Functional Debugging:

Rapid prototyping and debugging for DSLs using functional programming

Albert Gerritsen
18th April 2011
Abstract

Domain Specific Languages (DSLs) are languages designed to solve problems in specific domains. For software projects with a lot of functional requirements, a DSL could be designed to allow domain experts to write functional specifications. Implementation of such a DSL might be time-consuming, thereby delaying feedback on design decisions. Furthermore, there is a need for tools that support debugging of specifications. If specifications cannot be executed or debugged during development, chances of mistakes going unnoticed will be greatly increased.

We suggest that, by embedding the DSL in a lazy functional programming language, a prototype can rapidly be constructed. Functional programming allows for concise formulation of a prototype interpreter and offers interesting ways to leverage this prototype for debugging purposes. We describe a real world system that is being developed using a DSL. We demonstrate the feasibility of our approach by constructing an interpreter for a simplified version of that DSL. We also show how this interpreter can be used to build a delta debugger and a breakpoint debugger. Such a delta debugger will automatically minimize failure inducing input. Breakpoint debugging allows the user of the DSL to execute the specification step by step.

We hereby show that, for DSL based projects, functional programming offers an interesting platform for rapid prototyping of an interpreter and debugging tools. This provides feedback on design decisions and a means to debug specifications, early on in the development process.
# Contents

1 Introduction 1

2 TSL 4
   2.1 Requirements 4
   2.2 Architecture 5
   2.3 Functional Model 6
   2.4 Abstracting over time 7
   2.5 Input, state and output 8
   2.6 Towards a prototype interpreter 9

3 Interpreter 10
   3.1 Overview 10
   3.2 Meta model 11
   3.3 Execution context 12
   3.4 Values and types 13
   3.5 Timed values 14
   3.6 State 17
   3.7 Expressions 20
   3.8 Statements 22
   3.9 Summary 24

4 Debugging techniques 25
   4.1 Narrowing your focus 26
   4.2 Breakpoint debugging 27
   4.3 Beyond breakpoints 29
   4.4 Algorithmic debugging 30

5 Functional debugging 31
   5.1 Delta debugging 32
   5.2 Breakpoint debugging 34
   5.3 Summary 44
CONTENTS

6 Future and related work 46
7 Conclusion 52
Bibliography 55
A Domain Specific Language (DSL) is a language tailored to solve problems in a specific domain. Such a language offers a convenient level of abstraction to the user of the language. Sometimes the domain is of a technical nature and the DSL offers the right abstractions and operations to quickly define a solution. A DSL could also be used to bridge the communication gap between domain expert and software engineer. Ideally, this DSL can be directly used by the domain expert and implementation of the DSL is left for the software engineer. There are many different approaches to the implementation of a DSL, for an overview see [DKV00].

Having a DSL that enables functional specification of software has many benefits. The DSL is designed such that the user of the language need not be aware of its technical implementation. Often such a DSL lacks the functionality of a full-blown programming language and the user need not be a programmer. The user could be a domain expert whom, instead of producing informal documentation, will produce a specification that can directly be used in the software. Because the technical implementation is kept separate from the functional specification, a clean separation of concerns is achieved. When functional requirements change, only the specification needs to be updated. For software projects with a large number of functional requirements, these benefits will outweigh the cost of implementing the DSL.

Capgemini uses this approach in their Functional Model Driven Development (FMDD) methodology. A DSL is used by domain experts to construct Functional Models (FM). A transformation architecture will translate a FM into software. The transformation architecture can easily be reused when requirements change or for products in a similar domain. A FMDD project begins with the design of the DSL by a language engineer. The structure of the language will be defined using a software tool and the semantics of the language will be described informally. Once the structure of the language has been defined, domain engineers can begin writing specifications using the DSL. Simultaneously, transformation engineers can start implementing the transformations that generate code from models.

Unfortunately, implementing a transformation architecture for nontrivial DSLs takes time. Until the transformations are finished, models can only be
understood in terms of their informal specification. Only once the transformations have been implemented, the system can be generated and its behavior will become observable. As a consequence there will be a significant delay before:

- language engineers can see the consequences of their design decisions;
- domain engineers can observe the behavior of their models;
- transformation engineers can verify whether their transformations work as desired.

Another problem is the lack of tools to debug functional models. The DSL is a formal language with a very specific meaning. Whenever people write something in a formal language, what they write is not necessarily what they mean. Language constructions offered by the DSL might not mean what the domain engineer expects them to mean. Also a domain engineer can simply make a mistake while constructing the model. As models become larger and more complicated, the need arises for tools that help the domain engineer understand the behavior of system in terms of the model.

To remedy these problems we propose to build a prototype at an early stage in development. This prototype should only be concerned with the functional behavior of the system. We claim that such a prototype can quickly be built using a lazy functional programming language. We will describe an existing FMDD project at Capgemini and demonstrate how to implement an interpreter for a simplified version of the DSL used in that project. Then we will show how to extend this interpreter such that it can be used for debugging purposes.

TSL

Capgemini used FMDD to build a system named TSL for an unnamed governmental organization. Since we cannot disclose the name of that organization, we will refer to this organization as UGO. TSL processes information about citizens and determines whether they are eligible for social welfare benefits. Whether or not citizens are eligible is determined by legislation. Understanding this legislation is not straightforward as it requires domain knowledge. Moreover, when legislation changes, the system also needs to be changed accordingly. A DSL was designed that allows specification of this legislation. A dedicated team of domain engineers builds these specifications. When the legislation changes, only the specifications need to be updated and a new version of the system can be generated.

During development of this system, engineers at Capgemini encountered the problems mentioned in the previous section. Not only do the transformations handle non-trivial abstractions, but they also handle technical details such as interaction with a database and communication with other systems.
CHAPTER 1. INTRODUCTION

Not surprisingly development of these transformations was found to be time consuming.

Furthermore, there was a need for tools that support the debugging of models. Once the transformations were available, the system could be tested. When a test fails, the cause of the failure needs to be determined. Since there were no debuggers available for models, debugging had to be performed at the level of abstraction of the generated code. Tools were developed to inspect the behavior of the system in terms of the model, but these tools were made available too late in the development process.

Functional programming

In 2009 Bjorn Lamers, a master student of the Radboud University, investigated the use of functional programming in FMDD at Capgemini. In his thesis he investigated how one could build a transformation architecture using a functional language [Lam09]. Parser combinators and generic programming are used to build a parser that transforms text into instances of algebraic datatypes. A generic pretty printer is used to do the reverse. Arrows and GADTs are used to define transformations between data structures of different types. The emphasis was on type safe transformations from one datatype (or language) to another.

The objective of this project is different. Instead of generating code that, when compiled and executed, is semantically equivalent we choose to write an interpreter. An interpreter can execute the semantic actions directly. We therefore expect an interpreter to be shorter and more suitable for rapid prototyping. Because we are only interested in the functional behavior of the system, we also abstract from technical details like a database server.

Techniques applied in the area of functional programming are an important source of inspiration. Not only because of the parsing and transformation techniques, functional languages are also successfully used to embed DSLs [BCS98, LM00, JPKA10]. By embedding the DSL we do not need to implement a parser ourselves, specifications are simply expressions in the host language. We take this approach and demonstrate how this results in an interpreter that concisely captures the semantics of the DSL. This interpreter can be used to build a functional debugger. This opens up the possibility of debugging functional models early on in the development process.

The structure of the thesis is as follows. In the first chapter we will describe the system Capgemini is implementing for the UGO. Then, in Chapter 3, we will implement a prototype interpreter for this DSL. Next we will give an overview of debugging techniques in Chapter 4. In Chapter 5 we will integrate two debugging techniques with our interpreter. Then we will go over some future directions for research. Finally, we will present our conclusions.
Chapter 2

The UGO is responsible distribution of social welfare benefits e.g. childcare benefits, child budgets, rent benefits and health care benefits. Annually, millions of forms are processed. The UGO communicates with numerous sources to gather relevant information. The UGO needs to ensure this information is reliable and consistent. Then regulations are applied to determine which citizens are eligible for the benefit in question. Automating this process is a challenging endeavor.

A few years ago Capgemini was tasked with the development of TSL, a new system that automatically assigns benefits to citizens. The old system proved to be unreliable and lacked the flexibility to adapt to quickly changing regulations. TSL is built using FMDD, which is central to this thesis.

The first section will explore the requirements for such a new system, giving insight into the complexity that such a system has to deal with. TSL is built using FMDD and is embedded in a larger architecture. The second section will outline the architecture of the entire system so the reader is familiar with the context of TSL. Then we will cover how FMDD can be used to design and build a system that meets the requirements. In Section 2.4 we will describe how the system deals with the complexity arising from timed information and asynchronous processing. Finally, in Section 2.5 we will define the input, state and output of the system.

2.1 Requirements

For many years, the UGO has been responsible for the implementation of several government regulations. Up until now, citizens sent in their forms once a year. A filled out form contained all necessary information. Automatic processing is done once a year, one form at a time.

Recently, the UGO has also become responsible for distributing benefits. Dutch citizens receive these benefits throughout the year. Whether or not they are eligible for such a benefit depends on many variables, e.g. their income, the size of their household and their rent. The UGO gets notified of changes to these variables and the benefits they receive should change accordingly.
Information necessary for determining the height of the benefits is no longer presented in a single form. This requires a new approach.

The new system is not based on annual forms but centered around events in the lives of citizens. These events can happen throughout the year and the UGO might come to know about these events through various sources. Sometimes the UGO will get notified ahead of time that some event will take place, but usually the notification takes place after the event. Regardless, the system should be able to determine the consequences of the event for the past and the future and notify the citizen about the consequences in a timely fashion.

Information can reach the UGO through various channels and not all information is reliable and complete. Usually the UGO has reliable sources of information to determine the validity and to complete missing information. It is policy to first consult these sources before acquiring this information directly from the citizen themselves. This means that sometimes an event reaches the UGO and can be processed by the system without any interference from the citizen, they just get notified about the consequences of the change. Sometimes, it may be necessary to overrule some piece of information because of a decision from an official. The system should provide a generic way to deal with unreliable or incomplete information and facilitate the possibility to overrule information manually.

Automatic processing of information is only useful if the cost of implementing the system and running it does not exceed the cost of processing the information manually. In the case of TSL, there are many different rules and special cases that can be considered exceptional. The system should not try to correctly process these rare situations, but instead redirect them for manual processing.

These requirements should be met while maintaining modularity and facilitating oversight. The next section will describe the architecture which aims to solve the problems described in this section.

2.2 Architecture

TSL is the core component of the system that actually determines the eligibility of benefits. It is embedded in a larger architecture. This section will describe this architecture to provide a frame of reference to the reader.

Because of the need for modularity and the asynchronous nature of notifications, the system is structured as a Service Oriented Architecture (SOA). In such an architecture functionality is split up in terms of services which communicate by sending messages over the Enterprise Service Bus (ESB). Each service maintains its own state and thus is only loosely coupled with other services. This allows for interoperability with: existing services, services which
handle manual processing, services from external parties, services that handle payments etc.

Notifications about events in the life of citizens are processed by the Fact Registration Service (FRS). These notifications can be unreliable and incomplete, the FRS is a service dedicated for dealing with these problems. It will verify and complete the information based on other sources of information. When the information is reliable and complete we know for sure the event took place and it is registered as a fact. After an event is registered as a fact, an event is sent over the ESB which can be processed by TSL.

TSL is a collection of services which communicate via events sent over the ESB. Functionality that belongs together for organizational reasons is grouped together into a service. Since each service maintains its own state, they all have their own database server. Services that are part of TSL will only start processing information once an event is received. Once the service finished processing, other services will be notified about the result. Then the service will remain idle until a new event needs to be processed.

2.3 Functional Model

TSL is organized in a number of services that have either verified facts or results from other TSL services as inputs. Each of these services implements a piece of legislation or some procedure and is built using FMDD. This section explains the FM that contains the description used for generating TSL.

A FM is built up from a number of components. The most fundamental being the Business Idiom Set (BIS). The BIS defines the structure of the language and the elementary data types that are available in specifications. The set of elementary data types is sometimes referred to as domains. Some elements in the BIS are generic, others are introduced specifically for the domain we are modeling. The BIS defines what the DSL looks like and what it means.

Using the BIS a Business Concept Model (BCM) is created. The BCM describes the objects that exist in the specifications, these are sometimes re-
ferred to as *Business Concepts* (BC). An BC is made up of: data attributes, relations and methods. Data attributes are simply stored values from some domain. Objects can have different types of relationships with other objects from the BCM. Such a relation can be characterized as either aggregation or association which, in the case of aggregation, has implications for the lifetime of the child object. The relations can be either one-to-one or one-to-many. One can also specify whether there exists an inverse relationship or whether the relationship is symmetrical. The database schemes needed to store instances from the BCM can be generated from the data attributes and the relations. Finally, one is allowed to specify behavior of BC in methods, these methods are also implemented using the DSL.

A simplified example of a business concept from TSL is a citizen which contains:

- **attributes:**
  - `id` (domain: social security numbers)
  - `date of birth` (domain: dates)

- **relations:**
  - `address` (association with BC Address)
  - `income` (aggregation of BC Income)
  - `partner` (symmetrical association with BC Citizen)
  - `children` (association with BC Citizen)

- **methods:**
  - `determine age`
  - `relatives`
  - `is adult`

The final component of the FM is the *Business Specification Model* (BSM). The BSM consists of *procedures* and *declarations* that describe how a particular event should be processed. Processing an event could result in changes to BC instances. Other services get updated of these changes. For example, given an event that the income of citizen X has changed, the BSM could prescribe that X is no longer entitled to health care benefits. When we receive such an event the affected instances are updated and other services are notified about the changes.

### 2.4 Abstracting over time

TSL receives notifications about events in the lives of citizens and determines the consequences of these events for their benefits. These notifications originate from within TSL or from other services. Because TSL is made up of multiple services, processing of these notifications happens in parallel. Some notifications that belong together might get split up and may be delivered out of order. Therefore, we can make no assumptions on the order in which
TSL will receive notifications. Furthermore, these notifications might concern events in the future or in the past.

Despite these complications TSL should still correctly determine the consequences for the instances of the BCM. Notifications are tagged with three different timestamps:

**Valid time** This time stamp reflects the actual time of the event. This is the most relevant time stamp for determining the consequences for a citizen’s benefits.

**Report time** The report time is the time at which FRS registered a fact. There might be a significant delay between receiving and processing a notification. The report time is the point in time at which the UGO could have known about the consequences of a particular event.

**Transaction time** The time at which the service actually processed a notification. This time dimension allows for a paper trail that explains why a service made a particular decision at a particular point in time. This can be used for debugging purposes and is necessary for accountability.

For a specification to be truly functional, it need not be concerned with transaction times. Also the need to refer to the report times is rare. Therefore, we want our specifications to abstract from these timing dimensions. Specifications state how events should be processed in general and the implications for the objects in the three dimensional timed space are determined automatically. This brings us to the topic of the next section: what is exactly the state of the system.

### 2.5 Input, state and output

Because events are associated with three different points in time we need TSL to be able to determine decisions within this three dimensional space. For example, if we need to determine whether citizen X is eligible for health benefits, then we should check whether he has had health insurance for each month of a particular year. In TSL, this is achieved by lifting the instances of the BCM to the timed domain. Most attributes, relations and methods are timed. We can query these values for an arbitrary point in the three dimensional space. The current implementation achieves this by, instead of using regular values, using lists of *timeboxes*. A timebox is a value together with the three dimensional time interval in which it has that particular value.

As mentioned earlier, the input of the specification is a notification of some event. When TSL receives such a notification it will apply a *mapping* that updates the affected instances of the BCM. Then the BSM will apply procedures and declarations that will make more changes to the instances. Finally the system checks what parts of the BCM have changed during the last transaction and *publishes* these changes to other services. So the output
of a specification is a sequence of notifications.

2.6 Towards a prototype interpreter

The actual specification language used for TSL is too detailed to be useful as a starting point for this research. Furthermore many “declarations” contain side effects, resulting in a specification style that closely resembles an imperative style of programming. An example of this is that many declarations specify what should be done in the case of missing information, to handle the situation side effects are allowed. Also it is possible to call a procedure from a declaration. For this research we have split the language into a procedural part and a declarative part. We also made a number of simplifications in order to obtain a similar simpler language that remains interesting enough to debug. The most significant modifications are:

- declarations are without side effects;
- methods are represented by declarations;
- mappings are represented by procedures;
- we only support a few domains: dates, strings, integers, booleans and enumeration types;
- the system will only consider one time dimension: valid time;
- many rare constructions are left out.

Originally the language is defined in different layers namely: BIS, BCM and BSM. The language workbench ensures these are consistent with one another. Since we will not be using a language workbench but a functional programming language we need to figure out how to represent these components in our prototype. For our prototype we will use algebraic datatypes (ADT) to represent the model. The definition of these types can be seen as the BIS. Using these datatypes we can define elements of the BCM and BSM by creating instances. We won’t distinguish between the BCM and the BSM, they are defined within the same data structure.
Chapter 3

Interpreter

This chapter describes an interpreter for the DSL used in the TSL project. This interpreter simulates services that would normally be generated. It is a prototype which serves as an executable specification of the functional semantics. We abstract from technical details such as the enterprise service bus and the database server and are only concerned with the functional behavior of the system. The chapter gives an example of how, by embedding the DSL in a lazy functional language, a prototype can quickly be constructed.

Such a prototype can be constructed as part of the design of the DSL. The core interpreter function can be regarded as a concise specification of the semantics. By constructing such a prototype, specifications written in the DSL become executable. The prototype interpreter can be used as the basis for tools that support the debugging of functional models. As a result experimentation and debugging become possible at an early stage of development.

3.1 Overview

The interpreter simulates one of the TSL services. Such a service first receives an event, then processes this event as described by the functional model which results in the publication of new events. Such a service is modeled by the function `processEvents` which, given a model and sequence of input events, produces a sequence of output events for each of the given input events. An event is represented by the datatype `Gebeurtenis`. UGO is a Dutch organization which is reflected in the DSL used in the TSL project. Therefore many datatypes and constructors will have Dutch names.

```plaintext
:: Identifier ::= String
:: TimePoint  = {y :: Int, m :: Int, d :: Int}
:: Gebeurtenis = Gebeurtenis Identifier TimePoint [(Identifier, Value)]
```

processEvents :: Model [Gebeurtenis] -> [Gebeurtenis]
interpretExpr :: Expr (ExecutionContext s) -> TimedValue | State s
interpretStmt :: Statement (ExecutionContext s) -> ExecutionContext s | State s

The second section describes how a functional model is represented using the `Model` datatype and how `processEvents` starts the interpretation pro-
cess. The state of the interpreter is stored in the ExecutionContext datatype which is explained in Section 3.3. The next two sections should give the reader an overview of how the interpreter is structured. We proceed in a bottom-up fashion for the rest of the chapter. Elementary values and types are explained in Section 3.4. From these elementary values TimedValues, which keep track of how a value changes over time, are constructed in Section 3.5. In Section 3.6 we define the State class that allows us store to BC instances, we also define an instance of this class: SimpleState. Then we are ready to describe the rest of the interpreter. In Section 3.7, we will define the Expr datatype for expressions together with the interpretExpr function that interprets these expressions. Finally, statements and their interpretation function interpretStm are defined.

3.2 Meta model

A model describes the business concepts, enumeration types, procedures, methods and declarations that define the behavior of the system. We use an algebraic datatype ModelElement to represent these things. Our model will simply be a list of model elements. A model element can define the following:

Business concept A BC consists of a number of fields, each with a name and type. A field can be an attribute or a relation to another BC. Note that methods are not declared here, they are stored as a separate model element.

Enumeration type An enumeration type is defined by the list of possible values.

Procedure A procedure is defined by its return type, formal parameters, local variables and a list of statements. Procedures are executed one statement at a time and have some effect on the state of the system.

Declaration A named expression with a return type and formal parameters. Declarations are without side effects.

Method A method for some BC. Other then the fact it is associated with a BC it is equivalent to a declaration. When a method is used, the BC instance on which the method is invoked will be passed as the first parameter.

:: Model  ::= [ModelElement]
:: ModelElement  ::= BC Identifier [(Identifier, Type)]
| Enum Identifier [Identifier]  
| Procedure Identifier Type Formals Variables [Statement]
| Declaration Identifier Type Formals Expr
| Method Identifier Identifier Type Formals Expr
:: Formals  ::= [(Identifier, Type)]
:: Variables  ::= [(Identifier, Type)]
Interpretation of a model starts with \texttt{processEvents}. Given a list of input events and a model it will construct an initial \texttt{ExecutionContext} and process the events one by one.

An event is processed by invoking the procedure "Verwerk gebeurtenis". Each model should contain this procedure as a point of entry for the interpreter. This procedure first performs a mapping, which updates BC instances with information from the event. Then it will apply declarations on instances that are changed by the mapping. Finally, it will publish updated information if necessary.

\begin{verbatim}
processEvents :: Model [Gebeurtenis] -> [[Gebeurtenis]]
processEvents model evt_in = result_ec(evt_out)
  where
      result_ec = while (\ec -> (ec.evt_idx < (length ec.evt_in)))
                       (\ec -> (processEvent ec & evt_idx = ec.evt_idx + 1,
                                 stack = []))
                       (initEC model evt_in)
      processEvent = interpretStm (StmProcedure "Verwerk gebeurtenis" [])
\end{verbatim}

\subsection{3.3 Execution context}

The state of our interpreter is determined by: the model, the input and output events, the state of BC instances and the call stack. Because we need to pass around all of this information in the interpreter it is bundled into an \texttt{ExecutionContext}.

The interpreter keeps track of the name, environment and source code location of the procedure that is currently executed in the head of the stack. Each time a new procedure is called, a new stack frame is pushed on the stack. Within a stack frame there can be a nested blocks of statements, each with their own scope of variables.

\begin{verbatim}
:: ExecutionContext s = { model :: Model,
                       evt_idx :: Int,
                       evt_in :: [Gebeurtenis],
                       evt_out :: [[Gebeurtenis]],
                       state :: s,
                       stack :: [StackFrame] }
:: StackFrame = { method :: Identifier,
                  env :: [AssocList TimedValue],
                  line :: Location }
:: AssocList a = AssocList [(Identifier, a)]
:: Location = (Identifier, Int)
\end{verbatim}

Operations for assigning and reading variables in the current scope are available. As are functions for pushing a new stack frame onto the stack. A nested scope could be added manually but there is also a function that performs a state changing function within a nested scope.
As a preparation for the debugger we need to keep track of the source code location of the statement that is currently executed. That allows us to highlight the right statement when the model is being pretty printed. A Location contains the name of the current procedure and a list that indicates which nested statement is being processed. The pushStack and pushBlock functions will automatically update the line field in the stack frame. In the interpreter we only need to use nextECLoc to update the location when we have finished processing a statement.

<table>
<thead>
<tr>
<th>function</th>
<th>signature</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>setVariable</td>
<td>(Identifier, TimedValue) (ExecutionContext s) -&gt; ExecutionContext s</td>
<td></td>
</tr>
<tr>
<td>setVariables</td>
<td>[(Identifier, TimedValue)] (ExecutionContext s) -&gt; ExecutionContext s</td>
<td></td>
</tr>
<tr>
<td>getVariable</td>
<td>Identifier (ExecutionContext s) -&gt; TimedValue</td>
<td></td>
</tr>
<tr>
<td>pushStack</td>
<td>Identifier Variables (ExecutionContext s) -&gt; ExecutionContext s</td>
<td></td>
</tr>
<tr>
<td>popStack</td>
<td>(ExecutionContext s) -&gt; ExecutionContext s</td>
<td></td>
</tr>
<tr>
<td>pushBlock</td>
<td>Variables (ExecutionContext s) -&gt; ExecutionContext s</td>
<td></td>
</tr>
<tr>
<td>popBlock</td>
<td>(ExecutionContext s) -&gt; ExecutionContext s</td>
<td></td>
</tr>
<tr>
<td>doInBlock</td>
<td>((ExecutionContext s) (Identifier, TimedValue)) -&gt; (ExecutionContext s)</td>
<td></td>
</tr>
<tr>
<td>nextECLoc</td>
<td>(ExecutionContext s) -&gt; ExecutionContext s</td>
<td></td>
</tr>
</tbody>
</table>

initEC constructs an initial execution context from a model and a sequence of input events. Furthermore there are functions to get or put events in the execution context. The function allChanges retrieves all BC instances that are changed during the current transaction. For each changed instance it will return a reference and the time point at which the change occurred.

<table>
<thead>
<tr>
<th>function</th>
<th>signature</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>initEC</td>
<td>Model [Gebeurtenis] -&gt; ExecutionContext SimpleState</td>
<td></td>
</tr>
<tr>
<td>putEvent</td>
<td>Gebeurtenis (ExecutionContext s) -&gt; ExecutionContext s</td>
<td></td>
</tr>
<tr>
<td>getEvent</td>
<td>(ExecutionContext s) -&gt; Gebeurtenis</td>
<td></td>
</tr>
<tr>
<td>allChanges</td>
<td>Identifier (ExecutionContext s) -&gt; [(TimedValue, TimePoint)]</td>
<td>State s</td>
</tr>
</tbody>
</table>

### 3.4 Values and types

The interpreter has four primitive types: dates, strings, integers and booleans. Furthermore, the user can define two kinds of types in the model: enumeration types and business concepts. A relation type is used to specify that a particular field of a BC is a relation to another BC. Such a type will only occur in the specification of BCs, at run-time relations are represented by values with type TypeBC. TypeNone is used to specify that a procedure does not have any return value.

<table>
<thead>
<tr>
<th>symbol</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>:: Type</td>
<td>TypeDatum</td>
</tr>
<tr>
<td></td>
<td>TypeString</td>
</tr>
<tr>
<td></td>
<td>TypeGetal</td>
</tr>
<tr>
<td></td>
<td>TypeWW</td>
</tr>
<tr>
<td></td>
<td>TypeEnum Identifier</td>
</tr>
<tr>
<td></td>
<td>TypeBC Identifier</td>
</tr>
<tr>
<td></td>
<td>TypeRelation Identifier [RelationFlag]</td>
</tr>
<tr>
<td></td>
<td>TypeNone</td>
</tr>
<tr>
<td>:: RelationFlag</td>
<td>One2One</td>
</tr>
<tr>
<td></td>
<td>Aggregates</td>
</tr>
<tr>
<td></td>
<td>Symmetric Identifier</td>
</tr>
</tbody>
</table>
Primitive values are simply represented by their corresponding representation in Clean. Enumeration values are represented by the name of the alternative. We don’t have values that represent BC instances but we can have values that refer to BC instances stored in the state. Every BC instance is uniquely identified by a BCID which we will use this to store such a reference. Finally it is possible for a Value to be ValueNietAanwezig, which means the value is unknown to the system.

```
:: BCID := Int
:: Value       = ValueDatum TimePoint
   | ValueString String
   | ValueGetal Int
   | ValueW Bool
   | ValueEnum Type Identifier
   | ValueBC Type BCID
   | ValueNietAanwezig Type
   | ValueNone
```

3.5 Timed values

A TimedValue records how a value changes over time, it is represented by a list of TimeBoxes. Each TimeBoxes contains: the value for that particular interval, a list integers to keep track of the events that triggered an update of the time box and the starting time point of the interval. To find the end time point of the interval one needs to look at the next time box in the timed value. Because manipulating time boxes can be rather tricky there are functions available that take care of creating, combining and inspecting timed values. We hide the internal representation from TimedValue to the rest of the program.

```
:: TimeBox := (TimePoint, [Int], Value)
:: TimedValue := [TimeBox]
timepoint (timepoint, _, _ ) = timepoint
origins ( _, origins, _ ) = origins
value ( _, _, value ) = value
```

We can construct two types of timed values: values that are always valid and facts. By always valid we mean that the timed value takes on the same value for the entire time domain. We represent the beginning of time by a constant MIN_TIMEPOINT and the end of time by the absence of a next time box. A fact is a notification, originating from some event, that a value needs to change at some point in time. Note that there is no end time associated with a fact, it just states that the value is changed from a particular point in time. We start off with values that have the same value for the entire time domain and while processing events, facts are inserted causing the timed value to take on different values at different points in time.
always :: Value -> TimedValue
always val = [(MIN_TIMEPOINT, [val])]

fact :: TimePoint Int Value -> TimedValue
fact = _ (ValueNietAanwezig _) = abort "A fact should have an actual value"
fact p t nr val | p <= MIN_TIMEPOINT = abort "A fact needs an actual point in time"
| otherwise = [(p, [t, nr], val)]

Applying Value based functions

apply1 applies a unary function for untimed values to a timed value. apply1 is basically the map function which keeps the time points and origins unchanged, but applies the function to the value in each time box.

apply1 :: (Value -> Value) TimedValue -> TimedValue
apply1 f tbs = [(t,o,f v) | (t,o,v) <- tbs]

When applying a binary function to a pair of timed values we run into the problem that the periods of the time boxes might not be aligned. This means that, while we walk over the two arguments, we need to determine where the periods of the resulting timed value begin and end.

apply2 :: (Value Value -> Value) TimedValue TimedValue -> TimedValue
apply2 f [] [] = []
apply2 f [l1:ls] [r1:rs] | move_both ls rs = [(new tp, new origins, new val) : apply2 f ls rs]
| move_left ls rs = [(new tp, new origins, new val) : apply2 f ls [r1:rs]]
| otherwise = [(new tp, new origins, new val) : apply2 f [l1:ls] rs]

where
move_both [l2:] [r2:] | timepoint l2 == timepoint r2 = True
move_both [] [] = True
move_both _ _ = False
move_left [] _ = True
move_left [l2:] _ | timepoint l2 < timepoint r2 = True
move_left _ _ = False
new tp = max (timepoint l1) (timepoint r1)
new val = f (value l1) (value r1)
new origins = mergeUniq (origins l1) (origins r1)

Foreach

foreach allows us to apply a function for every BC instance in the timed value. It has the following arguments:

- ref: The timed value which refers to the BC instances in question
- func: A function that constructs a timed value based on a specific BC instance
- combine: A function which tells us how to combine these timed values into a new result
- init: The initial value

foreach is basically a foldr which, before using the combine function, makes sure the second argument of combine is only valid for the period of the
CHAPTER 3. INTERPRETER

Assignment and clearing

A timed assignment merges information, not all time boxes are overwritten. An assignment combines two histories to form a more complete time line. Intervals with ValueNietAanwezig will be ignored. The source value could be a fact, which represents that the variable has a particular value at a particular point in time. The system will assume this value until another time box is encountered. If the source value is a fact, then it needs to be inserted and not assigned.

We can not remove values by assignment because ValueNietAanwezig boxes will be ignored. This function will clear all intervals marked by the source timed value.

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>foreach ref := [] init = init</td>
</tr>
<tr>
<td>17</td>
<td>foreach ref:= [(c ...) func combine init = foreach vs func combine init</td>
</tr>
<tr>
<td>18</td>
<td>scoped_value (foreach vs func combine init)</td>
</tr>
<tr>
<td></td>
<td>where /* Ensures changes are within the current TimeBox */ scoped_value := TimedValue</td>
</tr>
<tr>
<td>19</td>
<td>/* Constructs a new TimedValue containing only the hd of the input TimedValue. */ only_head := TimedValue -&gt; TimedValue</td>
</tr>
<tr>
<td>20</td>
<td>foreach ref := []</td>
</tr>
<tr>
<td>21</td>
<td>only_head :: TimedValue -&gt; TimedValue</td>
</tr>
<tr>
<td>22</td>
<td>only_head [(t0,o,v0)] t0 &gt;= MIN_TIMEPOINT = [(MIN_TIMEPOINT, o, v0)]</td>
</tr>
<tr>
<td>23</td>
<td>only_head [(t0,o,v0),(t1, ...)] t0 &gt;= MIN_TIMEPOINT = [(t1, ValueNietAanwezig (getType v0)), (t0, o, v0)]</td>
</tr>
<tr>
<td>24</td>
<td>only_head [(t0,o,v0)] t &lt; timepoint box = [(t0, o, v0)]</td>
</tr>
<tr>
<td>25</td>
<td>only_head [(t0,o,v0),(t1, ...); vs] t &lt; timepoint box = [(t0, o, v0), (t1, ValueNietAanwezig (getType v0))]</td>
</tr>
<tr>
<td>26</td>
<td>foreach ref = abort (“foreach: illegal TimedValue: “ +++(show1 (hd ref)))</td>
</tr>
<tr>
<td></td>
<td>where/insertBox :: TimedValue TimeBox -&gt; TimedValue</td>
</tr>
<tr>
<td>27</td>
<td>insertBox [] (t,o,v) = [(t, o, v)]</td>
</tr>
<tr>
<td>28</td>
<td>insertBox [box:rest] (t,o,v) t &gt;= timepoint box = [(t, o, v) : rest]</td>
</tr>
<tr>
<td>29</td>
<td>t &gt; timepoint box = [box:insertBox rest (t,o, v)]</td>
</tr>
<tr>
<td></td>
<td>assign :: TimedValue TimedValue -&gt; TimedValue</td>
</tr>
<tr>
<td>30</td>
<td>assign target source</td>
</tr>
<tr>
<td>31</td>
<td>otherwise = apply2 asg target source</td>
</tr>
<tr>
<td></td>
<td>where/insertBox :: TimedValue TimedValue -&gt; TimedValue</td>
</tr>
<tr>
<td>32</td>
<td>insertBox [] (t,o,v) = [(t, o, v)]</td>
</tr>
<tr>
<td>33</td>
<td>insertBox [box:rest] (t,o,v) t &gt;= timepoint box = [(t, o, v) : rest]</td>
</tr>
<tr>
<td>34</td>
<td>t &gt; timepoint box = [box:insertBox rest (t,o, v)]</td>
</tr>
<tr>
<td></td>
<td>assign :: Value Value -&gt; Value</td>
</tr>
<tr>
<td>35</td>
<td>assign original (ValueNietAanwezig _) = original</td>
</tr>
<tr>
<td>36</td>
<td>assign original new = new</td>
</tr>
<tr>
<td></td>
<td>clear :: TimedValue TimedValue -&gt; TimedValue</td>
</tr>
<tr>
<td>37</td>
<td>clear target source = apply2 clr target source</td>
</tr>
<tr>
<td></td>
<td>where/clear original (ValueNietAanwezig _) = ValueNietAanwezig (getType original)</td>
</tr>
<tr>
<td>39</td>
<td>clr original _ = original</td>
</tr>
</tbody>
</table>
Sometimes we want to clear only the intervals which are valid in the source timed value, \texttt{clearInv} does just that.

\begin{verbatim}
clearInv :: TimedValue TimedValue -> TimedValue
clearInv target source = apply2 clr target source
  where
  clr original (ValueNietAanwezig _) = original
  clr original _ = ValueNietAanwezig (getType original)
\end{verbatim}

\texttt{mask} clears a timed value based on a boolean typed timed value. All non-true periods will become \texttt{ValueNietAanwezig}.

\begin{verbatim}
mask target source = clear target (apply1 msk source)
  where
  msk :: Value -> Value
  msk (ValueWW True) = ValueWW True
  msk (ValueWW False) = ValueNietAanwezig TypeWW
  msk (ValueNietAanwezig _) = ValueNietAanwezig TypeWW
  msk _ = abort "Cannot mask using a non ValueWW typed TimedValue"
\end{verbatim}

**Inspecting TimedValues**

There are some functions available to inspect timed values who’s implementation is not particularly interesting, we will just give their function headers:

\begin{verbatim}
(-=) infix 4 :: TimedValue TimedValue -> Bool
instance == TimedValue
instance toString TimedValue
instance getType TimedValue
instance getEmpty TimedValue
getBCName :: TimedValue -> Identifier
getChanges :: TimedValue Int -> [(TimedValue, TimePoint)]
getValue :: TimedValue TimePoint -> Value
getUntimedValue :: TimedValue -> Value
sometimes :: (Value -> Bool) TimedValue -> Bool
\end{verbatim}

**3.6 State**

When a service receives an event, a mapping will update the relevant BC instances. Then, for BC instances that are changed, the model needs to be reevaluated causing again new changes to BC instances. These changes can in turn trigger the publication of an event to the service bus. After which the service is ready to process the next event.

While processing events we need to maintain a state that records the BC instances that are available and the values of their fields. Instances of BCs themselves are untimed but they do have timed attributes. We can however use a timed value as a time dependent reference to a BC instance. We identify a BC instance by an unique number: a \texttt{BCIID}.

Although some fields of BCs are immutable we represent all fields using timed values for reasons of simplicity. We also refrain from any run-time type
checking for the same reason. It is probably best to write a separate type checker which ensures that such mistakes cannot arise at run-time.

The interpreter can maintain its state using any instance of the `State` class. This allows us to replace this data structure by another implementation in the future. Most functions in this interface operate on a timed value that refers to a BC instance. This reference may have different values over time so any instance of `State` should properly account for that. The functions that such an instance should implement are:

```haskell
class State s where
  create :: Type s -> (s, TimedValue)
  getField :: TimedValue Identifier s -> TimedValue
  setField :: TimedValue Identifier TimedValue s -> s
  getBCIDxs :: Type s -> [BCIID]
  getChangeSlices :: TimedValue s
```

`SimpleState` is a very straightforward implementation of `State`. It consists of a model, a list of types and a list that associates fields with their actual timed values. We need the model and list of types to look up the type and structure of the BC instances.

```haskell
instance State SimpleState where
  create (t := TypeBC bcname) (SimpleState m ts is) = (SimpleState m (append t ts) (append fields is), ref)
 绁閺＜blesiscche = always (ValueBC t (length is))
  fields = AssocList [(name, getEmpty type) <- (getActualFields m bcname)]

  getField ref field (SimpleState m _ is) = foreach ref (
    bciid -> get2 is (bciid,field))
     (\_ -> assign)
     empty
 绁閺＜lesiscche = getEmpty (getActualField m (getBCName ref) field)

  setField ref field value (SimpleState m ts is) =
    | isSymmetric /= Intersect both values /
    # ref = clear ref value
    # value = clear value ref
```

:: SimpleState = SimpleState Model [Type] [AssocList TimedValue]
emptyState :: Model -> SimpleState
emptyState model = (SimpleState model [] [])
/* Clear old relations */

/* Set the relation and its inverse */

where

isSymmetric = isJust findSymmFlag
findSymmFlag = case fromJust findSymmFlag of
  Symmetric id -> id
  TypeRelation flags -> find check flags
  Nothing -> Nothing

check (Symmetric _) = True
check _ = False

clearValue ref field is = foreach ref (id -> ref)
  (\id val s -> set2 s (id, field) (clearInv (get2 s (id, field)) val))

setValue ref field val is = foreach ref (id -> val)
  (\id val s -> set2 s (id, field) (assign (get2 s (id, field)) val))

getBCIIDs type (SimpleState _ ts is) = [idx | idx <- indexList ts | ts!!idx == type]

getChangeSlices ref (s =: SimpleState m ts is) = do_fields ref (getFormalFields m (getBCName ref))

where

split dst src = apply2 (\x y -> x) dst src
do_fields dst [(f, type):fs] = do_fields (split dst (getChangeSlices (getField dst f s) s)) fs
| otherwise = do_fields (split dst (getField dst f s)) fs
do_fields dst [] = dst
isAggregatingRelation (TypeRelation bc flags) = isMember Aggregates flags
isAggregatingRelation _ = False

*/
3.7 Expressions

Expressions describe a side effect free computation resulting in a timed value. An expression is build up from the following alternatives:

\[
\begin{align*}
\text{:: Expr} & \quad \ast \text{ExprValue} \quad \text{Value} \\
& \quad | \quad \text{ExprVariable} \quad \text{Identifier} \\
& \quad | \quad \text{ExprMatchBool} \quad [(\text{Expr}, \text{Expr})] \quad \text{Expr} \\
& \quad | \quad \text{ExprDot} \quad \text{Expr} \quad \text{Identifier} \quad [\text{Expr}] \\
& \quad | \quad \text{ExprApply} \quad \text{Identifier} \quad [\text{Expr}] \\
& \quad | \quad \text{ExprDisjunction} \quad \text{Expr} \quad \text{Expr} \\
& \quad | \quad \text{ExprConjunction} \quad \text{Expr} \quad \text{Expr} \\
& \quad | \quad \text{ExprNegation} \quad \text{Expr} \\
& \quad | \quad \text{ExprComparison} \quad \text{Expr} \quad \text{Expr} \\
& \quad | \quad \text{ExprAanwezig} \quad \text{Expr}
\end{align*}
\]

Interpreting expressions is fairly straightforward because expressions don’t have any effect on the execution context. They do however use the execution context to look up the values of fields of BC instances and variables. When a method or declaration is applied we push a new stack frame with the actual parameters but this change need only be visible in the recursive call to interpretExpr. The semantics of expressions is as follows:

Value A value literal, represents the same value for the entire time domain.

Variable A variable from the environment.

MatchBool A conditional expression with multiple if-branches and an else-branche. If it encounters the else branch it should simply evaluate the value of that expression as a result. If there is a condition associated with the expression it should first evaluate the rest of the MatchBool expression and then overwrite the intervals where the condition is true.

Dot Depending on the context, a dot operator could mean a number things, it could be: a method application, traversal of a one2one-relation or it could refer to an attribute. The interpreter first evaluates the left hand side so it can determine the type of the BC instance. Then it checks whether there is a method defined for this BC and if so, it will evaluate that method. If the right hand side does not refer to a method it must refer to a field of the BC instance and it will retrieve that value from the state.

Apply Applies a predefined declaration to a number of parameters. The interpreter retrieves the right expression from the model and evaluates it using an appropriate environment.

Operators There are a couple of boolean typed operators available: disjunction, conjunction, negation, comparison, aanwezig. The semantics of these operators are defined by functions based on untimed values.
CHAPTER 3. INTERPRETER

interpretExpr :: Expr (ExecutionContext s) -> TimedValue | State s
interpretExpr (ExprValue value) ec = always value
interpretExpr (ExprVariable name) ec = getVariable name ec
interpretExpr (ExprMatchBool [] else) ec = interpretExpr else ec
interpretExpr (ExprMatchBool [(c,t):xs] else) ec = assign rest val
where
  val = mask (interpretExpr t ec) (interpretExpr c ec)
  rest = interpretExpr (ExprMatchBool xs else) ec
interpretExpr (ExprDot lhs rhs actuals) ec =
  case findMethod ec.model (getType lhs _ val) rhs of
    Just (Method bc _, _ f expr) # vs = [(n, interpretExpr e ec) \ (n,v) <- t1 f & e <- actuals]
    # ec = pushStack rhs f ec
    # ec = setVariables [(fst(hd f), lhs_val):vs] ec
    = interpretExpr expr ec
  Nothing
    = getField lhs_val rhs ec.state
  where
    lhs_val = interpretExpr lhs ec
interpretExpr (ExprApply name actuals) ec =
  case getElement ec.model name of
    Declaration _, _ f expr # vs = [(n, interpretExpr e ec) \ (n,v) <- t1 f & e <- actuals]
    # ec = pushStack name f ec
    # ec = setVariables vs ec
    = interpretExpr expr ec
interpretExpr (ExprDisjunction lhs rhs) ec = apply2 (ww_disjunction) (interpretExpr lhs ec) (interpretExpr rhs ec)
interpretExpr (ExprConjunction lhs rhs) ec = apply2 (ww_conjunction) (interpretExpr lhs ec) (interpretExpr rhs ec)
interpretExpr (ExprComparison lhs rhs) ec = apply2 (ww_comparison) (interpretExpr lhs ec) (interpretExpr rhs ec)
interpretExpr (ExprNegation expr) ec = apply1 (ww_negation) (interpretExpr expr ec)
interpretExpr (ExprAanwezig expr) ec = apply1 (ww_aanwezig) (interpretExpr expr ec)

ww_disjunction :: Value Value -> Value
ww_conjunction :: Value Value -> Value
ww_negation :: Value -> Value
ww_comparison :: Value Value -> Value
ww_aanwezig :: Value -> Value
CHAPTER 3. INTERPRETER

3.8 Statements

Statements describe the procedural aspect of the system. Statements are used to describe how events are mapped to the BC instances, what declarations need to be evaluated and when a new event needs to be published. The Statement datatype needs statements to express these different operations.

```plaintext
:: Statement = StmtWijziging Identifier [(Wijziging, [Identifier])] [Statement] |
| StmtGebeurtenis Identifier [Statement] |
| StmtRelevante Identifier Identifier Expr Bool [Statement] |
| StmtPubliceer Identifier [(Identifier, Expr)] |
| StmtKrijgtwaarde Expr Expr |
| StmtPerIngangsDatum Expr Expr |
| StmtProcedure Identifier [Expr] |

:: Wijziging = Gewijzigde
```

Interpretation of statements is slightly more complicated because each statement can have side effects which are recorded in the execution context.

**Wijziging** Checks whether a BC instance is changed during this transaction. In the statement we can specify whether we are interested in changes in specific fields or changes in the entire instance. The current implementation will ignore this specification and gather all changes for the entire instance. If changes are found the interpreter will execute the list of statements associated with the statement. The interpreter will make the changed BC instance available to the environment together with a value “wijzigings datum” that contains the date of the change.

**Gebeurtenis** Every kind of event needs a different treatment, so we need to be able to check what kind of event we are dealing with during this transaction. **StmtGebeurtenis** serves precisely this purpose. If the event in the statement matches the event in the execution context the interpreter executes the associated statements. The fields of the event will be made available to the environment.

**Relevante** A **StmtRelevante** statement will try to find the BC instance with the correct value for a particular field. If it cannot find such an instance it can create one if this is required. If it has found an appropriate instance it will execute the associated statements.

**Publiceer** Publishes an event.

**Krijgtwaarde** The left hand side can either be a variable or a dotted expression. If it is a variable we will reassign this variable. If it is a dotted expression we will set the field of the right BC instances.

**PerIngangsDatum** Assignment of a value that originate from an event. These values are facts which only have a starting date associated with them, therefore these values need to be inserted in the TimedValue.

**Procedure** Calling another procedure.
CHAPTER 3. INTERPRETER

interpretStm :: Statement (ExecutionContext s) -> ExecutionContext s | State s
interpretStm stm ec = case stm of
  StmtWijziging bc chs stms = handle (allChanges bc ec) ec
  where
      handle [] ec = ec
      handle [(ref, date):cs] ec = handle cs (doInBlock (interpretStms stms) ec ["wijzigings datum", always (ValueDatum date)], (bc , ref))

  StmtGebeurtenis e stms
  | s == name = doInBlock (interpretStms stms) ec values
  | otherwise = ec
  where
      (name, date, fields) = case (getEvent ec) of Gebeurtenis n d f -> (n, d, f)
      values = [ (n, always v) | (n,v) <- fields ]

  StmtRelevante bc _name field expr make stms
  | isEmpty rel _bcis && not make = ec
  | isEmpty rel _bcis && make
      # (s, ref) = create bc_type ec.state
      # s = setField ref field value s
      # state = s
      = doInBlock (interpretStms stms) ec [(bc _name, ref)]
  | length rel _bcis == 1 = doInBlock (interpretStms stms) ec [(bc _name, hd rel _bcis)]
  | otherwise
      = abort "StmtRelevante is only implemented to select a single element" where
      bc_type = TypeBC bc _name
      value = interpretExpr expr ec
      all_bcis = map (ValueBC bc_type) (getBCIIds bc_type ec.state)
      rel_bcis = filter exists (map narrow all_bcis)
      narrow ref = mask (always ref) (interpretExpr (ExprComparison (ExprDot (ExprValue ref) field [])) ec)
      exists ref = sometimes ((<>)(ValueNietAanwezig bc_type)) ref

  StmtPubliceer bc tvvalues = putEvent (Gebeurtenis bc date values) ec
      where
          date = case getUntimedValue (getVariable "wijzigings datum" ec) of Valuedatum d -> d
          values = [(name, getValue (interpretExpr expr ec) date) | (name,expr) <- tvvalues]

  StmtKrijgtwaarde (ExprVariable name) rhs =
      setVariable (name, interpretExpr rhs ec) ec

  StmtKrijgtwaarde (ExprDot lhs field []) rhs =
      { ec & state = setField (interpretExpr lhs ec) field (interpretExpr rhs ec) ec.state }

  StmtPerIngangsDatum (ExprDot lhs field []) rhs =
      { ec & state = setField bci field fact' ec.state }
      where
          bci = interpretExpr lhs ec
          fact' = fact date ec.evt_idx value
          value = getValue (interpretExpr rhs ec) date
          date = case (getEvent ec) of Gebeurtenis _date _ -> date

  StmtProcedure p actuals
      # vs = [(in, interpretExpr e ec) \ (n, ..) <- formals & e <- actuals]
      # ec = pushStack p (formals+vars) ec
      # ec = setVariables vs ec
      # ec = interpretStms stms ec
      = popStack ec
      where
          (formals, vars, stms) = case (getElement ec.model p) of Procedure _ _ f v s -> (f, v, s)

interpretStms :: [Statement] (ExecutionContext s) -> ExecutionContext s | State s
interpretStms [stm] ec # ec = interpretStms s ec
      # ec = nextECLoc ec
      = interpretStms ss ec
interpretStms [] ec = ec
3.9 Summary

When developing a DSL, there is a need to have a prototype as soon as possible. A prototype gives feedback on the effect of design decisions. It could also be used to execute specifications and confirm whether those are working as desired. If we want to be able to debug specifications early on in the development cycle, we need to have a prototype that can execute these specifications.

This chapter explained how we built such a prototype for a simplified version of the DSL from the TSL project. By embedding the DSL in a lazy functional programming language, the interpreter could be developed in a short period of time. The core interpreter is under 150 lines of code long. There are also 700 lines of support code that handle: values, timed values, types and BC instances. These supporting modules are most likely reusable in new projects where a similar DSL needs to be developed.
To err is human. People make mistakes, also when expressing themselves using language. In natural language such a mistake can usually be resolved using available redundancy, context and common sense. Whenever language is interpreted by a computer though, it might not be so forgiving. Computers interpret languages which have a very specific meaning and that is difficult for people to work with. When using such a *formal language* we make mistakes and we need debuggers to help us resolve them.

When we start using a debugger we already know there is something wrong. We feed the interpreter input and it returns something unexpected. We know the output is wrong, we just don’t know where the mistake is located. Debugging is the activity of finding the source of the problem and fixing it. A debugger is a tool that helps us do that. We start with the output and reason backwards to the source of the problem. In a debugging session we can distinguish the following steps:

1. Observe erroneous output
2. Come up with a hypothesis
3. Observe computation
4. Confirm or reject hypothesis
5. If possible: correct mistake, otherwise: goto step 2

Quite often when starting a debug session we are clueless about the origin, the mistake could be located anywhere. In the next section we will discuss two simple techniques that could help to narrow our focus. As a computation can be very complicated, the more we can narrow our focus, the easier it will be to come up with a good hypothesis. Most debuggers, however, come into play at step 3, they offer a way to observe the computation. We will discuss the most common technique, *breakpoint debuggers*, in Section 4.2. Then we will discuss some more recent developments to this class of debuggers in 4.3. Finally, we will briefly discuss *algorithmic debuggers* which offer a declarative way of debugging.
CHAPTER 4. DEBUGGING TECHNIQUES

4.1 Narrowing your focus

We can find out about our system producing the wrong output in various ways. A mistake could expose itself during testing, or a mistake could be pointed out by an end user. Usually, when a mistake is pointed out to us, it is not in its most simple form. The input could contain all sorts of irrelevant details which make it hard to understand what is causing the problem. A good first step in resolving the issue is simplifying the input such that all irrelevant aspects are removed. Simplifying input is often a tedious and boring task.

Luckily this task can be automated by using a technique called delta debugging \cite{ZH02}. Delta debugging involves an algorithm making changes to the circumstances that result in faulty behavior. By systematically exploring these circumstances it will determine the minimal set that results in wrong output. Delta debugging has been applied to run configurations, version control repositories and program input. We will start by explaining how the \texttt{ddmin} algorithm minimizes the input that produces an error. For reasons of simplicity we assume the input to be a set and that no erroneous output occurs when tested with an empty input. A more general version can be found in \cite{ZH02}.

We assume we have an automated test which allows the \texttt{ddmin} algorithm determine whether the program is exhibiting the faulty behavior or not. Given an input \( c \) this function will determine whether the test:

- passes: \( \text{test}(c) = \checkmark \)
- fails: \( \text{test}(c) = \times \)
- is unresolved: \( \text{test}(c) = \triangleleft \)

We minimize the input with respect to some baseline for which the program runs without any problem: \( \text{test}(\emptyset) = \checkmark \). Given an input \( c_f \) which results in faulty behavior we want to find the smallest \( c' \subseteq c_f \) which causes the test to fail. Actually trying all different subsets of \( c_f \) would be exponential as there are \( 2^{|c_f|} \) possibilities. Instead \texttt{ddmin} determines the minimal input by approximation. In his book \textit{why programs fail}, Andreas Zeller has a simplified but equivalent version of \texttt{ddmin} which we will discuss here \cite{Zel09}.

\def\ddm{\texttt{ddmin}}
\def\ddmp{\texttt{ddmin}'}
\[
\ddm(c) = \ddmp(c, 2)
\]
\[
\ddmp(c, n) = \begin{cases} 
  c & \text{if } |c| = 1 \\
  \ddmp(c \setminus c_i, \max(n - 1, 2)) & \text{else if } \exists i \text{ test}(c \setminus c_i) = \times \\
  \ddmp(c, \min(2n, |c|)) & \text{else if } n < |c| \\
  c & \text{otherwise}
\end{cases}
\]

On each recursive call \( c \) is divided into \( n \) equal sized partitions such that \( c = c_1 \cup \ldots \cup c_n \) and \( \forall c_i, c_j [c_i \cap c_j = \emptyset \land |c_i| \approx |c_j|] \). The algorithm tries to eliminate these partitions from \( c \). If \( |c| \) contains just a single element we
have found a minimal failing input. Otherwise we can try to eliminate all partitions $c_i$ and see if we still get a failing test. If so we can eliminate that subset and continue searching using the same granularity. If we can find no subset which can be eliminated we increase the granularity by doubling the number of subsets $n$. If that also fails, we have reached our approximation of the minimal input.

The algorithm has a worst case complexity of $|c_f|^2 + 3|c_f|$ and a logarithmic best case scenario (then it boils down to binary search). Case studies show that the worst case scenario is unlikely and the approach can be used on real world examples. For very large input and slow running tests there are some useful optimizations mentioned in [ZH02].

Delta debugging has also been used to locate the regression in a version control repository [Ze99]. In that case $c$ is not the input but a series of patches between a state in which the program was still responding correctly and the new state in which it exhibits erroneous behavior. Delta debugging will result in the minimum number of patches that cause the error to occur, which seriously reduces the amount of source code to be investigated. There are some subtle differences to the minimization algorithm to decrease the chance of non-working configurations, but roughly the approach is the same.

A similar technique is actually being used in practice. Although it is not quite so sophisticated as the delta debugging approach, the source control system $git$ offers a way to do a binary search through the source code history by means of the $bisect$ command. The programmer can perform the test manually or automatically by writing a small shell script which does the job. $git$ $bisect$ assumes the regression happens at a single point in time but still proved to be useful to Linux kernel developers.

4.2 Breakpoint debugging

The most common way of observing the computation is by instrumenting the code such that it activates the debugger when an interesting event occurs like: reaching a breakpoint or accessing a watched variable. The programmer then runs the program until such an event occurs. Then the debugger will take over and allow the programmer to inspect the state of the program. What the state of the program is depends on the kind of program that is being debugged. For most imperative programming languages this would be: the call stack, all variables in scope, relevant heap structures and the location of the current breakpoint. An important aspect of a debugger is not only the information it has available but also the way this information is presented.

\textsuperscript{1}http://git-scm.com/
\textsuperscript{2}http://www.kernel.org/pub/software/scm/git/docs/git-bisect-lk2009.html
to the programmer. Some examples of how representation can make all the difference in usability:

- Easy navigation of complicated structures on the heap
- Representing the instruction pointer by source code location
- Representing data structures by graphs

After inspection of the state the programmer can:

- `step`: Perform only the next statement
- `step into/over/out`: Step into/over/out a particular function call
- `continue`: Continue until we hit another breakpoint

Quite some debuggers allow the programmer not only to observe the values that make up the state, but also allow them to alter these values thereby influencing the behavior of the remaining execution. Some interpreters like `python` and GHCi\(^3\) have integrated debuggers which, on a breakpoint, leave the programmer in an interactive session with the interpreter. The programmer then has the full power of the programming language at his disposal when observing or altering the state of the program.

For lazy functional programming languages a breakpoint based debugger poses a number of problems. Because arguments are not evaluated until the moment they are needed, the order of computation might be different from what the programmer expects it to be. This means that the program will hit breakpoints in a seemingly random order. When inspecting the state at a breakpoint, the programmer could find that many variables are unevaluated. Furthermore, the translation to machine code is complicated making it very hard to maintain a link between the execution and the source code. For example during execution the program doesn’t keep a lexical call stack and tail-recursive calls might be removed during optimization.

For GHCi there exists a breakpoint based debugger, but the programmer has no access to the call stack. The programmer is forced to use tracing to figure out what is the context of the current reduction \[\text{MIPG07}\]. Some argue that breakpoint based debuggers deserve more attention for lazy languages are there might be solutions for the mentioned problems \[\text{EJ03}\]. One could for instance disable tail-call optimization, evaluate expressions up until a certain depth and maintain a separate stack that resembles the lexical stack.

But for now the breakpoint based debuggers are not really usable for lazy languages. There is an algorithmic debugger for Haskell which we will discuss in the final section of this chapter. Furthermore, there is the option of tracing the computation. Both Clean and Haskell have a function called `trace` that breaks referential transparency by having a side effect that prints a message to `stdout`. The order in which these messages get displayed corresponds to the order in which these expressions are evaluated, so the programmer should

---

\(^3\)The interpreter distributed with the Glasgow Haskell Compiler: http://www.haskell.org/ghc/
be aware of this. For Haskell there exist some more convenient libraries for displaying values during computation called Hood and Vacuum.

4.3 Beyond breakpoints

While debugging we typically do not inspect each step of the computation. We observe some states and make predictions of what the code will do between states we observe. These predictions are based on what we know about the flow of control of the program. There is a technique called slicing that makes making these predictions easier [Wei82, Wei81]. The code is analyzed to determine the control flow graph and the data dependence graph. The programmer can then be offered a source code editor which hides irrelevant statements. Because these graphs are determined statically, not that much code can be hidden. Other research shows this can be improved by using traced information to rule out certain dependencies [AH90, HRB90, HLP10].

Most debugging starts with the observation of incorrect output. Based on this output and knowledge of the program the programmer must reason backwards to the location of the mistake. Breakpoint debuggers offer a way to stop and observe the computation and then continue until the next breakpoint. So when we reach a breakpoint we can use the debugger to move the computation forward but there is no way to look back.

Some debuggers address this issue by offering a way to reverse the computation. An early example was the debugger that shipped with the SML/NJ distribution [TA90]. It records checkpoints and side effects and allows the programmer to revert to an earlier state of the computation. The SML/NJ debugger is no longer maintained but similar functionality has since been made available to the OCaml debugger [LDF+10], Lisp’s ZSTEP95 [LF98] and Scheme’s DrRacket [FCF+02]. GDB offers a record mode that will record every step of the computation. Since 2009 GDB can undo execution steps that have been recorded this way [SPS10]. For some computational steps reversing the computation is impossible, for example for system calls. This is usually resolved by only performing the side effect once and recording the result of that call. On all subsequent calls a stub is used with the same result.

The idea of recording all steps of a computation and then reviewing them in a debugger has resulted in the development of Omniscient Debuggers. The program is instrumented to trace all relevant steps and is then executed until the problem has been observed. Afterward this trace is displayed in an interface that resembles that of a breakpoint debugger. The programmer can navigate forward and backwards in time. This idea has been implemented for Java with good results [Lew03] and has recently been added to the Visual

---

4The debugger distributed with the GNU Compiler Collection: http://www.gnu.org/software/gdb/
Studio debugger\(^5\). A big advantage is that a trace with erroneous behavior need only be recorded once. The trace can then be shared among testers and programmers to work on it. Although recording such a trace might be expensive, navigating it will often be quicker than actually executing the code (especially if the program makes extensive use of libraries and system calls). Also it eliminates the problems with user interaction and non-deterministic behavior. The paper by Bil Lewis [Lew03] has some useful hints on how to improve performance, for example, by not recording every value but recomputing them on demand, or letting the programmer specify granularity.

### 4.4 Algorithmic debugging

The concept of algorithmic debugging has been introduced by Shapiro in the domain of logic programming [Sha82]. Inspired by this work, its applicability has been explored for procedural languages [FSK92] and lazy functional languages [NF94]. Algorithmic debugging offers a declarative approach to debugging which, although tricky to implement, seems to fit well with the declarative nature of lazy functional programming languages. An algorithmic debugger helps the programmer find his mistake by navigating through computation based on the dependencies within the code. For pure functional languages the result of an expression is only dependent on its constituents making it an ideal candidate for algorithmic debugging.

An algorithmic debugger will first perform a transformation on the program. The program needs to be modified such that it generates a trace while performing its computation. Then the program is run and the trace can be inspected. Not all debuggers choose to trace the same information and offer different views on the computation. The Haskell debuggers Freja [Nil01] and Buddha [Pop06] maintain an Evaluation Dependency Tree (EDT). The debugger performs traverses over the EDT, at each step asking the programmer whether a particular reduction produced the correct result. If a reduction produced an incorrect result but all of its constituents did work correctly, we have found the location of our mistake! The Haskell debugger Hat [WCBR01] originally performed the trace using a Redex Trail which gave the user a convenient way to navigate the redexes and their reducts. Later they have extended Hat to use Augmented Redex Trails (ART) so that the same kind of traversal that Buddha and Freja offer can be performed.

Chapter 5

Functional debugging

Now let us shift our attention from debugging in general towards the topic of functional debugging. In FMDD domain engineers first construct a functional model that describes the system, then code is generated from this model. A functional debugger is a tool for locating errors in functional models. Such a tool should help the domain engineer understand erroneous behavior in terms of the functional model. In this chapter, we will show how, using our interpreter and our knowledge about debugging techniques, to build debugging tools suitable for debugging functional models.

Before actually constructing our debugging tools, let us first look at how the FMDD group at Capgemini dealt with bugs up until now. In Chapter 2, we describe the current architecture. Since there is no interpreter, we can only observe the behavior of the system after it has been generated. Typically, a bug is observed on the live system or during testing. The first step in resolving the bug is reproducing it locally. The system consists of services running on multiple servers connected through an enterprise service bus. This environment is simulated on the system of the domain engineer. The engineer can feed the services events and observe their responses. This simulated environment allows us to determine whether or not the error is specific to the environment and identify which service is at fault.

The next step is to determine why the service produces erroneous output. To understand the behavior of the service in terms of its functional model, the domain engineer has two tools at his disposal: unit testing and ListIDE. Unit testing is embedded in the environment in which the domain engineer builds the functional models: Cheetah. The domain engineer can specify input for procedures and observe their output. This is fine for understanding individual methods and procedures but not very convenient for localizing bugs in a large model. For this reason ListIDE was developed. ListIDE is basically a breakpoint based debugger. The code generation is altered such that the resulting program, at each execution step, records a pretty printed version of what is executed together with the values of the variables in scope. This allows the domain engineer to step through the execution and observe the consequences of each step.
CHAPTER 5. FUNCTIONAL DEBUGGING

Finally, once the bug is well understood, it can be fixed. The source of the problem could be in the functional model or in one of the technical aspects of the system, such as the code generator. Because the source of the problem could be outside the model and because the current debugging tools are a bit lacking, we sometimes resort to debugging the generated code. This requires knowledge of how the generated code works and is beyond what is considered to be functional debugging.

So the FMDD group already has quite some tools available to resolve bugs. The most important problem with these tools is that they were available relatively late in the development process. To some extend this is inevitable because these tools rely on the availability of a code generator. But, because building a rudimentary interpreter is easier than building a code generator, we could ease this problem by basing these tools on our interpreter instead.

Unit testing and simulating a group of services is just a matter of invoking the interpretation functions with the right parameters and displaying the results in a nice user interface. Instead we will focus on the more interesting debugging techniques of delta debugging and breakpoint debugging. First we will implement delta debugging and show how it could be used in conjunction with our interpreter. Next, in Section 5.2, we will discuss how to construct a breakpoint debugger. First we will look at what requirements domain engineers have for a breakpoint debugger. Then, to make our interpreter suitable for debugging, some changes are needed. Then we will present our implementation of the breakpoint debugger. We used the iTasks framework to quickly build a user interface for the breakpoint debugger which is described in the last subsection of the chapter.

5.1 Delta debugging

As described in section 4.1 whenever we are presented with an input which causes an error, this input need not be minimal. The input could include all sorts of irrelevant detail which makes it harder to understand why the system fails to produce the correct output. The \texttt{ddmin} algorithm systematically changes the input to eliminate these irrelevant details until a minimal example is obtained.

We presented the \texttt{ddmin} algorithm in pseudo code in section 4.1, it turns out the actual implementation is not that different. We only need to make explicit how the algorithm tries to exclude partitions from the input. In Clean the \texttt{ddmin} algorithm can quite concisely be formulated as:
CHAPTER 5. FUNCTIONAL DEBUGGING

33

\[
\begin{align*}
\text{ddmin} :: ([a] -> \text{Bool}) [a] \rightarrow [a] \\
\text{ddmin test c} = \text{ddmin' c 2} \\
\text{where}
\end{align*}
\]

\[
\begin{align*}
\text{ddmin' c n} = \\
\quad | 1 == \text{length c} \quad & \quad = c \\
\quad | \text{not (isEmpty excl_find)} \quad = \text{ddmin' (hd excl_find) (max (n-1) 2)} \\
\quad | n < \text{length c} \quad & \quad = \text{ddmin' c (min (2*n) (length c))} \\
\quad | \text{otherwise} \quad & \quad = c \\
\text{excl_set n c} = \text{splitAt (length c / n) c} \\
\quad = \text{[ t : [ h ++ rest \ \text{\&\&} rest \text{\&\&} \text{excl_set (n-1) t ] ]}}
\end{align*}
\]

The \text{ddmin} algorithm can be used to minimize all sorts of circumstances that may be relevant for the bug. For example it has been used to minimize input, but also to minimize the set of patches between a working and a non–working version of the program. This generic aspect of the algorithm is reflected in the implementation by the fact it works on lists of arbitrary type. Therefore we expect the test function to be more then just a simple test, it should convert the subset of circumstances into a running configuration and then test whether the error still occurs. Let’s see how that works out for minimizing a set of input events.

While using the interpreter with the example model in the software distribution, you might notice that it will, at some point, return that a citizen is rightfully uninsured because he/she lives abroad. This could be considered a bug since we cannot construct input events about the home address of citizens yet. The service has no way of knowing whether the citizen does or does not live abroad. We can get one step closer to understanding this behavior if we could trim down the input events needed to produce this suspicious output.

We first need a test to recognize the suspicious output event, the function \text{wrongEvent} serves this purpose. The function that is passed to \text{ddmin} feeds a subset of input events to the interpreter and then tests whether or not any of the output events is suspicious. We also pass \text{ddmin} a set of input events which exhibits the suspicious behavior and it will automatically minimize it for us.

\[
\begin{align*}
\text{run_minimize} = \text{ddmin (\lambda es \rightarrow \text{test (processEvents bepalen_zorglasten es))}} \\
\quad \text{Gebeurtenis "Feit Burger actief militair"} \quad \{ y=2009,\text{m}=1,\text{d}=1 \} \quad \{ ["BSN-Subject", ValueGetal 0] \}, \\
\quad \text{Gebeurtenis "Feit Burger gemoedsbezwaard"} \quad \{ y=2009,\text{m}=1,\text{d}=1 \} \quad \{ ["BSN-Subject", ValueGetal 0] \}, \\
\quad \text{Gebeurtenis "Feit Burger gedetineerd"} \quad \{ y=2009,\text{m}=2,\text{d}=1 \} \quad \{ ["BSN-Subject", ValueGetal 0] \}, \\
\quad \text{Gebeurtenis "Feit Burger niet gedetineerd"} \quad \{ y=2009,\text{m}=7,\text{d}=1 \} \quad \{ ["BSN-Subject", ValueGetal 0] \}, \\
\quad \text{Gebeurtenis "Feit Burger niet actief militair"} \quad \{ y=2009,\text{m}=1,\text{d}=1 \} \quad \{ ["BSN-Subject", ValueGetal 0] \}, \\
\quad \text{Gebeurtenis "Feit Burger actief militair"} \quad \{ y=2010,\text{m}=1,\text{d}=1 \} \quad \{ ["BSN-Subject", ValueGetal 0] \} \\
\text{where}
\end{align*}
\]

\[
\begin{align*}
\text{test output} & = \text{not (any wrongEvent (flatten output))} \\
\text{wrongStatus} & = \text{ValueEnum (TypeEnum "Zorgverzekerd") "terecht onverzekerd wegens buitenland"} \\
\text{wrongEvent evt} & = \text{case evt of} \\
\quad \text{Gebeurtenis "Gebeurtenis Lasten Zorg" v} & \quad = \text{get (AssocList v)} \quad \text{"status verzekering" == wrongStatus} \\
\quad & \quad = \text{False}
\end{align*}
\]

It returns only one event named "Feit Burger niet actief militair" which indeed reproduces the problem. For those wondering, according to the spe-
cification the system should indeed assume a citizen to be living abroad if there is no known home address. So this behavior is to be considered a feature.

In this section we have shown how, with just a few lines of code, we can leverage the interpreter to minimize failure inducing input. Having such a minimal input will reduce the effort needed to understand why the error occurs. For the rest of the chapter we will focus on a more traditional debugging technique: breakpoint debugging.

5.2 Breakpoint debugging

Requirements

We now return to the task of building a breakpoint debugger. To find out what is expected of such a debugger I interviewed all domain engineers\(^1\). During this interview I asked how they use the available debugging tools and what they believe a good functional debugger is. Although their debugging tactics varied quite a bit I have distilled a number of observations:

**Visualization** Because values in the system can vary with respect to three different time axes, visualization is of great importance. Just presenting the data structures will render the debugger unusable. The obvious way around this problem is presenting values as a time line which can easily be navigated.

**Fine grained stepping** Earlier versions of ListIDE allowed domain engineers to step through the top-level statements only. Because of this limitation domain engineers could not narrow down the problem inside a set of nested statements. This was perceived as a huge shortcoming which has been resolved in newer versions of ListIDE.

**Breakpoints** The debugger allows the engineer to step through the computation and review the state of the program. The problem is the number of individual steps can be huge. Not having to review each step and being able to just skip to the part we are interested in, is essential. This feature was also recently added to ListIDE.

**Observing state** The state of a service can be quite complicated. During execution the model can reference variables. Some variables are timed, others are untimed. Some variables are simple values, others reference BC instances which can in turn refer to other instances. The ability to navigate all this data is important because it may influence the behavior of the system. Cheetah can only observe the input and output of a

\(^1\)At the time of writing the team of domain engineers consisted of six engineers
given procedure. ListIDE can only observe variables that are in scope. There exists a separate viewer that displays the BC instances and their relations in the database. The fact that the BC instances and their relations are not accessible from Cheetah and ListIDE is considered a shortcoming.

Because implementing a debugger which meets all these criteria will take too much time, choices have to be made. Our interpreter only uses a single time axis, so this already simplifies the problem of visualization. We will make a simple visualization to display a fixed interval of a timed value. The interpreter consists of two parts, one that is concerned with interpreting statements and one that is concerned with interpreting expressions. Our debugger will be able to step through statements (also nested ones) but not expressions. Each statement can be indicated to be a breakpoint and the debugger can run the computation until such a breakpoint is encountered. Furthermore the debugger will display the BC instances that are referenced from the variables, but not their relations.

**Stepped interpretation**

If we were to build a debugger for generated code the obvious choice would be changing the code generator such that it produces instrumented code. This would clutter the generated code, but most developers will not be reading that anyway. ListIDE takes exactly this approach. But for this to work we need have a code generator in the first place.

We have chosen to use our prototype interpreter. A functional language like Clean allows for a concise implementation of this interpreter which makes the semantics of the meta model explicit and executable. Ideally we would base our debugger on this interpreter because it is available in an early stage of development. But there is a risk of cluttering the interpreter with debugging related functionality. This clutter could make the interpreter less suitable as a concise description of the semantics.

So how do we make our interpreter debuggable? Upon closer inspection of the interpreter you might notice it uses recursion quite frequently. For example, it processes a list of statements by first computing the result of the first statement, and then recursively continuing with the rest of the list. So it will not just execute a single statement, it will automatically execute them all. Instead we need the interpreter to execute the statements one by one.

```plaintext
interpretStms :: [Statement] (ExecutionContext s) -> ExecutionContext s | State s
interpretStms [s:ss] ec # ec = interpretStm s ec
    # ec = nextECLoc ec
        # ec = interpretStms ss ec
interpretStms [] ec = ec
```
We could rewrite the interpreter such that it operates without state and executes only a single step at a time. But we will soon notice there is quite some state hidden within the recursion. For example when it is processing a `StmWijziging` statement it first determines a list of changes, then it will recursively execute the list of statements for each change. Because of the recursion we don’t need to keep track of the current statement, or which change we are currently processing. If we wanted the interpreter to do just a single step, it should be able to determine which statement and change it should process from its input parameters. Keeping track of this by hand would introduce quite some overhead in the interpreter.

Instead we will use a different approach. In Clean functions are first class citizens, meaning we can use them the same way we use other types. Functions can be used as argument, returned as a result, written to disk or stored in some data structure. And this is exactly what we will do: we store the function that is the recursive continuation in a data structure. We also need to store the intermediate state in this data structure.

```haskell
:: Computation s f = Finished f
   | Suspended s (s -> Computation s f)

class doStep i s f :: i s -> Computation s f
```

A `Computation` can either be finished or suspended. When it is finished it contains the resulting value, when it is suspended is contains the intermediate state and a function that, given a state, computes the rest of the interpretation. If we manage to rewrite our interpreter such that it returns a `Computation` we can interpret our model step by step. Notice that this also facilitates a debugger that allows the user to edit the state before continuing.

This time we define our interpreter as instance of a class `doStep` which will provide an interface to the debugging functions we will define later on. An instance of `doStep` will interpret some data structure of type `i` given a state `s` and when finished return a value of type `f`. When interpreting statements the intermediate type and the result type are the same: `ExecutionContext SimpleState`. But whenever we want to extend debugging functionality to the interpretation of expressions we need the possibility of different intermediate and result types.

We define two combinators to easily sequence computational steps:

```haskell
(>>>) infixl 1 :: (Computation s b) (b -> Computation s c) -> (Computation s c)
(>>>) c f = case c of (Finished result) = f result
                     (Suspended state rest) = Suspended state \ec -> rest ec >>> f

(>>+) infixl 1 :: (Computation s b) (b -> c) -> (Computation s c)
(>>+) c f = case c of (Finished result) = Finished (f result)
                     (Suspended state rest) = Suspended state \ec -> rest ec >>+ f
```

These functions check whether the left hand side computation is finished. If
CHAPTER 5. FUNCTIONAL DEBUGGING

37

this is the case it can proceed with the right hand side computation. If the left
hand side computation is suspended, then the entire computation is suspended
and it will try again after the next computational step.

Some alternatives of the interpreter contain just a single change to the
state of the computation, these are the base cases of the recursion. The result
of these alternatives can be converted by using the Finished constructor. Other
alternatives make some changes and then recurse to process the rest of
the model. The recursive calls to the interpreter need to be stringed together
by means of the combinators. The combinators will skip over Finished states
and respect Suspended states, such that the combined Computation will have
just a single finished state.

The combinators will not insert any suspended states of their own. We
need to choose which steps in the computation are visible to the debugger and
which are not. The Computation data structure and the doStep class allow
for some flexibility when it comes to this. Suppose we have an alternative in
which something complicated happens and we want to break this up in distinct
computational steps, then we could simply nest Suspended constructors. Sup-
pose we have a recursive call in which the domain engineer could not possibly
be interested. Then we could save us the trouble of juggling Computations by
writing a small function that skips all suspended states and directly computes
the result.

We place our Suspended constructor such that it on one hand exposes
the interesting intermediate states, and on the other hand does not clutter
our interpreter too much. Most debuggers will pause the computation be-
fore a statement is actually executed. Since we have a doStep instance for
[Statement], this would be a strategic choice for placing the Suspended con-
structor:

```haskell
instance doStep [Statement] (ExecutionContext SimpleState) (ExecutionContext SimpleState)
where

doStep [] ec = Finished ec

doStep [s:ss] ec = Suspended ec/ec -> doStep s ec
>>> nextECLoc
>>> doStep ss
```

The rest of the changes to the interpreter are small. We need to make sure
all alternatives either use the Finished constructor or they combine recursive
call by means of the combinators. Some alternatives used the doInBlock
function for which we provided an alternative implementation stepInBlock
which also strings together Computations appropriately. Compare the stepped
interpreter with the original interpreter and you will notice the changes are
minor. In the stepped interpreter below, these changes are indicated by a
cursive typeface.
CHAPTER 5. FUNCTIONAL DEBUGGING

38

stepInBlock f ec values # ec = pushBlock [(1, getType v) \ (i, v) \ values] ec
# ec = setVariables values ec
= f ec >> popBlock

instance doStep Statement (ExecutionContext SimpleState) (ExecutionContext SimpleState)
where
doStep s ec = case s of

StmWijziging bc chs stms = handle (allChanges bc ec) ec
where
  handle [] ec = Finished ec
  handle [(ref, date):cs] ec = stepInBlock (doStep stms) ec
        ("wijzigings datum", always (ValueDatum date)),
        (bc, ref) >> handle cs

StmGebeurtenis e stms =
  e = name = stepInBlock (doStep stms) ec values
| otherwise = Finished ec
where
  (name, date, fields) = case (getEvent ec) of
    Gebeurtenis n d f -> (n, d, f)
  values = [(n, always v) \ (n, v) <- fields]

StmRelevante bc name expr make stms =
  isEmpty rel bcis && not make
  isEmpty rel bcis && make
  bc _ type
  value = interpretExpr expr ec
  all bcis = map (ValueBC bc _ type) (getBCIIDs bc _ type ec . state)
  rel bcis = filter exists (map narrow all bcis)
  narrow ref = mask (always ref)
        (interpretExpr (ExprComparison
          (ExprDot (ExprValue ref) field [])
          expr)
        )
  exists ref = sometimes ((=)) (ValueNietAanwezig bc _ type) ref

StmPubliceer bc tvalues = Finished (putEvent (Gebeurtenis bc date values) ec)
where
  date = case getUntimedValue (getVariable "wijzigings datum" ec) of
    ValueDatum d -> d
  values = [(name, getValue (interpretExpr expr ec) date) \ (name, expr) <- tvalues]

StmKrijgtwaarde (ExprVariable name) rhs =
  Finished (setVariable (name, interpretExpr rhs ec) ec)

StmKrijgtwaarde (ExprDot lhs field []) rhs =
  Finished (ec & state = setField (interpretExpr lhs ec) field
            (interpretExpr rhs ec) ec . state)

StmPerIngangsDatum (ExprDot lhs field []) rhs =
  Finished (ec & state = setField lhs bci field fact' ec . state)
where
  bci = interpretExpr lhs ec
  fact' = fact date ec . evt_idx value
  value = getValue (interpretExpr rhs ec) date
  date = case (getEvent ec) of Gebeurtenis _ date _ -> date

StmProcedure p actuals
# vs = [(ln, interpretExpr e ec) \ (ln, e) <- formal & e <- actuals]
# ec = pushStack p (formals == vars) ec
# ec = setVariables vs ec
= doStep stms ec >> popStack
where
  (formals, vars, stms) = case (getElement ec . model p) of
    Procedure _ _ f v s -> (f, v, s)
Breakpoint debugger

Now that we have a stepped interpreter we can write functions that operate it to implement our debugger. But first, remember that functions are first class citizens. This enabled us to store the state and the recursive continuation in a `Computation`. Well, we can take this one step further and store all suspended states we have encountered so far:

\[
\text{Trace } s \ f \ := \ [\text{Computation } s \ f]
\]

That enables us to revert to an earlier state in our debugging session. The state of a debugging session is a `Trace` in which the head is the computational step we are currently inspecting and its tail contains the previous steps of the computation. We can move the computation forward or revert the computation to an earlier state.

Reverting to an earlier state is possible because the state of the computation is simply a value. There are no side effects that need to be reversed. In the generated system we would need to rollback the database. Furthermore, because Clean is referentially transparent, when continuing a suspended computation, the current state is passed by reference. As a result many values will be shared between different steps within the trace. Although we did not test this extensively, we expect this approach to scale well.

Typically we don’t want to inspect every single step but let it run until something interesting happens. The function `debugUntil` runs the computation until some condition on the state becomes `True`, and it can do so in both directions:

```clean
debugUntil :: (Trace s f) Direction (s -> Bool) -> Trace s f
debugUntil trace d cond = case d of Forward -> forward trace cond Backward -> backward trace cond
where
  backward :: (Trace s f) (s -> Bool) -> Trace s f
  backward [ el ] = cond = [ el ]
  backward [ Finished _ : trace ] cond = backward trace cond
  backward [ Suspended _ : trace ] cond = case hd trace of
    Finished _ -> abort "backwardUntil: Unexpected Finished"
    Suspended s rest -> if (cond s)
      trace
      ( backward trace cond )
      forward trace cond = case hd trace of
    Finished f -> trace
    Suspended s rest -> case rest s of
      Finished f -> [ Finished f : trace ]
      Suspended s rest -> if (cond s)
      [ Suspended s rest : trace ]
      ( forward [ Suspended s rest : trace ] cond )
```

Running the computation backwards is simply popping elements of the list. Running the computation forwards means we have to prepend new states to the trace. When testing whether a stopping condition is `True` we ignore the
CHAPTER 5. FUNCTIONAL DEBUGGING

head of the list since the programmer has already inspected that state.

By varying the stopping condition we can implement different debugging commandos. The `debugStep` commando will perform all computational steps until the next statement in the same procedure (at the same stack depth) is reached. `debugStepIn` will, if the current statement is a procedure call, step into that procedure. `debugStepOut` on the other hand will perform all computational steps until we return to the caller of the current procedure. The command `debugNext` will step through every single computational step. The command `debugContinue` will perform all computational steps until the current location is one of the breakpoint locations.

```plaintext
currentStackDepth [Finished ec :_] = length ec.stack
currentStackDepth [Suspended ec :_] = length ec.stack

debugStep trace dir = debugUntil trace dir (\ec -> length ec.stack <= max 1 (currentStackDepth trace))
debugStepIn trace dir = debugUntil trace dir (\ec -> True)
debugStepOut trace dir = debugUntil trace dir (\ec -> length ec.stack < max 2 (currentStackDepth trace))
debugNext trace dir = debugUntil trace dir (\ec -> True)
debugContinue trace dir bps = debugUntil trace dir (\ec -> case ec.stack of
    [] -> False
    [s:_] -> if (s.line == None)
        False
        (isMember s.line bps))
```

User interface

We will use iTasks to quickly build a user interface for our debugger. iTasks is a workflow management system embedded in Clean [JPKA10]. A workflow is a directed graph of tasks along which data flows. The iTasks library offers basic tasks that require user interaction and task combinators that specify how information flows from one task to the other. Together these form an embedded DSL in which workflows can be specified. Given a set of workflow specifications, the iTasks engine offers a web-based user interface in which users can initiate and interact with instances of these workflows.

```plaintext
UI :: *World -> *World
UI world = startEngine [ workflow "Debugger" runDebugger ] world
```

Embedding workflow specifications in Clean has a number of advantages. First of all: workflow specifications will always be correctly typed. When combinators are used to form new tasks, the compiler will check whether these combinations are consistently typed in terms of their input and output types. Secondly we can use Clean’s generic programming to give sensible default implementations for common operations on a particular piece of data. The iTasks library has generic functions that handle editing, visualization and persistence of data.
Our debugger uses only two basic tasks: \texttt{updateInformation} and \texttt{showMessageAboutA}. The first task presents the user with an editor that allows modification of the value. The second task only displays the value and offers the user a number of actions to perform. We can lift values to the task domain by using the \texttt{return} combinator. Finally we can perform two tasks in succession with the \texttt{>=} combinator. The result of the first task will be used in the second task.

```plaintext
updateInformation :: !String ! desc a -> Task a | html desc & iTask a
showMessageAboutA :: !String ! mesg ![TaskAction a] a -> Task (!Action, !a) | html mesg & iTask a
(>>>=) infixl 1 :: !Task a !(a -> Task b) -> Task b | iTask a & iTask b
return :: !a -> Task a | iTask a

class iTask a
| gVisualize(*{!})
  , gUpdate(*{!})
  , gVerify(*{!})
  , JSONEncode(*{!})
  , JSONDecode(*{!})
  , TC a
```

Values within the workflow should be instances of the \texttt{iTask} class. The \texttt{iTask} class ensures that instances are available of the generic functions needed for displaying, editing and storing values. These functions are generic so we can choose to either automatically derive the function, or to make a specific implementation ourselves.

For complicated data structures the generic implementation will often result in awkward editors and visualizations. A simple solution for this problem is to define a new datatype that hides irrelevant details. We can automatically derive the \texttt{iTask} class for this new datatype and define conversions to and from the desired datatype.

An example of such a complicated data structure is the \texttt{Gebeurtenis} datatype. It is designed to represent all sorts of events and relies on internal datatypes from our interpreter (like: \texttt{TimePoint}). We will replace it with a new datatype: \texttt{UIGebeurtenis}, which can only represent a selected number of events. The generic editor for this datatype will be much more pleasant to work with. We still need to derive the \texttt{iTask} class for \texttt{Gebeurtenis} so we can pass these values around in our workflow.

The \texttt{updateEvents} workflow will first convert the input from \texttt{[Gebeurtenis]} to \texttt{[UIGebeurtenis]}. Then the user can manipulate the list in the \texttt{updateInformation} task. Finally it will convert the result back into an \texttt{[Gebeurtenis]}:

```plaintext
The updateEvents workflow will first convert the input from [Gebeurtenis] to [UIGebeurtenis]. Then the user can manipulate the list in the updateInformation task. Finally it will convert the result back into an [Gebeurtenis]:
```
CHAPTER 5. FUNCTIONAL DEBUGGING

```haskell
updateEvents :: [Gebeurtenis] -> Task [Gebeurtenis]
updateEvents old = updateInformation "Gebeurtenissen"
  (map toUIGebeurtenis old)
  >>> new = return (map fromUIGebeurtenis new)

:: UITypeGebeurtenis = Militair | Gedetineerd | Gemoedsbezwaard
:: UIGebeurtenis = { type :: UITypeGebeurtenis,
                     beëindig :: Bool,
                     date :: Date,
                     bsn :: Int }

toUIGebeurtenis :: Gebeurtenis -> UIGebeurtenis
fromUIGebeurtenis :: UIGebeurtenis -> Gebeurtenis

derive class iTask UITypeGebeurtenis, UIGebeurtenis
derive class iTask Gebeurtenis, TimePoint, Value, ...
```

The graphical user interface of the debugger should:
- Present a nice visualization for the head of the trace
- Allow the user to change direction
- Allow the user to change the breakpoints
- Allow the user to perform one of the debugging commands

The state of the debugger consist of the current trace, direction and set of breakpoints, these are recorded in a new datatype `UIDebugger`. Because we want more control over how the computation is visualized we will implement `gVisualize` ourselves. A proper implementation of `gVisualize` is able to create different kinds of visualizations, we will just let it return a `HtmlFragment`.

```
:: UIDebugger = { trace :: Trace (ExecutionContext SimpleState) (ExecutionContext SimpleState),
                 direction :: Direction,
                 breakpoints :: [Location] }

where
  gVisualize (UIDebugger)! (VValue dbgst) vst = (HtmlFragment content, vst)
  ec = debugInspect dbgst.trace

htmlCodeWindow :: [StackFrame] Model -> [HtmlTag]
htmlStack :: [StackFrame] -> [HtmlTag]
htmlEnv :: StackFrame -> [HtmlTag]
htmlBCInstances :: [AssocList TimedValue] SimpleState Model -> [HtmlTag]
```

The current version (10.08) of iTasks does not offer generic implementations for `(->)`-typed values. This shortcoming is fixed in the upcoming version. As some generic functions are only used when editing values, something we don’t do for UIDebugger typed values, we will use stubs for these functions. For `JSONEncode` and `JSONDecode` we do need actual implementations because values governed by the workflow engine are repeatedly written to and read from disk.
CHAPTER 5. FUNCTIONAL DEBUGGING

The functions `dynamic_to_string` and `string_to_dynamic` serialize and deserialize arbitrary values, including functions. Because of a bug in the function that escapes JSON strings we need to store these strings in base64 encoding:

```plaintext
JSNDEncode!UIDebugger! debugger = [JSNSString (base64Encode (dynamic_to_string (dynamic debugger)))]
JSNDecode!UIDebugger! [JSNSString s: r] = case string_to_dynamic (c \ c <- base64Decode s) of
  (debugger :: UIDebugger) = (Just debugger, r)
  (Nothing, [JSNSString s: r])
JSNDecode!UIDebugger! rest = (Nothing, rest)
gUpdate {(<>): _ _ val updst = abort "This should not be called: gUpdate{(<>)}"
gVerify {(<>): _ _ val verst = abort "This should not be called: gVerify{(<>)}"
derive gUpdate UIDebugger, ...
derive gVerify UIDebugger, ...
```

Now we are ready to define the task that controls the state of the debugger. The user can edit breakpoints, toggle direction or perform one of the debugging commands. For each of these actions `ActionLabels` and corresponding `ButtonActions` are defined. `stepDebugger`, the task that performs a single debugging action, is defined using the basic task `showMessageAboutA`. The `showMessageAboutA` task will display the value, using our implementation of `gVisualize`, and allows the user to choose an action. Depending on the action chosen by the user we will update the `UIDebugger` value and recursively invoke the `stepDebugger` task so the user can observe the resulting state and choose the next action.

```plaintext
runDebugger :: Task Void
runDebugger = updateEvents input
  >>> \a -> stepDebugger { trace = debugStart model (initEC model es),
  direction = Forward, breakpoints = [] }
  
where
  input = zorglasten_input0
  model = bepalen_zorglasten
stepDebugger :: UIDebugger -> Task Void
stepDebugger dbg = showMessageAboutA "Debugger" ** DebugActions dbg
  >>> \(a, dbg) ->
    case a of
      ActionEdit -> updateBreakpoints dbg breakpoints >>> \bps ->
        stepDebugger (dbg & breakpoints = bps)
      ActionDirection -> case dbg.direction of
        Forward -> stepDebugger (dbg & direction = Backward)
        Backward -> stepDebugger (dbg & direction = Forward )
      ActionStep -> stepDebugger (dbg & trace = debugStep dbg.trace dbg.direction )
      ActionStepIn -> stepDebugger (dbg & trace = debugStepIn dbg.trace dbg.direction )
      ActionStepOut -> stepDebugger (dbg & trace = debugStepOut dbg.trace dbg.direction )
      ActionContinue -> stepDebugger (dbg & trace = debugContinue dbg.trace
dbg.direction dbg.breakpoints )
      ActionQuit -> stop
```
CHAPTER 5. FUNCTIONAL DEBUGGING

We conclude this section with a screenshot of the debugger in action in figure 5.1. Most of the screen is occupied with the visualization of UIDebugger which consists of:

- A pretty printed version of the model
- The stack
- The list of variables in scope
- The list of BC instances referenced by the variables

Below this visualization are a number of buttons which trigger the actions in stepDebugger.

Figure 5.1: The main screen of the breakpoint debugger

5.3 Summary

In this chapter we demonstrated how, using the prototype interpreter, we built two different debugging tools. First we implemented a delta debugger
that automatically minimizes failure inducing input. In situations where the domain engineer does not know what part of the input causes the erroneous behavior, this tool can be used. The delta debugger will systematically try to reduce the input and tests whether the error still occurs. Having a minimal input that triggers the erroneous behavior makes understanding the bug a lot easier.

The second debugging tool is a breakpoint debugger. We demonstrated that with minor modifications to the interpreter, the model can be executed step by step. Because the changes are only minor, the interpreter remains a concise description of the semantics of the DSL. Using this stepped interpreter we implemented a breakpoint debugger that supports: stepping (in/out/over), breakpoints and reverse stepping.
Research never complete. In the process of finishing this thesis, corners have been cut and newly discovered questions have been left unanswered. In the first section, we will go over the current limitations of the interpreter and debugger. Most of these are not fundamental limitations, but simply areas where the current implementation could be improved upon. Then we will consider the option of using our work for more than just prototyping. For example one could imagine static analysis or code generation to be included in the same framework. In the final section, we will explore some other debugging and testing techniques. Reusing our interpreter opens up new opportunities for testing and debugging which are not necessarily aimed at finding mistakes in functional models but are interesting nonetheless.

Limitations

To demonstrate our ideas, we implemented a proof of concept interpreter and debugger. Since they serve only as proof of concepts they are lacking in some respects. Most of these limitations are not fundamental limitations of our approach. Rather they are suggestions for improvement of the current implementation.

Time dimensions

The first shortcoming of the interpreter is that it only keeps track of a single time axis: valid time. As mentioned in section 2.4, there are three different time axes to consider: valid time, report time and transaction time. The report time records when something is reported to the organization, the transaction time records when something is processed by a service. Both of these are relevant for explaining previous behavior of the system but should not be relevant for the functional semantics. In theory the valid timeline is the only relevant timeline for functional modeling. In TSL though, some functional models do refer to the report time. One option for improvement is to change the models (and the semantics) such that they only rely on the valid time.
The other option is to change the interpreter such that it also allows models to refer to the report time.

**Modeling language**

Models for TSL are built up from a large number of language constructs. Our interpreter implements only a selection of these language constructs. Furthermore some parts of the language are implemented in the interpreter as expressions without side effects. Also, BC instances currently support only one-to-one relations. We choose our language such that it convincingly demonstrates the feasibility of our ideas. The language used in this prototype is neither a complete nor an accurate version of the language that specifies TSL.

**Origin tracking**

The `StmWijziging` statement checks whether BC instances have changed. For each change the interpreter will perform the associated list of statements. To implement this we need to keep track of which events changed which `TimeBox`. These are referred to as the origins of a time box. Currently, the interpreter handles this uniformly for all operations on timed values but this turns out to be too crude\(^1\). As a result the interpreter will detect a change when there is not really one, performing the computation superfluously.

**Database support**

In section 3.6 we introduce the `State` class that abstracts over the way we store BC instances. Currently only `SimpleState` implements this interface. Originally, we intended to pass instances of `State` uniquely through the interpreter so that we could later use the same interpreter with an actual database. The current interpreter does not allow for this, but this should not be too difficult to fix. The stepped interpreter on the other hand, relies on the fact that `SimpleState` is just a value. It is not clear how the breakpoint debugger described in this thesis can be used with an external database. If the debugger would need to be used with an actual database, some method for database rollbacks would have to be implemented.

\(^1\)Origins are stored in the `TimeBox`. All functions on timed values are implemented by means of the `apply1` and `apply2` functions. These functions are now responsible for merging the origins but this turns out to have some unwanted effects. For example the function `assign` is implemented using `apply2` which combines the origins of the left hand side with the origins of the right hand side. This is not what we want, we would rather have `assign` replace the origins if the right hand side has a different value then the left hand side.
CHAPTER 6. FUTURE AND RELATED WORK

Debugging expressions

The breakpoint debugger cannot debug inside expressions. Currently the stepped interpreter for statements uses the original interpreter to evaluate the result of an expression. The entire expression will be evaluated in a single step, so the debugger is unable step through smaller steps that the programmer can observe. The stepped interpretation pattern could also be used for expressions. The Computation datatype supports functions that have \( s \) as intermediate state and \( f \) as final result, and the combinators work with these accordingly. The only problem is that expressions only observe the intermediate state and have no effect on it. So implementing such a stepped debugger will allow the domain engineer to step through the computation but the debugger will not show any change in state! Somehow representing the result of a computational step in the ExecutionContext would resolve this problem.

User interfaces

For demonstration purposes we built a basic interface for the breakpoint debugger. A lot of useful functionality could also be made available by just defining appropriate user interfaces. For example one could build an user interface to simulate a group of services and control how events are directed from one service to the other. The interpreter could be used to simulate the individual services.

Another useful addition would be an interface that allows the user to indicate erroneous behavior. Then a test can be constructed that for delta debugging to minimize the error inducing input. This test could also be used by the breakpoint debugger to break at the exact position where the behavior occurs.

Finally, a user interface to observe and change the local variables and BC instances would improve the usability of the breakpoint debugger. Also it would enable the possibility of changing the intermediate state in the breakpoint debugger. The debugUntil function already supports this, so only the user interface needs to be updates before the user can start using this functionality.

Performance

Since there are no side effects, the Clean run-time will never destructively update values. When a value is passed to another function it can safely pass it by reference. Many data structures will be shared and therefore a data structure like Trace need not be expensive. This ought to keep down the memory usage of our breakpoint debugger (but this is not extensively tested). The use of Trace in the iTasks library however could result in performance
problems. The iTasks engine stores these traces as part of the tasks state. At each step the `Trace` will be serialized to disk. Because of the serialization traces cannot be shared within the task state and we run the risk of consuming more and more memory at each step, until it runs out. Our current task is tail recursive and therefore does not have this problem, previous traces will simply be garbage collected. For more complicated task structures this behavior could be problematic. We expect that requiring a tail recursive task definition will not be a huge limitation. Investigating whether this is a reasonable assumption and verifying whether our approach scales for larger models remains future work.

**Beyond prototyping**

In this thesis, we have argued that building a prototype in a lazy functional programming language is both feasible and worthwhile. The interpreter concisely captures the semantics and easily extends to simulation and debugging tools. This raises the question: why not build the rest of the transformation architecture in Clean?

There is quite some research available on the implementation of DSLs in lazy functional programming languages. Most of the research focuses on embedding the DSL by defining constructors and/or combinators as building blocks for models. Because models are just expressions in the host language, we get parsing and type checking for free. The iTasks framework is such an example of a DSL embedded in the Clean programming language [JPKA10, PAK07]. With this framework one can model workflows which can be executed by the iTasks engine. The Clean compiler implicitly checks whether these workflows are syntactically correct and consistently typed.

The prototype presented in this thesis could also be regarded as an embedded DSL. We did not implement a parser or type checker but instead defined an algebraic datatype and some convenience functions that help construct instances of that datatype. Because we needed an explicit representation for types we could not directly use the types already available in Clean. Instead types are represented with an explicit data structure of type `Type`. The explicit type representation opens up the possibility of having inconsistently typed models without the compiler noticing, they are consistent as far as the Clean compiler is concerned. For example, boolean conjunction is handled by a function called `ww_conjunction` which expects two arguments of type `Value`. For the Clean compiler any `Value` will do, but at run-time the interpreter expects both values to be of type `TypeWW`.

There a number of possible solutions for this problem. It might be possible to enforce restrictions on the expressions using shadow types [LM00] or by replacing our algebraic datatype with a *generalized algebraic data-
type \textsuperscript{[PVWW06]}. Another possible solution is to implement a type checker ourselves. The type system of our DSL is rather straightforward, so this should not pose any problems.

Typically, models will be constructed by domain engineers who may or may not be familiar with functional programming. If the DSL is embedded in a functional programming language, models are expressions in that language. Formulating these expressions without a solid understanding of the underlying language will be difficult. For example: syntactic errors will refer to the structure of the underlying language and not to the structure of the DSL. Typing errors will be unification errors, which are sometimes hard to understand. Another downside to embedding is that it confines the structure of the language to what is possible within the syntax of the host language.

To overcome these problems there are two options: either build a parser or build a projectional editor. For textual DSLs one can use parser combinators to define a parser that turns text into a syntax tree. Parser combinators offer a convenient way to define such a parser. They have been shown to scale well and allow for error reporting and automatic error correction \textsuperscript{[KP99, Swi01]}. Instead of a parser one could also build an editor that constructs models. There is also research available on how to build projectional editors in Haskell \textsuperscript{[Sch04]}.

Also functionality like determining testing coverage or code generation could be implemented in Clean. For example, in \textsuperscript{[BCS98]} a DSL for hardware design is implemented together with an interpreter which simulates the design. Symbolic interpretation is used to generate assertions for a theorem prover and the implementation in VHDL. Symbolic computation might not only be useful for code generation but also for determining test coverage. There have also been efforts to build code generators that produce optimized code \textsuperscript{[EFD03]}.

Since this is the future work section, all of the above is not without speculation. These suggestions need careful consideration before they can actually be applied. Perhaps an easier alternative is integrating with existing tools like MPS, Spoofax or xText. Serializing and deserializing the model is not difficult as both Clean and Haskell offer generic libraries for this purpose. For Clean the JSON library from the iTasks framework seems to be a good candidate \textsuperscript{[LP11]}. In Haskell the HaXML library allows saving or loading arbitrary data structures using the XML format \textsuperscript{[WR99]}.

More debugging and testing

In section 5.1 we demonstrated how, using just a few lines of code, the interpreter could be integrated in an algorithm that systematically reduces failure inducing input. Having a minimal interpreter opens up a lot of similar opportunities, not just for functional debugging. In this section we will discuss
three of these opportunities.

Version control management is standard practice while developing software. If a bug is encountered it may be a regression, locating it within the revision history can be very useful. If we can isolate the regression, it will be much easier to understand the erroneous behavior. Delta debugging can be used for this purpose [Zel99]. How this is implemented depends greatly on the version control management system that is used to manage models.

Another interesting option using the interpreter when testing the generated system. A test consist of input together with the expected output. The output is supplied by, what is known in testing terminology as, a test oracle [GVEB08]. Usually this expected output is constructed manually, but when searching for a regression it could also be a previous version of the system. The interpreter serves as an executable specification of the semantics of the system, therefore it is possible to use the interpreter as a test oracle. All is needed are sensible inputs to test whether the generated code conforms to the prototype. These inputs either be generated automatically, constructed by hand, or they can be recorded during real life usage of the system.

Finally it may be worthwhile to thoroughly test the interpreter. The interpreter serves as a specification of the functional semantics. Finding inconsistencies in our understanding of the semantics early on in the development cycle can save costs later on. Explicitly formulating properties of our language increases our understanding of the semantics. These properties can then be tested using a generic testing library like Gast for Clean [KATP03] or QuickCheck for Haskell [CH00]. This approach has been applied successfully to the while programming language and the iTasks DSL [KPA10]. The current prototype already tests a number of properties on: values, timed values and the state. Extending this test suite to the interpreter would increase our confidence in its quality.
DSLs are languages designed to solve problems in specific domains. A DSL could be used to write functional specifications in software projects with many functional requirements. Such a DSL abstracts from the technical implementation and allows domain experts to express the functional behavior of the system. Since the functional behavior is kept separately from the technical implementation, a clean separation of concerns is achieved.

But designing and implementing a DSL is hard. A number of tools exist to ease the burden: language workbenches, parser generators and meta modeling frameworks. These tools tend to focus on the syntax (or structure) of the language, but not on the semantics. Typically when designing a language, the semantics will only be described informally. Not until after the first implementation is finished, the consequences of design decisions will become apparent. This would not be a problem if the DSL in question were simple, but for nontrivial DSLs that might turn out to be problematic. An approach for rapid prototyping is needed.

Another frequently overlooked problem is that, once we write large specifications using a DSL, we need tools to debug these specifications. When confronted with erroneous behavior, it most likely will not be obvious where in the specification the error is located. Just executing the specification and reviewing the result will not be enough. Functional debugging tools make it easier to understand the behavior of the system in terms of the specification. These tools will be used throughout development for experimentation and debugging purposes.

In this thesis we address these two problems. We suggest that, by embedding the DSL in a lazy functional programming language, a prototype can rapidly be constructed. Instead of defining a meta model, one defines an algebraic datatype. Because specifications are expressions in the host language, there is no need to write a separate parser. Furthermore, functional programming offers interesting ways to leverage this prototype for debugging purposes. We demonstrated this approach by constructing an interpreter and two debugging tools for a system that is being implemented by Capgemini.
Capgemini implemented a system named TSL for a Dutch governmental organization. A DSL was used to specify the functional behavior of the system. Before the specifications could be executed, code generators had to be implemented. Furthermore, there was a need for tools to debug specifications. Eventually, the code generators and a functional debugger were developed, but not until late in the development process.

A code generator processes a model and generates code that, when executed, is semantically equivalent. Instead we chose to write an interpreter which directly performs the semantic actions. We define an interpretation function that takes as input a model and a state, and then produces a new state as output. Functional programming languages allow for a very concise formulation of such an interpreter. The interpreter is just a prototype and should only specify the functional semantics of the language. Therefore, we chose not to consider technical details such as databases, integration with existing systems and performance.

We demonstrated the feasibility of this approach by building a proof of concept implementation. This proof of concept interprets a simplified version of the DSL used by Capgemini. The core interpreter is under 150 lines of code in length. It relies on a few other modules as well (these are under 700 lines of code in length). Because the interpreter is so small, it could be considered a specification of the semantics.

For the implementation of a breakpoint debugger, we relied on the fact that the interpreter is a function in a functional programming language. Functional programming languages allow functions, even those not fully evaluated, to be used as values. Using this feature, we demonstrated a pattern that extends the interpreter such that the computation can be viewed step by step. This approach does not change the interpretation function significantly, so it still compactly represents the semantics. But it does offer a fully fledged debugger which can step in and out of procedures, stop on breakpoints and it can even revert to earlier states in the computation. All of this functionality is implemented outside of the interpreter.

We also demonstrated how delta debugging can be used with our interpreter to minimize failure inducing input. This results in a tool that systematically reduces the input until we have a minimal example. Such a tool makes debugging real life bugs a lot easier. This shows that having a simple interpreter opens up the possibility of developing a set of tools that might prove useful throughout development.

If specifications cannot be executed during development, chances of mistakes going unnoticed will be greatly increased. Also, the lack of a running system rules out experimentation. Writing small examples and observing their behavior will help engineers understand the new language. This is especially important since, during development of the DSL, the semantics are under de-
bate. Having a prototype and debugging tools available will benefit, not just users of the DSL, but everybody involved in such a project.

It is not for nothing that rapid prototyping is dictated by many software engineering methodologies. Designing software involves a bit of speculation, which is not without its risks. By quickly experimenting with an implementation these risks can be mitigated. For DSL based projects, functional programming offers an interesting platform for building such a prototype. This prototype can be leveraged to a fully fledged debugger without much effort. The prototype will not only be valuable in early stages of development, it can serve as the basis for tools that can be used throughout development of the system.


