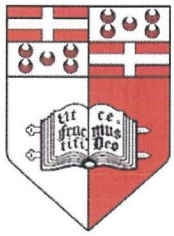
A Formal Semantics for Imperative Compensations

Jennifer Camilleri

Supervisor: Dr. Adrian Francalanza
Prof. Gordon Pace



Faculty of ICT

University of Malta

June 2013

Submitted in partial fulfillment of the requirements for the degree of B.Sc. I.C.T. (Hons.)

Acknowledgements

This project would not have been possible without the help and support of many people around me.

First and foremost, I would sincerely like to thank my supervisors, Dr. Adrian Francalanza and Prof. Gordon Pace for their continuous support, patience and motivation throughout this project.

Finally, I would like to thank my family and friends for believing in me and for the constant support during my studies.

Abstract

Formal Semantics deal with the study of meaning of a programming language. A formal specification is a way to describe a programming language precisely and unambiguously. At the moment, a lot of work was done on formalising compensations. A compensation is an action that is associated with another action. When the compensation is executed, it cancels the effects of the activity it is associated with.

Programming languages and libraries based on compensations are usually specified informally. The language behaviour is described using natural languages which tend to be ambiguous. The only interpretation of the language is what the compiler outputs.

The aim of this FYP is to develop an operational semantics for an imperative programming language that uses compensations as part of its error recovery mechanism. At the moment only a written documentation is available for the said language.

We will create a set of reduction rules that describe the behaviour of the language constructs. The proposed semantics could then be used as a precise and unambiguous standard for the language.

Contents

List of Figures

List of Tables

1. Introduction

Software has evolved a lot in the past years and the need of reliable software is essential. Nowadays, people expect their systems to operate correctly even under severe conditions. However, given the complexity of today's computer systems, the presence of certain bugs is inevitable [1]. The demand for highly-reliable software has led to fault tolerance mechanisms. A fault tolerant system is able to recover from errors and continue execution.

The use of compensating activities for reversing previously executed activities has proved to be quite a useful technique when handling failure in a program. A compensation can be described as an act of un-doing an action with another action thus cancelling the effects of the first action.

The use of compensations in an imperative programming language, have been investigated by Lydia Vella [9] in her dissertation. The developed language has the usual imperative language constructs together with four other constructs that deal with compensations. However, this language lacks a formal semantics.

1.1 Objectives

Informal descriptions are inherently ambiguous. There can be cases where this can lead to different interpretations of a given program. A recurring problem that causes ambiguities in how the if-then-else statement is interpreted in the *dangling else*

$$\texttt{if } a \texttt{ then if } b \texttt{ then } s_1 \texttt{ else } s_2$$

In the following example, s_2 can be executed either when a evaluates to `true` or else when a evaluates to `false` and b evaluates to `true`. In other words, the above statement can be interpreted as either

$$\texttt{if } a \texttt{ then } \{\texttt{if } b \texttt{ then } s_1 \texttt{ else } s_2\}$$

or

$$\texttt{if } a \texttt{ then } \{\texttt{if } b \texttt{ then } s_1\} \texttt{ else } s_2$$

The specification of a formal semantics is one way of overcoming such ambiguities. In this case, we can provide a formal interpretation whereas we attach the `else` to the nearest `if` statement.

There are a number of other reasons why we might want to describe our language formally:

1. With the help of formal semantics we can reason about the behaviour of certain programs. Apart from reasoning about single programs, formal descriptions can help us prove the correctness of the language compiler

2. Formal semantics can help us understand the language design better

3. By describing a programming language formally, we are separating the specification from the implementation. If the specification is complete and we can show that the implementation is correct with respect to the specification, then we have shown that the implementation is correct

1.2 Dissertation Overview

The first part of this dissertation presents some background information that the reader must understand before reading this FYP. Chapter 2 gives an overview of operational semantics while making a distinction between big-step and small-step semantics. In Chapter 3, a number of error recovery mechanisms are presented, mainly compensations. In chapter 4, the language syntax is presented while giving a brief description of the compensating constructs and their use. Further analysis of why a formal description is needed in Vella's language is given in chapter 5 where an ambiguous program is given using the language constructs.

The second part focuses on giving a formal description to our programming language. In chapter 7 we present a preliminary semantics for Vella's language. An evaluation of the semantics is given by comparing the output of the reduction rules with the output of the language compiler. In chapters 8, 9 and 10 we shall present an issue in our programming language while providing solutions on how to solve it. Finally, in chapter 11, we shall explore the notion of scope variables inside the compensation mechanism. Note that we don't have a specific design and evaluation section. However, the design issues will be included as we build the reduction rules while the evaluation will be given at the end of each chapter.

The relevant literature focusing on compensations, studied during this dissertation, is discussed in chapter 12. A description of two formalisms that deal with compensations is given while focusing mainly on the semantics of said formalisms.

Part I

Background Information

2. Operational Semantics

A language definition can be divided into syntax and semantics. Syntax refers to the building blocks of a programming language. These buliding blocks allow us to construct programs in our language. The semantics refer to the behaviour of said programs.

Operational semantics describe how a program is interpreted as a sequence of computational steps. Operational semantics give an abstraction of how a program is executed on a machine while ignoring details like use of registers and addresses for variables [2].

One way of describing the behaviour of a language's constructs is by using reduction rules. The rules take the form of:

$$\frac{premise \quad premise}{conclusion}$$

If we can show that all the premises are valid, we can deduct that the conclusion is valid as well.

Operational semantics can be split into two categories, namely, small-step and big-step semantics. In this section we will present a semantics for a simple impera-

tive programming language which we will call L_1. The language syntax is shown in Table 2.1. The following naming conventions are used: variables are denoted by x, values are denoted by v, expressions are denoted by e and commands are denoted by c. A value can only be an integer, i or a boolean, b.

e	$\in$	**Expression**
	::=	$v \mid x \mid$ `not` $e \mid e$ `and` $e \mid e$ `+` $e \mid e$ `-` $e \mid \ldots$
c	$\in$	**Command**
	::=	`skip` $\mid c$`;`$c \mid x$ `:=` e
		$\mid$ `if` e `then` c `else` $c \mid$ `while` e `do` c

Table 2.1: Syntax of L_1

2.1 Big-Step Semantics

Big-step semantics describe how the overall result of the executions are obtained. Big-step semantics is concerned on the relationship between the start and final state while ignoring the steps in between.

The semantics for expressions of L_1 will be given in terms of a big-step semantics. The main reason for choosing a big-step semantics is that expressions always return a value. Thus, there is no need to express the intermediary computation. Additionally, expressions are side-effect free. Since the state is never updated in expressions, we only need to focus on the final value.

To specify the operational semantics of an imperative programming language, we must specify how a program will transform the state of a system [3]. The program state, which is denoted by s, is a map between variables and values.

The big-step semantics take the form of a relation denoted by the symbol $\Downarrow$. The type of $\Downarrow$ is:

$$\Downarrow\colon \langle s,\, e\rangle \Downarrow v$$

where expression e is evaluated to value v under state s. The rules defining the relation consist of two axioms and 5 inductive rules:

$$\frac{}{s, w \Downarrow w}\ \textsc{eVal} \qquad \frac{}{s, v \Downarrow s(v)}\ \textsc{eVar}$$

$$\frac{s, e_1 \Downarrow w_1 \qquad s, e_2 \Downarrow w_2 \qquad w_3 = w_1 + w_2}{s, e_1\texttt{+}e_2 \Downarrow w_3}\ \textsc{eAdd}$$

$$\frac{s, e_1 \Downarrow w_1 \qquad s, e_2 \Downarrow w_2 \qquad w_3 = w_1 - w_2}{s, e_1\texttt{-}e_2 \Downarrow w_3}\ \textsc{eSub}$$

$$\frac{s, e_1 \Downarrow w_1 \qquad s, e_2 \Downarrow w_2 \qquad w_3 = w_1 \wedge w_2}{s, e_1 \texttt{ and } e_2 \Downarrow w_3}\ \textsc{eAnd}$$

$$\frac{s, e_1 \Downarrow w_1 \qquad s, e_2 \Downarrow w_2 \qquad w_3 = w_1 \leq w_2}{s, e_1 \texttt{ <= } e_2 \Downarrow w_3}\ \textsc{eLeq}$$

$$\frac{s, e \Downarrow w \qquad w' = \neg w}{s, \texttt{ not } e \Downarrow w'}\ \textsc{eNot}$$

Figure 2.1: Big-Step Semantics for Expressions of L_1

The rules defining expressions are pretty straight-forward. The axiom rule eVal express the fact that values do not need to evaluate further.

The rule eAdd states that if an expression e_1 can be evaluated to a value w_1 under state s and similarly an expression e_2 can be evaluated to a value w_2 under the same state s, then the expression e_1 + e_2 can be evaluated to a value w_3 under s. The rules eSub, eAnd and eLeq behave similarly to eAdd.

The rule eNot, relies on the premise that an expression e, is evaluated down to

a boolean value w. If the premise is satisfied, then we can deduct that the negation of e can also be evaluated down to a boolean value w'.

The rule EVAR differs from the other rules since we need the state to evaluate the expression. Rule EVAR states that a variable v under the state s can evaluate down to the value present in the stack that is assigned to that particular variable.

2.2 Small-Step Semantics

Small-step semantics is mainly concerned on how the individual steps of computation take place rather than the final result. To be exact, we are concerned about how the state of a program is modified during execution. In this dissertation, the small-step semantics take the form of a relation, denoted by the symbol $\longrightarrow$. In an imperative programming language, the most basic type of $\longrightarrow$ is:

$$\longrightarrow: \langle s, c\rangle \longrightarrow \langle s, c\rangle$$

whereas s represents the program state and c represents a command. After several computational steps, every command reduces down to a `skip` once it terminates.

$$\langle s_1, c_1\rangle \longrightarrow \langle s_2, c_2\rangle \longrightarrow \ldots \longrightarrow \langle s_n, \texttt{skip}\rangle$$

The multi step reductions can be denoted by the symbol $\xrightarrow{*}$, where * represents 0 or more steps. Each command that terminates can be represented as:

$$\langle s, c\rangle \xrightarrow{*} (s', \texttt{skip})$$

When describing commands in L_1, we are more interested on how the program state is updated after every step of execution and not what the final value is. Thus, the semantics for commands will be given in terms of a small-step semantics.

In small-step semantics, the axiom rules describe reductions involving commands that are already evaluated down to `skip`. On the other hand, the inductive rules describe how reductions occur inside larger expressions. The rules defining the relation are shown in Figure 2.2

$$\frac{}{s, \texttt{skip};c \longrightarrow s, c}\ \textsc{rSeq1} \qquad \frac{s, c_1 \longrightarrow s', c_3}{s, c_1;c_2 \longrightarrow s', c_3;c_2}\ \textsc{rSeq2}$$

$$\frac{s, e_1 \Downarrow v \qquad s, e_2 \Downarrow w}{s, e_1 := e_2 \longrightarrow (s[v \rightarrow w]), \texttt{skip}}\ \textsc{rAss}$$

$$\frac{s, e \Downarrow \texttt{true}}{s, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \longrightarrow s, c_1}\ \textsc{rIf} \qquad \frac{s, e \Downarrow \texttt{false}}{s, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \longrightarrow s, c_2}\ \textsc{rElse}$$

$$\frac{s, e \Downarrow \texttt{true}}{s, \texttt{while}\ e\ \texttt{do}\ c \longrightarrow s, c;\texttt{while}\ e\ \texttt{do}\ c}\ \textsc{rWhile1}$$

$$\frac{s, e \Downarrow \texttt{false}}{s, \texttt{while}\ e\ \texttt{do}\ c \longrightarrow s, \texttt{skip}}\ \textsc{rWhile2}$$

Figure 2.2: Small-Step Semantics for Commands of L_1

2.2.1 Sequencing

The behaviour of the sequencing operator can be defined using the rules rSeq1 and rSeq2. These rules state that to execute $c_1;c_2$, in state s, we must first execute c_1 one step from s.

If after one step, execution of c_1 has not yet terminated, we must first complete it before starting execution of c_2. This case is captured by the rule rSeq2. If after executing one step of $\langle s,\ c_1\rangle$ we get the intermediate configuration $\langle s',\ c_3\rangle$, then the next configuration would be $\langle s',\ c_3;c_2\rangle$ This shows that in order to begin executing c_2, we must first complete execution of c_1.

On the other hand, if execution of c_1 has been completed, then we can start executing c_2. This is captured by the rule RSEQ1. If the result from executing the first step of c_1 from state s is the intermediate configuration $\langle s, \texttt{skip}\rangle$, then the next configuration would be $\langle s, c_2\rangle$ so that we start executing c_2.

2.2.2 Assignment

The assignment operator $e_1 := e_2$ requires that e_1 is evaluated down to a variable v and e_2 is evaluated down to a value w. If both of these premises are satisfied, then the value w can be assigned to the variable v and stored on the stack s.

The rule RASS captures the behaviour of the assignment command. After one step of execution the assignment statement is fully executed. The next configuration would be $\langle s[v \rightarrow w], \texttt{skip}\rangle$. In this case the state $s[v \rightarrow w]$ is the same state s except that v now has a value w.

2.2.3 Branching

The rules RTHEN and RELSE describe the behaviour of the branching construct. If the boolean expression e is `true` in the current state s, then the first branch is executed. On the other hand, if the value of e is `false`, then the second branch is executed.

2.2.4 Looping

Similar to the branching command, the looping command relies on a boolean expression that determines the path of execution. The two rules that describe the looping command are RWHILE1 and RWHILE2. There are two possible cases in the construct `while` e `do` c:

- If the expression e, evaluates to `true` in the program state s, then we must first execute the body of the while loop. After execution of the body of the

loop, we must continue with the loop itself from the new state obtained.

- If the expression e, evaluates to `false` in the program state s, then the loop must terminate

The rule RWHILE1 captures the behaviour of the first case where the expression e is evaluated to `true`. The next configuration would be $\langle s,\ c\texttt{;while}\ e\ \texttt{do}\ c\rangle$ where the loop body c is executed first and it is then followed by the while loop again.

On the other hand, the rule RWHILE2 formalizes the second case where the expression e is evaluated to `false`. This causes the loop to terminate while leaving the state unchanged.

2.3 Conclusion

The aim of this chapter was to explain how to use operational semantics as a method to describe a programming language formally. First we started by explaining the use of operational semantics. Then, we distinguished between the two main categories of operational semantics, Big-Step and Small-Step semantics. Finally, we constructed a set of rules that describe formally a simple imperative programming language.

3. Error Recovery and Compensations

The demand for highly-reliable systems led to fault tolerance mechanisms. A fault-tolerant system allows certain errors to exist by applying error recovery techniques to restore normal computation [4]. Error recovery mechanisms can be divided into backward and forward recovery [5].

3.1 Backward and Forward Error Recovery

Backward recovery refers to strategies that first revert the program to a previous sane state before further continuing operation of the system, while forward recovery tries to recover the fault by correcting the state. The main difference between both strategies is that backward recovery depends on recovery points, which are points in time at which the state is saved in case it is required for future use [6]. On the other hand, forward error recovery makes further use of the faulty state and tries to correct it. Both strategies try to obtain an error-free state.

A simple and commonly used approach of forward error-recovery is the use of exception handling. A `try-catch` block is placed around parts of the code which might generate an error.

```
1 try {
2     int array[] = new int[2];
3     System.out.println(array[3]);
4 } catch(ArrayIndexOutOfBoundsException e) {
5     System.out.println("Error!");
6 }
```

A program first tries to execute a `try` block. If execution fails, the `catch` block is executed which tries to repair the state of that program. In the following example, the program tries to access an element in an array that does not exist. The program will terminate with an error message instead of crashing. Note that exception handlers are simply aware of a failure since no historic data is available.

On the other hand, backward recovery requires additional data to repair the state of a program. One approach of backward recovery is rollback [7] which is a perfect reversal of a previous action which leaves no evidence of said action. All past actions must be saved in the case an error occurs and these actions must be reverted back perfectly. After the application of a rollback, it is like the error never happened.

However, perfect reversal is not always possible. Example, errors may arise while recording transactions. In accountancy, errors are not corrected by undoing the incorrect transaction but rather by making other entries to set off such errors. In this case, a rollback would not be possible. A compensation mechanism works very similar to the technique used to correct errors in accounting.

A compensation is executed to try and undo an error as much as possible. Unlike rollback, it does not cancel the error but simply works on the state to try and correct it. Compensations can be used as a mean of failure handling whereas a compensating activity is associated with an activity. If a failure happens in the system, the compensating activity is executed to repair the state of the program.

Both compensations and exception handling are forward error recovery techniques. However, their mechanism is completely different. One difference between exception handling and compensations is that a compensation can only be performed on an activity that has completed successfully, not one that is in its faulting state. Additionally, compensations are history dependent. Whereas exception handlers only know about the occurrence of an error, compensations are aware of when the error occurred. Occasionally, the occurrence of an error is not enough to repair it. Therefore, many formalisms decide to implement both compensations and exception handlers, where the compensating activity can be executed in the catch block. Thus, when an activity throws an exception, the compensating activity is executed.

3.2 Compensations

Compensations are a form of forward error recovery mechanism [8]. A compensation is associated to another activity, such that when executed it cancels the effects of the activity to which it is associated.

For example, consider a party-planning system in which a user selects the venue and catering according to his preferences. For a transaction to be successful, it must succeed in full. Therefore, the user must book both catering and venue for the transaction to be successful. If the user declines either the venue or the catering, the whole transaction fails.

Upon receiving a request from a user, the system tries to book a venue and catering according to the users preferences. If the user accepts both choices, the system then charges the user's bank account and an email is sent to the user containing all details. Figure 3.1, represents the party planning system. The arrows represent the flow of execution while the boxes represent the forward action.

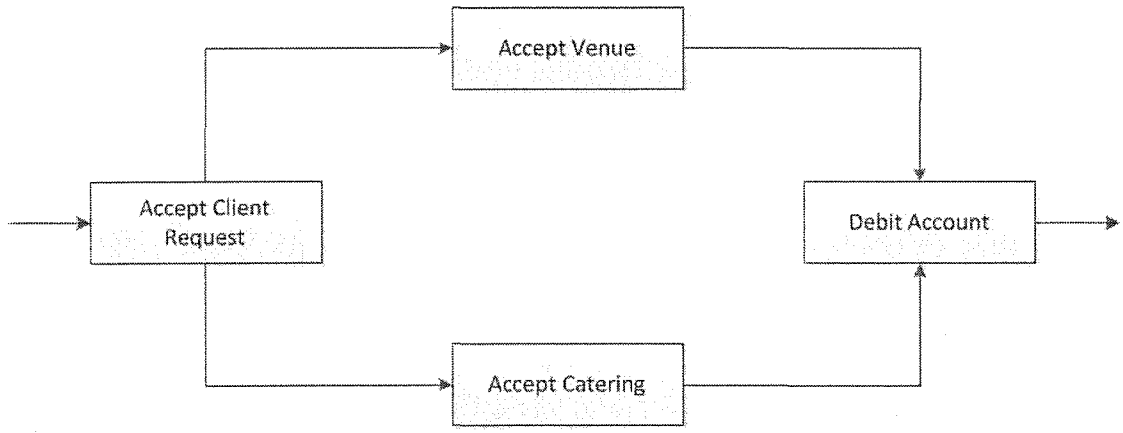


Figure 3.1: Party Planning System Forward Action

The system accepts a request from the client and makes the necessary reservations. When the user accepts both options, the system tries to debit the client's account. Should this process fail, the system must cancel the whole transaction. At the moment, the forward actions do not present a means by which this can be done. However, compensations do. Compensations can be associated with the forward actions shown in Figure 3.1. If a failure occurs in any of these activities, the previously completed activities must be compensated. Below is a list of activities together with their associated compensation.

- If the user declines the choice of venue, the reservation of the venue must be cancelled and an email is sent to notify the user about the cancellation
- If the user declines the choice of caterer, the catering order must be cancelled. Similarly an email is sent to the user to notify him/her about the cancellation

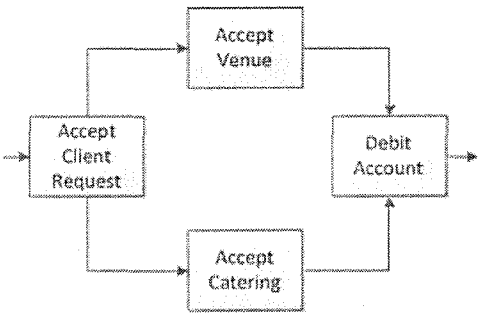


Figure 3.2: Party Planning System with Compensations

- If the user has insufficient funds in his bank account, both venue and catering must be cancelled and an email is sent to the client informing him that the transaction failed
- If the user cancels the whole reservation after he has already paid, both venue and catering must be cancelled. The system must also credit the user's bank account. Additionally, the user is charged a fee and an email is sent notifying the user about the cancellation. On the other hand, if it is too late to cancel the reservations, an email is sent to the user explaining the situation

A representation of the party planning system enriched with compensations is shown in Figure 3.2. The top part of the boxes represents the forward action. On the other hand, the bottom part of the boxes represent their associated compensating action. The normal arrows represent the flow of execution when an action executes successfully. The dashed arrows represent a fault.

Using this approach, if debiting the client's account fails, the whole process may simply be reversed by executing the compensations for the completed actions. Generally the order of executing compensating activities is in reverse order (as shown by the arrows in Figure 3.2). Thus, if debiting the client's account fails, the system first cancels both reservations and then notifies the client. After executing all the compensating actions, the system is restored to a state similar to the state that the system was before the transaction that was compensated, was executed.

3.3 Conclusion

This aim of this section was to give an overview of existing error recovery techniques while focusing mainly on the compensation mechanism. The differences between these techniques were highlighted. Lastly, a practical example where the compensation mechanism would be a useful error recovery technique was presented.

4. A Compensation Language

The language studied in this FYP [9] is an imperative programming language that uses compensations as part of its error handling mechanism. It assumes the usual constructs found in an imperative programming language plus an extra four constructs which handle compensations.

4.1 Syntax

n, m	$\in$	**ScopeName**
x, y	$\in$	**VarName**
e	$\in$	**Expression**
	::=	x | `not` e | e `and` e | e `+` e | e `-` e | ...
c	$\in$	**Command**
	::=	`skip` | c;c | x := e | `if` e `then` c `else` c
		| `while` e `do` c | `try` c `catch` c | `scope` n c
		| `undo` c `with` c | `compensate` n | `purge` n

Table 4.1: Language Syntax

In Vella's language, each expression represents a value. Expressions do not modify the state and are side-effect free.

A command c can be the no-op command `skip`; a sequential composition of two commands c;c; the assignment operator $x := e$, which assigns the value of e to variable x; the conditional operator `if` e `then` c `else` c, which executes the first

command if e is `true` or the second command if otherwise; the looping operator `while` e `do` c, which repeatedly executes command c until e becomes `true`; and the exception handling operator represented by `try` c_1 `catch` c_2 such that if c_1 terminates abnormally, c_2 is executed to repair the state of the program. Additionally, the language has four other constructs that handle compensations.

Compensation Installation

`undo` c_1 `with` c_2

This construct associates the compensating block of code c_2 with the block of code c_1 such that if the program fails, c_1 can be reversed. The compensating activity c_2 is installed once c_1 has terminated successfully.

Scoping

`scope` n c

This construct creates a new compensating scope with the name n, such that all compensations installed during execution of c, belong to that scope. In our language the scope name has two roles. First, it acts as an identifier for a group of compensations. For this reason, scope names must be unique. Second, it acts as a marker up to which point we have to compensate.

Compensation Activation

`compensate` n

Using this construct, the programmer can explicitly execute compensating activities belonging to scope name n. Compensations are executed in reverse order of installation. The use of `compensate` n outside of scope n must not be permitted.

Compensation Deletion

`purge` n

Similar to the compensate construct, the programmer can explicitly discard installed compensations. When using this construct, all compensations belonging to scope n are deleted. The programmer must use this construct only within scope name n.

We shall now present a simple program that uses the compensation constructs:

```
1   scope transaction
2   {
3       account := 100;
4       price := 10;
5
6       undo {
7           account := account - price;
8       }
9       with {
10          account := account + price;
11      }
12
13      if(error == true)
14          compensate transaction;
15  }
```

Figure 4.1: Practical example using compensation constructs

In the above program, the compensation **account := account + price** is associated with the activity **account := account - price**. If the variable **error** is `true` in the program state, all compensations belonging to scope transaction must be executed. In the next section we will use the same example and show how the stack is affected with these commands.

4.2 Compiling Compensations

Compensations must be stored in the appropriate data structure which allows us to effectively execute the compensations in reverse order of installation. Being a LIFO data structure, the stack was considered the best option.

The stack can contain either a string representing the scope name or a block of code representing a compensation action for a previously executed action. The different instructions supported by our language affect the compensation stack in different ways

- Normal imperative language instructions leave the stack unaffected
- `undo` a `with` a': the compensating action a' is pushed onto the stack when the action a has terminated successfully.
- `scope` n c: upon entering a new scope, a string containing the scope name n, is pushed onto the stack. This string acts as a delimeter up to which point we have to compensate. When the sub-program c terminates, the string containing the scope name is popped from the stack. Any compensations that were installed during execution of c are not removed from the stack.
- `compensate` n: all compensations found on the stack are popped and executed until scope name n is encountered at the head of the stack. Since the stack is a LIFO data structure, the last pushed compensation is the first to be executed. Any scope names found on the stack are not removed.
- `purge` n: this construct affects the stack exactly like the `compensate` construct. The only difference is that when using the `compensate` construct, the compensations are executed. On the other hand, the `purge` construct simply pops the compensations from the stack.

We shall now take the example shown in Figure 4.1 to see how the constructs affect the compensation state.

```
1   scope transaction                          [<transaction>]
2   {
3       account := 100;
4       price := 10;
5
6       undo {                                 [c::<transaction>]
7           account := account - price;
8       }
9       with {
10          account := account + price;
11      }
12
13      if(error == true)
14          compensate transaction;            [<transaction>]
15  }                                          [ ]
```

where c is equal to **account := account + price**. The right hand side of the example above represents how the stack is updated by the compensation constructs.

In line 1, the scope name is pushed onto the stack. The scope name will act as a marker up until which the compensations will be executed. The assignment statements in lines 4 and 5 leave the stack unaffected. Once we finish executing the command in line 7, its associated compensation is pushed onto the stack. Let us assume that the variable **error** in line 13 evaluates to `true`. Therefore, we can proceed to execute the `compensate` command. This forces all compensations in the compensation stack to be executed up until the scope name **transaction** is found at the top of the stack. Finally, when we reach the end of the scope, the scope name must be popped out of the stack. Thus, leaving us with an empty stack.

5. Problem Definition

Many programming languages that we are familiar with nowadays are described using a natural language. These descriptions usually take the form of a manual which consists of hundreds of pages. The ambiguity and incompleteness of natural language descriptions led to problems with understanding the specification of a programming language. For example, the semantics of Java threads, which were written in English, were difficult to interpret [10].

At the moment, the language designed by Lydia Vella lacks a formal semantics and only a written documentation is given of the language. It is usually very hard to write precise natural language descriptions. These usually end up being very large, ambiguous and incomplete. We need our language to have clear and precise definitions. An ambiguous specification usually leads to different implementations due to lack of common understanding between the language designers and the users.

The aim of this FYP is to develop a formal semantics for the said language. This will be used as a precise and unambiguous description of the language. To illustrate more the need of a formal semantics, we shall consider the following example:

```
1    ...
2    scope transaction {
3        price := 5;
4        undo
5            pay(price);
6        with
7            refund(price);
8        price := 10;
9        compensate transaction;
10   }
11   ...
```

This example represents a simple transaction that allows the user to pay for a single item. The variable **price** is being used to hold the monetary value of an item that the user is buying. The compensating activity `refund(price)` is associated with the activity `pay(price)` such that if the program fails, the action can be reversed. Therefore as soon as we finish executing line 5, its associated compensation must be pushed on the stack. When we reach line 9, the compensating activities belonging to scope **transaction** must be executed. In this case only one compensation will be executed

Vella's syntax allows us to construct programs such as the one above where a variable is bound more than once. At this point there can be two different interpretation of this program. The most intuitive way is to bind the variable **price** to the value 5. This is because we would want to refund the user's account with the amount that he paid for.

Another interpretation of the above program is to bind the variable **price** with 10. This is not an ideal interpretation since we would end up refunding the user's account with a value that is bigger than the amount he paid for. However, some might argue that it is the correct way to interpret this program.

This example is very similar to the `let` construct

```
let x = 10 in
    let y = x + 5 in
        let x = 2 in
            y + 4
```

Some programming languages bind the value of x in the expression $y = x + 5$ to the value 10. On the other hand, other programming languages might bind the value of x in the expression $y = x + 5$ to 2. This will lead to two different outputs.

In this FYP we will try to create a formal semantics that describes the behaviour of the language constructs while at the same time eliminate any ambiguities present in the language.

Part II

Semantics

6. Overview

In this part, we shall present one possible semantics for Vella's syntax. In Chapter 7, the preliminary operational semantics will be presented. The output of the reduction rules should behave the same way as the output of the language compiler. In Chapters 8, 9 and 10, three edge cases will be explained while discussing various solutions to each edge case.

In Chapter 8, we will focus on how variables are handled in compensations. Firstly, we will start by discussing how Vella's compiler handles variables inside compensating activities. Then we will present a set of reduction rules to remove any ambiguities with variable handling. Chapter 9 deals with the problems brought with nested compensations. Chapter 10 focuses mainly on compensation scopes. Lastly, in chapter 11 we will tackle the notion of local variables inside the compensation mechanism. At the moment, this is not supported in Vella's language. However, we shall explore the difficulties presented by local variables when using compensations.

7. Semantics

In this section, we will illustrate a method of giving the operational semantics for our compensation language. The semantics will take the form of reduction rules that model the behaviour of the language constructs. The syntax for Vella's language is summarized in Table 4.1.

For Vella's language, the meta-variables are as follows:

n, m will range over scope names, **Scope**

x will range over variables, **Var**

w will range over values, **Val**

s will range over stacks, **Stack**

e will range over expressions, **Exp**

c will range over commands, **Cmd**

Note that a value can either be a boolean b, an integer i , `null` or an exception `exc`. The stack will be used in all the reduction rules to hold all installed compensations. Lastly, the meta-variables can also be subscripted or primed. So, both s_1 and s' will also represent a state.

The semantics will take the form of reduction rules that model the behaviour of the commands and expressions of the language. Vella's language is a very simple

imperative programming language that operates on the state. The state consists of a number of locations used to store variables.

The semantics will be partitioned in two:

1. Expressions: the semantics for expressions will be given in terms of a big step semantics
2. Commands: the semantics for commands will be given in terms of a small-step semantics

While the semantics for expressions only inspect the state to determine the value of variables in an expression, the semantics for commands will change the state as well [2]. The program state will be used in all the reduction rules. Vella's language does not have a notion of local variables. Since all variables are declared globally, they can be used inside expressions even if they are not initialized. Therefore, in the domain of the initial state, all free variables contain the value `null`.

7.1 Evaluation Semantics for Expressions

The big-step semantics will take the form of a ternary relation between states, expressions and values.

$$\langle s,\ e\rangle \Downarrow v$$

Here, the expression e is evaluated in state s to the value v. In expressions, the state is only used to evaluate variables (rule EVAR). At the moment, there is no type system for Vella's language. Thus, expressions like `true + 5` are allowed. However, we are assuming that all expressions type check.

An alternative relation would be to include the final state on the right hand side of the relation

$$\langle s,\ e\rangle \Downarrow \langle s',\ v\rangle$$

However, expression evaluation in Vella's language have no side effects. Thus, s' is always identical to s. Since after expression evaluation, the state remains unchanged, a three tuple relation is enough.

The rules defining expressions are exaclty the same as the rules presented in Figure 2.1.

7.2 Reduction Semantics for Commands

When describing commands formally, we needed a more detailed description than a simple relation between the starting state and the final value. Therefore, we opted for a small step semantics. The program state will be needed in both the left hand side and the right hand side of the relation as certain commands update the state. Apart from commands and states, another element was needed in the relation to store compensating activities. We opted for a stack so as to be similar to the implementation.

Our small step semantics for commands will be phrased in terms of a transition system whose configuration is a tuple consisting of a compensation stack, the program state and a command.

$$\langle t,\ s,\ c\rangle$$

where c is the next command to be executed and s describes the current state of the program. The compensation stack, t is particularly needed in the rules regarding the compensating constructs since any installed compensating activities

are stored in a stack. The stack can either hold a string representing a scope name or a command.

$$t \in \textbf{Stack: Scope} \times \textbf{Cmd}$$

The transition relation will then describe how execution of commands takes place. For example:

$$\langle t, s, c \rangle \longrightarrow \langle t', s', c' \rangle$$

Unlike in a big-step semantics, execution of command c is not completed. In this rule, only a single step of execution is shown and the remaining computation of c is expressed by the intermediate configuration $\langle t', s', c' \rangle$. Note that every command eventually reduces down to `skip` in 0 or more reductions.

Vella's language contains the same constructs presented in L_1 together with additional constructs for error recovery. The reduction rules are exactly the same as the ones describing the behaviour of L_1, with the exception of a new element in the relation representing the compensation stack. Being the usual imperative language constructs, the stack remains unaffected in these reduction rules. The full set of reduction rules can be found in Appendix A.1.2.

7.2.1 Exception Handling

Four rules are present for dealing with exception handling. Firstly, we must execute the `try` block, one step from s. In this case, there are three possible outcomes:

- The execution of the `try` block terminates
- The execution of the `try` block has not yet completed, however we have reached an intermediate configuration
- The execution of the `try` block throws an exception

$$\frac{}{t, s, (\texttt{try skip catch } c) \longrightarrow t, s, \texttt{skip}}\ \textsc{rTry1}$$

$$\frac{t, s, c_1 \longrightarrow t', s', c_1'}{t, s, (\texttt{try } c_1 \texttt{ catch } c) \longrightarrow t', s', \texttt{try } c_1' \texttt{ catch } c}\ \textsc{rTry2}$$

The rule rTry1 formalises the first case. When the `try` block is terminated without throwing an exception, the `catch` block is ignored. Neither the program state nor the compensation stack are changed.

On the other hand, the second case is captured by the rule rTry2. In this case, if after executing the first step, the result is an intermediate configuration, $\langle t',\ s',\ c_1' \rangle$, then we must continue execution from this intermediate configuration. This is repeated until the whole `try` block reduces to `skip`.

$$\frac{}{t, s, (\texttt{try throw exc catch } c) \longrightarrow t, s, c}\ \textsc{rCatch}$$

$$\frac{}{t, s, (\texttt{throw } exc; c) \longrightarrow t, s, \texttt{throw exc}}\ \textsc{rThrow}$$

The last case is captured by the rule rCatch. If during execution of the `try` block, an exception is thrown, the execution of the `try` block must be aborted and we start executing the `catch` block instead.

Lastly, the rule rThrow states that when an exception is thrown, everything that is followed by it is ignored. Note that the command `throw` forces the program to terminate immediately, if it is not surrounded by a `try-catch` block.

7.2.2 Compensation Installation

A compensation is an activity that is associated with another activity, such that when said activity fails, the compensating activity is executed to repair the state of the program. The following rules describe the behaviour when a compensation is pushed on the stack.

$$\frac{}{t, s, (\texttt{undo skip with } c) \longrightarrow c :: t, s, \texttt{skip}} \text{ RUNDO1}$$

$$\frac{t, s, c_1 \longrightarrow t', s', c_1'}{t, s, (\texttt{undo } c_1 \texttt{ with } c_2) \longrightarrow t', s', \texttt{undo } c_1' \texttt{ with } c_2} \text{ RUNDO2}$$

Before installing a compensating activity c_2 which is associated with activity c_1, we must first execute successfully c_1. Since small-step semantics focuses on the intermediate steps of a computation, we first execute one step of c_1 from the starting state s. These rules are very similar to the rules capturing the behaviour of the sequence operator. After executing one single step, there are two possible outcomes:

- The activity c_1 has terminated successfully
- The activity c_1 has not yet completed, however we moved one step of execution

The rule RUNDO1 describes the first case. When c_1 terminates, the associated compensating activity must be pushed on the stack. Note that the program state remains unaffected.

The second case is captured by the rule RUNDO2. If c_1 has not yet terminated, we must first complete it before pushing c_2 on the stack. This shows that a compensation can only be installed once the activity it is associated with has terminated successfully.

7.2.3 Compensation Activation

The construct `compensate` n pops and executes all compensating activities present on the stack until scope name n is encountered. Note that any scope names found on the stack including n must not be removed. The following reduction rules describe formally the behaviour of the `compensate` construct.

$$\frac{}{n :: t, s, (\texttt{compensate}\ n) \longrightarrow n :: t, s, \texttt{skip}}\ \textsc{rComp1}$$

$$\frac{}{t, s, (\texttt{compensate}\ n) \longrightarrow t', s, c; \texttt{compensate}\ n}\ \textsc{rComp2}$$

where

$t' = \textbf{pop\_first\_c}\ t\ n$

$c = \textbf{get\_first\_c}\ t\ n$

Whenever n is found at the head of the stack, we must stop executing compensations. This case is captured by the rule rComp1. When this happens, we immediately reduce the command to `skip` to denote termination. Both the state and the stack are not affected.

When either a command is found at the top of the stack, we must first pop the command and execute it. Afterwards, we must continue compensating scope n. On the other hand, when a scope name that is not equal to n is found, we must ignore that scope name and continue compensating scope n. Two options were considered to describe this behaviour:

- One option was to include a temporary stack, where any scope names found on the stack while compensating, are pushed on it. Finally, when n is found,

the temporary stack and the compensation stack are merged

- The other option was to use two recursive functions that work on the compensation stack. The first function returns the first command in the stack, while the second function returns the compensation stack after removing the first command from it

We decided to use the second option as it was unfeasible to add an extra element to the relation, which will only be used for the `compensate` and `purge` constructs. We decided to make use of two recursive functions that work on the compensating stack:

- **get_first_c**: a function that takes a stack t, and a scope name n, and returns the first command that appears inside the stack.

  get_first_c:: **Stack** $\rightarrow$ **Scope** $\Rightarrow$ **Cmd**

  get_first_c (n:t) n = `skip`

  get_first_c (m:t) n = **get_first_c** t n

  get_first_c (c:t) _ = c

  If at the top of t, the scope name n is found, the function returns `skip`. This is the case when all compensating activities belonging to n were already executed. If at the head of the stack, a command c is found, c is returned. Finally, if a scope name is found which is not equivalent to n, it removes the head of the stack and performs the function **get_first_c** again on the remaining stack.

- **pop_first_c**: is a function that removes the first command in a stack and returns the remaining stack.

  pop_first_c:: **Stack** $\rightarrow$ **Scope** $\Rightarrow$ **Stack**

  pop_first_c (n:t) n = n:t

  pop_first_c (m:t) n = m:**pop_first_c** t n

  pop_first_c (c:t) n = t

This function is inputted the compensating stack and a scope name n. When n is found at the top of the stack, the whole stack is returned. This is the case when no compensating activities pertaining to that scope were found in the compensating stack.

When a scope name m, that is not equal to n, is found at the top of the stack, we must continue to search the stack for a command. Therefore we remove m from the stack and apply the function to the remaining stack. Note however that m must not be removed from the stack.

Lastly, when a command is found at the top of the stack, we pop the command and return the remaining stack.

7.2.4 Compensation Deletion

The construct `purge` n is similar to `compensate` n. However, it deletes all compensating activities belonging to scope n instead of executing them. Similar to RCOMP2, the rule RPURGE makes use of a recursive function **super_pop**.

$$\frac{}{t, s, (\texttt{purge}\ n) \longrightarrow t', s, \texttt{skip}}\ \text{RPURGE}$$

where

$t' =$ **super_pop** $t\ n$

The rule used to describe the behaviour of the compensate construct makes use of a recursive function **super_pop**:

super_pop:: **Stack** $\rightarrow$ **Scope** $\Rightarrow$ **Stack**

super_pop $(n{:}t)\ n = n{:}t$

super_pop $(m{:}t)\ n = m{:}$**super_pop** $t\ n$

super_pop $(c{:}t)\ n =$ **super_pop** $t\ n$

The function **super_pop** works very similar to the function **pop_first_c**. However, instead of popping the first command in a stack, it pops all the commands until scope name n is found.

7.2.5 Compensation Scope

The command `scope` n c is used to create a scope having scope name n, such that any compensating activities installed during execution of command c belong to said scope. n must be pushed on the stack when we enter the scope and popped out when we finish executing c.

$$\frac{}{t, s, \texttt{scope}\ \ n\ c \longrightarrow n :: t, s, c; \texttt{end}\ \ n}\ \textsc{rScope1}$$

$$\frac{}{t, s, \texttt{end}\ \ n \longrightarrow t', s, \texttt{skip}}\ \textsc{rScope2}$$

where

$$t' = \textbf{pop\_n}\ t\ n$$

To describe the behaviour of the `scope` construct, we needed a mechanism to determine when we reach the end of the scope. We considered two options:

- Include a reduction rule, where the contents of a scope is executed up until it reaches the command `skip`. When this happens, the scope name is then popped out of the stack
- We use a marker to denote the end of the scope

The first option behaves very similar to the rules describing the sequence operator. Firstly, we push scope name n on the stack and start executing the contents

of the scope. If after one step, c reduces to `skip`, we pop n out of the stack and proceed to execute the next command. On the other hand, if after one step of execution, we reach the intermediate configuration $\langle t', s', c' \rangle$, then the next configuration would be $\langle t', s', \texttt{scope}\ n\ c' \rangle$. With this approach, after every step of executing the contents of a scope, we would end up pushing the scope name on the stack.

The other option was to add a marker at the end of a scope. When we reach the marker, the scope name is popped out of the stack. With this method, we eliminated the problem of pushing the scope name on the stack multiple times while at the same time, the reduction rules are very easy to understand.

The rule RSCOPE1 deals with when we enter a new scope. In this case, the scope name must be pushed on the stack. Additionally, we must execute the contents of the scope. The command `end` n is added at the end of the scope to denote scope termination.

After the scope is successfully executed and we reach `end` n, the scope name must be popped from the stack. This is described in the rule RSCOPE2. A function pop_n was defined that given a stack, t and a scope name n, pops n out of t and returns the remaining stack.

pop_n:: **Stack** $\rightarrow$ **Scope** $\Rightarrow$ **Stack**

pop_n (n:xs) n = xs

pop_n (m:xs) n = m:**pop_n** xs n

pop_n (c:xs) n = c:**pop_n** xs n

7.3 Evaluation

In this section, we will use the example presented in Figure 4.1 and use the reduction rules to deduce the final output of the program. The output is then compared with the output of the language compiler.

$$\frac{}{\emptyset, [(\textbf{error} \rightarrow \texttt{true})], \texttt{scope}\ \textbf{transaction}\{\textbf{account} := 100; \ldots\} \longrightarrow} \textsc{rScope1}$$
$$[\textbf{transaction}], [(\textbf{error} \rightarrow \texttt{true})], \textbf{account} := 100; \ldots; \textbf{end transaction} \tag{7.1}$$

When we enter scope **transaction**, the scope name is pushed on the stack and we immediately proceed to execute the contents of the scope. Additionally, the marker **end transaction** is placed at the end of the scope to denote scope termination.

$$\frac{\dfrac{}{[(\textbf{error} \rightarrow \texttt{true})], \textbf{account} \Downarrow \texttt{null}}\ \textsc{eVar} \qquad \dfrac{}{[(\textbf{error} \rightarrow \texttt{true})], 100 \Downarrow 100}\ \textsc{eVal}}{[\textbf{transaction}], [(\textbf{error} \rightarrow \texttt{true})], \textbf{account} := 100; \ldots \longrightarrow} \textsc{rAss}$$
$$[\textbf{transaction}], [(\textbf{error} \rightarrow \texttt{true}), (\textbf{account} \rightarrow 100)], \texttt{skip}; \ldots \tag{7.2}$$

$$\frac{}{[\textbf{transaction}], [(\textbf{error} \rightarrow \texttt{true}), (\textbf{account} \rightarrow 100)], \texttt{skip}; \textbf{price} := 10 \longrightarrow} \textsc{rSeq1}$$
$$[\textbf{transaction}], [(\textbf{error} \rightarrow \texttt{true}), (\textbf{account} \rightarrow 100)], \textbf{price} := 10 \tag{7.3}$$

After executing the assignment statement, the program state is updated with the new values for variables **price** and **account**. From the next rule, let's assume that

$s = [(\textbf{error} \rightarrow \texttt{true}), (\textbf{account} \rightarrow 100)]$

$s_2 = [(\textbf{error} \rightarrow \texttt{true}), (\textbf{account} \rightarrow 100), (\textbf{price} \rightarrow 10)]$

$$\dfrac{\dfrac{}{s, \mathbf{price} \Downarrow \mathtt{null}}\ \textsc{eVar} \qquad \dfrac{}{s, 10 \Downarrow 10}\ \textsc{eVal}}{[\mathbf{transaction}], s, \mathbf{price} := 10; \ldots \longrightarrow [\mathbf{transaction}], s_2, \mathtt{skip}; \ldots}\ \textsc{rAss} \tag{7.4}$$

$$\dfrac{}{[\mathbf{transaction}], s_2, \mathtt{skip}; \mathtt{undo}\{\ldots\}\mathtt{with}\{\ldots\} \longrightarrow [\mathbf{transaction}], s_2, \mathtt{undo}\{\ldots\}\mathtt{with}\{\ldots\}}\ \textsc{rSeq1} \tag{7.5}$$

After executing the command **account := account - price**, the state will be updated to s_3 where

$$s_3 = [(\mathbf{error} \rightarrow \mathtt{true}), (\mathbf{account} \rightarrow 90), (\mathbf{price} \rightarrow 10)]$$

$$\dfrac{\dfrac{}{s_2, \mathbf{account} \Downarrow 100}\ \textsc{eVar} \qquad \dfrac{}{s_2, \mathbf{price} \Downarrow 10}\ \textsc{eVar}}{s_2, \mathbf{account - price} \Downarrow 90}\ \textsc{eSub} \tag{7.6}$$

$$\dfrac{\dfrac{}{s_2, \mathbf{account} \Downarrow 100}\ \textsc{eVar} \qquad \dfrac{}{s_2, \mathbf{account - price} \Downarrow 90}\ \text{subproof (7.6)}}{[\mathbf{transaction}], s_2, \mathbf{account := account - price} \longrightarrow [\mathbf{transaction}], s_3, \mathtt{skip}}\ \textsc{rAss}$$
$$\dfrac{}{[\mathbf{transaction}], s_2, \mathtt{undo}\{\mathbf{account := account - price}\}\ \mathtt{with}\{\ldots\} \longrightarrow [\mathbf{transaction}], s_3, \mathtt{undo}\{\mathtt{skip}\}\ \mathtt{with}\{\ldots\}}\ \textsc{rUndo2} \tag{7.7}$$

When we reach the `undo-with` command, we must first execute the contents of the undo block before pushing the compensation of the stack.

$$\dfrac{}{[\mathbf{transaction}], s_3, \mathtt{undo}\{\mathtt{skip}\}\ \mathtt{with}\{\mathbf{account := account + price}\} \longrightarrow [c :: \mathbf{transaction}], s_3, \mathtt{skip}; \mathtt{if}(\mathbf{error == true})\ \mathtt{then}\ \{\ldots\}}\ \textsc{rUndo1} \tag{7.8}$$

As soon as the contents of the undo block is evaluated to `skip`, the compensation can be pushed on the stack. From the sub-proof (7.7) onwards, let's assume that

c = **(account := account + price)**

$$\frac{}{[c :: \textbf{transaction}], s_3, \texttt{skip}; \texttt{if}(\textbf{error} == \textbf{true})\ \texttt{then}\ \{\ldots\} \longrightarrow [c :: \textbf{transaction}], s_3, \texttt{if}(\textbf{error} == \textbf{true})\ \texttt{then}\ \{\ldots\}}\ \textsc{rSeq1} \tag{7.9}$$

$$\frac{\dfrac{}{s_3, \textbf{error} \Downarrow \texttt{true}}\ \textsc{eVar}}{[c :: \textbf{transaction}], s_3, \texttt{if}(\textbf{error} == \textbf{true})\ \texttt{then}\{\texttt{compensate}\ \textbf{transaction}\} \longrightarrow [c :: \textbf{transaction}], s_3, \texttt{compensate}\ \textbf{transaction}}\ \textsc{rThen} \tag{7.10}$$

$$\frac{}{[c :: \textbf{transaction}], s_3, \texttt{compensate}\ \textbf{transaction} \longrightarrow [\textbf{transaction}], s_3, c; \texttt{compensate}\ \textbf{transaction}}\ \textsc{rComp2} \tag{7.11}$$

When we reach the `compensate` construct, we must first pop the first command present on the stack and start executing it. From this sub-proof onwards,

s_4 = **[(error** $\rightarrow$ `true`**), (account** $\rightarrow$ 100**), (price** $\rightarrow$ 10**)]**

$$\frac{\dfrac{}{s_3, \textbf{account} \Downarrow 90}\ \textsc{eVar} \quad \dfrac{}{s_3, \textbf{price} \Downarrow 10}\ \textsc{eVal}}{s_3, \textbf{account} + \textbf{price} \Downarrow 100}\ \textsc{eAdd} \tag{7.12}$$

$$\dfrac{\dfrac{}{s_3, \mathbf{account} \Downarrow 90}\ \textsc{eVar} \qquad \dfrac{}{s_3, \mathbf{account} + \mathbf{price} \Downarrow 100}\ \text{sub-proof (7.12)}}{[\mathbf{transaction}], s_3, \mathbf{account} := \mathbf{account} + \mathbf{price} \longrightarrow [\mathbf{transaction}], s_4, \mathtt{skip}}\ \textsc{rAss}$$

$$\dfrac{}{[\mathbf{transaction}], s_3, \mathbf{account} := \mathbf{account} + \mathbf{price}; \mathtt{compensate}\ \mathbf{transaction} \longrightarrow [\mathbf{transaction}], s_4, \mathtt{skip}; \mathtt{compensate}\ \mathbf{transaction}}\ \textsc{rSeq2} \tag{7.13}$$

$$\dfrac{}{[\mathbf{transaction}], s_4, \mathtt{skip}; \mathtt{compensate}\ \mathbf{transaction} \longrightarrow [\mathbf{transaction}], s_4, \mathtt{compensate}\ \mathbf{transaction}}\ \textsc{rSeq1} \tag{7.14}$$

$$\dfrac{}{[\mathbf{transaction}], s_4, \mathtt{compensate}\ \mathbf{transaction}; \mathtt{end}\ \mathbf{transaction} \longrightarrow [\mathbf{transaction}], s_4, \mathtt{skip}; \mathtt{end}\ \mathbf{transaction}}\ \textsc{rComp1} \tag{7.15}$$

In sub-proof (7.15), the scope name **transaction** is found at the top of the stack. Moreover, we need to compensate up until the scope name **transaction** is present at the head of the stack. Therefore, we need to stop executing compensations and proceed to the next command.

$$\dfrac{}{[\mathbf{transaction}], s_4, \mathtt{skip}; \mathtt{end}\ \mathbf{transaction} \longrightarrow [\mathbf{transaction}], s_4, \mathtt{end}\ \mathbf{transaction}}\ \textsc{rSeq1} \tag{7.16}$$

$$\dfrac{}{[\mathbf{transaction}], s_4, \mathtt{end}\ \mathbf{transaction} \longrightarrow [\mathbf{transaction}], s_4, \mathtt{skip}}\ \textsc{rScope2} \tag{7.17}$$

When the whole program is reduced down to `skip`, we can examine the final state of the program. Note that the final state s_4 is equal to s_2 which was the state at which the program was before executing the command **account := account - price**. Therefore, by executing the compensations present in scope **transaction**, we managed to cancel the command **account := account - price**. The final state produced by the language compiler is shown in Figure 7.1

```
{
    scope transaction
    {
        account := 100;
        price := 10;
        error := true;

        undo {
            account := account - price;
        }
        with {
            account := account + price;
        }

        if(error == true)
            compensate transaction;
    };
}
```

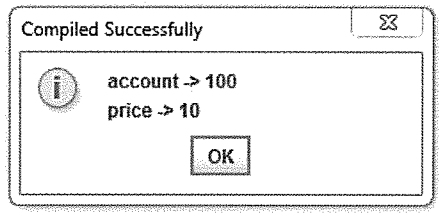


Figure 7.1: Output generated by Vella's Compiler

The current version of semantics conforms to the textual description of the language. The output generated by the reduction rules is the same as the output produced by the language compiler.

8. Variable Handling

Since compensating activities are installed and executed at different times in a program, variable handling might be a fairly complicated task. Consider the following programs:

```
1    ...                                    ...
2    scope S {                              scope S {
3        account := 100;                        account := 100;
4        price := 10;                           price := 10;
5                                                fee := 1;
6        undo {                                 undo {
7            account := account - price;            account := account - price;
8        }                                      }
9        with {                                 with {
10           account := account + price;            account := account + price;
11       }                                          account := account - fee;
12                                               }
13       price := 20;                           price := 20;
14                                              fee := 2;
15       compensate S;                          compensate S;
16   }                                      }
17   ...                                    ...
```

In the first example, the compensating activity **account := account + price** is associated with the activity **account := account - price** such that if an error

occurs in the program, the action can be reversed. We will use the current version of semantics to derive the final state of the program. The first part of the proof is exactly the same as the one presented in the evaluation section of chapter 7 (sub-proofs 7.1-7.9). The difference occurs in line 13, when the variable **price** is updated. Note that the state s represents the state that the program was before executing the command **account := account - price** while state s_2 represents the program state after the compensation c, is pushed on the stack.

$s = [(\textbf{account} \rightarrow 100), (\textbf{price} \rightarrow 10)]$

$s_2 = [(\textbf{account} \rightarrow 90), (\textbf{price} \rightarrow 10)]$

$c =$ **account := account + price;**

$$\dfrac{\dfrac{}{s_2, \textbf{price} \Downarrow 10}\ \textsc{eVar} \qquad \dfrac{}{s, 20 \Downarrow 20}\ \textsc{eVal}}{[c :: \textbf{transaction}], s_2, \textbf{price} := 20; \ldots \longrightarrow [c :: \textbf{transaction}], s_3, \texttt{skip}; \ldots}\ \textsc{rAss} \tag{8.1}$$

After executing the assignment statement, the state is updated to s_3 where

$s_3 = [(\textbf{account} \rightarrow 90), (\textbf{price} \rightarrow 20)]$

$$\dfrac{}{[c :: \textbf{transaction}], s_3, \texttt{skip;compensate}\ \textbf{transaction} \longrightarrow [c :: \textbf{transaction}], s_3, \texttt{compensate}\ \textbf{transaction}}\ \textsc{rSeq1} \tag{8.2}$$

$$\dfrac{}{[c :: \textbf{transaction}], s_3, \texttt{compensate}\ \textbf{transaction} \longrightarrow [\textbf{transaction}], s_3, c; \texttt{compensate}\ \textbf{transaction}}\ \textsc{rComp2} \tag{8.3}$$

When we reach the command `compensate` **transaction**, we are required to reverse the whole transaction. The first compensation found on the stack is popped and

executed.

$$\dfrac{\dfrac{}{s_3, \textbf{account} \Downarrow 90}\ \textsc{eVar} \qquad \dfrac{}{s_3, \textbf{price} \Downarrow 20}\ \textsc{eVal}}{s_3, \textbf{account} + \textbf{price} \Downarrow 110}\ \textsc{eAdd} \tag{8.4}$$

$$\dfrac{\dfrac{}{s_3, \textbf{account} \Downarrow 90}\ \textsc{eVar} \qquad \dfrac{}{s_3, \textbf{account} + \textbf{price} \Downarrow 110}\ \text{sub-proof (8.4)}}{[\textbf{transaction}], s_3, \textbf{account} := \textbf{account} + \textbf{price} \longrightarrow [\textbf{transaction}], s_4, \texttt{skip}}\ \textsc{rAss}$$

$$\dfrac{}{[\textbf{transaction}], s_3, \textbf{account} := \textbf{account} + \textbf{price}; \texttt{compensate}\ \textbf{transaction} \longrightarrow [\textbf{transaction}], s_4, \texttt{skip}; \texttt{compensate}\ \textbf{transaction}}\ \textsc{rSeq2} \tag{8.5}$$

After executing the compensation, the state is updated to s_4 where

$$s_4 = [(\textbf{account} \rightarrow 110), (\textbf{price} \rightarrow 20)]$$

$$\dfrac{}{[\textbf{transaction}], s_4, \texttt{skip}; \texttt{compensate}\ \textbf{transaction} \longrightarrow [\textbf{transaction}], s_4, \texttt{compensate}\ \textbf{transaction}}\ \textsc{rSeq1} \tag{8.6}$$

$$\dfrac{}{[\textbf{transaction}], s_4, \texttt{compensate}\ \textbf{transaction}; \texttt{end}\ \textbf{transaction} \longrightarrow [\textbf{transaction}], s_4, \texttt{skip}; \texttt{end}\ \textbf{transaction}}\ \textsc{rComp1} \tag{8.7}$$

$$\dfrac{}{[\textbf{transaction}], s_4, \texttt{skip}; \texttt{end}\ \textbf{transaction} \longrightarrow [\textbf{transaction}], s_4, \texttt{end}\ \textbf{transaction}}\ \textsc{rSeq1} \tag{8.8}$$

$$\frac{}{[\textbf{transaction}], s_4, \texttt{end}\ \textbf{transaction} \longrightarrow [\textbf{transaction}], s_4, \texttt{skip}}\ \textsc{rSeq1} \quad (8.9)$$

When we compare the program state before executing the `undo-with` command s, and the program state after executing the compensation s_4, we can notice that the value held by the variable **account** is different. This is because we executed the compensation with the new value of **price**. Ideally, the value of **price** when executing the compensating activity should be the same as its respective value at the time of installation of said compensating activity. However, with our current semantics, the user will end up being refunded with the newly updated price.

The second program behaves very similar to the first one. However when refunding the user's account, an additional fee is charged to the user's account. We will use the existing reduction rules to derive the output of the second program. We will skip the first part of the proof as it is very similar to the one shown in the Evaluation Section of Chapter 7 (sub-proofs 7.1 - 7.9). We will use s to denote the state at which the program was before executing command **account := account - price**, s_2 to denote the state after pushing the compensation on the stack and c to denote the compensation

$s = [(\textbf{account} \rightarrow 100), (\textbf{price} \rightarrow 10), (\textbf{fee} \rightarrow 1)]$

$s_2 = [(\textbf{account} \rightarrow 90), (\textbf{price} \rightarrow 10), (\textbf{fee} \rightarrow 1)]$

$c =$ **account := account + price; account := account - fee;**

$$\frac{\dfrac{}{s_2, \textbf{price} \Downarrow 10}\ \textsc{eVar} \qquad \dfrac{}{s_2, 20 \Downarrow 20}\ \textsc{eVal}}{[c :: \textbf{transaction}], s_2, \textbf{price} := 20; \ldots \longrightarrow [c :: \textbf{transaction}], s_3, \texttt{skip}; \ldots}\ \textsc{rAss} \quad (8.10)$$

After executing the assignment statement, the program state is updated to s_3 where

$s_3 = [(\textbf{account} \rightarrow 90), (\textbf{price} \rightarrow 20), (\textbf{fee} \rightarrow 1)]$

$$\frac{}{[c :: \textbf{transaction}], s_3, \texttt{skip};\textbf{fee} := 2 \longrightarrow [c :: \textbf{transaction}], s_3, \textbf{fee} := 2}\ \textsc{rSeq1} \tag{8.11}$$

$$\frac{\dfrac{}{s_3, \textbf{fee} \Downarrow 1}\ \textsc{eVar} \qquad \dfrac{}{s_3, 2 \Downarrow 2}\ \textsc{eVal}}{[c :: \textbf{transaction}], s_3, \textbf{fee} := 2; \ldots \longrightarrow [c :: \textbf{transaction}], s_4, \texttt{skip}; \ldots}\ \textsc{rAss} \tag{8.12}$$

Similarly, after executing the assignment statement, the program state is updated to s_4 where

$$s_4 = [(\textbf{account} \rightarrow 90), (\textbf{price} \rightarrow 20), (\textbf{fee} \rightarrow 2)]$$

$$\frac{}{\begin{array}{c}[c :: \textbf{transaction}], s_4, \texttt{skip};\texttt{compensate}\ \textbf{transaction} \longrightarrow \\ [c :: \textbf{transaction}], s_4, \texttt{compensate}\ \textbf{transaction}\end{array}}\ \textsc{rSeq1} \tag{8.13}$$

$$\frac{}{\begin{array}{c}[c :: \textbf{transaction}], s_4, \texttt{compensate}\ \textbf{transaction} \longrightarrow \\ [\textbf{transaction}], s_4, c;\texttt{compensate}\ \textbf{transaction}\end{array}}\ \textsc{rComp2} \tag{8.14}$$

$$\frac{\dfrac{}{s_4, \textbf{account} \Downarrow 90}\ \textsc{eVar} \qquad \dfrac{}{s_4, \textbf{price} \Downarrow 20}\ \textsc{eVar}}{s_4, \textbf{account} + \textbf{price} \Downarrow 110}\ \textsc{eAdd} \tag{8.15}$$

$$\dfrac{\dfrac{}{s_4, \mathbf{account} \Downarrow 90}\ \textsc{eVar} \qquad \dfrac{}{s_4, \mathbf{account} + \mathbf{price} \Downarrow 110}\ \text{sub-proof (8.15)}}{[\mathbf{transaction}], s_4, \mathbf{account} := \mathbf{account} + \mathbf{price} \longrightarrow [\mathbf{transaction}], s_5, \mathtt{skip}}\ \textsc{rAss}$$

$$\dfrac{}{[\mathbf{transaction}], s_4, \mathbf{account} := \mathbf{account} + \mathbf{price}; \mathtt{compensate}\ \mathbf{transaction} \longrightarrow [\mathbf{transaction}], s_5, \mathtt{skip}; \mathtt{compensate}\ \mathbf{transaction}}\ \textsc{rSeq2} \tag{8.16}$$

After executing the first part of the compensation, the state is updated to s_5 where

$s_5 = [(\mathbf{account} \rightarrow 110), (\mathbf{price} \rightarrow 20), (\mathbf{fee} \rightarrow 2)]$

$$\dfrac{}{[\mathbf{transaction}], s_5, \mathtt{skip}; \mathbf{fee} := 2; \mathtt{compensate}\ \mathbf{transaction} \longrightarrow [\mathbf{transaction}], s_5, \mathbf{fee} := 2; \mathtt{compensate}\ \mathbf{transaction}}\ \textsc{rSeq1} \tag{8.17}$$

$$\dfrac{\dfrac{}{s_5, \mathbf{account} \Downarrow 110}\ \textsc{eVar} \qquad \dfrac{}{s_5, \mathbf{fee} \Downarrow 2}\ \textsc{eVar}}{s_5, \mathbf{account} - \mathbf{fee} \Downarrow 108}\ \textsc{eSub} \tag{8.18}$$

$$\dfrac{\dfrac{}{s_5, \mathbf{account} \Downarrow 110}\ \textsc{eVar} \qquad \dfrac{}{s_5, \mathbf{account} - \mathbf{fee} \Downarrow 108}\ \text{sub-proof (8.18)}}{[\mathbf{transaction}], s_5, \mathbf{account} := \mathbf{account} - \mathbf{fee} \longrightarrow [\mathbf{transaction}], s_6, \mathtt{skip}}\ \textsc{rAss}$$

$$\dfrac{}{[\mathbf{transaction}], s_5, \mathbf{account} := \mathbf{account} - \mathbf{fee}; \mathtt{compensate}\ \mathbf{transaction} \longrightarrow [\mathbf{transaction}], s, \mathtt{skip}; \mathtt{compensate}\ \mathbf{transaction}}\ \textsc{rSeq2} \tag{8.19}$$

After executing the last part of the compensation, the state is updated to s_6 where

$s_6 = [(\mathbf{account} \Downarrow 108), (\mathbf{price} \Downarrow 20), (\mathbf{fee} \Downarrow 2)]$

$$\frac{}{[\textbf{transaction}], s_6, \texttt{skip;compensate}\ \textbf{transaction} \longrightarrow [\textbf{transaction}], s_6, \texttt{compensate}\ \textbf{transaction}}\ \text{RSEQ1} \quad (8.20)$$

The final part of the proof is exactly as sub-proofs 8.7 - 8.9. If we compare s, which is the state at which the program was before executing the command **account := account - price**, and the final state s_6, we can note that the variable **account** holds a different value than what was expected.

Like the previous example, when compensating the scope **transaction**, we want to refund the user with the same amount that he has paid. However, we want to charge the user with the latest value of **fee**.

With the following examples, we showed that both the current state and the state at compensation installation are needed to compensate correctly an activity. At the moment, our semantics only allows us to execute compensations with respect to the current program state.

A solution to this problem is to save a copy of the state at which the program was when the compensation was being installed. Both the compensating activity together with the state are pushed on the stack. We also introduce a new element, $s^{\leftarrow}$ in the relation, representing the state at which the program was when the currently executing compensation was installed. Thus, before executing a compensation, we will update $s^{\leftarrow}$ with the value found on the stack. To support this change, the stack will be updated as follows:

t	$\in$	**Stack**
	$::=$	$\epsilon \mid (c,s)::t \mid n::t$

Table 8.1: Syntax of the Compensation Stack

The stack can either hold a scope name or a tuple containing a compensation and its respective program state at compensation installation. Note that in this version of semantics, we are storing two separate memories in the relation. Thus, when using a variable, we can access the current value or the value held in state $s^{\leftarrow}$. The semantics of expressions are updated as follows:

e	$\in$	**Exp**
	$::=$	$n \mid v \mid v.\texttt{back} \mid \texttt{not}\ e \mid e\ \texttt{and}\ e \mid e\ \texttt{+}\ e \mid e\ \texttt{-}\ e \mid ...$

Table 8.2: Syntax of Expressions

With the expressions $v.\texttt{back}$ and v, the user has the option to choose explicitly whether to use either the current value of a variable or else the value of that same variable held in $s^{\leftarrow}$.

The evaluation semantics for expressions will remain exactly the same with the exception of a new element $s^{\leftarrow}$ representing the state at which the program was when the currently executing compensation was installed. The current state will be represented by $s^{\rightarrow}$. Initially, $s^{\leftarrow}$ will be identical to $s^{\rightarrow}$ i.e. it will contain all the free variables pointing to the value `null`.

The only major difference in the evaluation semantics is:

1. The inclusion of a new rule EVARB that evaluates a variable with respect to $s^{\leftarrow}$
2. Rule EVAR is renamed to EVARF. It evaluates a variable with respect to the current state $s^{\rightarrow}$

$$\frac{}{s^{\leftarrow}, s^{\rightarrow}, v \Downarrow s^{\rightarrow}(v)}\ \textsc{eVarF} \qquad \frac{}{s^{\leftarrow}, s^{\rightarrow}, v.\texttt{back} \Downarrow s^{\leftarrow}(v)}\ \textsc{eVarB}$$

Figure 8.1: Big-Step Semantics for Variable Evaluation

Similarly, most of the semantic rules describing the behaviour of commands remain the same with the exception of a new element, $s^{\leftarrow}$ in the relation. Only, the reduction rules regarding the compensating constructs need to be changed. The full set of reduction rules can be found in Appendix A.2.

$$\frac{}{t, s^{\leftarrow}, s^{\rightarrow}, (\texttt{undo skip with } c) \longrightarrow (c, s^{\rightarrow}) :: t, s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}}\ \textsc{rUndo1}$$

$$\frac{t, s^{\leftarrow}, s^{\rightarrow}, c_1 \longrightarrow t', s^{\leftarrow\prime}, s^{\rightarrow\prime}, c_1'}{t, s^{\leftarrow}, s^{\rightarrow}, (\texttt{undo } c_1 \texttt{ with } c_2) \longrightarrow t', s^{\leftarrow\prime}, s^{\rightarrow\prime}, \texttt{undo } c_1' \texttt{ with } c_2}\ \textsc{rUndo2}$$

When the undo block is reduced down to skip the compensation c must be pushed on the stack together with the current state $s^{\rightarrow}$.

$$\frac{}{t, s^{\leftarrow}, s^{\rightarrow}, (\texttt{purge } n) \longrightarrow t', s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}}\ \textsc{rPurge}$$

where

$t' = \text{super\_pop } t\ n$

$$\frac{}{t, s^{\leftarrow}, s^{\rightarrow}, \texttt{scope}\ n\ c \longrightarrow n :: t, s^{\leftarrow}, s^{\rightarrow}, c; \texttt{end}\ n}\ \textsc{rScope1}$$

$$\frac{}{t, s^{\leftarrow}, s^{\rightarrow}, \texttt{end}\ n \longrightarrow t', s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}}\ \textsc{rScope2}$$

where

$t' = \textbf{pop\_n}\ t\ n$

$$\frac{}{n :: t, s^{\leftarrow}, s^{\rightarrow}, (\texttt{compensate}\ n) \longrightarrow n :: t, s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}}\ \textsc{rComp1}$$

$$\frac{}{t, s^{\leftarrow}, s^{\rightarrow}, (\texttt{compensate}\ n) \longrightarrow t', s', s^{\rightarrow}, c; \texttt{compensate}\ n}\ \textsc{rComp2}$$

where

$t' = \textbf{pop\_first\_c}\ t\ n$

$s' = \textbf{snd}\ (\textbf{get\_first\_c}\ t\ n)$

$c = \textbf{fst}\ (\textbf{get\_first\_c}\ t\ n)$

Since we are saving both the state and the compensation inside the compensating stack, the function **get_first_c** now returns a pair instead of just one element. Two functions were needed to get the first and second items of the pair returned. The function **fst** returns the first item of the pair which corresponds to the compensation part. On the other hand, the function **snd** returns the second item of the pair, which is the state.

The newly added element in the relation, $s^{\leftarrow}$, will only be used when executing the `compensate` construct. When executing a compensating activity, $s^{\leftarrow}$ must be updated with the state at which the program was when the compensation was installed. This state is saved on the compensation stack together with the compensating activity. Therefore, before we start executing a compensation, we must first update the value of $s^{\leftarrow}$ with the state present on the compensation stack.

8.1 Evaluation

The issue of variable handling inside compensations is quite complicated. Since compensation installation and activation happens at different times in a program, it is unclear which value of a variable is to be used to successfully compensate a sub-program. In this chapter we have presented a solution to overcome the shortcomings presented with variable handling. We will now use the new reduction rules to derive the final state of the second program presented in the beginning of the chapter. Since, we introduced a new expression v.`back`, the program has to be updated as follows

```
1  scope S {
2        account := 100;
3        price := 10;
4        fee := 1;
5        undo {
6            account := account - price;
7        }
8        with {
9            account := account + price.back;
10           account := account - fee;
11       }
12       price := 20;
13       fee := 2;
14       compensate S;
15 }
```

Initially $s^{\leftarrow}$ and $s^{\rightarrow}$ contain the free variables pointing to the value **null**.

$$\frac{}{\emptyset, s^{\leftarrow}, s^{\rightarrow}, \texttt{scope}\ \textbf{transaction}\{\textbf{account} := 100; \ldots\} \longrightarrow [\textbf{transaction}], s^{\leftarrow}, s^{\rightarrow}, \textbf{account} := 100; \ldots; \texttt{end}\ \textbf{transaction}}\ \textsc{rScope1} \tag{8.21}$$

$$\frac{\dfrac{}{s^{\leftarrow}, s^{\rightarrow}, \textbf{account} \Downarrow \texttt{null}}\ \textsc{eVarF} \qquad \dfrac{}{s^{\leftarrow}, s^{\rightarrow}, 100 \Downarrow 100}\ \textsc{eVal}}{[\textbf{transaction}], s^{\leftarrow}, s^{\rightarrow}, \textbf{account} := 100; \ldots \longrightarrow [\textbf{transaction}], s^{\leftarrow}, [(\textbf{account} \rightarrow 100)], \texttt{skip}; \ldots}\ \textsc{rAss} \tag{8.22}$$

$$\frac{}{[\textbf{transaction}], s^{\leftarrow}, [(\textbf{account} \rightarrow 100)], \texttt{skip}; \textbf{price} := 10 \longrightarrow [\textbf{transaction}], s^{\leftarrow}, [(\textbf{account} \rightarrow 100)], \textbf{price} := 10}\ \textsc{rSeq1} \tag{8.23}$$

After executing the assignment statement, the program state is updated with the new values for the variables **price**, **fee** and **account**. From the next rule, lets assume that

$s = [(\textbf{account} \rightarrow 100)]$

$s_1 = [(\textbf{account} \rightarrow 100), (\textbf{price} \rightarrow 10)]$

$s_2 = [(\textbf{account} \rightarrow 100), (\textbf{price} \rightarrow 10), (\textbf{fee} \rightarrow 1)]$

$$\frac{\dfrac{}{s^{\leftarrow}, s, \textbf{price} \Downarrow \texttt{null}}\ \textsc{eVarF} \qquad \dfrac{}{s^{\leftarrow}, s, 10 \Downarrow 10}\ \textsc{eVal}}{[\textbf{transaction}], s^{\leftarrow}, s, \textbf{price} := 10; \ldots \longrightarrow [\textbf{transaction}], s^{\leftarrow}, s_1, \texttt{skip}; \ldots}\ \textsc{rAss} \tag{8.24}$$

$$\frac{}{[\textbf{transaction}], s^{\leftarrow}, s_1, \texttt{skip;fee} := 1 \longrightarrow [\textbf{transaction}], s^{\leftarrow}, s_1, \textbf{fee} := 1}\ \textsc{rSeq1} \tag{8.25}$$

$$\frac{\dfrac{}{s^{\leftarrow}, s_1, \textbf{fee} \Downarrow \texttt{null}}\ \textsc{eVarF} \qquad \dfrac{}{s^{\leftarrow}, s_1, 1 \Downarrow 1}\ \textsc{eVal}}{[\textbf{transaction}], s^{\leftarrow}, s_1, \textbf{fee} := 1; \ldots \longrightarrow [\textbf{transaction}], s^{\leftarrow}, s_2, \texttt{skip}; \ldots}\ \textsc{rAss} \tag{8.26}$$

$$\frac{}{[\textbf{transaction}], s^{\leftarrow}, s_2, \texttt{skip;undo}\{\ldots\}\texttt{with}\{\ldots\} \longrightarrow [\textbf{transaction}], s^{\leftarrow}, s_2, \texttt{undo}\{\ldots\}\texttt{with}\{\ldots\}}\ \textsc{rSeq1} \tag{8.27}$$

After executing the command **account := account - price**, the state will be updated to s_3 where

$s_3 = [(\textbf{account} \rightarrow 90), (\textbf{price} \rightarrow 10), (\textbf{fee} \rightarrow 1)]$

$$\frac{\dfrac{}{s^{\leftarrow}, s_2, \textbf{account} \Downarrow 100}\ \textsc{eVarF} \qquad \dfrac{}{s^{\leftarrow}, s_2, \textbf{price} \Downarrow 10}\ \textsc{eVarF}}{s^{\leftarrow}, s_2, \textbf{account - price} \Downarrow 90}\ \textsc{eSub} \tag{8.28}$$

$$\frac{\dfrac{}{s^{\leftarrow}, s_2, \textbf{account} \Downarrow 100}\ \textsc{eVar} \qquad \dfrac{}{s^{\leftarrow}, s_2, \textbf{account} - \textbf{price} \Downarrow 90}\ \text{subproof (8.28)}}{\begin{array}{l}[\textbf{transaction}], s^{\leftarrow}, s_2, \textbf{account := account - price} \\ \qquad \longrightarrow [\textbf{transaction}], s^{\leftarrow}, s_3, \texttt{skip}\end{array}}\ \textsc{rAss}}{\begin{array}{l}[\textbf{transaction}], s^{\leftarrow}, s_2, \textbf{undo\{account := account - price\}}\ \texttt{with}\{\ldots\} \\ \qquad \longrightarrow [\textbf{transaction}], s^{\leftarrow}, s_3, \texttt{undo\{skip\}}\ \texttt{with}\{\ldots\}\end{array}}\ \textsc{rUndo2} \tag{8.29}$$

$$\frac{}{[\textbf{transaction}], s_3, \texttt{undo\{skip\}}\ \texttt{with\{}\textbf{account} := \textbf{account} + \textbf{price}\texttt{\}} \longrightarrow [c :: \textbf{transaction}], s_3, \texttt{skip;}\textbf{price} := 20}\ \textsc{rUndo1} \tag{8.30}$$

As soon as the contents of the undo block evaluates to skip, the compensation together with the program state must be pushed on the stack. Lets assume that

$c = ((\textbf{account} := \textbf{account} + \textbf{price}; \textbf{account} := \textbf{account} - \textbf{fee};),$
$[(\textbf{account} \rightarrow 90), (\textbf{price} \rightarrow 10)])$

$$\frac{\dfrac{}{s^{\leftarrow}, s_3, \textbf{price} \Downarrow 10}\ \textsc{eVar} \qquad \dfrac{}{s^{\leftarrow}, s_3, 20 \Downarrow 20}\ \textsc{eVal}}{[c :: \textbf{transaction}], s^{\leftarrow}, s_3, \textbf{price} := 20 \longrightarrow [c :: \textbf{transaction}], s^{\leftarrow}, s_4, \texttt{skip}}\ \textsc{rAss} \tag{8.31}$$

After executing the assignment statement, the program state is updated to s_4 where

$s_4 = [(\textbf{account} \rightarrow 90), (\textbf{price} \rightarrow 20), (\textbf{fee} \rightarrow 1)]$

$$\frac{}{[c :: \textbf{transaction}], s^{\leftarrow}, s_4, \texttt{skip;}\textbf{fee} := 2 \longrightarrow [c :: \textbf{transaction}], s^{\leftarrow}, s_4, \textbf{fee} := 2}\ \textsc{rSeq1} \tag{8.32}$$

$$\frac{\dfrac{}{s^{\leftarrow}, s_4, \textbf{fee} \Downarrow 1}\ \textsc{eVar} \qquad \dfrac{}{s^{\leftarrow}, s_4, 2 \Downarrow 2}\ \textsc{eVal}}{[c :: \textbf{transaction}], s^{\leftarrow}, s_4, \textbf{fee} := 2 \longrightarrow [c :: \textbf{transaction}], s^{\leftarrow}, s_5, \texttt{skip}}\ \textsc{rAss} \tag{8.33}$$

Similarly, after executing the assignment statement, the program state is updated to s_5 where

$s_5 = [(\textbf{account} \rightarrow 90), (\textbf{price} \rightarrow 20), (\textbf{fee} \rightarrow 2)]$

$$\frac{}{[c :: \textbf{transaction}], s^{\leftarrow}, s_5, \texttt{skip}; \texttt{compensate}\ \textbf{transaction} \longrightarrow [c :: \textbf{transaction}], s^{\leftarrow}, s_5, \texttt{compensate}\ \textbf{transaction}}\ \textsc{rSeq1} \quad (8.34)$$

$$\frac{}{[c :: \textbf{transaction}], s^{\leftarrow}, s_5, \texttt{compensate}\ \textbf{transaction} \longrightarrow [\textbf{transaction}], s', s_5, c'; \texttt{compensate}\ \textbf{transaction}}\ \textsc{rComp2} \quad (8.35)$$

We know that c is a tuple containing a compensation together with the state at compensation installation. Therefore, we must take the first element c' to denote the compensation, and the second element s' to denote the state where

$c' = \textbf{account} := \textbf{account} + \textbf{price}.\texttt{back}; \textbf{account} := \textbf{account} - \textbf{fee}$

$s' = [(\textbf{account} \rightarrow 90), (\textbf{price} \rightarrow 10), (\textbf{fee} \rightarrow 1)]$

$$\frac{\dfrac{}{s', s_5, \textbf{account} \Downarrow 90}\ \textsc{eVarF} \qquad \dfrac{}{s', s_5, \textbf{price}.\texttt{back} \Downarrow 10}\ \textsc{eVarB}}{s', s_5, \textbf{account} + \textbf{price}.\texttt{back} \Downarrow 100}\ \textsc{eAdd} \quad (8.36)$$

In the compensation, it is explicitly stated that we must use the value held by the variable **price** at compensation installation. Therefore we must use state s' to get the value of **price**.

$$\frac{\dfrac{}{s', s_5, \textbf{account} \Downarrow 90}\ \textsc{eVarF} \qquad \dfrac{}{s', s_5, \textbf{account} + \textbf{price}.\texttt{back} \Downarrow 100}\ \text{sub-proof (8.36)}}{[\textbf{transaction}], s', s_5, \textbf{account} := \textbf{account} + \textbf{price}.\texttt{back} \longrightarrow [\textbf{transaction}], s_6, \texttt{skip}}\ \textsc{rAss}$$

$$\frac{}{[\textbf{transaction}], s', s_5, \textbf{account} := \textbf{account} + \textbf{price}.\texttt{back}; \texttt{compensate}\ \textbf{transaction} \longrightarrow [\textbf{transaction}], s', s_6, \texttt{skip}; \texttt{compensate}\ \textbf{transaction}}\ \textsc{rSeq2} \quad (8.37)$$

After executing the first part of the compensation, the state is updated to s_6 where

$s_6 = [(\textbf{account} \rightarrow 100), (\textbf{price} \rightarrow 20), (\textbf{fee} \rightarrow 2)]$

$$\frac{}{[\textbf{transaction}], s', s_6, \texttt{skip};\textbf{account} := \textbf{account} - \textbf{fee} \longrightarrow [\textbf{transaction}], s', s_6, \textbf{account} := \textbf{account} - \textbf{fee}}\ \textsc{rSeq1} \tag{8.38}$$

$$\frac{\dfrac{}{s', s_6, \textbf{account} \Downarrow 100}\ \textsc{eVar} \quad \dfrac{}{s's_6, \textbf{fee} \Downarrow 2}\ \textsc{eVar}}{s', s_6, \textbf{account} - \textbf{fee} \Downarrow 98}\ \textsc{eSub} \tag{8.39}$$

On the other hand, we want the most recent value of variable **fee**. Therefore we must use the current state to get its value.

$$\frac{\dfrac{\dfrac{}{s', s_6, \textbf{account} \Downarrow 100}\ \textsc{eVar} \quad \dfrac{}{s', s_6, \textbf{account} - \textbf{fee} \Downarrow 98}\ \text{sub-proof (8.39)}}{[\textbf{transaction}], s', s_6, \textbf{account} := \textbf{account} - \textbf{fee} \longrightarrow [\textbf{transaction}], s', s_7, \texttt{skip}}\ \textsc{rAss}}{[\textbf{transaction}], s', s_6, \textbf{account} := \textbf{account} - \textbf{fee};\texttt{compensate}\ \textbf{transaction} \longrightarrow [\textbf{transaction}], s', s_7, \texttt{skip};\texttt{compensate}\ \textbf{transaction}}\ \textsc{rSeq2} \tag{8.40}$$

After executing the last part of the compensation, the state is updated to s_7 where

$s_7 = [(\textbf{account} \rightarrow 98), (\textbf{price} \rightarrow 20), (\textbf{fee} \rightarrow 2)]$

$$\frac{}{[\textbf{transaction}], s_7, \texttt{skip};\texttt{compensate}\ \textbf{transaction} \longrightarrow [\textbf{transaction}], s_7, \texttt{compensate}\ \textbf{transaction}}\ \textsc{rSeq1} \tag{8.41}$$

If we compare the state before executing command **account := account - fee**, s_2 with the final state s_7, we notice that there is a descrepancy of 2 in the variable **account**. This is because, to successfully compensate the activity, we had to charge the user a fee of 2. By storing both the current state and the state at compensation installation, we managed to successfully compensate the command **account := account - fee**.

9. Nested Compensations

Nested compensations present another difficulty in our formal semantics. Consider the following example

```
1   scope Z {
2       x := 0
3       undo c1 with {
4           x := 5;
5           undo c2 with {
6               x := 10;
7           }
8           compensate Z;
9               x := x.back + 1;
10      }
11      compensate Z
12  }
```

In this case, the variable x is present in all nested compensations while holding a different value in each compensation. After successfully executing command c_1, its associated compensation together with the current state are pushed on the stack. Let's assume that the stack t is equal to $[(c_1', s_1)::\mathbf{Z}]$ where

$c_1' = x$:= 5;undo c_2 with{ x := 10};compensate **Z**;x := x.back + 1;

$s_1 = [(x \rightarrow 0)]$

$$\frac{}{t, s^{\leftarrow}, s_1, \texttt{compensate}\ \mathbf{Z} \longrightarrow, [\mathbf{Z}], s_1, s_1, x := 5; \ldots}\ \textsc{rComp2} \tag{9.1}$$

$$\frac{\dfrac{}{s_1, s_1, x \Downarrow 0}\ \textsc{eVarF} \qquad \dfrac{}{s_1, s_1, 5 \Downarrow 5}\ \textsc{eVal}}{[\mathbf{Z}], s_1, s_1, x := 5 \longrightarrow [\mathbf{Z}], s_1, s_2, \texttt{skip}}\ \textsc{rAss} \tag{9.2}$$

After executing the assignment statement, the current state is updated to s_2 where

$s_2 = [(x \rightarrow 5)]$

$$\frac{}{[\mathbf{Z}], s_1, s_2, \texttt{skip;undo}\ c_2\ \texttt{with}\{\ldots\} \longrightarrow [\mathbf{Z}], s_1, s_2, \texttt{undo}\ c_2\ \texttt{with}\{\ldots\}} \tag{9.3}$$

In this example we are not interested in how c_2 is executed, therefore we will immediately reduce c_2 to `skip`

$$\frac{}{[\mathbf{Z}], s_1, s_2, \texttt{undo skip with}\ x := 10 \longrightarrow t', s_1, s_2, \texttt{skip}} \tag{9.4}$$

After pushing the compensation on the stack, the stack is updated to t' which contains $[(c_2, s_2)::\mathbf{Z}]$ where

$c_2 = x := 10;$

$s_2 = [(x \rightarrow 5)]$

$$\frac{}{t', s_1, s_2, \texttt{skip;compensate}\ \mathbf{Z} \longrightarrow t', s_1, s_2, \texttt{compensate}\ \mathbf{Z}}\ \textsc{rSeq1} \tag{9.5}$$

$$\frac{}{t', s_1, s_2, \texttt{compensate}\ \mathbf{Z} \longrightarrow [\mathbf{Z}], s_2, s_2, c_2\texttt{;compensate}\ \mathbf{Z}}\ \textsc{rSeq1} \tag{9.6}$$

Recall that at the head of t', we have the tuple (c_2, s_2) where

$c_2 = x := 10;$

$s_2 = [(x \rightarrow 5)]$

$$\dfrac{\dfrac{}{s_2, s_2, x \Downarrow 5}\ \textsc{eVarF} \qquad \dfrac{}{s_2, s_2, 10 \Downarrow 10}\ \textsc{eVal}}{[\mathbf{Z}], s_2, s_2, x := 10 \longrightarrow [\mathbf{Z}], s_2, s_3, \texttt{skip}}\ \textsc{rAss} \tag{9.7}$$

$$\dfrac{}{[\mathbf{Z}], s_2, s_3, \texttt{skip}; x := x.\texttt{back} + 1 \longrightarrow [\mathbf{Z}], s_2, s_3, x := x.\texttt{back} + 1} \tag{9.8}$$

Ideally, when executing the command $x := x.\texttt{back} + 1$, we would want to evaluate $x.\texttt{back}$ with respect to the state at compensation installation i.e. s_1. However, our current version of semantics allows us to store only the state of the currently executing compensating activity. Thus s_1 was overwritten by s_2.

One solution to this problem is to store an additional stack of states. Every time a compensating activity executes successfully, the state belonging to that compensation is pushed on the stack. Once the compensating activity terminates, the state is popped out of the stack.

To support this change, our evaluation semantics for expressions will be unaffected. However, the semantics regarding commands will have to change. The relation will now consist of five elements. An additional element containing the stack of current nested scopes activated will be needed.

The only change that is needed in the majority of the reduction rules is the inclusion of another element in the relation, namely the stack containing the nested compensations currently activated. This new element r, will only be used in the rules regarding the `compensate` construct. For this reason, only the rules regarding the `compensate` construct will be presented. The full set of reduction rules can be

found in Appendix A.3

r is a stack of pairs similar to the compensations stack t. Each pair contains the next command to be executed after the current compensation is terminated, together with the state of its parent compensation.

$$\frac{}{n :: t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{compensate}\ n) \longrightarrow n :: t, r, s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}}\ \text{RCOMP1}$$

The only difference in the rule RCOMP1 is the inclusion of a new element r. This rule is used to to show termination of the `compensate` construct.

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{compensate}\ n) \longrightarrow t', (\texttt{compensate}\ n, s^{\leftarrow}) :: r, s', s^{\rightarrow}, c}\ \text{RCOMP2}$$

where

$t' = \textbf{pop\_first\_c}\ t\ n$

$s' = \textbf{snd}\ (\textbf{get\_first\_c}\ t\ n)$

$c = \textbf{fst}\ (\textbf{get\_first\_c}\ t\ n)$

When executing a compensating activity, we may need to refer to the values of variables at compensation installation. Thus $s^{\leftarrow}$ must be updated with the value present in the stack. However the value of $s^{\leftarrow}$ may still be needed after c is executed successfully. Thus, $s^{\leftarrow}$ is pushed on top of r together with the command to be executed after c terminates. Additionally, the compensation must be popped from the compensating stack.

$$\frac{}{t, (\texttt{compensate}\ n, s) :: r, s^{\leftarrow}, s^{\rightarrow}, \texttt{skip} \longrightarrow t, r, s, s^{\rightarrow}, \texttt{compensate}\ n}\ \text{RCOMP3}$$

When the compensation terminates, we must go back to the outer compensating activity, in the case of nested compensating activities. In the previous rule, we noted that the next command to be executed was stored in the new stack r. Thus, after successful execution of the command c, the command found at the head of r must be executed next. Since c terminated, the state when it was installed is not needed anymore. Thus, $s^{\leftarrow}$ is updated with the value found on top of r.

10. Scope Handling

In the textual description of our programming language, it is stated that the commands `compensate` n and `purge` n can never be called from outside scope n. When the user explicitly calls these constructs outside scope n, the compiler displays an error stating that it cannot compensate that scope.

However, since compensation installation and activation do not occur at the same time, there may be cases where this is allowed by the language compiler. Consider the following example

```
1    ...
2      undo a with a';
3      scope one {
4          undo b with b';
5          scope two {
6              undo c with c';
7              undo d with compensate two;
8          }
9          compensate one;
10         ...
11     }
12     ...
```

In this case after successfully executing scope **two**, the stack, t will hold the following items.

$t = [(\texttt{compensate}\ \textbf{two},\ s_1)::(c',\ s_2)::(\mathrm{b}',\ s_3)::\langle\textbf{one}\rangle::(a',\ s_4)::t']$

s_1, s_2, s_3 and s_4 represent the state at compensation installation. In this example, the state is not used.

The element t at the end of the stack is used to denote the remaining part of the stack. Note that the scope name **two** is not present on the stack since after successfully executing the contents of a scope, the scope name is popped.

$$\frac{}{t, s^{\leftarrow}, s^{\rightarrow}, \texttt{compensate}\ \textbf{one} \longrightarrow t_2,\ \mathrm{s}_1,\ \mathrm{s}^{\rightarrow}, \texttt{compensate}\ \textbf{two}\texttt{;}\texttt{compensate}\ \textbf{one}}\ \textsc{rComp2} \tag{10.1}$$

After popping the first compensation from the stack, the stack is updated to t_2 where

$t_2 = [(c',\ s_2)::(\mathrm{b}',\ s_3)::\langle\textbf{one}\rangle::(a',\ s_4)::t']$

The next command forces all compensations to be executed until scope name **two** is found at the head of the stack. Recall that **two** was popped when scope **two** finished executing.

$$\frac{}{t_2, s_1, s^{\rightarrow}, \texttt{compensate}\ \textbf{two} \longrightarrow t_3, s_2, s^{\rightarrow}, c'\texttt{;}\texttt{compensate}\ \textbf{two}}\ \textsc{rComp2} \tag{10.2}$$

We will assume that command c' is equal to `skip`. Additionally, the stack is updated to t_3 where

$t_3 = [(b',\ s_3)::\langle\textbf{one}\rangle::(a',\ s_4)::t']$

$$\frac{}{t_3,\ s_1, s^{\rightarrow}, \texttt{skip;compensate}\ \textbf{two} \longrightarrow t_3,\ s_1, s^{\rightarrow}, \texttt{compensate}\ \textbf{two}} \tag{10.3}$$

$$\frac{}{\begin{array}{l} t_3, s_2, s^{\rightarrow}, \texttt{compensate}\ \textbf{two} \longrightarrow \\ \\ t_4, s_3, s^{\rightarrow}, b'\texttt{;compensate}\ \textbf{two} \end{array}}\ \textsc{rComp2} \tag{10.4}$$

Similarly, we will assume that command b' is equal to `skip`. The stack is updated to t_4 where

$t_4 = [\ \langle \textbf{one} \rangle :: (a',\ s_4) :: t']$

$$\frac{}{t_4,\ s^{\leftarrow}, s^{\rightarrow}, \texttt{skip;compensate}\ \textbf{two} \longrightarrow t_4,\ s^{\leftarrow}, s^{\rightarrow}, \texttt{compensate}\ \textbf{two}} \tag{10.5}$$

This will force the system to keep executing all compensations present on the stack until scope name **two** is found. Since it is not present on the stack, the program will fail.

There are a number of solutions present to solve this problem. One alternative is to not allow the constructs `compensate` and `purge` to break out of a `with` block. However, this can rule out certain valid programs example:

```
1   scope A
2   {
3       undo a with a';
4       undo b with compensate A;
5       compensate A;
6   }
```

Another way how to solve this problem is to search through the stack for the scope name before beginning to execute the compensating activities. If the scope

name is not found, an exception is thrown. To accommodate this change, the rules RCOMP2 and RPURGE need to be modified as follows:

$$\frac{}{n :: t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{compensate}\ n) \longrightarrow n :: t, r, s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}}\ \text{RCOMP1}$$

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{compensate}\ n) \longrightarrow t', (\texttt{compensate}\ n, s^{\leftarrow}) :: r, s', s^{\rightarrow}, a; c}\ \text{RCOMP2}$$

where

$t' =$ **pop_first_c** t n

$s\prime =$ **snd (get_first_c** $t\ n$**)**

$a =$ **find_n** $t\ n$

$c =$ **fst (get_first_c** $t\ n$**)**

Before starting to execute the compensation at the head of the stack, a recursive function **find_n** is called to search through the stack for the scope name **n**. The function is defined as follows

find_n :: **Stack** $\rightarrow$ **Scope** $\Rightarrow$ **Scope**

find_n $m{:}t\ n =$ `skip`

find_n $m{:}t\ n =$ **find_n** $t\ n$

find_n $(c,s){:}t\ n =$ **find_n** $t\ n$

find_n $\epsilon\ n\ =$ `throw exc`

If n is found on the stack, then the `skip` command is returned. This will cause the system to automatically start executing the compensations on the stack. On the other hand, if any element apart from n is found at the head of the stack, we must continue to search for scope name n in the remaining part of the stack. Lastly, if n is not found, an exception is thrown. In this case it would be ideal if the `compensate` command is surrounded with a `try-catch` block so that execution is not aborted.

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{purge}\ n) \longrightarrow t', r, s^{\leftarrow}, s^{\rightarrow}, a; \texttt{skip}}\ \text{RPURGE}$$

where

$t' = \textbf{super\_pop}\ t\ n$

$a = \textbf{find\_n}\ t\ n$

The changes applied to rule RPURGE are the same as those in RCOMP2. The same function **find_n** is used to check whether a scope name is present on the stack.

An optimized algorithm would be to check whether the scope name is being referenced in the compensation stack before you pop it. Consider the following example

```
1   scope one {
2       undo a with a';
3       scope two {
4           undo b with b';
5           undo c with compensate two;
6       }
7       compensate one;
8       ...
9   }
10  ...
```

After successfully executing command c, the stack will contain the following elements

$$[(\texttt{compensate}\ \textbf{two},\ s_1)::(b',\ s_2)::\langle\textbf{two}\rangle::(a',\ s_3)::\langle\textbf{one}\rangle::t]$$

The next step would be to check whether the scope name **two** is being referenced in the compensation stack. There is no way that scope name **two** is called before **two** was pushed on the stack. Thus, we only need to search up until we find scope name **two**. If scope name **two** is being called in any of the compensating activities, we change that command to throw an exception. In this example, the stack will be updated as follows

$$[(\texttt{throw exc},\ s_1)::(b',\ s_2)::(a',\ s_3)::\langle\textbf{one}\rangle::t]$$

When we move on to execute the compensating activities on the stack, an exception will be thrown. This would be more suitable to implement since searching through a stack every time before executing a compensation would be very time consuming.

11. Local Variables and Compensations

In Vella's language, all variables are declared globally. The reason for declaring each variable globally was that compensation installation and activation is performed at different times. Thus, the use of local variables would cause a lot of difficulties. Consider the following example

```
1  scope A {
2      x := 10;
3      declare y in {
4          scope B {
5              y := x;
6              undo {
7                  x := x + y;
8              }
9              with {
10                 x := x - y;
11             }
12         }
13     }
14     compensate A;
15 }
```

In this example, the variable y is declared only for scope **B**. When we try to execute the expression x - y outside of scope **B**, the program will fail since y is not an element of the current state. Hence, declaring all variables globally will eliminate this problem

In this section, we shall explore the use of compensations in a language that has both global and scope variables. A new construct will be needed that will act as a local scope for a variable

`declare` x `in` c: This construct will limit the variable x only to exist in c. Any references made to variable x outside of c will fail the program.

The difference between the constructs `declare-in` and `scope` will be that `declare-in` will represent a chunk of code where the variable may be accessed. On the other hand, `scope` acts as an identifier for a group of compensations.

In previous sections, we mentioned that all the free variables are contained in the domain of the initial state. When using local variables, the starting state is empty. The semantic rules for the `declare-in` construct will be as follows

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, \texttt{declare}\ x\ \texttt{in}\ c \longrightarrow t, r, s^{\leftarrow}, s^{\rightarrow}[x \rightarrow \texttt{null}], c; \texttt{end}\ x}\ \text{RDECL1}$$

When the command `declare` x `in` c is encountered, we must add the new variable x in the program state. We shall assign the value null for the moment. Additionally, we must start executing command c. The command `end` x will be added after c to denote the point up until which x can be accessed.

$$\frac{}{t, r, s \leftarrow, s^{\rightarrow}, \texttt{end}\ x \longrightarrow t, s^{\leftarrow}, s', \texttt{skip}}\ \text{RDECL2}$$

where $s' = s^{\rightarrow} \mathbin{/} x$

When we reach the command `end` x, we must remove the variable x from the program state. This rule cannot be used if we allow the same variables to be named in different scopes example:

```
1   scope A {
2       declare x in {
3           x := 0;
4           declare x in {
5               scope B {
6                   x := 2;
7               }
8           }
9           x := x + 1;
10      }
11  }
```

The inner `declare-in` removes x out of the current state as soon as we finish executing line 8. However, x was also declared for line 9. This problem is known as variable shadowing. At this stage, we will limit our programs to use unique variable names.

Additionally, the reduction rules for the assignment statement need to be changed.

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e_1 \Downarrow v \quad s^{\leftarrow}, s^{\rightarrow}, e_2 \Downarrow w \quad e_1 \in s^{\rightarrow}}{t, r, s^{\leftarrow}, s^{\rightarrow}, (e_1 := e_2) \longrightarrow t, r, s^{\leftarrow}, (s^{\rightarrow}[v \rightarrow w]), \texttt{skip}}\ \text{RASS}$$

A value can only be assigned to a variable, if that variable is already found in the state. Thus, we can never assign a value to a variable, if it has not yet been

declared.

This will bring up similar problems that were discussed in the previous chapters. Since compensation installation and activation happens at different times in a program, there may be cases that we will end up executing scope variables from outside their scope.

```
1   ...
2   declare x in {
3       scope A {
4           x := 10;
5           declare y in {
6               scope B {
7                   y := x;
8                   undo {
9                       x := x + y;
10                  }
11                  with {
12                      x := x - y;
13                  }
14              }
15          }
16          compensate A;
17      }
18  }
```

In this example when we finish executing scope **B** any local variables pertaining only to that scope are removed from the current state. In this case the variable y is removed. When we try to execute the compensation $x := x - y$, the program will fail since the variable y is not present in the program state.

Additionally, there are cases where the program does not fail. However an incorrect output is produced.

```
1  scope A {
2      declare x in {
3          scope B {
4              x := 10;
5              undo {
6                  x := x + 1;
7              }
8              with {
9                  x := x - 1;
10             }
11         }
12     }
13     declare x in {
14         scope C {
15             x : = 5;
16             compensate A;
17         }
18     }
19 }
```

In this example, when scope **B** terminated, the variable x is removed out of the current state. When we enter scope **C**, a new variable, x, is added to the current state. When we start executing the compensating activity `x := x - 1;`, the value of x in the compensating activity would be the new value assigned to x in scope **C**.

There are a number of options to solve this problem. One solution is to add another reduction rule, RAss2, that throws an exception every time a variable is not present in the current state. It would be ideal to surround the compensation with a `try-catch` block so that the program does not end with an exception. The

reduction rule would be very similar to RAss.

$$\frac{s^{\leftarrow}, s^{\rightarrow} e_1 \Downarrow v \quad s^{\leftarrow}, s^{\rightarrow}, e_2 \Downarrow w \quad e_1 \notin s^{\rightarrow}}{t, r, s^{\leftarrow}, s^{\rightarrow}, (e_1 := e_2) \longrightarrow t, r, s^{\leftarrow}, s^{\rightarrow}), \texttt{throw exc}} \text{ RAss2}$$

An alternative solution is to use the same technique that was used when we handling scopes. Before removing a variable from the current state, we must first check the compensation stack to see if this variable is being referenced. If the variable is being accessed in a compensating activity, we change that activity to throw an exception.

The final solution is to update any referenced variables in the stack with the value held by them, before removing them from the state. Consider the following example:

```
1   scope A {
2       x := 10;
3       declare y in {
4           scope B {
5               y := x;
6               undo
7                   x := y + 1;
8                   y := x;
9               with
10                  x := y - 1;
11          }
12      }
13      compensate(A);
14  }
```

After successfully executing the activity $y := x + 1$, the stack and state would contain the following elements:

$$\text{Stack: } [(x := y - 1, s_1) :: \langle \mathbf{B} \rangle :: \langle \mathbf{A} \rangle]$$

$$\text{State: } [y \mapsto 11, x \mapsto 11]$$

When scope **B** terminates, variable y must be removed from the current state. Before doing so, any references made to y are updated to the value that y holds in the state. The stack and state are updated as follows:

$$\text{Stack: } [(x := 11 - 1, s_1) :: \langle \mathbf{A} \rangle]$$
$$\text{State: } [x \mapsto 11]$$

After executing the compensating activities belonging to **A**, no exceptions will be thrown and the program will terminate successfully.

11.1 Evaluation

In previous sections we mentioned the issues brought up when we try to handle variables inside compensations. The introduction of local variables complicates things even more. Problems like variable shadowing are only brought up with the introduction of local variables. Additionally, local variables inside compensations can be very tricky since a compensation can be executed any time in a program.

In this section, three solutions were presented to overcome most of the problem brought with local variables and compensations. The best solution out of the three is the last one whereas, if a local variable inside a compensation is accessed outside its variable scope, we use its most recent value. However, it is up to the programmer whether to allow local variables inside a compensation to be accessed outside of their scope.

In this section, we came up with a set of reduction rules that handle local variables inside compensations. The rules are not finalized and more work is needed to overcome all the problems.

12. Related Work

Along the years, many formalisms have emerged which handle compensations. In this section we will review two notations namely StAC and Sagas. Additionally we will compare both formalisms with our solution.

The main reason for choosing StAC and Sagas is because both of them provide a formal description of compensations by using operational semantics. The similarities and differences with our formal semantics will be provided as we discuss more about each formalism.

12.1 StAC

In StAC [12], compensations are not only associated with failures. StAC stores the accumulated compensating activities in a stack. Thus compensations can be installed, executed and deleted using the usual stack operations. Additionally, StAC supports both exception handling and compensations.

12.1.1 Syntax

$StAC_i$ is an extension of StAC which supports concurrent compensation stacks. Thus several compensations can be maintained concurrently. The syntax for the compensating constructs in $StAC_i$ is presented in Table 12.1.

new(i).P_i	used to create a new compensation stack
$\mid \boxtimes_i$	indexed reverse
$\mid \boxdot_i$	indexed accept
$\mid \uparrow_i$P	push
$\mid$ J $\rhd ?$ i	merge

Table 12.1: StAC Syntax

In $StAC_i$, each compensation task must have a compensation stack of its own where compensations are added to it. The new(i).P_i construct is used to create a new compensation stack i, where tasks are added in it throughout the execution of P. The $\uparrow_i$P operator is used to push tasks onto the stack. Thus a compensation pair can be defined using the push construct

$$P \; ; \uparrow_i Q$$

In the following example, Q is pushed onto stack i after successful conclusion of P.

In $StAC_i$ execution of compensations must be programmed explicitly. Thus the $\boxtimes_i$ must be used to reverse the order of execution of stack i. The accept construct $\boxdot_i$ is used to discard all tasks residing in stack i.

Lastly, StAC has a merge construct $J \rhd ?\ i$ which allows multiple compensation tasks to be merged with one another. For example $c \rhd ?\ s$ merges all the processes belonging to c into the compensation stack of task s.

12.1.2 Semantics

The semantics of StAC is given in terms of a small-step semantics that describe the language constructs. We shall only focus on the constructs regarding compensations.

$$\frac{C(k) = \bot}{(new(i).P_i, C, \sigma) \xrightarrow{T} (P_k, C[k := skip], \sigma)}$$

Create Compensation Task: in this rule we must first check whether the new compensation task, k is not in use. If this premise holds, we create a new compensation task and set it to *skip*.

$$\frac{}{(\uparrow_i Q, C, \sigma) \longrightarrow (skip, C[i := (Q; C(i))], \sigma)}$$

Push: this operator, adds the compensation Q, to the compensation function C. Similar to our compensation language, C is added in sequence with the previous compensations such that the compensations will be executed in reverse order.

$$\frac{}{(\boxtimes_i, C, \sigma) \longrightarrow (C(i), C[i := skip], \sigma)}$$

Reverse: the $\boxtimes_i$ operator is very similar to the `compensate` construct. In this rule we are formalizing the behaviour when a compensation task i, is to be executed. After execution of i the contents of the compensation task is set to *skip*.

$$\frac{}{(\boxdot_i, C, \sigma) \longrightarrow (skip, C[i := skip], \sigma)}$$

Accept: the $\boxtimes_i$ operator works very similar to the purge construct. Whenever we reach $\boxtimes_i$, the whole compensation task i is discarded and i is reset to *skip*.

$$\frac{}{(J \triangleright ?i, C, \sigma) \longrightarrow (skip, C[i := (\|_{j \in J}.C(j)); C(i), J := \bot], \sigma)}$$

Merge: all compensation tasks of J are merged in parallel on the front of the compensation task i. The value of J is set to $\bot$ meaning that the tasks in J are no longer in use.

Similar to our approach, small-step semantics are used to describe the behaviour of commands. The rules take the form of a relation between two configurations, where the configuration is a tuple containing a compensation stack, a command and the current state. This is very similar to the the reduction rules specified for Vella's language.

One difference is that in StAC, each compensation task has a compensation stack of its own. In this project, we use a single stack where all compensations are stored. Additionally, StAC, does not save the state at which the compensations are installed. On the other hand, the current state is always used when executing compensations.

12.2 Sagas

Sagas [13] is a flow composition language based on the idea of saga. A saga is a transaction composed of a sequence of activities which either all of them succeed or else all activities should be compensated.

12.2.1 Syntax

X	::=	0 | A | A ÷ B	step
P	::=	X | P;P | P|P	process
S	::=	{P}	saga

Table 12.2: Sagas Syntax

A saga S, consists of a process enclosed in a scope. Each step in P may refer to either the inert process 0, or an activity A or a compensated activity $A \div B$. Processes may be composed of a single process X, a sequential composition of processes $P;P$, or a parallel composition of processes $P|P$.

A transaction may terminate in three different ways. ⊡ signifies that a transaction has terminated successfully. ⊠ signifies that the transaction was aborted and then it was successfully compensated. Lastly, ⊞ signifies a transaction that was aborted but not successfully compensated.

12.2.2 Semantics

The semantics of Sagas is given as big-step semantics. Unlike StAC, activities in Sagas operate under a context Γ, which determines the success or failure of the activity. Γ is a partial function that maps any activity A, to the result obtained from execution

$$\Gamma : A \rightharpoonup \{\boxdot, \boxtimes\}$$

The semantics of saga is given by the relation

$$\Gamma \vdash S \xrightarrow{\alpha} \square$$

We shall now consider the semantic rule in the case of a successful completion of a step

$$\mathrm{A} \rightarrow \boxdot \, , \Gamma \vdash \langle \, \mathrm{A} \div \mathrm{B}, \beta \, \rangle \xrightarrow{A} \langle \, \boxdot \, , \mathrm{B}; \beta \, \rangle$$

This rule states that if A terminates successfully, then so does $A \div B$ and thus it evolves to $\boxdot$. Additionally β evolves to $B \, ; \beta$ where the compensation B is added in sequence to the previous compensations. We will now consider the semantics rules of the compensating mechanism.

$$\frac{\Gamma \vdash \langle \beta, 0 \rangle \xrightarrow{\alpha} \langle \boxdot, 0 \rangle}{A \rightarrow \boxtimes, \Gamma \vdash \langle A \div B, \beta \rangle \xrightarrow{\alpha} \langle \boxtimes, 0 \rangle} \qquad \frac{\Gamma \vdash \langle \beta, 0 \rangle \xrightarrow{\alpha} \langle \boxtimes, 0 \rangle}{A \rightarrow \boxtimes, \Gamma \vdash \langle A \div B, \beta \rangle \xrightarrow{\alpha} \langle \circledast, 0 \rangle}$$

The first rule above is saying that when the compensation succeeds, if β, the compensation, progresses to $\boxdot$, when A fails, $A \div B$ evolves to $\boxtimes$. Otherwise, if we look at the second rule, when the compensating activity fails as well i.e. β progresses to $\boxtimes$, then if A fails, $A \div B$ evolves to $\circledast$

13. Conclusions

The aim of this FYP was to create a formal description for an imperative programming language that is enriched with the compensation construct. The use of a formal description would help us to prove the correctness of the language.

Initially, we started by creating a preliminary operational semantics that described the behaviour of Vella's language. The output of the reduction rules was then compared with the output generated by the compiler. Moreover, we became familiar with the programming language and came up with a number of edge cases that caused our programs to terminate abnormally or produce incorrect outputs. Starting from the formalization of our language constructs, we have progressively enriched our language to deal with variable and scope handling. Careful analysis of both variable and scope handling were needed to give a convenient solution.

The main problem with the use of compensations as a fault handling mechanism was that compensation installation and activation happens at different times in a program. This caused a lot of problems when handling both variables and scopes.

Finally, We have given a precise operational semantics of our programming language which is adequate with respect to the textual description available. Additionally the reduction rules provided a way to solve the problems that were unforeseen

when implementing the programming language.

13.1 Future Work

The principal focus of this dissertation was to present a formal description for Vella's language. Additionally, we explored the use of local variables within the compensation mechanism. Due to lack of time available, we did not manage to come up with a set of rules that describe the behaviour of local variables. Extending the rules with the `declare-in` command would definitely be a great addition to our formal semantics.

Currently, Vella's compiler does not detect the erroneous programs presented in the previous sections. Ideally, the compiler is optimized to handle the anomalies brought forward by the reduction rules.

Bibliography

[1] Jie Xu, Brian Randell, Alexander Romanovsky, Cecilia M F Rubira, Robert J Stroud, and Zhixue Wu, *Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery* 1995

[2] Hanne Riis Nielson, Flemming Nielson, *Semantics with Applications, A Formal Introduction.* 1995.

[3] Koen V. Hindriks, Frank S. de Boer, Wiebe van der Hoek, John-Jules Ch. Meyer *Formal semantics for an abstract agent programming language* Lecture Notes in Computer Science Volume 1365, 1998, pp 215-229

[4] Roy H. Campbell, Brian Randell, *Error recovery in asynchronous systems* IEEE Transactions on Software Engineering, vol. 12, no. 8, pp. 811-826, 1986

[5] Brian Randell, P. A. Lee and P. C. Treleaven, *Reliability Issues in Computing System Design.* 1978

[6] Tom Anderson, Brian Randell, *Computing Systems Reliability* Cambridge Univ. Press, pp. 160-162, 1986

[7] Christian Colombo and Gordon J. Pace, *Semantics of Compensations.* 2011

[8] B. Randell, P.A. Lee, P.C. Treleaven, *Reliability Issues in Computing System Design* 1978

[9] Lydia Vella, *Compensations in an Imperative Programming Language.* 2010

[10] William Pugh, *The Java Memory Model is Fatally Flawed* Deptartment of Computer Science, University of Maryland, College Park

[11] Ken Slonneger and Barry L. Kurtz, *Formal Syntax and Semantics of Programming Langauges: A Laboratory-Based Approach.* 1995.

[12] Michael Butler and Carla Ferreira, *An Operational Semantics for StAC, a Language for Modelling Long-running Business Transactions.* 2004

[13] Roberto Bruni, Hernan Melgratti, Ugo Montanari, *Theoretical foundations for compensations in ow composition languages* 2005

A. Appendix

A.1 Preliminary Semantics

A.1.1 Big-Step Semantics for Expressions

$$\frac{}{s, w \Downarrow w}\ \textsc{eVal} \qquad \frac{}{s, v \Downarrow s(v)}\ \textsc{eVar}$$

$$\frac{s, e_1 \Downarrow w_1 \qquad s, e_2 \Downarrow w_2 \qquad w_3 = w_1 + w_2}{s, e_1 \texttt{+} e_2 \Downarrow w_3}\ \textsc{eAdd}$$

$$\frac{s, e_1 \Downarrow w_1 \qquad s, e_2 \Downarrow w_2 \qquad w_3 = w_1 - w_2}{s, e_1 \texttt{-} e_2 \Downarrow w_3}\ \textsc{eSub}$$

$$\frac{s, e_1 \Downarrow w_1 \qquad s, e_2 \Downarrow w_2 \qquad w_3 = w_1 \wedge w_2}{s, e_1 \ \texttt{and}\ e_2 \Downarrow w_3}\ \textsc{eAnd}$$

$$\frac{s, e_1 \Downarrow w_1 \qquad s, e_2 \Downarrow w_2 \qquad w_3 = w_1 \leq w_2}{s, e_1 \ \texttt{<=}\ e_2 \Downarrow w_3}\ \textsc{eLeq}$$

$$\frac{s, e \Downarrow w \qquad w' = \neg w}{s, \ \texttt{not}\ e \Downarrow w'}\ \textsc{eNot}$$

A.1.2 Small-Step Semantics for Commands

$$\frac{}{t, s, \texttt{skip}; c \longrightarrow t, s, c}\ \textsc{rSeq1}$$

$$\frac{t, s, c_1 \longrightarrow t', s', c_3}{t, s, c_1; c_2 \longrightarrow t', s', c_3; c_2}\ \textsc{rSeq2}$$

$$\frac{s, e_1 \Downarrow v \qquad s, e_2 \Downarrow w}{t, s, e_1 := e_2 \longrightarrow t, (s[v \rightarrow w]), \texttt{skip}}\ \textsc{rAss}$$

$$\frac{s, e \Downarrow \texttt{true}}{t, s, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \longrightarrow t, s, c_1}\ \textsc{rIf}$$

$$\frac{s, e \Downarrow \texttt{false}}{t, s, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \longrightarrow t, s, c_2}\ \textsc{rElse}$$

$$\frac{s, e \Downarrow \texttt{true}}{t, s, \texttt{while}\ e\ \texttt{do}\ c \longrightarrow t, s, c; \texttt{while}\ e\ \texttt{do}\ c}\ \textsc{rWhile1}$$

$$\frac{s, e \Downarrow \texttt{false}}{t, s, \texttt{while}\ e\ \texttt{do}\ c \longrightarrow t, s, \texttt{skip}}\ \textsc{rWhile2}$$

$$\frac{}{t, s, \texttt{try skip catch}\ c \longrightarrow t, s, \texttt{skip}}\ \textsc{rTry1}$$

$$\frac{t, s, c_1 \longrightarrow t', s', c_1'}{t, s, (\texttt{try}\ c_1\ \texttt{catch}\ c) \longrightarrow t', s', \texttt{try}\ c_1'\ \texttt{catch}\ c}\ \textsc{rTry2}$$

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{try throw } exc \texttt{ catch } c) \longrightarrow} \text{ rCATCH}$$

$$\frac{}{t, s, (\texttt{throw } exc; c) \longrightarrow t, s, \texttt{throw exc}} \text{ rTHROW}$$

$$\frac{}{t, s, (\texttt{undo skip with } c) \longrightarrow c :: t, s, \texttt{skip}} \text{ rUNDO1}$$

$$\frac{t, s, c_1 \longrightarrow t', s', c_1'}{t, s, (\texttt{undo } c_1 \texttt{ with } c_2) \longrightarrow t', s', \texttt{undo } c_1' \texttt{ with } c_2} \text{ rUNDO2}$$

$$\frac{}{n :: t, s, (\texttt{compensate } n) \longrightarrow n :: t, s, \texttt{skip}} \text{ rCOMP1}$$

$$\frac{}{t, s, (\texttt{compensate } n) \longrightarrow t', s, c; \texttt{compensate } n} \text{ rCOMP2}$$

where

$t' = \textbf{pop\_first\_c}\ t\ n$

$c = \textbf{get\_first\_c}\ t\ n$

$$\frac{}{t, s, (\texttt{purge } n) \longrightarrow t', s, \texttt{skip}} \text{ rPURGE}$$

where

$t' = \textbf{super\_pop}\ t\ n$

$$\frac{}{t, s, \texttt{scope}\ \ n\ c \longrightarrow n :: t, s, c; \texttt{end}\ \ n}\ \textsc{rScope1}$$

$$\frac{}{t, s, \texttt{end}\ \ n \longrightarrow t', s, \texttt{skip}}\ \textsc{rScope2}$$

where

$\mathrm{t}' = \mathbf{pop\_n}\ t\ n$

A.2 Variable Handling

A.2.1 Evaluation Semantics for Expressions

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e_1 \Downarrow w_1 \quad \mathrm{s}^{\leftarrow}, s^{\rightarrow}, e_2 \Downarrow w_2 \quad w_3 = w_1 + w_2}{s^{\rightarrow}, s^{\leftarrow}, e_1 + \mathrm{e}_2 \Downarrow w_3} \textsc{eAdd}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e_1 \Downarrow w_1 \quad \mathrm{s}^{\leftarrow}, s^{\rightarrow}, e_2 \Downarrow w_2 \quad w_3 = w_1 - w_2}{s^{\leftarrow}, s^{\rightarrow}, e_1 \text{ - } \mathrm{e}_2 \Downarrow w_3} \textsc{eSub}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e_1 \Downarrow w_1 \quad \mathrm{s}_{\leftarrow}, s^{\rightarrow}, e_2 \Downarrow w_2 \quad b = w_1 \leq w_2}{s^{\leftarrow}, s^{\rightarrow}, e_1 <= \mathrm{e}_2 \Downarrow w_3} \textsc{eLeq}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e_1 \Downarrow b_1 \quad \mathrm{s}^{\leftarrow}, s^{\rightarrow}, e_2 \Downarrow b_2 \quad b_3 = b_1 \text{ and } \mathrm{b}_2}{s^{\leftarrow}, s^{\rightarrow}, e_1 \text{ and } \mathrm{e}_2 \Downarrow b_3} \textsc{eAnd}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e \Downarrow b \quad b' = \neg b}{s^{\leftarrow}, s^{\rightarrow}, \texttt{not } e \Downarrow b'} \textsc{eNot} \qquad \frac{}{s^{\leftarrow}, s^{\rightarrow}, w \Downarrow w} \textsc{eVal}$$

$$\frac{}{s^{\leftarrow}, s^{\rightarrow}, v \Downarrow s^{\rightarrow}(v)} \textsc{eVarF} \qquad \frac{}{s^{\leftarrow}, s^{\rightarrow}, v.\texttt{back} \Downarrow s^{\leftarrow}(v)} \textsc{eVarB}$$

A.2.2 Reduction Semantics for Commands

$$\frac{}{t, s^{\leftarrow}, s^{\rightarrow}, (\texttt{skip}; c) \longrightarrow t, s^{\leftarrow}, s^{\rightarrow}, c} \text{ rSeq1}$$

$$\frac{t, s^{\leftarrow}, s^{\rightarrow}, c_1 \longrightarrow t', s^{\leftarrow\prime}, s^{\rightarrow\prime}, c_1'}{t, s^{\leftarrow}, s^{\rightarrow}, c_1; c_2 \longrightarrow t', s^{\leftarrow\prime}, s^{\rightarrow\prime}, c_1'; c_2} \text{ rSeq2}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e_1 \Downarrow v \qquad s^{\leftarrow}, s^{\rightarrow}, e_2 \Downarrow w}{t, s^{\leftarrow}, s^{\rightarrow}, (e_1 := e_2) \longrightarrow t, s^{\leftarrow}, (s^{\rightarrow}[v \rightarrow w]), \texttt{skip}} \text{ rAss}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e \Downarrow \mathit{true}}{t, s^{\leftarrow}, s^{\rightarrow}, (\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2) \longrightarrow t, s^{\leftarrow}, s^{\rightarrow}, c_1} \text{ rThen}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e \Downarrow \mathit{false}}{t, s^{\leftarrow}, s^{\rightarrow}, (\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2) \longrightarrow t, s^{\leftarrow}, s^{\rightarrow}, c_2} \text{ rElse}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e \Downarrow \mathit{true}}{t, s^{\leftarrow}, s^{\rightarrow}, (\texttt{while } e \texttt{ do } c) \longrightarrow t, s^{\leftarrow}, s^{\rightarrow}, c; \texttt{while } e \texttt{ do } c} \text{ rWhile1}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e \Downarrow \mathit{false}}{t, s^{\leftarrow}, s^{\rightarrow}, (\texttt{while } e \texttt{ do } c) \longrightarrow t, s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}} \text{ rWhile2}$$

$$\frac{}{t, s^{\leftarrow}, s^{\rightarrow}, (\texttt{try skip catch } c) \longrightarrow t, s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}} \text{ rTry1}$$

$$\frac{t, s \leftarrow, s^{\rightarrow}, c_1 \longrightarrow t', s \leftarrow', s^{\rightarrow\prime}, c_1'}{t, s \leftarrow, s^{\rightarrow}, (\texttt{try}\ c_1\ \texttt{catch}\ c) \longrightarrow t', s^{\leftarrow\prime}, s^{\rightarrow\prime}, \texttt{try}\ c_1'\ \texttt{catch}\ c}\ \textsc{rTry2}$$

$$\frac{}{t, s^{\leftarrow}, s^{\rightarrow}, (\texttt{try throw}\ exc\ \texttt{catch}\ c) \longrightarrow t, s^{\leftarrow}, s^{\rightarrow}, c}\ \textsc{rCatch}$$

$$\frac{}{t, s^{\leftarrow}, s^{\rightarrow}, (\texttt{throw}\ exc; c) \longrightarrow t, s^{\leftarrow}, s^{\rightarrow}, \texttt{throw}\ exc}\ \textsc{rThrow}$$

$$\frac{}{t, s^{\leftarrow}, s^{\rightarrow}, (\texttt{undo skip with}\ c) \longrightarrow (c, s^{\rightarrow}) :: t, s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}}\ \textsc{rUndo1}$$

$$\frac{t, s^{\leftarrow}, s^{\rightarrow}, c_1 \longrightarrow t', s^{\leftarrow\prime}, s^{\rightarrow\prime}, c_1'}{t, s^{\leftarrow}, s^{\rightarrow}, (\texttt{undo}\ c_1\ \texttt{with}\ c_2) \longrightarrow t', s^{\leftarrow\prime}, s^{\rightarrow\prime}, \texttt{undo}\ c_1'\ \texttt{with}\ c_2}\ \textsc{rUndo2}$$

$$\frac{}{n :: t, s^{\leftarrow}, s^{\rightarrow}, (\texttt{compensate}\ n) \longrightarrow n :: t, s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}}\ \textsc{rComp1}$$

$$\frac{}{t, s^{\leftarrow}, s^{\rightarrow}, (\texttt{compensate}\ n) \longrightarrow t', s', s^{\rightarrow}, c; \texttt{compensate}\ n}\ \textsc{rComp2}$$

where

$t' = \textbf{pop\_first\_c}\ t\ n$

$s' = \textbf{snd}\ (\textbf{get\_first\_c}\ t\ n)$

$c = \textbf{fst}\ (\textbf{get\_first\_c}\ t\ n)$

$$\frac{}{t, s^{\leftarrow}, s^{\rightarrow}, (\texttt{purge}\ n) \longrightarrow t', s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}}\ \textsc{rPurge}$$

where

$t' = \textbf{super\_pop}\ t\ n$

$$\frac{}{t, s^{\leftarrow}, s^{\rightarrow}, \texttt{scope}\ n\ c \longrightarrow t, s^{\leftarrow}, s^{\rightarrow}, c\texttt{;end}\ n}\ \textsc{rScope1}$$

$$\frac{}{t, s^{\leftarrow}, s^{\rightarrow}, \texttt{end}\ n \longrightarrow t', s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}}\ \textsc{rScope2}$$

where

$t' = \textbf{pop\_n}\ t\ n$

A.3 Nested Compensations

A.3.1 Evaluation Semantics for Expressions

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e_1 \Downarrow w_1 \quad s^{\leftarrow}, s^{\rightarrow}, e_2 \Downarrow w_2 \quad w_3 = w_1 + w_2}{s^{\rightarrow}, s^{\leftarrow}, e_1 + \mathrm{e}_2 \Downarrow w_3} \text{ EADD}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e_1 \Downarrow w_1 \quad s^{\leftarrow}, s^{\rightarrow}, e_2 \Downarrow w_2 \quad w_3 = w_1 - w_2}{s^{\leftarrow}, s^{\rightarrow}, e_1 \text{ - } \mathrm{e}_2 \Downarrow w_3} \text{ ESUB}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e_1 \Downarrow w_1 \quad \mathrm{s}_{\leftarrow}, s^{\rightarrow}, e_2 \Downarrow w_2 \quad b = w_1 \leq w_2}{s^{\leftarrow}, s^{\rightarrow}, e_1 <= \mathrm{e}_2 \Downarrow w_3} \text{ ELEQ}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e_1 \Downarrow b_1 \quad s^{\leftarrow}, s^{\rightarrow}, e_2 \Downarrow b_2 \quad b_3 = b_1 \text{ and } \mathrm{b}_2}{s^{\leftarrow}, s^{\rightarrow}, e_1 \text{ and } \mathrm{e}_2 \Downarrow b_3} \text{ EAND}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e \Downarrow b \quad b' = \neg b}{s^{\leftarrow}, s^{\rightarrow}, \texttt{not } e \Downarrow b'} \text{ ENOT} \qquad \frac{}{s^{\leftarrow}, s^{\rightarrow}, w \Downarrow w} \text{ EVAL}$$

$$\frac{}{s^{\leftarrow}, s^{\rightarrow}, v \Downarrow s^{\rightarrow}(v)} \text{ EVARF} \qquad \frac{}{s^{\leftarrow}, s^{\rightarrow}, v.\texttt{back} \Downarrow s^{\leftarrow}(v)} \text{ EVARB}$$

A.3.2 Reduction Semantics for Commands

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{skip}; c) \longrightarrow t, r, s^{\leftarrow}, s^{\rightarrow}, c} \text{ rSeq1}$$

$$\frac{t, r, s^{\leftarrow}, s^{\rightarrow}, c_1 \longrightarrow t', r', s^{\leftarrow\prime}, s^{\rightarrow\prime}, c_1'}{t, s^{\leftarrow}, s^{\rightarrow}, c_1 ; c_2 \longrightarrow t', r', s^{\leftarrow\prime}, s^{\rightarrow\prime}, c_1' ; c_2} \text{ rSeq2}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e_1 \Downarrow v \qquad s^{\leftarrow}, s^{\rightarrow}, e_2 \Downarrow w}{t, r, s^{\leftarrow}, s^{\rightarrow}, (e_1 := e_2) \longrightarrow t, r, s^{\leftarrow}, (s^{\rightarrow}[v \rightarrow w]), \texttt{skip}} \text{ rAss}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e \Downarrow true}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2) \longrightarrow t, r, s^{\leftarrow}, s^{\rightarrow}, c_1} \text{ rThen}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e \Downarrow false}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2) \longrightarrow t, r, s^{\leftarrow}, s^{\rightarrow}, c_2} \text{ rElse}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e \Downarrow true}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{while } e \texttt{ do } c) \longrightarrow t, r, s^{\leftarrow}, s^{\rightarrow}, c; \texttt{while } e \texttt{ do } c} \text{ rWhile1}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e \Downarrow false}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{while } c \texttt{ do } c) \quad \rangle\, t, r, s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}} \text{ rWhile2}$$

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{try skip catch } c) \longrightarrow t, r, s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}} \text{ rTry1}$$

$$\frac{t, r, s \leftarrow, s^{\rightarrow}, c_1 \longrightarrow t', r', s \leftarrow', s^{\rightarrow\prime}, c_1'}{t, r, s \leftarrow, s^{\rightarrow}, (\texttt{try } c_1 \texttt{ catch } c) \longrightarrow t', r', s^{\leftarrow\prime}, s^{\rightarrow\prime}, \texttt{try } c_1' \texttt{ catch } c} \text{ rTry2}$$

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{try throw } exc \texttt{ catch } c) \longrightarrow t, r, s^{\leftarrow}, s^{\rightarrow}, c} \text{ rCatch}$$

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{throw } exc; c) \longrightarrow t, r, s^{\leftarrow}, s^{\rightarrow}, \texttt{throw } exc} \text{ rThrow}$$

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{undo skip with } c) \longrightarrow (c, s^{\rightarrow}) :: t, r, s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}} \text{ rUndo1}$$

$$\frac{t, r, s^{\leftarrow}, s^{\rightarrow}, c_1 \longrightarrow t', r', s^{\leftarrow\prime}, s^{\rightarrow\prime}, c_1'}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{undo } c_1 \texttt{ with } c_2) \longrightarrow t', r', s^{\leftarrow\prime}, s^{\rightarrow\prime}, \texttt{undo } c_1' \texttt{ with } c_2} \text{ rUndo2}$$

$$\frac{}{n :: t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{compensate } n) \longrightarrow n :: t, r, s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}} \text{ rComp1}$$

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{compensate } n) \longrightarrow t', (\texttt{compensate } n, s^{\leftarrow}) :: r, s', s^{\rightarrow}, c} \text{ rComp2}$$

where

$t' = \textbf{pop\_first\_c}\ t\ n$

$s' = \textbf{snd}\ (\textbf{get\_first\_c}\ t\ n)$

$c = \textbf{fst}\ (\textbf{get\_first\_c}\ t\ n)$

$$\frac{}{t, (\texttt{compensate}\ n, s) :: r, s^{\leftarrow}, s^{\rightarrow}, \texttt{skip} \longrightarrow t, r, s, s^{\rightarrow}, \texttt{compensate}\ n}\ \textsc{rComp3}$$

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{purge}\ n) \longrightarrow t', r', s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}}\ \textsc{rPurge}$$

where

$t' = \textbf{super\_pop}\ t\ n$

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, \texttt{scope}\ n\, c \longrightarrow t, r, s^{\leftarrow}, s^{\rightarrow}, c\texttt{;end}\ n}\ \textsc{rScope1}$$

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, \texttt{end}\ n \longrightarrow t', r', s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}}\ \textsc{rScope2}$$

where

$t' = \textbf{pop\_n}\ t\ n$

A.4 Scope Handling

A.4.1 Evaluation Semantics for Expressions

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e_1 \Downarrow w_1 \quad s^{\leftarrow}, s^{\rightarrow}, e_2 \Downarrow w_2 \quad w_3 = w_1 + w_2}{s^{\rightarrow}, s^{\leftarrow}, e_1 + \mathrm{e}_2 \Downarrow w_3} \text{ eADD}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e_1 \Downarrow w_1 \quad s^{\leftarrow}, s^{\rightarrow}, e_2 \Downarrow w_2 \quad w_3 = w_1 - w_2}{s^{\leftarrow}, s^{\rightarrow}, e_1 \text{ - } \mathrm{e}_2 \Downarrow w_3} \text{ eSUB}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e_1 \Downarrow w_1 \quad \mathrm{s}_{\leftarrow}, s^{\rightarrow}, e_2 \Downarrow w_2 \quad b = w_1 \leq w_2}{s^{\leftarrow}, s^{\rightarrow}, e_1 <= \mathrm{e}_2 \Downarrow w_3} \text{ eLEQ}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e_1 \Downarrow b_1 \quad s^{\leftarrow}, s^{\rightarrow}, e_2 \Downarrow b_2 \quad b_3 = b_1 \text{ and } \mathrm{b}_2}{s^{\leftarrow}, s^{\rightarrow}, e_1 \text{ and } \mathrm{e}_2 \Downarrow b_3} \text{ eAND}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e \Downarrow b \quad b' = \neg b}{s^{\leftarrow}, s^{\rightarrow}, \texttt{not } e \Downarrow b'} \text{ eNOT} \qquad \frac{}{s^{\leftarrow}, s^{\rightarrow}, w \Downarrow w} \text{ eVAL}$$

$$\frac{}{s^{\leftarrow}, s^{\rightarrow}, v \Downarrow s^{\rightarrow}(v)} \text{ eVARF} \qquad \frac{}{s^{\leftarrow}, s^{\rightarrow}, v.\texttt{back} \Downarrow s^{\leftarrow}(v)} \text{ eVARB}$$

A.4.2 Reduction Semantics for Commands

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{skip}; c) \longrightarrow t, r, s^{\leftarrow}, s^{\rightarrow}, c} \text{ rSeq1}$$

$$\frac{t, r, s^{\leftarrow}, s^{\rightarrow}, c_1 \longrightarrow t', r', s^{\leftarrow\prime}, s^{\rightarrow\prime}, c_1'}{t, s^{\leftarrow}, s^{\rightarrow}, c_1; c_2 \longrightarrow t', r', s^{\leftarrow\prime}, s^{\rightarrow\prime}, c_1'; c_2} \text{ rSeq2}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e_1 \Downarrow v \quad s^{\leftarrow}, s^{\rightarrow}, e_2 \Downarrow w}{t, r, s^{\leftarrow}, s^{\rightarrow}, (e_1 \texttt{:=}\ e_2) \longrightarrow t, r, s^{\leftarrow}, (s^{\rightarrow}[v \rightarrow w]), \texttt{skip}} \text{ rAss}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e \Downarrow \mathit{true}}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2) \longrightarrow t, r, s^{\leftarrow}, s^{\rightarrow}, c_1} \text{ rThen}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e \Downarrow \mathit{false}}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2) \longrightarrow t, r, s^{\leftarrow}, s^{\rightarrow}, c_2} \text{ rElse}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e \Downarrow \mathit{true}}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{while}\ e\ \texttt{do}\ c) \longrightarrow t, r, s^{\leftarrow}, s^{\rightarrow}, c; \texttt{while}\ e\ \texttt{do}\ c} \text{ rWhile1}$$

$$\frac{s^{\leftarrow}, s^{\rightarrow}, e \Downarrow \mathit{false}}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{while}\ e\ \texttt{do}\ c) \longrightarrow t, r, s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}} \text{ rWhile2}$$

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{try skip catch}\ c) \longrightarrow t, r, s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}} \text{ rTry1}$$

$$\frac{t, r, s \leftarrow, s^{\rightarrow}, c_1 \longrightarrow t', r', s \leftarrow', s^{\rightarrow\prime}, c_1'}{t, r, s \leftarrow, s^{\rightarrow}, (\texttt{try } c_1 \texttt{ catch } c) \longrightarrow t', r', s^{\leftarrow\prime}, s^{\rightarrow\prime}, \texttt{try } c_1' \texttt{ catch } c} \text{ rTRY2}$$

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{try throw } exc \texttt{ catch } c) \longrightarrow t, r, s^{\leftarrow}, s^{\rightarrow}, c} \text{ rCATCH}$$

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{throw } exc; c) \longrightarrow t, r, s^{\leftarrow}, s^{\rightarrow}, \texttt{throw } exc} \text{ rTHROW}$$

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{undo skip with } c) \longrightarrow (c, s^{\rightarrow}) :: t, r, s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}} \text{ rUNDO1}$$

$$\frac{t, r, s^{\leftarrow}, s^{\rightarrow}, c_1 \longrightarrow t', r', s^{\leftarrow\prime}, s^{\rightarrow\prime}, c_1'}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{undo } c_1 \texttt{ with } c_2) \longrightarrow t', r', s^{\leftarrow\prime}, s^{\rightarrow\prime}, \texttt{undo } c_1' \texttt{ with } c_2} \text{ rUNDO2}$$

$$\frac{}{n :: t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{compensate } n) \longrightarrow n :: t, r, s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}} \text{ rCOMP1}$$

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{compensate } n) \longrightarrow t', (\texttt{compensate } n, s^{\leftarrow}) :: r, s', s^{\rightarrow}, a; c} \text{ rCOMP2}$$

where

$t' =$ **pop_first_c** t n

$s\prime =$ **snd** (**get_first_c** t n)

a = **find_n** t n

$c = \textbf{fst}\ (\textbf{get\_first\_c}\ t\ n)$

$$\frac{}{t, (\texttt{compensate}\ n, s) :: r, s^{\leftarrow}, s^{\rightarrow}, \texttt{skip} \longrightarrow t, r, s, s^{\rightarrow}, \texttt{compensate}\ n}\ \textsc{rComp3}$$

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, (\texttt{purge}\ n) \longrightarrow t', r, s^{\leftarrow}, s^{\rightarrow}, a; \texttt{skip}}\ \textsc{rPURGE}$$

where

$t' = \textbf{super\_pop}\ t\ n$

$a = \textbf{find\_n}\ t\ n$

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, \texttt{scope}\ n\, c \longrightarrow t, r, s^{\leftarrow}, s^{\rightarrow}, c; \texttt{end}\ n}\ \textsc{rScope1}$$

$$\frac{}{t, r, s^{\leftarrow}, s^{\rightarrow}, \texttt{end}\ n \longrightarrow t', r', s^{\leftarrow}, s^{\rightarrow}, \texttt{skip}}\ \textsc{rScope2}$$

where

$t' = \textbf{pop\_n}\ t\ n$