointed, but took an important lesson from the results, that small changes are more effective when trying to improve the level of abstraction in an industrial environment.

In the following years, I was involved in different projects of writing useful libraries for the Haskell programming language. In these projects the main focus was to create
a system that works well within the existing programming language, extending it to provide a higher level of abstraction, but without the intention of replacing or modifying it. I described the result in sections 3.2 and 3.3.

One of my colleagues, Máté Karácsony, developed a method of describing C preprocessor macros with elements of functional programming. The idea was interesting for me, because it connected to my research about describing complex systems with simple tools. I applied my previous experiences about Haskell and programming languages to develop a way to automatically translate Haskell programs into preprocessor macros. The results are described in Section 4.4.

Later I was introduced to executable UML models in a new industrial project. Here my research group was presented with the problem of executing these models. Previous solutions for executing the models were implemented as interpreters, but we knew that these were not efficient enough for automated test execution. We decided on translating the UML models into Java source code and executing them as Java programs. This allowed us to execute the models efficiently and reuse the already existing technologies for debugging Java programs. This is described in Section 4.2.

Gradually, we found out that the editor provided by a third party was not stable enough for supporting industrial model-based development. This was a serious problem for us, since we translated models to Java that had no textual representation, but a model file that was only accessible through the editor. In response we changed our approach and designed a way to embed the UML model into the Java language, as it is described in Section 4.3. As an embedded language, it was accessible by the very mature Java tooling, and we did not have to rely on one model editor supported by a third-party.

Contributions of the dissertation

- We inspected existing techniques and optimizations that allow writing recursive functions that run in constant stack space. We focused on Scala and Haskell, and covered tail-call optimization and continuation passing style.

- We introduced the Miller programming language for defining the behavior of high-performance software without using in-memory stack. We describe the
features of Miller that allow the high-level specification of strictly restricted
program, such as bubbles, interfaces, limited function calls and sandbox. These
contributions are supported by our paper on Miller [13].

- We designed a library that allow Haskell programs to define the handling of
  the program state on a higher abstraction level. It introduced the concept of
  references, effectful property abstractions. This is detailed in our paper on
  property abstractions [29].

- We provided a way to automatically define how the data structures are handled
  by multi-threaded execution. It enables the user to separate business logic and
  parallelization and automatically transform sequential code to run in a multi-
  threaded environment. This method is reported in our paper [30].

- We created a method of generating Java source code to execute UML models.
  This gives good performance for automatic test execution and enables the use
  of existing Java debugging tools. This contribution is supported by the paper
  [69] to which I also contributed.

- We introduced a representation of executable UML models as an embedded
  language in Java. Using this language provides a way to re-use the highly so-
  phisticated Java development tools as well as to efficiently and conveniently
  test the described models. Using the Java compiler, the embedded represent-
  ation can be analyzed and exported as an UML model. This technique is
  described in our paper [68].

- We designed a cross-language transformation that maps Haskell programs to
  preprocessor macros. This transformation utilize the similarities between pre-
  processing and purely functional programming languages, and enables the user
  to write complex preprocessing libraries on a higher abstraction level. This
  contribution is described in our paper [60].

Benefits of the research

The results of the dissertation can be beneficial for further research and develop-
ment in multiple ways. The research is focused around methods to design complex
systems, especially in cases when there are restrictions on the execution platform of the system.

The lessons shown in this dissertation are useful for the design of programming languages. The dissertation contains case studies of language design but also show the pitfalls of developing a new programming language. It also describes issues of support and tooling that are related to language design.

The experiences from the embedding languages and high abstraction libraries are especially useful because they can be easily applied to the already existing toolchain and can be used to refactor already existing code.

Finally, the last thesis contains two examples of cross-translation between languages. Some modern programming languages are already using this technique, but in my dissertation, I described cases where the translation between languages is a challenging task. The results may help language designers who are interested in developing languages that are translated to other languages for compatibility reasons or more efficient execution.
BIBLIOGRAPHY


(MoDeVVa) co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013). pp. 1-10.


[40] Javassist library.
http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/

[41] Java Specification Requests 45: Debugging Support for Other Languages.
https://jcp.org/en/jsr/detail?id=45

[42] Java Platform Debugger Architecture (JPDA)
http://docs.oracle.com/javase/8/docs/technotes/guides/jpda/

http://www.eclipse.org/jdt/


https://pdfs.semanticscholar.org/55a1/417c034899636e736cfb168071555641dece.pdf


https://wiki.haskell.org/Yhc

https://pdfs.semanticscholar.org/2de6/ec680a18a163af3a4ace0eda2f4df50923c78.pdf#page=65


https://pdfs.semanticscholar.org/d1f7/3d7c073c8c5547082001cf5c6db156ac479c.pdf


[56] The MetaC Language.  
http://www.maier-komor.de/metac/

[57] M4 preprocessor language.  


[61] B. C. Pierce (2002). In Types and Programming Languages, pages 341–344. MIT.


[65] The txtUML embedded domain-specific language and toolset.  
https://github.com/ELTE-Soft/txtUML


https://github.com/nboldi/hs2cpp


ADATLAP
a doktori értekezés nyilvánosságra hozatalához

I. A doktori értekezés adatai
A szerző neve: Németh Boldizsár
MTMT-azonosító: 10046233
A doktori értekezés címe és alcíme: Program abstraction and efficient execution
DOI-azonosító: 10.15476/ELTE.2019.217
A doktori iskola neve: Informatika Doktori Iskola
A doktori iskolán belüli doktori program neve: Az Informatika alapjai és módszertana
A témavezető neve és tudományos fokozata: Tejfel Máté, PhD
A témavezető munkahelye: Eötvös Loránd Tudományegyetem, Programozási Nyelvek és Fordítóprogramok Tanszék

II. Nyilatkozatok
1. A doktori értekezés szerzőjeként
   a) hozzájárulok, hogy a doktori fokozat megszerzését követően a doktori értekezésem és a tézisek nyilvánosságára kerüljenek az ELTE Digitális Intézményi Tudástárban. Felhatalmazom az Informatika Doktori Iskola hivatalának ügyintézőjét, Kulcsár Adinát, hogy az értekezést és a téziseket feltöltse az ELTE Digitális Intézményi Tudástárba, és ennek során kitöltesse a feltöltéshez szükséges nyilatkozatokat.
   b) kérem, hogy a mellékelt kérelemben részletezett szabadalmi, illetőleg oltalmi bejelentés közveteteléig a doktori értekezést ne bocsássák nyilvánosságra az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban;
   c) kérem, hogy a nemzetbiztonsági okból minősített adatot tartalmazó doktori értekezést a minősítés (datum)-ig tartó időtartama alatt ne bocsássák nyilvánosságra az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban;
   d) kérem, hogy a mű kiadására vonatkozó mellékelt kiadó szerződésre tekintettel a doktori értekezést a könyv megjelenéséig ne bocsássák nyilvánosságra az Egyetemi Könyvtárban, és az ELTE Digitális Intézményi Tudástárban csak a könyv bibliográfiai adatait tegyék közzé. Ha a könyv a fokozatszerzést követően egy évig nem jelenik meg, hozzájárulok, hogy a doktori értekezésem és a tézisek nyilvánosságra kerüljenek az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban.
2. A doktori értekezés szerzőjeként kijelentem, hogy
   a) az ELTE Digitális Intézményi Tudástárba feltöltendő doktori értekezés és a tézisek saját eredeti, önálló szellemi munkám és legjobb tudomásom szerint nem sértet véle senki szerzői jogait;
   b) a doktori értekezés és a tézisek nyomtatott változatai és az elektronikus adathordozón benyújtott tartalmak (szöveg és ábrák) mindenben megegyeznek.
3. A doktori értekezés szerzőjeként hozzájárulok a doktori értekezés és a tézisek szövegének plágiumkereső adatbázisba helyezéséhez és plágiumellenőrző vizsgálatok lefuttatásához.

Kelt: 2019.09.03.

a doktori értekezés szerzőjének aláírása

2 A kari hivatal ügyintézője tölti ki.
3 A megfelelő szöveg aláhúzandó.
4 A doktori értekezés benyújtásával egyidejűleg be kell adni a tudományági doktori tanácshoz a szabadalmi, illetőleg oltalmi bejelentést tanúsító okiratot és a nyilvánosságra hozatal elhalasztása iránti kérelmet.
5 A doktori értekezés benyújtásával egyidejűleg be kell nyújtani a minősített adatra vonatkozó közokiratot.
6 A doktori értekezés benyújtásával egyidejűleg be kell nyújtani a mű kiadásáról szóló kiadói szerződést.