# Compiling and Reasoning about Exceptions and Interrupts

by Joel J Wright, BSc (Hons)

Thesis submitted to The University of Nottingham

for the degree of Doctor of Philosophy, June 2005

# Contents

**1 Background** **1**

1.1 Exceptions 2

1.2 Implementing Exceptions 5

1.3 Reasoning about Exceptions 6

1.4 Interrupts 8

1.5 Implementing Interrupts 9

1.6 Reasoning about Interrupts 10

1.7 Summary 10

**2 A Minimal Language** **12**

2.1 The Language 12

2.2 Big-Step Operational Semantics 13

2.3 Small-Step Operational Semantics 14

2.4 Equivalence of the Implementations 15

2.5 Equivalence of the Semantics 20

2.6 Summary 25

**3 A Simple Compiler** **26**

3.1 Compiler 26

3.2 Compiler Correctness 27

3.3 Generalised Compiler Correctness 30

3.4 Summary 32

**4 Adding Exceptions** **33**

4.1 The Extended Language 33

4.2 Big-Step Operational Semantics 34

4.3 Small-Step Operational Semantics 36

4.4 Equivalence of the Semantics 38

4.5 Summary 48

**5 Compiling Exceptions** **49**

5.1 Compiler 49

5.2 Compiler Correctness 52

5.3 Compiler Variations 60

5.4 Summary 82

**6 *finally*, an Example** **83**

6.1 What is *finally*? 83

6.2 *finally*, a Definition 84

6.3 Correctness of *finally* 85

6.4 Summary 87

**7 Interrupts** **88**

7.1 Big-Step Semantics and Interrupts 88

7.2 Machine Interrupts 91

7.3 Compiler Correctness 94

7.4 Summary 118

**8 Interrupts and *finally*** **119**

8.1 A New Specification of *finally* 119

8.2 Another *finally* Definition 120

8.3 Safety in the Presence of Interrupts 121

8.4 Summary 123

**9 Summary and Further Work** **124**
9.1 Further Work 126

**References** **129**

**A Rule Induction** **134**
A.1 An Example 135

## Abstract

Exceptions and interrupts are important for programming reliable software systems, yet our methods for reasoning about them are weak. In this thesis we explore the problem of compiling and reasoning about exceptions and interrupts in the context of a simple language. We develop the language in three stages, initially without any form of exceptions, then adding exceptions, and finally adding interrupts. At each stage we produce a formal semantics for the language, a compiler, and a proof of its correctness with respect to the semantics. We also reason about a *finally* combinator using the high level semantics, developing a new version at each stage to satisfy our specifications.

## Acknowledgments

I would like to thank the following people for their help throughout the course of this thesis: my supervisor, Graham Hutton, for always pointing me in the right direction, and for always having time to help, no matter how basic the question; Thorsten Altenkirch and Conor McBride for always having a helpful suggestion on escaping the stickier parts of proofs; Simon Peyton Jones and Simon Marlow at Microsoft Research for their guidance on the semantics of exceptions and interrupts in ghc, and helpful comments on their compilation; and all the other members of the FOP group postgrad office: James Chapman, Jon Grattage, Catherine Hope, Mark Jago, Peter Morris, Pablo Noguiera, and Dmitri Schkatov, for making my time in Nottingham fun as well as intellectual.

Special thanks also go to my parents and grandparents, for their unwavering support throughout my time at university, and to my fiancee Katherine, for always providing encouragment, and for tolerating my many bad moods during the production of this thesis.

Finally, I would like to thank Microsoft Research and The University of Nottingham for providing joint funding for my PhD, without which none of this work would have been possible.

Chapter 1

# Background

Events which cause computations to terminate in a non-standard way, known as *exceptions*, are an important consideration in modern programming languages[1]. What should a program do if something goes wrong during its execution? How does a floating point error such as a division by zero, or an attempt to open a file that does not exist, affect the execution of a program? Exception handling mechanisms give the programmer control over these issues, and provide the additional benefit of keeping code for normal excecution separate from code for error handling.

Exceptions come in two basic varieties, those that arise as a direct result of the computation being performed, such as division by zero or an exception raised directly by the programmer, and those that arise externally to a computation, such as a timeout or an interrupt. These two kinds of exceptions are known as *synchronous exceptions* and *asynchronous exceptions* respectively; however, for ease of reference we shall refer to them simply as *exceptions* and *interrupts* throughout this thesis.

In this chapter we briefly review exception and interrupt handling mechanisms in a number of languages, simple methods for implementing both exceptions and interrupts, and some approaches to describing both semantically.

[1]The crash of the Ariane 5 launcher in 1996 is a good example of the importance of proper exception handling. Caused by an uncaught exception [Lio96], and destroying a rocket the European Space Agency had spent $7 billion developing along with its payload of four expensive scientific satellites, this serves as a devastating example of the importance of proper exception handling.

## 1.1 Exceptions

Early programming languages had no means to deal with exceptions: when an unexpected error occured the program simply terminated and returned control to the operating system along with an error message. Early attempts to handle exceptions were rather low level, borrowing ideas from hardware, with a raised exception being indicated using a special return value from a computation. However, this low level approach caused problems because programs rapidly became overrun by tests on return values.

A more modern approach is to provide high level, language based, support for dealing with exceptions, allowing programmers to *throw* exceptions when something unexpected happens and *catch* any errors raised. This high level approach avoids cluttering code with tests for abnormal behaviour, by keeping all code for recovering from errors in discrete *catch* blocks.

We shall continue our brief review of exception handling by taking a look at the exception handling mechanisms available in a number of programming languages. These range in flexibility and scope, from a complete lack of explicit exception handling to high level language support:

- C [KR88] has no specific language features for handling exceptions. Success or failure of a computation is modelled using return values from functions. Return values allow exceptions to be "handled" by testing for possible errors and returning corresponding values, which can then be acted upon; however, runtime exceptions such as stack overflow and division by zero cannot be handled and will result in the program exiting ungracefully.
- PL/I [Sib65] was the first language to allow user programs to handle exceptions. Both language-defined and user-defined exceptions are supported and both are handled using the same language mechanisms.
- ADA [Led81] implements full exception handling and was the first major language to provide users with full interrupt handling. The language specifications for ADA were written in 1977 and finalised in 1980.
- C++ [Str86] exception handling was accepted by the ANSI standardisation committee

in 1990. The design draws ideas from a number of languages including ADA and ML [Led81, MTHM97], and uses a special language contruct to specify the scope of a piece of code which may raise an exception, along with code to recover from a raised exception, the `try-catch` block. Code that is expected to raise an exception is placed in the `try` block and the `catch` block contains one or more *exception handlers* which may deal with specific exceptions or provide a way to deal with whole classes of errors. The basic syntax of a `try-catch` block is as follows:

```
try{
/* Code that may raise an exception */}
catch(){
/* Handler code */}
[catch(){
/* Handler code */}
```

The `try-catch` block executes according to the following rules:

- If the code in the `try` block executes without raising an exception the code in the `catch` blocks is ignored and the program execution continues after the `catch` statements.
- If the code in the `try` block raises an exception and it is handled by one of the `catch` blocks, then the code in that particular `catch` block is executed and the program continues to run normally.
- If the code in the `try` block raises an exception and it is not handled by one of the `catch` blocks, then two possible behaviours apply:
  - If the current `try-catch` block is contained within another, then the exception is propagated to this enclosing set of `catch` blocks.
  - If the current `try-catch` block is the outermost, then the program fails with the unhandled exception.

Exceptions in C++ may be raised only by explicit use of the `throw` command. In general, C++ code cannot catch system generated exceptions such as *DivByZero*, only those explicitly raised by the programmer; however, some implementations allow any exception to be handled. The syntax of a `throw` command in C++ is as follows:

```
throw [expression]
```

- Java [GJSB00] has a similar exception handling syntax to C++; however, Java also has an extension to the `try-catch` mechanism, namely `try-catch-finally`, which executes as follows:

  – If the code in the `try` block executes without raising an exception, then the code in the `finally` block is executed.

  – If the code in the `try` block raises an exception and the exception is handled by one of the `catch` blocks, then the code in the `finally` block is executed after the exception is handled.

  – If the code in the `try` block raises an exception and the exception is not handled by one of the `catch` blocks, then the code in the `finally` block is executed before the exception is propagated.

  This extended catch construct is useful because some pieces of code must always be executed, regardless of whether exceptions are raised. Closing file handles and releasing locks on shared variables are two such examples. The finally block allows repeated code to be avoided because the "clean up" code need be written only once. The basic syntax of the `try-catch-finally` block is as follows:

```
try{
/* Code that may raise an exception */}
catch(){
/* Handler code */}
[catch(){
/* Handler code */}]
[finally{
/* Code to always be executed */}]
```

  Java also includes another useful addition to its exception handling mechanism. The Java compiler detects whether a method may raise an exception, and if so forces the programmer to either handle the exception or specify that the new method may

raise that exception. The only exception to this rule are *RuntimeExceptions* such as *DivByZero* or *StackOverflow* since virtually any piece of code could generate them.

Java code may *throw* an exception using the `throw` method

```
throw new Exception("message")
```

where `Exception` can be any Exception class defined in the Java specification or one explicitly defined by the programmer extending the built-in exception classes.

- Haskell exceptions, as defined in the Haskell 98 standard [Jon03], allow for any type of exception to be thrown and caught, but are limited to use within the IO Monad [Rei98, Pey01]. The types of the functions used to raise and catch exceptions in Haskell are given below:

  $$\begin{aligned} &\mathit{throwIO} :: \mathit{Exception} \rightarrow \mathit{IO}\ a \\ &\mathit{catch} \quad\ :: \mathit{IO}\ a \rightarrow (\mathit{Exception} \rightarrow \mathit{IO}\ a) \rightarrow \mathit{IO}\ a \end{aligned}$$

  It is now also possible for Haskell programs to raise exceptions in purely functional code [PRH$^+$99, MLP99]; however, handling must still be done within the IO monad as previously. This new type of *imprecise* exceptions provides the *throw* function below, which is similar to previously, but does not return an *IO* type:

  $$\mathit{throw} :: \mathit{Exception} \rightarrow a$$

  A full formal semantics has been presented for both exceptions (and interrupts) in GHC, in the articles *Tackling the Awkward Squad* [Pey01] and *Asynchronous Exceptions in Haskell* [MPMR01]. It was these semantics which were the starting point of, and inspiration for, this thesis.

## 1.2 Implementing Exceptions

Exceptions may be implemented in a variety of ways; however, it is helpful to look at a simple method, allowing us to explain the basic ideas. We now present the basic techniques required to implement an exception handling mechanism:

- When the scope of a catch is entered the address of the exception handler code is pushed onto the stack. This is referred to as *marking* the stack.

- If the code executes without raising an exception the handler address is popped off the stack. This is referred to as *unmarking* the stack.
- If the code raises an exception during execution then the current computation is abandoned and any values pushed onto the stack are popped off searching for a handler address.
- If a handler address is found, the exception raised is passed to the exception handler and the exception handler is run.
- If a handler address is not found on the stack, execution is terminated with an *uncaught exception* error.

These rules for implementating exceptions present all the necessary ideas without the complication of optimising techniques. Optimisations include keeping a stack of handler addresses to remove the need for searching the program stack for handlers, although it is still necessary to remove intermediate values from the program stack. Another method for optimising exception handling is so-called *Low Overhead* (or *Zero Overhead*) exceptions [DGL95], which are designed to minimise the run-time cost for utilising the *catch* construct in programs.

## 1.3 Reasoning about Exceptions

Reasoning about programs that include exceptions can be hard due to the fact that a number of "taken for granted" algebraic properties of programs can be rendered invalid due to their presence [AWW90]. For example, the property $x + y = y + x$ is no longer valid if both $x$ and $y$ raise distinct exceptions. In the first case, $x + y$ would evaluate to the exception raised by $x$, however $y + x$ would evaluate to the exception raised by $y$. A number of techniques do, however, exist for describing the semantics of exceptions:

- *weakest preconditions* [LvdS94]. An extension of weakest precondition semantics which includes exceptions has been used to prove some basic algebraic properties of *catch* and other primitives. However, this semantics has not been used for reasoning about programs.

- *game semantics* [Lai01]. A game-semantic approach to describing exceptions in the setting of idealised Algol has been proven to be fully abstract with respect to an operational semantics.
- *direct* [Spi90]. A semantics for exceptions based around a *Maybe* type. The use of this technique for actual programming as well as proving simple properties of primitives using equational reasoning is demonstrated.

The semantics of exceptions have been studied in a number of modern programming languages. We shall now look at some of the work done on reasoning about exceptions, and the settings in which this work was done:

- A considerable amount of work has been done on the semantics of exceptions in Java:
  - A semantics of *throw*, *catch* and *finally* has been studied using type theory. This work formalises the case-based approach of the Java specifications, but is a fairly low-level approach. This semantics has been used to reason about programs using exceptions [Jac01].
  - A semantics for a minimal Java-like language has been produced; however, it has not been applied to reasoning about programs [ALZ01].
  - A semantics of the core features of Java and a corresponding virtual machine have been used to give a proof of correctness of a compiler including exceptions [KN05].
  - An operational semantics of exceptions in Java using a type system that covers both normal and exceptional behaviour has been used to prove a subject reduction theorem (type preservation); however, no further examples of reasoning are given [DV00]. This is part of a larger work on the semantics of Java.
  - An abstract state machine model of exceptions in Java and a JVM have been used to show the correctness of a compiler by showing the equivalence of states in the Java source and corresponding compiled JVM code [BS00].
  - A denotational semantics for Java including exceptions, but excluding concurrency, has been completed and has been stable since 2000. This semantics provides tool support for program verification; however, the examples are mainly Java Card programs of at most hundreds of lines of code [JP03].

- Various research exists on detecting potentially uncaught exceptions in ML. Both an efficient static analysis of potentially uncaught exceptions at compile time, and the use of types to detect potentially uncaught exceptions have been explored [YR02, PL00].
- Haskell has seen much work on exceptions in recent years; for example, see [Pey01] for a summary. Exceptions were originally accessible only from within the IO monad [Rei98]; however, the ability to raise exceptions in pure functional code was added later [PRH+99, MLP99]. Recently [HMPH05] a new approach to concurrency based on *software transactional memory* which incorporates both exceptions and interrupts in a more compositional framework has been implemented. None of this work, however, includes examples of reasoning about programs.

This concludes our brief survey of exception handling mechanisms, their implementation, and semantics. We now move on to consider interrupts in the same way.

## 1.4 Interrupts

Interrupts are usually viewed as a special kind of exception, differing only in that they do not arise as a direct result of a computation. Instead they can occur at any time, regardless of the current instruction, because they arise from concurrently executing program threads as well as from user signals and timeouts. Interrupts are a useful programming tool because they allow messages to be delivered to programs, as well as between program threads, without the need for polling global variables.

A language which features support for interrupts and interrupt-handling will typically provide features for enabling and disabling the delivery of interrupts to a thread as well as a mechanism for threads to raise exceptions in one another. Ada has a full interrupt-handling mechanism, and was the first major language to do so. However, GHC's implementation of Haskell not only has full support for exception and interrupt handling, but is the only implementation of a language to include a full formal semantics for these features. The semantics of interrupts has also been considered in the context of the process calculus CSP, which provides a simple, yet powerful, interrupt combinator.

Work on reasoning about, and programming with, interrupts is likely to become more

important to programmers in the near future, because concurrent programming is becoming increasingly widespread as hardware manufacturers begin to reach the limits of single processor systems, and are exploiting multi-core designs in order to increase performance [Sut05].

## 1.5 Implementing Interrupts

A similar approach to implementing exceptions can be applied to interrupts. In this case we must also keep track of whether interrupts are enabled or disabled rather than just the address of the current handler. We achieve this by pushing the interrupt state onto the stack as we move between interrupt scopes.

- When the scope of an operator which changes the interrupt status (either enabling or disabling interrupts, which is often called *masking* or *unmasking* interrupts) is entered, the current interrupt state is saved on the top of the stack and the interrupt state of the program thread is updated to reflect the scope entered.
- If the scope is exited without an interrupt being received the original state is restored by retrieving it from the stack.
- If an interrupt is received whilst interrupts are unmasked we simply raise an interrupt exception. The interrupt state of the computation is updated to reflect stored states as the stack is popped searching for an exception handler.

Again, this is an introduction to the basic techniques for implementing interrupt handling and various optimisations are possible. For example, we can avoid pushing the interrupt state onto the stack every time we enter a new scope by pushing only the interrupt state when entering a scope which actually *changes* the current interrupt state. In this case we simplify the stack as we require only a single operation that flips the current interrupt status. This can be removed from the stack in the same manner as before if no interrupt is delivered; however, when an interrupt is delivered we need to do less work whilst unwinding the stack.

## 1.6 Reasoning about Interrupts

Most previous work on semantics of interrupts has been within process calculi such as CSP [Hoa85], CCS [Mil82] and the pi-calculus [Mil99], usually within the more general setting of prioritized processes [CLN01]. We shall now look at some examples of reasoning about interrupts as well as some languages in which interrupts have been implemented:

- The process calculus CSP [Hoa85] defines a kind of sequential composition, $P^\wedge Q$, which behaves as a simple form of interrupt; $P^\wedge Q$ behaves as $P$ until interrupted by the first event of $Q$, and from then on behaves as $Q$. Despite its simplicity, this basic form of interrupts can be used to define a number of useful combinators.
- *A Typed Interrupt Calculus* [PM02] describes a type system for ensuring stack boundness in the presence of interrupts. It gives a minimal calculus with support for interrupts and an operational semantics via an abstract machine, and is used to prove that no cycles of handlers interrupt each other.
- GHC's implementations of Haskell is currently the only language to provide both a full asynchronous exception mechanism and a formal semantics for this feature [Pey01, MPMR01]. Interrupting is provided via the *throwTo* function and the *ThreadID* data type, which are extensions to Concurrent Haskell [PGF96, GF96]. When a new program thread is forked a *ThreadID* is generated, which can be used in conjunction with the *throwTo* function to allow any thread to raise an arbitrary exception in another. The types and functions required to use interrupts in Haskell are given below:

  **data** *ThreadID*

  $\mathit{forkIO} \quad :: \mathit{IO}\ () \rightarrow \mathit{IO}\ \mathit{ThreadID}$

  $\mathit{throwTo} :: \mathit{ThreadID} \rightarrow \mathit{Exception} \rightarrow \mathit{IO}\ ()$

## 1.7 Summary

In this chapter we have seen basic methods for implementing both exception handling and interrupt handling features, as well examples of the syntax for these features in real

programming languages. We have also seen that, even though work on reasoning about exceptions, and programs which make use of them, exists for a variety of settings, reasoning about interrupts in the context of programming languages is rather sparse.

A full interrupt-handling mechanism has recently been added to the purely functional language Haskell, complete with a full formal semantics for both exceptions and interrupts. This is the first example of a formal semantics for interrupts, and combined with the well-known benefits of the functional paradigm for reasoning about programs [Bir98], is the reason that a Haskell-inspired language with exceptions and interrupts will be the main focus of our work in this thesis. It should be noted that much of the content of Chapters 2 to 5 is based on the author's previously published work [HW04]. These chapters do, however, also include interesting and previously unpublished alternative views of the problem of compiling and reasoning about a simple exception handling language. It should also be noted that in some of the examples presented we use Haskell notation, however we assume that our expressions are finite, and our functions strict.

Chapter 2

# A Minimal Language

We wish to study exceptions in a minimal setting, to allow us to study the effects of exceptions without getting caught up in the details of a real language. In this chapter we begin by defining a minimal language to which we can add exceptions later. We shall also define a big-step and a small-step operational semantics for the language and give a proof of their equivalence, both by defining the language and semantics in terms of Haskell data types and functions, and directly on the formal rules for the semantics.

## 2.1 The Language

The langauge we shall look at initially is simply integers with addition, and is described by the following BNF grammar, in which $\mathbb{Z}$ is the set of integers:

$$\begin{aligned} \mathbb{E} &::= \mathbb{Z} \\ &\mid \ \mathbb{E} + \mathbb{E} \end{aligned}$$

It should be noted that although we do not explicitly allow brackets in our grammar, we shall use them in expressions to avoid ambiguity. Because we shall be using Haskell for some of the early examples and reasoning we also define the syntax of the language in terms of a Haskell data type:

```
data Expr = Val Int
          | Add Expr Expr
```

## 2.2 Big-Step Operational Semantics

In order to reason about the behaviour of a language we need to describe its behaviour in terms of a semantics. We begin by defining a big-step operational semantics (sometimes called a natural semantics) [Gun92]. Formally, we define an evaluation relation $\Downarrow$ on expressions as the least relation satisfying the following inference rules:

$$\frac{}{\overline{n} \Downarrow \overline{n}}\ \textsc{Val}_b \qquad \frac{x \Downarrow \overline{n} \qquad y \Downarrow \overline{m}}{x + y \Downarrow \overline{n + m}}\ \textsc{Add}_b$$

This semantics states that a number, $\overline{n}$, is completely evaluated, and that the evaluation of an expression $x + y$ is the result of adding the evaluation of $x$ to the evaluation of $y$. We use the overline notation, $\overline{n}$, to denote an evaluated integer.

Using the semantics we can show how expressions are evaluated. Big step semantics allow us to draw *evaluation trees* which describe the evaluation of an expression. For example, consider the expression $1 + (2 + 3)$. Then, the following evaluation tree describes the evaluation of this expression using the rules defined in the big-step semantics:

$$\frac{\frac{}{1 \Downarrow 1}\ \textsc{Val}_b \qquad \frac{\frac{}{2 \Downarrow 2}\ \textsc{Val}_b \qquad \frac{}{3 \Downarrow 3}\ \textsc{Val}_b}{2 + 3 \Downarrow 5}\ \textsc{Add}_b}{1 + (2 + 3) \Downarrow 6}\ \textsc{Add}_b$$

We can also define our big-step semantics as a Haskell function:

$$\begin{aligned} &\mathit{eval} && :: \mathit{Expr} \to \mathit{Int} \\ &\mathit{eval}\ (\mathit{Val}\ n) && = n \\ &\mathit{eval}\ (\mathit{Add}\ x\ y) && = (\mathit{eval}\ x) + (\mathit{eval}\ y) \end{aligned}$$

This function allows us to evaluate expressions defined in terms of the Haskell data type definition of our language in the same way that the big-step operational semantics allows

us to evaluate expressions defined in terms of the BNF grammar, with the added advantage that the evaluations can be run in a Haskell environment such as ghci [Ghc] or Hugs [Hug], and properties can be tested using *QuickCheck* [CH00]. We give a trace of the evaluation function below, using a Haskell data type definition of the previous example, $Add\ (Val\ 1)\ (Add\ (Val\ 2)\ (Val\ 3))$:

$$
\begin{array}{ll}
 & eval\ (Add\ (Val\ 1)\ (Add\ (Val\ 2)\ (Val\ 3))) \\
= & \{\ \text{definition of}\ eval\ \} \\
 & eval\ (Val\ 1) + eval\ (Add\ (Val\ 2)\ (Val\ 3)) \\
= & \{\ \text{definition of}\ eval\ \} \\
 & eval\ (Val\ 1) + (eval\ (Val\ 2) + eval\ (Val\ 3)) \\
= & \{\ \text{definition of}\ eval\ \} \\
 & 1 + (eval\ (Val\ 2) + eval\ (Val\ 3)) \\
= & \{\ \text{definition of}\ eval\ \} \\
 & 1 + (2 + eval\ (Val\ 3)) \\
= & \{\ \text{definition of}\ eval\ \} \\
 & 1 + (2 + 3) \\
= & \{\ \text{definition of}\ +\ \} \\
 & 6
\end{array}
$$

## 2.3 Small-Step Operational Semantics

A small-step operational semantics [Plo81] allows us to study the effects of exceptions at a finer level of detail than the big-step operational semantics allows, because it describes the progress of execution in terms of simple transitions of expressions. The small step semantics of our language is defined as the least relation $\rightarrow$ on expressions satisfying the following inference rules:

$$
\frac{}{\overline{n} + \overline{m} \rightarrow \overline{n + m}}\ \textsc{Add}_s1
\qquad
\frac{x \rightarrow x'}{x + y \rightarrow x' + y}\ \textsc{Add}_s2
\qquad
\frac{y \rightarrow y'}{x + y \rightarrow x + y'}\ \textsc{Add}_s3
$$

These rules state that two evaluated numbers $\overline{n}$ and $\overline{m}$ may make a transition to a single evaluated number $\overline{n+m}$ (the result of adding $n$ and $m$), and that an addition expression $x + y$ may make a transition if either the $x$ or $y$ can make a transition. We now give a trace of the execution of our simple example using the small-step semantics:

$$
\begin{array}{ll}
 & 1 + (2 + 3) \\
\rightarrow & \{ \textsc{Add}_s 3 \text{ and } \textsc{Add}_s 1 \text{ rules} \} \\
 & 1 + 5 \\
\rightarrow & \{ \textsc{Add}_s 1 \text{ rule} \} \\
 & 6
\end{array}
$$

The small-step semantics is more complex in Haskell, because we must keep track of all possible transitions. In a deterministic language we will reach only a single value, however there may be a number of paths (e.g. evaluating the left-hand side of an addition before evaluating the right-hand side and vice versa).

$$
\begin{array}{ll}
\mathit{trans} & :: \mathit{Expr} \rightarrow [\mathit{Expr}] \\
\mathit{trans}\ (\mathit{Val}\ n) & = [\,] \\
\mathit{trans}\ (\mathit{Add}\ (\mathit{Val}\ n)\ (\mathit{Val}\ m)) & = [\mathit{Val}\ (n + m)] \\
\mathit{trans}\ (\mathit{Add}\ x\ y) & = [\mathit{Add}\ x'\ y \mid x' \leftarrow \mathit{trans}\ x] \\
 & \quad +\!\!+ [\mathit{Add}\ x\ y' \mid y' \leftarrow \mathit{trans}\ y]
\end{array}
$$

We will not give an example of evaluation using this function as even simple examples get very large and complex. However, it is useful for testing purposes.

## 2.4 Equivalence of the Implementations

In order to prove the two semantics equivalent we must prove that for every expression both semantics produce the same result. By tracing every possible small-step execution of the expression and comparing each result to the big-step evaluation of the expression, we can determine whether the semantics differ. In this section we prove our Haskell implementations of the semantics equivalent.

**Theorem 2.4.1 (Equivalence of the Semantics).**

$$and\ [e' = Val\ (eval\ e) \mid e' \leftarrow leaves\ (exec\ e)]$$

That is, the result of every possible evaluation in the small-step semantics produces produces the same result as that produced by an evaluation in the big-step semantics. The function *exec* fully evaluates the operational semantics into a tree, storing final results as leaves, and the function *leaves* retrieves the results from the tree:

```
data Tree = Node Expr [Tree]

exec   :: Expr → Tree
exec e = Node e [exec e' | e' ← trans e]

leaves              :: Tree → [Expr]
leaves Node e [ ]   = [e]
leaves Node e xs = concat (map leaves xs)
```

To prove theorem 2.4.1 we first prove that a transition in the operational semantics does not alter the value of the expression. Again this notion is captured as a list comprehension, which states that every expression obtained from a legal transition in the operational semantics evaluates to the same value as the original expression:

**Lemma 2.4.2.**

$$and\ [eval\ e = eval\ e' \mid e' \leftarrow trans\ e]$$

Before we move to formally proving this result, we can test it informally using QuickCheck. Since we defined the equivalence of the two semantics in Haskell, this is a simple case of running lemma 2.4.2 and theorem 2.4.1 as QuickCheck properties:

```
prop_trans   :: Expr → Bool
prop_trans e = and [eval e' = eval e | e' ← trans e]
prop_equiv   :: Expr → Bool
prop_equiv e = and [e' = Val (eval e) | e' ← leaves (exec e)]
```

Note that it is also necessary to write a random expression generator for QuickCheck, but we will not discuss it here. For further details see *QuickCheck: a lightweight tool for random*

*testing of Haskell programs* [CH00]. We can now test our properties using QuickCheck. When run in Hugs we see that they both pass 100 random tests:

```
Main> quickCheck prop_trans
OK, passed 100 tests.

Main> quickCheck prop_equiv
OK, passed 100 tests.
```

Having gained confidence from testing our properties using QuickCheck, we can now move on to proving them formally. We start by proving lemma 2.4.2 by induction on $e$. The base cases of $Val\ n$ and $Add\ (Val\ n)\ (Val\ m)$ are trivial, so we only prove the inductive case, $Add\ x\ y$, which can be done as follows:

*Proof.*

$$
\begin{aligned}
& \quad and\ [eval\ (Add\ x\ y) = eval\ e' \mid e' \leftarrow trans\ (Add\ x\ y)] \\
= & \quad \{ \text{ definition of } eval,\ trans \ \} \\
& \quad and\ [eval\ x + eval\ y = eval\ e' \mid e' \leftarrow [Add\ x'\ y \mid x' \leftarrow trans\ x] \\
& \qquad +\!\!+ [Add\ x\ y' \mid y' \leftarrow trans\ y]] \\
= & \quad \{ \text{ both cases are similar } \} \\
& \quad and\ [eval\ x + eval\ y = eval\ e' \mid e' \leftarrow [Add\ x'\ y \mid x' \leftarrow trans\ x]] \\
= & \quad \{ \text{ substituting for } e' \ \} \\
& \quad and\ [eval\ x + eval\ y = eval\ (Add\ x'\ y) \mid x' \leftarrow trans\ x] \\
= & \quad \{ \text{ definition of } eval \ \} \\
& \quad and\ [eval\ x + eval\ y = eval\ x' + eval\ y \mid x' \leftarrow trans\ x] \\
= & \quad \{ \text{ cancelling the } eval\ y \ \} \\
& \quad and\ [eval\ x = eval\ x' \mid x' \leftarrow trans\ x] \\
= & \quad \{ \text{ induction hypothesis } \} \\
& \quad True
\end{aligned}
$$

□

In order to prove theorem 2.4.1, we use induction over the size of the expression $e$, defined simply as the number of constructors in $e$. Again we do not show the proof for the base cases and proceed straight to the inductive case, *Add x y*:

*Proof.*

$$
\begin{aligned}
& \quad \mathit{and}\ [e' = \mathit{Val}\ (\mathit{eval}\ e) \mid e' \leftarrow \mathit{leaves}\ (\mathit{exec}\ e)] \\
= & \quad \{ \text{ definition of } \mathit{exec} \ \} \\
& \quad \mathit{and}\ [e' = \mathit{Val}\ (\mathit{eval}\ e) \mid e' \leftarrow \mathit{leaves}\ (\mathit{Node}\ e\ [\mathit{exec}\ e'' \mid e'' \leftarrow \mathit{trans}\ e])] \\
= & \quad \{ \text{ definition of } \mathit{leaves} \ \} \\
& \quad \mathit{and}\ [e' = \mathit{Val}\ (\mathit{eval}\ e) \mid e' \leftarrow \mathit{concat}\ [\mathit{leaves}\ (\mathit{exec}\ e'') \mid e'' \leftarrow \mathit{trans}\ e]] \\
= & \quad \{ \text{ list comprehensions } \} \\
& \quad \mathit{and}\ [e' = \mathit{Val}\ (\mathit{eval}\ e) \mid e'' \leftarrow \mathit{trans}\ e, e' \leftarrow \mathit{leaves}\ (\mathit{exec}\ e'')] \\
= & \quad \{ \text{ list comprehensions } \} \\
& \quad \mathit{and}\ [\mathit{and}\ [e' = \mathit{Val}\ (\mathit{eval}\ e) \mid e' \leftarrow \mathit{leaves}\ (\mathit{exec}\ e'')] \mid e'' \leftarrow \mathit{trans}\ e] \\
= & \quad \{ \text{ lemma 2.4.2 } \} \\
& \quad \mathit{and}\ [\mathit{and}\ [e' = \mathit{Val}\ (\mathit{eval}\ e'') \mid e' \leftarrow \mathit{leaves}\ (\mathit{exec}\ e'')] \mid e'' \leftarrow \mathit{trans}\ e] \\
= & \quad \{ \text{ induction hypothesis, lemma 2.4.3 } \} \\
& \quad \mathit{and}\ [\mathit{True} \mid e'' \leftarrow \mathit{trans}\ e] \\
= & \quad \{ \text{ definition of } \mathit{and}, \text{ list comprehensions } \} \\
& \quad \mathit{True}
\end{aligned}
$$

□

Lemma 2.4.3 states that transitions always decrease the size of an expression:

**Lemma 2.4.3.**

$$and\ [size\ e' < size\ e \mid e' \leftarrow trans\ e]$$

where the function *size* is defined as follows:

$$\begin{aligned}
&size && :: Expr \rightarrow Int \\
&size\ (Val\ _) && = 1 \\
&size\ (Add\ x\ y) && = 1 + (size\ x) + (size\ y)
\end{aligned}$$

We prove lemma 2.4.3 by induction on $e$. Note that again we do not show the proofs for the base cases *Val n* and *Add* (*Val n*) (*Val m*), and so proceed straight to proving the case *Add x y*:

*Proof.*

$$\begin{aligned}
& and\ [size\ e' < size\ (Add\ x\ y) \mid e' \leftarrow trans\ (Add\ x\ y)] \\
= \quad & \{ \text{ definition of } trans \ \} \\
& and\ ([size\ (Add\ x'\ y) < size\ (Add\ x\ y) \mid x' \leftarrow trans\ x] \\
& \quad +\!\!+ [size\ (Add\ x\ y') < size\ (Add\ x\ y) \mid y' \leftarrow trans\ y]) \\
= \quad & \{ \text{ both cases are similar } \} \\
& and\ [size\ (Add\ x'\ y) < size\ (Add\ x\ y) \mid x' \leftarrow trans\ x] \\
= \quad & \{ \text{ definition of } size \ \} \\
& and\ [(1 + size\ x' + size\ y) < (1 + size\ x + size\ y) \mid x' \leftarrow trans\ x] \\
= \quad & \{ \text{ cancel on either side } \} \\
& and\ [size\ x' < size\ x \mid x' \leftarrow trans\ x] \\
= \quad & \{ \text{ induction hypothesis } \} \\
& True
\end{aligned}$$

□

In conclusion, by proving theorem 2.4.1 we have shown our Haskell implementations of the two semantics to be equivalent.

## 2.5 Equivalence of the Semantics

We can also prove the equivalence of our two semantics directly without referring to their implementations in Haskell, by using rule induction (which is explained in more detail in Appendix A). Before we can proceed we need to define the reflexive transitive closure of the small-step semantics which formalises the idea of making zero or more transitions. This is defined in the standard way by the following two rules:

### 2.5.1 Reflexive Transitive Closure

$$\frac{}{x \to^* x}\ \textsc{NoTrans} \qquad \frac{x \to x'' \qquad x'' \to^* x'}{x \to^* x'}\ \textsc{Trans}$$

We can now proceed to prove the big and small-step operational semantics equivalent.

**Theorem 2.5.1 (Equivalence of the Semantics).**

$$x \Downarrow \overline{n} \;\Leftrightarrow\; x \to^* \overline{n}$$

The proof of this theorem comprises two parts, *soundness* and *completeness*;

### 2.5.2 Soundness

The first step in proving the equivalence of our two operational semantics is *soundness*, which states that if an expression can evaluate in the big-step semantics then the same expression must be able to evaluate to the same result using the small-step semantics:

**Lemma 2.5.2.**

$$x \Downarrow \overline{n} \;\Rightarrow\; x \to^* \overline{n}$$

*Proof.* By rule induction on $x \Downarrow \overline{n}$.

Case: $\textsc{Val}_b$

We need to verify that $\overline{n} \to^* \overline{n}$, which is immediate because $\to^*$ is reflexive.

Case: ADD$_b$

We can assume $x \Downarrow \overline{n}$ and $y \Downarrow \overline{m}$ by rule ADD$_b$ for $\Downarrow$, and that $x \rightarrow^* \overline{n}$ and $y \rightarrow^* \overline{m}$ as our induction hypotheses. We now verify that $x + y \rightarrow^* \overline{n+m}$ as follows:

$$
\begin{array}{ll}
 & x + y \\
\rightarrow^* & \{ \text{ assumption that } x \rightarrow^* \overline{n}, \text{ repeated use of ADD}_s2 \text{ rule } \} \\
 & \overline{n} + y \\
\rightarrow^* & \{ \text{ assumption that } y \rightarrow^* \overline{m}, \text{ repeated use of ADD}_s3 \text{ rule } \} \\
 & \overline{n} + \overline{m} \\
\rightarrow & \{ \text{ ADD}_s1 \text{ rule } \} \\
 & \overline{n+m}
\end{array}
$$

□

To be fully formal we also need to verify the first and second steps in the calculation above.

**Lemma 2.5.3.**

$$x \rightarrow^* \overline{n} \;\Rightarrow\; x + y \rightarrow^* \overline{n} + y$$

*Proof.* by rule induction on $x \rightarrow^* \overline{n}$

Case: NOTRANS

We need to verify that $\overline{n} + y \rightarrow^* \overline{n} + y$, which is immediate because $\rightarrow^*$ is reflexive.

Case: TRANS

We can assume $x \rightarrow x'$ and $x' \rightarrow^* \overline{n}$ by TRANS rule for $\rightarrow^*$ (here we are exploiting the fact that an existential quantification on the left of an implication is equivalent to a universal quantification at the outer level), and that $x' + z \rightarrow^* \overline{n} + z$ as our induction hypothesis, and verify that $x + y \rightarrow^* \overline{n} + y$ using the following simple calculation:

$$\begin{array}{ll} & x + y \\ \rightarrow & \{ \text{ assumption that } x \rightarrow x', \text{ Application of rule } \textsc{Add}_s 2 \ \} \\ & x' + y \\ \rightarrow^* & \{ \text{ assumption that } \forall z.\ x' + z \rightarrow^* \overline{n} + z \ \} \\ & \overline{n} + y \end{array}$$

□

### 2.5.3 Completeness

The second step in proving the equivalence of our two operational semantics is *completeness*, which states that if an expression can evaluate in the small-step semantics then the same expression must be able to evaluate to the same value using the big-step semantics:

**Lemma 2.5.4.**

$$x \rightarrow^* \overline{n} \ \Rightarrow \ x \Downarrow \overline{n}$$

*Proof.* by rule induction on $x \rightarrow^* \overline{n}$

Case: NoTrans

We verify that $\overline{n} \Downarrow \overline{n}$, which follows immediately from rule Val for $\Downarrow$.

Case: Trans

We can assume $x \rightarrow x'$ and $x' \rightarrow^* \overline{n}$ by rule Trans for $\rightarrow^*$, and that $x' \Downarrow \overline{n}$ as our induction hypothesis. The fact that $x \Downarrow \overline{n}$ is then immediate by the following lemma, which we prove separately. □

**Lemma 2.5.5.**

$$x \rightarrow x' \wedge x' \Downarrow \overline{n} \ \Rightarrow \ x \Downarrow \overline{n}$$

*Proof.* by rule induction on $x \rightarrow x'$

Case: $\text{ADD}_s 1$

We verify that $\overline{n+m} \Downarrow \overline{o} \Rightarrow \overline{n} + \overline{m} \Downarrow \overline{o}$ by the following calculation, using two auxiliary results:

$$
\begin{array}{ll}
 & \overline{n} + \overline{m} \Downarrow \overline{o} \\
\Leftrightarrow & \{ \text{ lemma 2.5.7 } \} \\
 & \exists\, a, b.\ \overline{n} \Downarrow \overline{a} \wedge \overline{m} \Downarrow \overline{b} \wedge a + b = o \\
\Leftrightarrow & \{ \text{ lemma 2.5.6 } \} \\
 & \exists\, a, b.\ a = n \wedge b = m \wedge a + b = o \\
\Leftrightarrow & \{ \text{ logic } \} \\
 & n + m = o \\
\Leftrightarrow & \{ \text{ lemma 2.5.6 } \} \\
 & \overline{n+m} \Downarrow \overline{o}
\end{array}
$$

Case: $\text{ADD}_s 2$

We can assume $x \rightarrow x'$ by rule $\text{ADD}_s 2$, and that $x' \Downarrow \overline{n} \Rightarrow x \Downarrow \overline{n}$ as our induction hypothesis, and verify that $x' + y \Downarrow \overline{n+m} \Rightarrow x + y \Downarrow \overline{n+m}$ as follows, also using the auxiliary results:

$$
\begin{array}{ll}
 & x' + y \Downarrow \overline{n+m} \\
\Leftrightarrow & \{ \text{ lemma 2.5.7 } \} \\
 & \exists\, a, b.\ x' \Downarrow \overline{a} \wedge y \Downarrow \overline{b} \wedge a + b = n + m \\
\Rightarrow & \{ \text{ induction hypothesis } \} \\
 & \exists\, a, b.\ x \Downarrow \overline{a} \wedge y \Downarrow \overline{b} \wedge a + b = n + m \\
\Leftrightarrow & \{ \text{ lemma 2.5.7 } \} \\
 & x + y \Downarrow \overline{n+m}
\end{array}
$$

Case: $\text{ADD}_s 3$

Similarly to the previous case. $\square$

### 2.5.4 Required Lemmas

**Lemma 2.5.6.**

$$\overline{n} \Downarrow \overline{m} \;\;\Leftrightarrow\;\; n = m$$

*Proof.* $\Rightarrow$ The only way to produce a valid judgement of the form $\overline{n} \Downarrow x$ is to use rule $\textsc{Val}_b$ for $\Downarrow$, which implies the truth of this direction. □

*Proof.* $\Leftarrow$ Rule $\textsc{Val}_b$ for $\Downarrow$ □

**Lemma 2.5.7.**

$$x + y \Downarrow \overline{n} \Leftrightarrow \exists\, a, b.\; (x \Downarrow \overline{a} \wedge y \Downarrow \overline{b} \wedge a + b = n)$$

*Proof.* $\Rightarrow$ The only way to produce a valid judgement of the form $x + y \Downarrow \overline{n}$ is to use rule $\textsc{Add}_b$ for $\Downarrow$, which implies the truth of this direction. □

*Proof.* $\Leftarrow$

$$\begin{array}{ll}
 & \exists\, a, b.\; (x \Downarrow \overline{a} \wedge y \Downarrow \overline{b} \wedge a + b = n) \Rightarrow x + y \Downarrow \overline{n} \\
\Leftrightarrow & \{ \text{ logic } \} \\
 & \forall\, a, b.\; (x \Downarrow \overline{a} \wedge y \Downarrow \overline{b} \wedge a + b = n \Rightarrow x + y \Downarrow \overline{n}) \\
\Leftrightarrow & \{ \text{ eliminating } n \text{ } \} \\
 & \forall\, a, b.\; (x \Downarrow \overline{a} \wedge y \Downarrow \overline{b} \Rightarrow x + y \Downarrow \overline{a + b}) \\
\Leftrightarrow & \{ \text{ rule } \textsc{Add}_b \text{ for } \Downarrow \} \\
 & \mathit{True}
\end{array}$$

□

The conclusion of this proof, in conjunction with lemma 2.5.2, completes a proof of theorem 2.5.1.

## 2.6 Summary

At this point both the direct rule inductive proofs and the Haskell style proofs are relatively straightforward. The Haskell style has the advantage of actually being executable, allowing us to test results we may wish to prove, however we run the risk of introducing errors in the implementation. Conversely the rule inductive proofs operate directly on the semantics but require additional lemmas. Whether both styles of proof remain useful in the presence of exceptions remains to be seen in later chapters of this thesis.

CHAPTER 3

# A Simple Compiler

In this chapter we produce a simple compiler for our minimal language and prove this compiler correct with respect to the semantics presented in the previous chapter. We shall compile expressions to a simple stack language, and produce an interpreter for executing the compiled code. Using the definition of this interpreter we shall prove the compiler equivalent to our big step semantics.

## 3.1 Compiler

In order to define a compiler for our minimal language we require two stack machine instructions: an operator which pushes a number onto the stack and an operator which adds the top two stack elements together. These instructions are sufficient to execute any expression in our language and are given by the following Haskell data type:

**data** $Op = PUSH\ Int \mid ADD$

The compiler takes as input an expression and produces code to be run on the stack machine, where code is simply a list of instructions to be executed. A compiled value is a push operation which pushes the corresponding number onto the stack, and a compiled addition expression is the list of operations consisting of compiling the first argument of the *Add* followed by compiling the second argument and finally an *ADD* operation.

$$
\begin{aligned}
&\textbf{type } \mathit{Code} && = [\,\mathit{Op}\,] \\
&\mathit{compile} && :: \mathit{Expr} \rightarrow \mathit{Code} \\
&\mathit{compile}\ (\mathit{Val}\ n) && = [\,\mathit{PUSH}\ n\,] \\
&\mathit{compile}\ (\mathit{Add}\ x\ y) && = \mathit{compile}\ x +\!\!+\ \mathit{compile}\ y +\!\!+\ [\,\mathit{ADD}\,]
\end{aligned}
$$

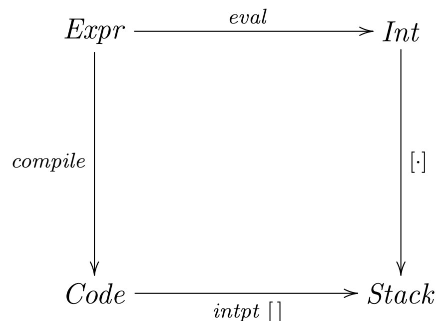
In order to execute the compiled code we define an interpret function. This function takes a stack (a list of integers) and some code to execute and produces the stack which results from executing the operations in the code.

$$
\begin{aligned}
&\textbf{type } \mathit{Stack} && = [\,\mathit{Int}\,] \\
&\mathit{intpt} && :: \mathit{Stack} \rightarrow \mathit{Code} \rightarrow \mathit{Stack} \\
&\mathit{intpt}\ s\ [\,] && = s \\
&\mathit{intpt}\ s\ (\mathit{PUSH}\ x : \mathit{ops}) && = \mathit{intpt}\ (x : s)\ \mathit{ops} \\
&\mathit{intpt}\ (y : x : s)\ (\mathit{ADD} : \mathit{ops}) && = \mathit{intpt}\ (x + y : s)\ \mathit{ops}
\end{aligned}
$$

Note that we assume the stack for the $\mathit{ADD}$ instruction has the correct form of $(y : x : s)$. The stack not having this form corresponds to a *stack underflow* error, but our compiler ensures that this situation can never arise in practice.

## 3.2 Compiler Correctness

We have produced a compiler for our arithmetic expression language and now proceed to show it correct with respect to our semantics. The correctness of the compiler is described by the following commuting diagram:



That is, if we compile an expression and interpret it with an initially empty stack, we should get the result of evaluating that expression pushed onto an empty stack. We need only prove one of the semantics equivalent to the interpreted code since we have proved both semantics equivalent. The correctness of the compiler with respect to the big step semantics is captured in the following QuickCheck property:

$$\begin{aligned} &\mathit{prop_Compile} \quad :: \mathit{Expr} \rightarrow \mathit{Bool} \\ &\mathit{prop_Compile}\ e = \mathit{intpt}\ [\,]\ (\mathit{compile}\ e) = [\mathit{eval}\ e] \end{aligned}$$

This test passes the 100 random cases generated by QuickCheck, so we now formally prove this property.

However, this form of the correctness statement causes the inductive hypothesis to be too weak, so we shall consider a more general version, where the code is executed with an arbitrary initial stack, and not the empty stack:

**Theorem 3.2.1 (compiler correctness).**

$$\mathit{intpt}\ s\ (\mathit{compile}\ e) = (\mathit{eval}\ e) : s$$

*Proof.* by induction on $e$

Case: $\mathit{Val}\ n$

$$\begin{aligned} & \quad \mathit{intpt}\ s\ (\mathit{compile}\ (\mathit{Val}\ n)) \\ = & \quad \{ \text{ definition of } \mathit{compile} \} \\ & \quad \mathit{intpt}\ s\ [\mathit{PUSH}\ n] \\ = & \quad \{ \text{ definition of } \mathit{intpt} \} \\ & \quad \mathit{intpt}\ (n : s)\ [\,] \\ = & \quad \{ \text{ definition of } \mathit{intpt} \} \\ & \quad n : s \\ = & \quad \{ \text{ definition of } \mathit{eval} \} \\ & \quad \mathit{eval}\ (\mathit{Val}\ n) : s \end{aligned}$$

Case: $(\mathit{Add}\ x\ y)$

$$\mathit{intpt}\ s\ (\mathit{compile}\ (\mathit{Add}\ x\ y))$$

$$
\begin{array}{ll}
= & \{ \text{ definition of } \mathit{compile} \ \} \\
 & \mathit{intpt}\ s\ (\mathit{compile}\ x \mathbin{+\!\!+} \mathit{compile}\ y \mathbin{+\!\!+} [\mathit{ADD}]) \\
= & \{ \text{ lemma 3.2.2 (see below) } \} \\
 & \mathit{intpt}\ (\mathit{intpt}\ (\mathit{intpt}\ s\ (\mathit{compile}\ x))\ (\mathit{compile}\ y))\ [\mathit{ADD}] \\
= & \{ \text{ induction hypothesis } \} \\
 & \mathit{intpt}\ (\mathit{intpt}\ (\mathit{eval}\ x : s)\ (\mathit{compile}\ y))\ [\mathit{ADD}] \\
= & \{ \text{ induction hypothesis } \} \\
 & \mathit{intpt}\ (\mathit{eval}\ y : \mathit{eval}\ x : s)\ [\mathit{ADD}] \\
= & \{ \text{ definition of } \mathit{intpt} \ \} \\
 & \mathit{intpt}\ ((\mathit{eval}\ x + \mathit{eval}\ y) : s)\ [\,] \\
= & \{ \text{ definition of } \mathit{intpt} \ \} \\
 & (\mathit{eval}\ x + \mathit{eval}\ y) : s \\
= & \{ \text{ definition of } \mathit{eval} \ \} \\
 & \mathit{eval}\ (\mathit{Add}\ x\ y) : s
\end{array}
$$

□

The proof above makes use of lemma 3.2.2, which states that in order to interpret the concatenation of two pieces of code, we can first interpret the first piece of code, then use the resulting stack to interpret the second piece of code.

**Lemma 3.2.2 (interpretation distributivity).**

$$\mathit{intpt}\ s\ (\mathit{xs} \mathbin{+\!\!+} \mathit{ys}) = \mathit{intpt}\ (\mathit{intpt}\ s\ \mathit{xs})\ \mathit{ys}$$

We prove lemma 3.2.2 by induction on $\mathit{xs}$. We do not show the proof for the base case, [], and proceed straight to the inductive cases, $\mathit{PUSH}\ n : \mathit{xs}$ and $\mathit{ADD} : \mathit{xs}$.

*Proof.* by induction on $\mathit{xs}$

Case: $\mathit{PUSH}\ n : \mathit{xs}$

$$
\begin{array}{ll}
 & \mathit{intpt}\ s\ (\mathit{PUSH}\ n : \mathit{xs} \mathbin{+\!\!+} \mathit{ys}) \\
= & \{ \text{ definition of } \mathit{intpt} \ \} \\
 & \mathit{intpt}\ (n : s)\ (\mathit{xs} \mathbin{+\!\!+} \mathit{ys})
\end{array}
$$

$$
\begin{aligned}
= \quad & \{ \text{ induction hypothesis } \} \\
& \quad intpt\ (intpt\ (n : s)\ xs)\ ys \\
= \quad & \{ \text{ definition of } intpt \ \} \\
& \quad intpt\ (intpt\ s\ (PUSH\ n : xs))\ ys
\end{aligned}
$$

Case: $ADD : xs$

We need only prove the case when the stack is of the correct form, $(x : y : s)$. If the stack is not of this form, the equation to be proved is trivially true, because the result of both sides is undefined ($\bot$), provided that we assume that *intpt* is strict in its stack argument ($intpt\ \bot\ ops = \bot$). This extra strictness assumption could be avoided by representing and managing stack underflow explicitly, rather than doing so implicitly using $\bot$.

$$
\begin{aligned}
& \quad intpt\ (x : y : s)\ (ADD : xs +\!\!+\ ys) \\
= \quad & \{ \text{ definition of } intpt \ \} \\
& \quad intpt\ ((x + y) : s)\ (xs +\!\!+\ ys) \\
= \quad & \{ \text{ induction hypothesis } \} \\
& \quad intpt\ (intpt\ ((x + y) : s)\ xs)\ ys \\
= \quad & \{ \text{ definition of } intpt \ \} \\
& \quad intpt\ (intpt\ (y : x : s)\ (ADD : xs))\ ys
\end{aligned}
$$

□

## 3.3 Generalised Compiler Correctness

By further generalising our correctness theorem we can eliminate the need for the interpretation distributivity lemma, avoiding the issue of stack underflow.

**Theorem 3.3.1 (generalised compiler correctness).**

$$intpt\ s\ (compile\ e +\!\!+\ ops) = intpt\ ((eval\ e) : s)\ ops$$

That is, compiling an expression and then executing the resulting code appended together with arbitrary additional code (which we can think of as a *code continuation*) gives

the same result as pushing the value of the expression to give a new stack, which is then used to interpret the additional code. Note that in the case where $s = ops = [\,]$, theorem 3.3.1 simplifies to our original statement of correctness.

*Proof.* By induction on $e :: Expr$

Case: $e = Val\ n$

$$
\begin{array}{ll}
 & intpt\ s\ (compile\ (Val\ n) \mathbin{+\!\!+} ops) \\
= & \{\ \text{definition of } compile\ \} \\
 & intpt\ s\ ([PUSH\ n] \mathbin{+\!\!+} ops) \\
= & \{\ \text{definition of } intpt\ \} \\
 & intpt\ (n : s)\ ops \\
= & \{\ \text{definition of } eval\ \} \\
 & intpt\ (eval\ (Val\ n) : s)\ ops
\end{array}
$$

Case: $e = Add\ x\ y$

$$
\begin{array}{ll}
 & intpt\ s\ (compile\ (Add\ x\ y) \mathbin{+\!\!+} ops) \\
= & \{\ \text{definition of } compile\ \} \\
 & intpt\ s\ (compile\ x \mathbin{+\!\!+} compile\ y \mathbin{+\!\!+} [ADD] \mathbin{+\!\!+} ops) \\
= & \{\ \text{induction hypothesis}\ \} \\
 & intpt\ (eval\ x : s)\ (compile\ y \mathbin{+\!\!+} [ADD] \mathbin{+\!\!+} ops) \\
= & \{\ \text{induction hypothesis}\ \} \\
 & intpt\ (eval\ y : eval\ x : s)\ ([ADD] \mathbin{+\!\!+} ops) \\
= & \{\ \text{definition of } intpt\ \} \\
 & intpt\ (eval\ x + eval\ y : s)\ ops \\
= & \{\ \text{definition of } eval\ \} \\
 & intpt\ (eval\ (Add\ x\ y) : s)\ ops
\end{array}
$$

□

The above proof has two benefits: it avoids the problem of stack underflow, and is somewhat shorter than our previous proof. This demonstrates the fact that generalising a theorem in the appropriate manner can often considerably simplify its proof.

## 3.4 Summary

We produced a compiler for our minimal language and proved that execution of the compiled code produces the same results as directly evaluating an expression using our semantics. We have also demonstrated that the *right* formulation of a correctness statement is just as important as a *correct* statement. In fact at the end of chapter 6 we will introduce a modified compiler which itself takes a code continuation, much like our generalised correctness statement. This has additional benefits from an efficiency and reasoning point of view. It is now time to add exceptions to our language and see what effects they have on reasoning about the semantics and compiler.

Chapter 4

# Adding Exceptions

In this chapter we extend our minimal language to include exceptions. We shall define both a small-step and big-step semantics for our extended language and provide a proof of their equivalence. By contrasting these proofs with those in Chapter 2 we can begin to see how adding exceptions to a language affects reasoning about its semantics. In Chapter 6 we will consider reasoning about programs in the presence of exceptions.

## 4.1 The Extended Language

To add support for exceptions we extend our language with two new primitives, *Throw*, which simply models an arbitrary raised exception, and *Catch*, which takes two expressions as arguments, the first being the expression to be evaluated, and the second the expression to evaluate in the case an exception is raised by the evaluation of the first. We also add support for sequencing of expressions, which simply evaluates its first argument followed by its second, and which is necessary for some particular examples that we wish to consider later.

$$
\begin{aligned}
\mathbb{E} ::= {} & \mathbb{Z} \\
\mid {} & \mathit{throw} \\
\mid {} & \mathbb{E} + \mathbb{E} \\
\mid {} & \mathit{catch}\ \mathbb{E}\ \mathbb{E} \\
\mid {} & \mathbb{E} ; \mathbb{E}
\end{aligned}
$$

We also extend the Haskell data type definition accordingly:

$$\begin{aligned}\textbf{data}\ \mathit{Expr} = \ & \mathit{Val}\ \mathit{Int} \\ \mid\ & \mathit{Throw} \\ \mid\ & \mathit{Add}\ \mathit{Expr}\ \mathit{Expr} \\ \mid\ & \mathit{Catch}\ \mathit{Expr}\ \mathit{Expr} \\ \mid\ & \mathit{Seq}\ \mathit{Expr}\ \mathit{Expr}\end{aligned}$$

There are three points to note about our extended language. Firstly, that we do not distinguish types of exceptions, we simply have a *Throw* expression representing all types of exceptions. Secondly, *Catch* takes as its arguments two expressions, not an expression and a collection of exception handling functions. And finally, we add sequencing of expressions, which will allow us to express more interesting combinators later in this thesis.

We make these three simplifications to better enable us to study the basic *behaviour* of programs which make use of exceptions, as we are not interested in writing practical programs. Bearing this goal in mind, it is unnecessary for us to distinguish between exceptions and therefore to consider which exception was raised when handling one, as these language features all deal with producing a result and do not affect *how* the program executes.

## 4.2 Big-Step Operational Semantics

We add support for the new primitives to the big-step semantics. Note that we extend the + operator with two rules to propagate uncaught exceptions, i.e. if either the left or right operand of a + evaluates to a raised exception, the whole addition raises an exception. A *catch* expression evaluates its first argument, and if it raises no exception is the result of the *catch*. If the first argument raises an exception the result of a *catch* is the result of evaluating the second argument. A sequence of expressions evaluates its first argument, and if it raises an exception is the result of the sequence, otherwise the resulting value is discarded and the result of the sequence is the result of its second argument.

$$\frac{}{\overline{n} \Downarrow \overline{n}}\ \textsc{Val}_b \qquad \frac{}{\mathit{throw} \Downarrow \mathit{throw}}\ \textsc{Throw}_b$$

$$\frac{x \Downarrow \mathit{throw}}{x + y \Downarrow \mathit{throw}}\ \textsc{Add}_b 1 \qquad \frac{y \Downarrow \mathit{throw}}{x + y \Downarrow \mathit{throw}}\ \textsc{Add}_b 2 \qquad \frac{x \Downarrow \overline{n} \qquad y \Downarrow \overline{m}}{x + y \Downarrow \overline{n + m}}\ \textsc{Add}_b 3$$

$$\frac{x \Downarrow \overline{n}}{\mathit{catch}\ x\ y \Downarrow \overline{n}}\ \textsc{Catch}_b 1 \qquad \frac{x \Downarrow \mathit{throw} \qquad y \Downarrow v}{\mathit{catch}\ x\ y \Downarrow v}\ \textsc{Catch}_b 2$$

$$\frac{x \Downarrow \overline{n} \qquad y \Downarrow v}{x; y \Downarrow v}\ \textsc{Seq}_b 1 \qquad \frac{x \Downarrow \mathit{throw}}{x; y \Downarrow \mathit{throw}}\ \textsc{Seq}_b 2$$

The Haskell definition of the big-step semantics is now somewhat more complicated:

```
eval               :: Expr → Maybe Int
eval (Val n)       = Just n
eval (Add x y)     = case eval x of
                       Nothing → Nothing
                       Just a  → case eval y of
                                   Nothing → Nothing
                                   Just b  → Just (a + b)
eval (Throw)       = Nothing
eval (Catch x h)   = case eval x of
                       Just a  → Just a
                       Nothing → eval h
eval (Seq x y)     = case (eval x) of
                       Nothing → Nothing
                       Just a  → eval y
```

We can exploit the fact that *Maybe* forms a *monad* [Wad92], and its membership of the *MonadPlus* class, to express the definition of *eval* more clearly and concisely using Haskell's "do notation" [LP95]:

$$
\begin{array}{lll}
eval & :: & Expr \rightarrow Maybe\ Int \\
eval\ (Val\ n) & = & return\ n \\
eval\ (Add\ x\ y) & = & \mathbf{do}\ a \leftarrow eval\ x \\
 & & \quad\ b \leftarrow eval\ y \\
 & & \quad\ return\ (a + b) \\
eval\ (Throw) & = & mzero \\
eval\ (Catch\ x\ h) & = & eval\ x\ `mplus`\ eval\ h \\
eval\ (Seq\ x\ y) & = & \mathbf{do}\ eval\ x \\
 & & \quad\ eval\ y
\end{array}
$$

The *MonadPlus* class adds two new monad operations, *mzero* and *mplus*. Using these in combination we can model a raised exception as *mzero*, which always fails, and catch exceptions using *mplus*. The function *mplus* for *Maybe* takes two monadic operations — if the first succeeds the result is simply returned, otherwise the second is evaluated, which is exactly the behaviour we require for *catch*.

Although the monadic definition of *eval* is more clearly expressed, we shall use our original definition for the purposes of proofs.

## 4.3 Small-Step Operational Semantics

We also produce a new small-step semantics for our extended language, which propagates and handles exceptions in a similar way to the big-step semantics:

$$
\frac{}{\overline{n} + \overline{m} \rightarrow \overline{n+m}}\ \textsc{Add}_s1 \qquad \frac{}{throw + y \rightarrow throw}\ \textsc{Add}_s2
$$

$$
\frac{}{x + throw \rightarrow throw}\ \textsc{Add}_s3 \qquad \frac{x \rightarrow x'}{x + y \rightarrow x' + y}\ \textsc{Add}_s4 \qquad \frac{y \rightarrow y'}{x + y \rightarrow x + y'}\ \textsc{Add}_s5
$$

$$\frac{}{catch\ \overline{n}\ y \to \overline{n}}\ \textsc{Catch}_s1 \qquad \frac{}{catch\ throw\ y \to y}\ \textsc{Catch}_s2$$

$$\frac{x \to x'}{catch\ x\ y \to catch\ x'\ y}\ \textsc{Catch}_s3$$

$$\frac{}{\overline{n}; y \to y}\ \textsc{Seq}_s1 \qquad \frac{}{throw; y \to throw}\ \textsc{Seq}_s2 \qquad \frac{x \to x'}{x; y \to x'; y}\ \textsc{Seq}_s3$$

We can also express the small-step semantics as the Haskell function below. This is useful as it allows us to perform some simple testing of theorems using QuickCheck, but is even now starting to get too cumbersome for pen and paper proofs:

```
trans                :: Expr -> [Expr]
trans (Val n)        = []
trans (Add x y)      = case (x, y) of
                         (Val n, Val m) -> [Val (n + m)]
                         (_, Throw)     -> [Throw] ++ [Add x' y | x' <- trans x]
                         (Throw, _)     -> [Throw] ++ [Add x y' | y' <- trans y]
                         (_, _)         -> [Add x' y | x' <- trans x]
                                             ++ [Add x y' | y' <- trans y]
trans (Throw)        = []
trans (Catch x h)    = case x of
                         Val n          -> [Val n]
                         Throw          -> [h]
                         _              -> [Catch x' h | x' <- trans x]
trans (Seq x y)      = case x of
                         Val n          -> [y]
                         Throw          -> [Throw]
                         _              -> [Seq x' y | x' <- trans x]
```

The cases for *Seq* and *Catch* are simpler than that for the *Add* expression because both can only make transitions on their first argument until it is fully evaluated. This means that the only source of multiple transition paths is an *Add* expression.

## 4.4 Equivalence of the Semantics

We now wish to prove the extended semantics equivalent. We begin be stating a theorem describing what we mean for the semantics to be equivalent:

**Theorem 4.4.1 (equivalence of the semantics).**

$$x \Downarrow v \Leftrightarrow x \rightarrow^* v$$

That is, an expression may evaluate to a value in the big-step semantics if and only if it can make some number of steps to that same value using the transitions described in the small-step semantics. As in Chapter 2 we prove this equivalence using rule induction and the proof consists of two parts, *soundness* and *completeness*.

**Lemma 4.4.2 (soundness).**

$$x \Downarrow v \quad \Rightarrow \quad x \rightarrow^* v$$

That is, if an expression evaluates to a value in our big-step semantics then it is possible to make transitions from the expression to that value using our small-step semantics. We show separately that the appropriate notion of *value* for our language is the set $\mathbb{Z} \cup \{throw\}$ (lemmas 4.4.5 and 4.4.6).

*Proof.* by rule induction on $x \Downarrow v$. There are nine cases to consider, one for each rule in the big-step semantics:

Case: $\text{VAL}_b$

We need to verify that $\overline{n} \rightarrow^* \overline{n}$, which is immediate because $\rightarrow^*$ is reflexive.

Case: $\text{THROW}_b$

We need to verify that $throw \rightarrow^* throw$, which is again immediate.

Case: $\text{ADD}_b 1$

We can assume $x \Downarrow throw$, and that $x \rightarrow^* throw$ as our induction hypothesis. We now verify that $x + y \rightarrow^* throw$ as follows:

$$
\begin{array}{ll}
 & x + y \\
\to^* & \{ \text{ assumption that } x \to^* \mathit{throw}, \text{ repeated use of } \textsc{Add}_s4 \text{ rule } \} \\
 & \mathit{throw} + y \\
\to & \{ \text{ application of } \textsc{Add}_s2 \text{ rule } \} \\
 & \mathit{throw}
\end{array}
$$

The case for $\textsc{Add}_b2$ is similar.

Case: $\textsc{Add}_b3$

We can assume $x \Downarrow \overline{n}$ and $y \Downarrow \overline{m}$, and that $x \to^* \overline{n}$ and $y \to^* \overline{m}$ as our induction hypotheses. We now verify that $x + y \to^* \overline{n + m}$ as follows:

$$
\begin{array}{ll}
 & x + y \\
\to^* & \{ \text{ assumption that } x \to^* \overline{n}, \text{ repeated use of } \textsc{Add}_s4 \text{ rule } \} \\
 & \overline{n} + y \\
\to^* & \{ \text{ assumption that } y \to^* \overline{m}, \text{ repeated use of } \textsc{Add}_s5 \text{ rule } \} \\
 & \overline{n} + \overline{m} \\
\to & \{ \textsc{Add}_s1 \text{ rule } \} \\
 & \overline{n + m}
\end{array}
$$

(For formal verification of the repeated steps see lemma 2.5.3)

Case: $\textsc{Catch}_b1$

We can assume that $x \Downarrow \overline{n}$, and that $x \to^* \overline{n}$ as our induction hypothesis. We now verify that $\mathit{catch}\ x\ y \to^* \overline{n}$ as follows:

$$
\begin{array}{ll}
 & \mathit{catch}\ x\ y \\
\to^* & \{ \text{ assumption that } x \to^* \overline{n}, \text{ repeated use of } \textsc{Catch}_s3 \text{ rule } \} \\
 & \mathit{catch}\ \overline{n}\ y \\
\to & \{ \text{ application of } \textsc{Catch}_s1 \text{ rule } \} \\
 & \overline{n}
\end{array}
$$

Case: $\text{CATCH}_b 2$

We can assume that $x \Downarrow \mathit{throw}$ and $y \Downarrow v$, and that $x \rightarrow^* \mathit{throw}$ and $y \rightarrow^* v$ as our induction hypotheses. We now verify that $\mathit{catch}\ x\ y \rightarrow^* v$ as follows:

$$
\begin{array}{ll}
 & \mathit{catch}\ x\ y \\
\rightarrow^* & \{ \text{ assumption that } x \rightarrow^* \mathit{throw}\text{, repeated use of } \text{ADD}_s 4 \text{ rule } \} \\
 & \mathit{catch}\ \mathit{throw}\ y \\
\rightarrow & \{ \text{ application of rule } \text{CATCH}_s 2 \ \} \\
 & y \\
\rightarrow^* & \{ \text{ assumption that } y \rightarrow^* v \ \} \\
 & v
\end{array}
$$

Case: $\text{SEQ}_b 1$

We can assume that $x \Downarrow \overline{n}$ and that $y \Downarrow v$, and that $x \rightarrow^* \overline{n}$ and $y \rightarrow^* v$ as our induction hypotheses. We now verify that $x; y \rightarrow^* v$ as follows:

$$
\begin{array}{ll}
 & x; y \\
\rightarrow^* & \{ \text{ assumption that } x \rightarrow^* \overline{n}\text{, repeated use of } \text{SEQ}_s 3 \ \} \\
 & \overline{n}; y \\
\rightarrow & \{ \text{ application of rule } \text{SEQ}_s 1 \ \} \\
 & y \\
\rightarrow^* & \{ \text{ assumption that } y \rightarrow^* v \ \} \\
 & v
\end{array}
$$

Case: $\text{SEQ}_b 2$

We can assume that $x \Downarrow \mathit{throw}$, and that $x \rightarrow^* \mathit{throw}$ as our induction hypothesis. We now verify that $x; y \rightarrow^* \mathit{throw}$ as follows:

$x; y$

$\rightarrow^*$ { assuption that $x \rightarrow^* \mathit{throw}$, repeated use of $\text{SEQ}_s3$ rule }

$\mathit{throw}; y$

$\rightarrow$ { application of $\text{SEQ}_s2$ rule }

$\mathit{throw}$

□

This concludes the proof that our big-step semantics is sound with respect to our small-step semantics. We now wish to prove that our big-step semantics is complete with respect to our small step semantics. This is captured by lemma 4.4.3.

**Lemma 4.4.3 (completeness).**

$$x \rightarrow^* v \quad \Rightarrow \quad x \Downarrow v$$

That is, if an expression can make transitions to a value in our small-step semantics then it is also possible for the expression to evaluate to that value in our big-step semantics.

*Proof.* by rule induction on $x \rightarrow^* v$

Case: NoTrans

We need to verify that $v \Downarrow v$, which is immediate by rules $\text{VAL}_b$ and $\text{THROW}_b$, combined with lemma 4.4.6, which states that a value produced by a sequence of transitions in the small-step semantics is a member of the set $\mathbb{Z} \cup \{\mathit{throw}\}$.

Case: Trans

We can assume that $x \rightarrow x'$ and $x' \rightarrow^* v$, and that $x' \Downarrow v$ as our induction hypothesis. We show in lemma 4.4.6 that a value produced by a sequence of transitions in the small-step semantics is a member of the set $\mathbb{Z} \cup \{\mathit{throw}\}$; the fact that $x \Downarrow v$ is then immediate by the following lemma, which we prove separately.

□

**Lemma 4.4.4.**

$$x \to x' \wedge x' \Downarrow v \quad \Rightarrow \quad x \Downarrow v$$

*Proof.* by rule induction on $x \to x'$. There are eleven cases to consider, one for each rule in the small-step semantics.

Case: $\textsc{Add}_s 1$

We need to verify that $\overline{n+m} \Downarrow \overline{o} \Rightarrow \overline{n} + \overline{m} \Downarrow \overline{o}$ by the following calculation (in fact we show an equivalence) using two auxiliary results:

$$
\begin{array}{cl}
 & \overline{n} + \overline{m} \Downarrow \overline{o} \\
\Leftrightarrow & \{ \text{ lemma 2.5.7 } \} \\
 & \exists\, a, b.\ \overline{n} \Downarrow \overline{a} \wedge \overline{m} \Downarrow \overline{b} \wedge a + b = o \\
\Leftrightarrow & \{ \text{ lemma 2.5.6 } \} \\
 & \exists\, a, b.\ a = n \wedge b = m \wedge a + b = o \\
\Leftrightarrow & \{ \text{ logic } \} \\
 & n + m = o \\
\Leftrightarrow & \{ \text{ lemma 2.5.6 } \} \\
 & \overline{n+m} \Downarrow \overline{o}
\end{array}
$$

Case: $\textsc{Add}_s 2$

The fact that $\mathit{throw} \Downarrow \mathit{throw} \Rightarrow x + \mathit{throw} \Downarrow \mathit{throw}$ is immediate by rule $\textsc{Add}_b 2$.

Case: $\textsc{Add}_s 3$

Similarly to the previous case applying rule $\textsc{Add}_b 1$.

Case: $\textsc{Add}_s 4$

We can assume $x \to x'$, and that $x' \Downarrow \overline{n} \Rightarrow x \Downarrow \overline{n}$ and $x' \Downarrow \mathit{throw} \Rightarrow x \Downarrow \mathit{throw}$ as our induction hypotheses. We now verify that $x' + y \Downarrow \overline{m} \Rightarrow x + y \Downarrow \overline{m}$ and $x' + y \Downarrow \mathit{throw} \Rightarrow x + y \Downarrow \mathit{throw}$ as follows, also using auxiliary results:

- $x' + y \Downarrow \overline{n + m}$

  $\Leftrightarrow$ { lemma 2.5.7 }

  $\exists\, a, b.\ x' \Downarrow \overline{a} \wedge y \Downarrow \overline{b} \wedge a + b = n + m$

  $\Rightarrow$ { induction hypotheses }

  $\exists\, a, b.\ x \Downarrow \overline{a} \wedge y \Downarrow \overline{b} \wedge a + b = n + m$

  $\Leftrightarrow$ { lemma 2.5.7 }

  $x + y \Downarrow \overline{n + m}$

- $x' + y \Downarrow \mathit{throw}$

  $\Leftrightarrow$ { lemma 4.4.7 }

  $x' \Downarrow \mathit{throw} \vee y \Downarrow \mathit{throw}$

  $\Rightarrow$ { induction hypotheses }

  $x \Downarrow \mathit{throw} \vee y \Downarrow \mathit{throw}$

  $\Leftrightarrow$ { lemma 4.4.7 }

  $x + y \Downarrow \mathit{throw}$

Case: $\textsc{Add}_s 5$

We can assume $y \rightarrow y'$, and that $y' \Downarrow \overline{m} \Rightarrow y \Downarrow \overline{m}$ and $y' \Downarrow \mathit{throw} \Rightarrow y \Downarrow \mathit{throw}$ as our induction hypotheses. We now verify that $\overline{n} + y' \Downarrow \overline{n + m} \Rightarrow \overline{n} + y \Downarrow \overline{n + m}$ and $\overline{n} + y' \Downarrow \mathit{throw} \Rightarrow \overline{n} + y \Downarrow \mathit{throw}$ as follows:

- $\overline{n} + y' \Downarrow \overline{n + m}$

  $\Leftrightarrow$ { lemma 2.5.7 }

  $\exists\, a, b.\ x \Downarrow \overline{a} \wedge y' \Downarrow \overline{b} \wedge a + b = n + m$

  $\Rightarrow$ { induction hypothesis }

  $\exists\, a, b.\ x \Downarrow \overline{a} \wedge y \Downarrow \overline{b} \wedge a + b = n + m$

  $\Leftrightarrow$ { lemma 2.5.7 }

  $x + y \Downarrow \overline{n + m}$

- $x + y' \Downarrow \mathit{throw}$

  $\Leftrightarrow$ { lemma 4.4.7 }

  $x \Downarrow \mathit{throw} \vee y' \Downarrow \mathit{throw}$

  $\Rightarrow$ { induction hypotheses }

$$
\begin{aligned}
& x \Downarrow \mathit{throw} \vee y \Downarrow \mathit{throw} \\
\Leftrightarrow \quad & \{ \text{ lemma 4.4.7 } \} \\
& x + y \Downarrow \mathit{throw}
\end{aligned}
$$

Case: $\textsc{Catch}_s 1$

The fact that $\overline{n} \Downarrow \overline{n} \Rightarrow \mathit{catch}\ \overline{n}\ h$ is immediate by rule $\textsc{Catch}_b 1$.

Case: $\textsc{Catch}_s 2$

- The fact that $y \Downarrow \overline{n} \Rightarrow \mathit{catch}\ \mathit{throw}\ y \Downarrow \overline{n}$ is immediate by rule $\textsc{Catch}_b 2$.
- The fact that $y \Downarrow \mathit{throw} \Rightarrow \mathit{catch}\ \mathit{throw}\ y \Downarrow \mathit{throw}$ is also immediate by rule $\textsc{Catch}_b 2$.

Case: $\textsc{Catch}_s 3$

We can assume that $x \rightarrow x'$, and that $x' \Downarrow \overline{n} \Rightarrow x \Downarrow \overline{n}$ and $x' \Downarrow \mathit{throw} \Rightarrow x \Downarrow \mathit{throw}$ as our indeuction hypotheses. We now verify that $\mathit{catch}\ x'\ h \Downarrow \overline{n} \Rightarrow \mathit{catch}\ x\ h \Downarrow \overline{n}$ and $\mathit{catch}\ x'\ h \Downarrow \mathit{throw} \Rightarrow \mathit{catch}\ x\ h \Downarrow \mathit{throw}$ as follows:

- $$
\begin{aligned}
& \mathit{catch}\ x'\ h \Downarrow \overline{n} \\
\Leftrightarrow \quad & \{ \text{ lemma 4.4.8 } \} \\
& \exists\, a.\ x' \Downarrow \overline{a} \vee (x' \Downarrow \mathit{throw} \wedge h \Downarrow \overline{a}) \wedge a = n \\
\Rightarrow \quad & \{ \text{ induction hyptheses } \} \\
& \exists\, a.\ x \Downarrow \overline{a} \vee (x \Downarrow \mathit{throw} \wedge h \Downarrow \overline{a}) \wedge a = n \\
\Leftrightarrow \quad & \{ \text{ lemma 4.4.8 } \} \\
& \mathit{catch}\ x\ h \Downarrow \overline{n}
\end{aligned}
$$
- $$
\begin{aligned}
& \mathit{catch}\ x'\ h \Downarrow \mathit{throw} \\
\Leftrightarrow \quad & \{ \text{ lemma 4.4.9 } \} \\
& x' \Downarrow \mathit{throw} \wedge h \Downarrow \mathit{throw} \\
\Rightarrow \quad & \{ \text{ induction hypotheses } \} \\
& x \Downarrow \mathit{throw} \wedge h \Downarrow \mathit{throw} \\
\Leftrightarrow \quad & \{ \text{ lemma 4.4.9 } \} \\
& \mathit{catch}\ x\ h \Downarrow \mathit{throw}
\end{aligned}
$$

Case: $\textsc{Seq}_s1$

- The fact that $y \Downarrow \overline{m} \Rightarrow \overline{n}; y \Downarrow \overline{m}$ is immediate by rule $\textsc{Seq}_b1$.
- The fact that $y \Downarrow \mathit{throw} \Rightarrow \overline{n}; y \Downarrow \mathit{throw}$ is also immediate by rule $\textsc{Seq}_b1$.

Case: $\textsc{Seq}_s2$

The fact that $\mathit{throw} \Downarrow \mathit{throw} \Rightarrow \mathit{throw}; y \Downarrow \mathit{throw}$ is immediate by rule $\textsc{Seq}_b2$.

Case: $\textsc{Seq}_s3$

We can assume that $x \rightarrow x'$, and that $x' \Downarrow \overline{n} \Rightarrow x \Downarrow \overline{n}$ and $x' \Downarrow \mathit{throw} \Rightarrow x \Downarrow \mathit{throw}$ as our induction hypotheses. We now verify that $x'; y \Downarrow \overline{m} \Rightarrow x; y \Downarrow \overline{m}$ and $x'; y \Downarrow \mathit{throw} \Rightarrow x; y \Downarrow \mathit{throw}$ as follows:

- $$\begin{aligned} & x'; y \Downarrow \overline{m} \\ \Leftrightarrow \quad & \{\text{ lemma 4.4.10 }\} \\ & \exists\, a, b.\ (x' \Downarrow \overline{a} \wedge y \Downarrow \overline{b} \wedge b = m) \\ \Rightarrow \quad & \{\text{ induction hypotheses }\} \\ & \exists\, a, b.\ (x \Downarrow \overline{a} \wedge y \Downarrow \overline{b} \wedge b = m) \\ \Leftrightarrow \quad & \{\text{ lemma 4.4.10 }\} \\ & x; y \Downarrow \overline{m} \end{aligned}$$
- $$\begin{aligned} & x'; y \Downarrow \mathit{throw} \\ \Leftrightarrow \quad & \{\text{ lemma 4.4.11 }\} \\ & x' \Downarrow \mathit{throw} \vee \exists\, a.\ (x' \Downarrow \overline{a} \wedge y \Downarrow \mathit{throw}) \\ \Rightarrow \quad & \{\text{ induction hypotheses }\} \\ & x \Downarrow \mathit{throw} \vee \exists\, a.\ (x \Downarrow \overline{a} \wedge y \Downarrow \mathit{throw}) \\ \Leftrightarrow \quad & \{\text{ lemma 4.4.11 }\} \\ & x; y \Downarrow \mathit{throw} \end{aligned}$$

□

By combining lemma 4.4.2 and lemma 4.4.3 we have shown that the two semantics are equivalent, namely $x \Downarrow v \;\Leftrightarrow\; x \rightarrow^* v$.

### 4.4.1 Required Lemmas

The following lemmas are required for the above proof:

**Lemma 4.4.5.**

$$x \Downarrow v \quad \Rightarrow \quad v \in \mathbb{Z} \cup \{\mathit{throw}\}$$

That is, evaluation in the big-step semantics always produces either a number or *throw*.

*Proof.* By induction on $x$ □

**Lemma 4.4.6.**

$$x \not\to \quad \Rightarrow \quad x = \mathit{throw} \lor (\exists\, n.\; x = \overline{n})$$

That is, if an expression $x$ can make no further transitions then $x$ is either *throw* or a number. We cannot prove this statement as it stands, so we rewrite into a form we can prove directly by induction on $x$. We begin by formally defining $x \not\to$ :

$$x \not\to \quad = \neg\, \exists x'.\; x \to x'$$

and now:

$$
\begin{array}{ll}
 & x \not\to \quad \Rightarrow \quad x = \mathit{throw} \lor \exists\, n.\; x = \overline{n} \\
= & \{ \text{ definition of } \not\to \ \} \\
 & \neg\, (\exists x'.\; x \to x') \quad \Rightarrow \quad x = \mathit{throw} \lor \exists\, n.\; x = \overline{n} \\
= & \{ \text{ logic } \neg\, A \Rightarrow B \lor C \;=\; A \lor B \lor C \ \} \\
 & \exists x'.\; x \to x' \lor x = \mathit{throw} \lor \exists\, n.\; x = \overline{n}
\end{array}
$$

*Proof.* By induction on $x$ □

**Lemma 4.4.7.**

$$x + y \Downarrow \mathit{throw} \quad \Leftrightarrow \quad x \Downarrow \mathit{throw} \lor y \Downarrow \mathit{throw}$$

That is, an expression of the form $x + y$ raises an exception if and only if $x$ raises an exception or $y$ raises an exception.

*Proof.* $\Rightarrow$ The only way to produce a valid judgement of the form $x + y \Downarrow \mathit{throw}$ is to use either rule $\text{ADD}_b1$ or $\text{ADD}_b2$, which implies the truth of this direction. □

*Proof.* $\Leftarrow$ Directly by inspection of the $\text{ADD}_b$ rules for $\Downarrow$. □

**Lemma 4.4.8.**

$$\mathit{catch}\ x\ h \Downarrow \overline{n} \iff x \Downarrow \overline{n} \vee (x \Downarrow \mathit{throw} \wedge h \Downarrow \overline{n}))$$

That is, an expression of the form *catch x h* evaluates to a number if and only if $x$ evaluates to a number, or if $x$ evaluates to *throw* and $h$ evaluates to a number.

*Proof.* $\Rightarrow$ The only way to produce a valid judgement of the form $\mathit{catch}\ x\ h \Downarrow \overline{n}$ is to use either rule $\text{CATCH}_b1$ or $\text{CATCH}_b2$, which implies the truth of this direction. □

*Proof.* $\Leftarrow$ Directly by inspection of the $\text{CATCH}_b$ rules for $\Downarrow$. □

**Lemma 4.4.9.**

$$\mathit{catch}\ x\ h \Downarrow \mathit{throw} \iff x \Downarrow \mathit{throw} \wedge h \Downarrow \mathit{throw}$$

That is, an expresion of the form *catch x h* evaluates to *throw* if and only if both $x$ and $h$ evaluate to *throw*.

*Proof.* $\Rightarrow$ The only way to produce a valid judgement of the form $\mathit{catch}\ x\ h \Downarrow \mathit{throw}$ is to use rule $\text{CATCH}_b2$ for $\Downarrow$, which implies the truth of this direction. □

*Proof.* $\Leftarrow$ Directly by inspection of rule $\text{CATCH}_b2$. □

**Lemma 4.4.10.**

$$x; y \Downarrow \overline{m} \iff \exists a.\ (x \Downarrow \overline{a} \wedge y \Downarrow \overline{m})$$

That is, a sequence $x; y$ evaluates to a number if and only if both $x$ and $y$ evaluate to a number.

*Proof.* $\Rightarrow$ The only way to produce a valid judgement of the form $x; y \Downarrow \overline{m}$ is to use rule $\text{SEQ}_b1$ for $\Downarrow$, which implies the truth of this direction. □

*Proof.* $\Leftarrow$ Directly by inspection of rule $\text{SEQ}_b1$. □

**Lemma 4.4.11.**

$$x; y \Downarrow \textit{throw} \;\;\Leftrightarrow\;\; x \Downarrow \textit{throw} \lor \exists\, a.\, (x \Downarrow \overline{a} \land y \Downarrow \textit{throw})$$

That is, a sequence $x; y$ evaluates to *throw* if and only if either $x$ or $y$ evaluates to *throw*.

*Proof.* $\Rightarrow$ The only way to produce a valid judgement of the form $x; y \Downarrow \textit{throw}$ is to use either rule $\text{SEQ}_b1$ or $\text{SEQ}_b2$, which implies the truth of this direction. □

*Proof.* $\Leftarrow$ Directly by inspection of rules $\text{SEQ}_b1$ and $\text{SEQ}_b2$. □

## 4.5 Summary

In this chapter we have proved our big and small-step semantics equivalent for our extended language, comprising integers, addition, throw and catch. These proofs are still readily understood, simply involving more cases because we now have to deal with two kinds of results, rather than actually being *harder*. We can now extend our stack machine and compiler to include the new primitives and prove its equivalence to the semantics.

Chapter 5

# Compiling Exceptions

In this chapter we extend our original compiler to include support for the *catch*, *throw* and sequencing language features, and again prove this compiler correct with respect to our big-step semantics. The new language features will require both an extension to our stack machine operations and a new type of stack item. At the end of the chapter we look at two alternative versions of the compiler which handle recovering from exceptions in different ways to that presented first. These variations are interesting from a reasoning point of view, and in fact we shall use one of them in future chapters.

## 5.1 Compiler

In order to extend our compiler to include support for exceptions and sequencing we require four new stack machine instructions:

$$\textbf{data}\ Op = \ldots \mid \mathit{THROW} \mid \mathit{MARK}\ \mathit{Code} \mid \mathit{UNMARK} \mid \mathit{POP}$$

We now have a machine instruction to *THROW* an exception, a *MARK* instruction for placing exception handling code onto the stack, and *UNMARK* instruction for removing handler code from the stack, and a *POP* instruction for removing a value from the top of the stack.

Because we now mark the stack with handler code, we also require new type for stack items. We modify our existing notion of stack items to include both integers, which are now tagged, and handler code:

$$\textbf{type}\ \mathit{Stack} = [\mathit{Item}]$$
$$\textbf{data}\ \mathit{Item}\ = \mathit{VAL}\ \mathit{Int} \mid \mathit{HAN}\ \mathit{Code}$$

Our existing compiler for expressions from Chapter 3 is extended to work with the new kinds of expressions as follows:

$$\begin{aligned}
&\mathit{comp}\ (\mathit{Throw}) &&= [\mathit{THROW}] \\
&\mathit{comp}\ (\mathit{Catch}\ x\ h) &&= [\mathit{MARK}\ (\mathit{comp}\ h)] +\!\!+\ \mathit{comp}\ x +\!\!+ [\mathit{UNMARK}] \\
&\mathit{comp}\ (\mathit{Seq}\ x\ y) &&= (\mathit{comp}\ x) +\!\!+ (\mathit{POP} : (\mathit{comp}\ y))
\end{aligned}$$

That is, a *Throw* expression simply compiles to a *THROW* instruction. In order to compile a *Catch* expression, we mark the stack with the compiled expression handler $h$, followed by the compiled expression $x$, and an *UNMARK* which removes the handler code from the stack if the execution of $x$ was successful. A sequence of expressions is simply the compiled code for the expression $x$, followed by a *POP* to remove the resulting value if execution of $x$ was successful, then the compiled code for the expression $y$. A simple example of output from the compiler is given below:

$$\begin{aligned}
&\quad \mathit{comp}\ (\mathit{Catch}\ (\mathit{Val}\ 1)\ (\mathit{Val}\ 2)) \\
&= \\
&\quad [\mathit{MARK}\ [\mathit{PUSH}\ 2], \mathit{PUSH}\ 1, \mathit{UNMARK}]
\end{aligned}$$

Our interpreter is also extended to run the new code produced by the compiler:

$$\begin{aligned}
&\mathit{intpt} &&:: \mathit{Stack} \rightarrow \mathit{Code} \rightarrow \mathit{Stack} \\
&\mathit{intpt}\ s\ [\,] &&= s \\
&\mathit{intpt}\ s\ (\mathit{PUSH}\ x : \mathit{ops}) &&= \mathit{intpt}\ (\mathit{VAL}\ x : s)\ \mathit{ops} \\
&\mathit{intpt}\ (\mathit{VAL}\ x : s)\ (\mathit{POP} : \mathit{ops}) &&= \mathit{intpt}\ s\ \mathit{ops} \\
&\mathit{intpt}\ (\mathit{VAL}\ y : \mathit{VAL}\ x : s)\ (\mathit{ADD} : \mathit{ops}) &&= \mathit{intpt}\ (\mathit{VAL}\ (x + y) : s)\ \mathit{ops} \\
&\mathit{intpt}\ s\ (\mathit{THROW} : \mathit{ops}) &&= \mathit{unwind}\ s\ (\mathit{skip}\ \mathit{ops}) \\
&\mathit{intpt}\ s\ (\mathit{MARK}\ \mathit{ops}' : \mathit{ops}) &&= \mathit{intpt}\ (\mathit{HAN}\ \mathit{ops}' : s)\ \mathit{ops} \\
&\mathit{intpt}\ s\ (\mathit{UNMARK} : \mathit{ops}) &&= \textbf{case}\ s\ \textbf{of} \\
& && \qquad (x : \mathit{HAN}\ _ : s') \rightarrow \mathit{intpt}\ (x : s')\ \mathit{ops}
\end{aligned}$$

For the *PUSH* and *ADD* instructions the interpreter behaves in the same way as previously, except that stack items are now tagged. The *POP* instruction simply removes the top

element of the stack if it is a value. *MARK* places a compiled handler onto the stack, and *UNMARK* removes a handler as the second-top stack element, the top of the stack being the result value produced by the body of the *catch* to which the *UNMARK* refers.

The most interesting case of the interpreter involves the *THROW* operation. When a *THROW* operation is reached the code being executed has raised an exception. In this case we need to recover by performing the following two actions. Firstly, we must remove all the intermediate values placed on the stack since entering the scope of the last *catch*, if there is one, and retrieve an exception handler to execute; and secondly, we must jump to the end of the code representing that same *catch* block. Here these two actions are represented by the functions *unwind* and *skip*. Intermediate values are removed from the stack, and a handler retrieved, using *unwind*:

$$
\begin{aligned}
&unwind && :: Stack \rightarrow Code \rightarrow Stack \\
&unwind\ [\,]\ _ && = [\,] \\
&unwind\ (VAL\ _ : s)\ ops && = unwind\ s\ ops \\
&unwind\ (HAN\ ops' : s)\ ops && = intpt\ s\ (ops' \mathbin{+\!\!+} ops)
\end{aligned}
$$

That is, we simply keep removing values searching for an exception handler, if we find one we run the handler code followed by the second argument of *unwind*, which in our interpreter is the result of applying *skip* to the currently executing code. If no handler is found when unwinding the stack, the interpreter ends, returning an empty stack, which represents an uncaught exception. Jumping to the end of the current *catch* block is achieved by applying *skip* to the currently executing code.

$$
\begin{aligned}
&skip && :: Code \rightarrow Code \\
&skip\ [\,] && = [\,] \\
&skip\ (UNMARK : ops) && = ops \\
&skip\ (MARK\ _ : ops) && = skip\ (skip\ ops) \\
&skip\ (_ : ops) && = skip\ ops
\end{aligned}
$$

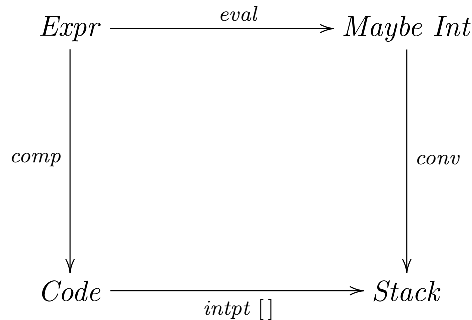
That is, any operation other than *MARK* and *UNMARK* is simply discarded. If we come across a *MARK* operation we need to completely skip over that *catch* block, hence the call of *skip* within *skip* in the definition. If we find an *UNMARK* we are done skipping code. The multiple calls to *skip* have the effect that we always skip to the end of the *catch* block we are currently executing within. This method of moving to the next instruction

to execute has the advantage of not requiring explicit labelling of points in the code, as would be necessary for direct jumps. At the end of this chapter we will consider alternative approaches to recovering from exceptions, one of which is based upon *jumps*.

We now proceed to prove this compiler correct with respect to our big-step semantics.

## 5.2 Compiler Correctness

The correctness of the new exception handling compiler can be described by the commuting diagram below. Note that we now convert a *Maybe Int* into a *Stack* using a function *conv*, and that the result of a failed computation is an empty final stack.




**where**

$conv \quad :: \; Maybe\ Int \rightarrow Stack$
$conv\ Nothing = [\,]$
$conv\ (Just\ n) = [\,VAL\ n\,]$

Similarly to previously, we also require a number of generalisations in order to allow the proof to proceed more smoothly. In particular, we modify the above statement to include an arbitrary initial stack and arbitrary code to be executed after the compiled expression. This change also requires a corresponding generalisation to the *conv* function.

**Theorem 5.2.1 (compiler correctness).**

$$intpt\ s\ (comp\ e +\!\!+\ ops) = conv\ s\ (eval\ e)\ ops$$

**where**

$conv :: Stack \rightarrow Maybe\ Int \rightarrow Code \rightarrow Stack$

$conv\ s\ Nothing\ ops = unwind\ s\ (skip\ ops)$

$conv\ s\ (Just\ n)\ ops = intpt\ (VAL\ n : s)\ ops$

That is, the result of compiling and interpreting an expression with an arbitrary initial stack followed by arbitrary code is the same as evaluating that expression and converting it to a stack using the same initial stack and code. Note that with $s = ops = [\,]$, this theorem simplifies to our original statement of correctness above. The right-hand side of theorem 5.2.1 could also be written as $intpt\ s\ (conv\ (eval\ e) : ops)$ using a simpler version of $conv$ with type $Maybe\ Int \rightarrow Op$, but the above formulation leads to simpler proofs.

*Proof.* By induction on $e :: Expr$

Case: $e = Val\ n$

$intpt\ s\ (comp\ (Val\ n) +\!\!+\ ops)$

$=$ { definition of $comp$ }

$intpt\ s\ ([PUSH\ n] +\!\!+\ ops)$

$=$ { definition of $intpt$ }

$intpt\ (VAL\ n : s)\ ops$

$=$ { definition of $conv$ }

$conv\ s\ (Just\ n)\ ops$

$=$ { definition of $eval$ }

$conv\ s\ (eval\ (Val\ n))\ ops$

Case: $e = Throw$

$intpt\ s\ (comp\ Throw +\!\!+\ ops)$

$=$ { definition of $comp$ }

$intpt\ s\ ([THROW] +\!\!+\ ops)$

$$
\begin{aligned}
= &\quad \{ \text{ definition of } intpt \ \} \\
&\quad unwind\ s\ (skip\ ops) \\
= &\quad \{ \text{ definition of } conv \ \} \\
&\quad conv\ s\ Nothing\ ops \\
= &\quad \{ \text{ definition of } eval \ \} \\
&\quad conv\ s\ (eval\ Throw)\ ops
\end{aligned}
$$

Case: $e = Add\ x\ y$

$$
\begin{aligned}
&\quad intpt\ s\ (comp\ (Add\ x\ y) +\!\!+\ ops) \\
= &\quad \{ \text{ definition of } comp \ \} \\
&\quad intpt\ s\ (comp\ x +\!\!+\ comp\ y +\!\!+\ [ADD] +\!\!+\ ops) \\
= &\quad \{ \text{ induction hypothesis } \} \\
&\quad conv\ s\ (eval\ x)\ (comp\ y +\!\!+\ [ADD] +\!\!+\ ops) \\
= &\quad \{ \text{ definition of } conv \ \} \\
&\quad \textbf{case}\ eval\ x\ \textbf{of} \\
&\qquad Nothing \rightarrow unwind\ s\ (skip\ (comp\ y +\!\!+\ [ADD] +\!\!+\ ops)) \\
&\qquad Just\ n \rightarrow intpt\ (VAL\ n : s)\ (comp\ y +\!\!+\ [ADD] +\!\!+\ ops)
\end{aligned}
$$

The two possible results from this expression are simplified below.

1:

$$
\begin{aligned}
&\quad unwind\ s\ (skip\ (comp\ y +\!\!+\ [ADD] +\!\!+\ ops)) \\
= &\quad \{ \text{ skipping compiled code (lemma 5.2.2) } \} \\
&\quad unwind\ s\ (skip\ ([ADD] +\!\!+\ ops)) \\
= &\quad \{ \text{ definition of } skip \ \} \\
&\quad unwind\ s\ (skip\ ops)
\end{aligned}
$$

2:

$$
\begin{aligned}
&\quad intpt\ (VAL\ n : s)\ (comp\ y +\!\!+\ [ADD] +\!\!+\ ops) \\
= &\quad \{ \text{ induction hypothesis } \} \\
&\quad conv\ (VAL\ n : s)\ (eval\ y)\ ([ADD] +\!\!+\ ops)
\end{aligned}
$$

$=$ { definition of $\mathit{conv}$ }

$\quad \textbf{case}\ \mathit{eval}\ y\ \textbf{of}$

$\qquad \mathit{Nothing} \rightarrow \mathit{unwind}\ (\mathit{VAL}\ n : s)\ (\mathit{skip}\ ([\mathit{ADD}] +\!\!+\ \mathit{ops}))$

$\qquad \mathit{Just}\ m \rightarrow \mathit{intpt}\ (\mathit{VAL}\ m : \mathit{VAL}\ n : s)\ ([\mathit{ADD}] +\!\!+\ \mathit{ops})$

$=$ { definition of $\mathit{unwind}$, $\mathit{skip}$ and $\mathit{intpt}$ }

$\quad \textbf{case}\ \mathit{eval}\ y\ \textbf{of}$

$\qquad \mathit{Nothing} \rightarrow \mathit{unwind}\ s\ (\mathit{skip}\ \mathit{ops})$

$\qquad \mathit{Just}\ m \rightarrow \mathit{intpt}\ (\mathit{VAL}\ (n + m) : s)\ \mathit{ops}$

We now continue the calculation using the two simplified results.

$\quad \textbf{case}\ \mathit{eval}\ x\ \textbf{of}$

$\qquad \mathit{Nothing} \rightarrow \mathit{unwind}\ s\ (\mathit{skip}\ \mathit{ops})$

$\qquad \mathit{Just}\ n \rightarrow \textbf{case}\ \mathit{eval}\ y\ \textbf{of}$

$\qquad\quad \mathit{Nothing} \rightarrow \mathit{unwind}\ s\ (\mathit{skip}\ \mathit{ops})$

$\qquad\quad \mathit{Just}\ m \rightarrow \mathit{intpt}\ (\mathit{VAL}\ (n + m) : s)\ \mathit{ops}$

$=$ { definition of $\mathit{conv}$ }

$\quad \textbf{case}\ \mathit{eval}\ x\ \textbf{of}$

$\qquad \mathit{Nothing} \rightarrow \mathit{conv}\ s\ \mathit{Nothing}\ \mathit{ops}$

$\qquad \mathit{Just}\ n \rightarrow \textbf{case}\ \mathit{eval}\ y\ \textbf{of}$

$\qquad\quad \mathit{Nothing} \rightarrow \mathit{conv}\ s\ \mathit{Nothing}\ \mathit{ops}$

$\qquad\quad \mathit{Just}\ m \rightarrow \mathit{conv}\ s\ (\mathit{Just}\ (n + m)\ \mathit{ops})$

$=$ { distribution over **case** }

$\quad \mathit{conv}\ s\ (\textbf{case}\ \mathit{eval}\ x\ \textbf{of}$

$\qquad \mathit{Nothing} \rightarrow \mathit{Nothing}$

$\qquad \mathit{Just}\ n \rightarrow \textbf{case}\ \mathit{eval}\ y\ \textbf{of}$

$\qquad\quad \mathit{Nothing} \rightarrow \mathit{Nothing}$

$\qquad\quad \mathit{Just}\ m \rightarrow \mathit{Just}\ (n + m))\ \mathit{ops}$

$=$ { definition of $\mathit{eval}$ }

$\quad \mathit{conv}\ s\ (\mathit{eval}\ (\mathit{Add}\ x\ y))\ \mathit{ops}$

Case: $e = \mathit{Catch}\ x\ h$

$$
\begin{aligned}
& \quad \mathit{intpt}\ s\ (\mathit{comp}\ (\mathit{Catch}\ x\ h) +\!\!+\ \mathit{ops}) \\
= & \quad \{\ \text{definition of } \mathit{comp}\ \} \\
& \quad \mathit{intpt}\ s\ ([\mathit{MARK}\ (\mathit{comp}\ h)] +\!\!+\ \mathit{comp}\ x +\!\!+ [\mathit{UNMARK}] +\!\!+\ \mathit{ops}) \\
= & \quad \{\ \text{definition of } \mathit{intpt}\ \} \\
& \quad \mathit{intpt}\ (\mathit{HAN}\ (\mathit{comp}\ h) : s)\ (\mathit{comp}\ x +\!\!+ [\mathit{UNMARK}] +\!\!+\ \mathit{ops}) \\
= & \quad \{\ \text{induction hypothesis}\ \} \\
& \quad \mathit{conv}\ (\mathit{HAN}\ (\mathit{comp}\ h) : s)\ (\mathit{eval}\ x)\ ([\mathit{UNMARK}] +\!\!+\ \mathit{ops}) \\
= & \quad \{\ \text{definition of } \mathit{conv}\ \} \\
& \quad \mathbf{case}\ \mathit{eval}\ x\ \mathbf{of} \\
& \qquad \mathit{Nothing} \rightarrow \mathit{unwind}\ (\mathit{HAN}\ (\mathit{comp}\ h) : s)\ (\mathit{skip}\ ([\mathit{UNMARK}] +\!\!+\ \mathit{ops})) \\
& \qquad \mathit{Just}\ n \rightarrow \mathit{intpt}\ (\mathit{VAL}\ n : \mathit{HAN}\ (\mathit{comp}\ h) : s)\ ([\mathit{UNMARK}] +\!\!+\ \mathit{ops}) \\
= & \quad \{\ \text{definition of } \mathit{unwind},\ \mathit{skip} \text{ and } \mathit{intpt}\ \} \\
& \quad \mathbf{case}\ \mathit{eval}\ x\ \mathbf{of} \\
& \qquad \mathit{Nothing} \rightarrow \mathit{intpt}\ s\ (\mathit{comp}\ h +\!\!+\ \mathit{ops}) \\
& \qquad \mathit{Just}\ n \rightarrow \mathit{intpt}\ (\mathit{VAL}\ n : s)\ \mathit{ops} \\
= & \quad \{\ \text{induction hypothesis}\ \} \\
& \quad \mathbf{case}\ \mathit{eval}\ x\ \mathbf{of} \\
& \qquad \mathit{Nothing} \rightarrow \mathit{conv}\ s\ (\mathit{eval}\ h)\ \mathit{ops} \\
& \qquad \mathit{Just}\ n \rightarrow \mathit{intpt}\ (\mathit{VAL}\ n : s)\ \mathit{ops} \\
= & \quad \{\ \text{definition of } \mathit{conv}\ \} \\
& \quad \mathbf{case}\ \mathit{eval}\ x\ \mathbf{of} \\
& \qquad \mathit{Nothing} \rightarrow \mathit{conv}\ s\ (\mathit{eval}\ h)\ \mathit{ops} \\
& \qquad \mathit{Just}\ n \rightarrow \mathit{conv}\ s\ (\mathit{Just}\ n)\ \mathit{ops} \\
= & \quad \{\ \text{distribution over } \mathbf{case}\ \} \\
& \quad \mathit{conv}\ s\ (\mathbf{case}\ \mathit{eval}\ x\ \mathbf{of} \\
& \qquad \mathit{Nothing} \rightarrow \mathit{eval}\ h \\
& \qquad \mathit{Just}\ n \rightarrow \mathit{Just}\ n)\ \mathit{ops} \\
= & \quad \{\ \text{definition of } \mathit{eval}\ \} \\
& \quad \mathit{conv}\ s\ (\mathit{eval}\ (\mathit{Catch}\ x\ h))\ \mathit{ops}
\end{aligned}
$$

Case: $e = \mathit{Seq}\ x\ y$

$$
\mathit{intpt}\ s\ (\mathit{comp}\ (\mathit{Seq}\ x\ y) +\!\!+\ \mathit{ops})
$$

$$
\begin{array}{ll}
= & \{ \text{ definition of } comp \ \} \\
 & intpt\ s\ ((comp\ x) +\!\!+\ POP : (comp\ y) +\!\!+\ ops) \\
= & \{ \text{ Induction Hypothesis } \} \\
 & conv\ s\ (eval\ x)\ (POP : (comp\ y) +\!\!+\ ops) \\
= & \{ \text{ definition of } conv \ \} \\
 & \textbf{case}\ eval\ x\ \textbf{of} \\
 & \quad Nothing \rightarrow unwind\ s\ (skip\ (POP : (comp\ y) +\!\!+\ ops)) \\
 & \quad Just\ n \rightarrow intpt\ (VAL\ n : s)\ (POP : (comp\ y) +\!\!+\ ops) \\
= & \{ \text{ definition of } unwind, skip, intpt \text{ and lemma 5.2.2 } \} \\
 & \textbf{case}\ eval\ x\ \textbf{of} \\
 & \quad Nothing \rightarrow unwind\ s\ (skip\ ops) \\
 & \quad Just\ n \rightarrow intpt\ s\ ((comp\ y) +\!\!+\ ops) \\
= & \{ \text{ definition of } conv\text{, Induction Hypothesis } \} \\
 & \textbf{case}\ eval\ x\ \textbf{of} \\
 & \quad Nothing \rightarrow conv\ s\ Nothing\ ops \\
 & \quad Just\ n \rightarrow conv\ s\ (eval\ y)\ ops \\
= & \{ \text{ definition of } conv \ \} \\
 & \textbf{case}\ eval\ x\ \textbf{of} \\
 & \quad Nothing \rightarrow conv\ s\ Nothing\ ops \\
 & \quad Just\ n \rightarrow \textbf{case}\ eval\ y\ \textbf{of} \\
 & \qquad Nothing \rightarrow unwind\ s\ (skip\ ops) \\
 & \qquad Just\ m \rightarrow intpt\ (VAL\ m : s)\ ops \\
= & \{ \text{ definition of } conv \ \} \\
 & \textbf{case}\ eval\ x\ \textbf{of} \\
 & \quad Nothing \rightarrow conv\ s\ Nothing\ ops \\
 & \quad Just\ n \rightarrow \textbf{case}\ eval\ y\ \textbf{of} \\
 & \qquad Nothing \rightarrow conv\ s\ Nothing\ ops \\
 & \qquad Just\ m \rightarrow conv\ s\ (Just\ m)\ ops \\
= & \{ \text{ distribution over } \textbf{case} \ \} \\
 & conv\ s\ (\textbf{case}\ eval\ x\ \textbf{of} \\
 & \quad Nothing \rightarrow Nothing \\
 & \quad Just\ n \rightarrow \textbf{case}\ eval\ y\ \textbf{of}
\end{array}
$$

$$
\begin{array}{ll}
& \qquad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
& \qquad \mathit{Just}\ m \rightarrow \mathit{Just}\ m)\ \mathit{ops} \\
= & \{ \text{definition of } \mathit{eval} \} \\
& \mathit{conv}\ s\ (\mathit{eval}\ (\mathit{Seq}\ x\ y))\ \mathit{ops}
\end{array}
$$

□

The two distribution over **case** steps in the above proof rely on the fact that *conv* is strict in its second argument ($conv\ s\ \bot\ ops = \bot$), which is indeed the case because *conv* is defined by pattern matching on this argument.

**Lemma 5.2.2 (skipping compiled code).**

$$\mathit{skip}\ (\mathit{comp}\ e +\!\!+\ \mathit{ops}) = \mathit{skip}\ \mathit{ops}$$

That is, skipping to the next unmark in compiled code followed by arbitrary additional code gives the same result as simply skipping the additional code. Intuitively, this is the case because the compiler ensures that all *UNMARKS*s are matched by preceding *MARK*s.

*Proof.* By induction on $e :: \mathit{Expr}$

Case: $e = \mathit{Val}\ n$

$$
\begin{array}{ll}
& \mathit{skip}\ (\mathit{comp}\ (\mathit{Val}\ n) +\!\!+\ \mathit{ops}) \\
= & \{ \text{definition of } \mathit{comp} \} \\
& \mathit{skip}\ ([\mathit{PUSH}\ n] +\!\!+\ \mathit{ops}) \\
= & \{ \text{definition of } \mathit{skip} \} \\
& \mathit{skip}\ \mathit{ops}
\end{array}
$$

Case: $e = \mathit{Throw}$

$$
\begin{array}{ll}
& \mathit{skip}\ (\mathit{comp}\ \mathit{Throw} +\!\!+\ \mathit{ops}) \\
= & \{ \text{definition of } \mathit{comp} \} \\
& \mathit{skip}\ ([\mathit{THROW}] +\!\!+\ \mathit{ops}) \\
= & \{ \text{definition of } \mathit{skip} \} \\
& \mathit{skip}\ \mathit{ops}
\end{array}
$$

Case: $e = Add\ x\ y$

$$
\begin{array}{ll}
 & skip\ (comp\ (Add\ x\ y) +\!\!+ ops) \\
= & \{\ \text{definition of } comp\ \} \\
 & skip\ (comp\ x +\!\!+ comp\ y +\!\!+ [ADD] +\!\!+ ops) \\
= & \{\ \text{induction hypothesis}\ \} \\
 & skip\ (comp\ y +\!\!+ [ADD] +\!\!+ ops) \\
= & \{\ \text{induction hypothesis}\ \} \\
 & skip\ ([ADD] +\!\!+ ops) \\
= & \{\ \text{definition of } skip\ \} \\
 & skip\ ops
\end{array}
$$

Case: $e = Catch\ x\ h$

$$
\begin{array}{ll}
 & skip\ (comp\ (Catch\ x\ h) +\!\!+ ops) \\
= & \{\ \text{definition of } comp\ \} \\
 & skip\ ([MARK\ (comp\ h)] +\!\!+ comp\ x +\!\!+ [UNMARK] +\!\!+ ops) \\
= & \{\ \text{definition of } skip\ \} \\
 & skip\ (skip\ (comp\ x +\!\!+ [UNMARK] +\!\!+ ops)) \\
= & \{\ \text{induction hypothesis}\ \} \\
 & skip\ (skip\ ([UNMARK] +\!\!+ ops)) \\
= & \{\ \text{definition of } skip\ \} \\
 & skip\ ops
\end{array}
$$

Case: $e = Seq\ x\ y$

$$
\begin{array}{ll}
 & skip\ (comp\ (Seq\ x\ y) +\!\!+ ops) \\
= & \{\ \text{definition of } comp\ \} \\
 & skip\ ((comp\ x) +\!\!+ POP : (comp\ y) +\!\!+ ops) \\
= & \{\ \text{induction hypothesis}\ \} \\
 & skip\ (POP : (comp\ y) +\!\!+ ops) \\
= & \{\ \text{definition of } skip\ \} \\
 & skip\ ((comp\ y) +\!\!+ ops)
\end{array}
$$

$=$ { induction hypothesis }
*skip ops*

□

## 5.3 Compiler Variations

We now introduce two variations on the compiler we have been looking at so far, namely a compiler that employs explicit jumps to reach handler code, and a compilation that employs a *code continuation* to avoid the need for the *skip* function. These compilers give two alternative approaches to reaching handler code in the case of a raised exception.

### 5.3.1 Explicit Jump Compiler

The compiler we have seen so far pushes handler code onto the stack, and skips instructions one at a time when an exception is raised. Whilst this method captures the *essence* of handling exceptions [HW04], in practice only the memory address of an exception handler is placed onto the stack, and programs jump to a handler when an exception is raised. We now introduce a version of the compiler which includes explicit *jumps* and *labels*, which are used to move the point of execution to a handler if an exception is raised.

First of all, we modify our type of machine operations to mark the stack with an address rather than actual code, to introduce a new label, and to jump to a specific address:

$$
\begin{aligned}
\textbf{data}\ \mathit{Op} = {} & \mathit{PUSH\ Int} \mid \mathit{THROW} \mid \mathit{ADD} \mid \mathit{POP} \\
\mid {} & \mathit{MARK\ Addr} \mid \mathit{UNMARK} \\
\mid {} & \mathit{LABEL\ Addr} \mid \mathit{JUMP\ Addr}
\end{aligned}
$$

For simplicity, an address is represented as an integer:

$$\textbf{type}\ \mathit{Addr} = \mathit{Int}$$

It is important that no two exception handlers have the same address, so we use the *fresh* function to generate fresh labels during compilation.

$$\mathit{fresh} \quad :: \mathit{Addr} \rightarrow \mathit{Addr}$$

$$fresh\ a = a + 1$$

Finally, the type of stacks must also be modified to reflect the change from handler code to handler address:

$$\textbf{data}\ Item = VAL\ Int \mid HAN\ Addr$$

The compiler intself now takes a starting address and an expression to compile, and produces code containing jump labels:

$$\begin{aligned}
&comp && :: Addr \to Expr \to Code \\
&comp\ a\ e && = fst\ (comp'\ a\ e) \\
&comp' && :: Addr \to Expr \to (Code, Addr) \\
&comp'\ a\ (Val\ n) && = ([PUSH\ n], a) \\
&comp'\ a\ (Add\ x\ y) && = (xs \mathbin{+\!\!+} ys \mathbin{+\!\!+} [ADD], c) \\
& && \quad\ \textbf{where} \\
& && \qquad (xs, b) = comp'\ a\ x \\
& && \qquad (ys, c) = comp'\ b\ y \\
&comp'\ a\ (Throw) && = ([THROW], a) \\
&comp'\ a\ (Catch\ x\ h) && = ([MARK\ a] \mathbin{+\!\!+} xs \\
& && \quad\ \mathbin{+\!\!+} [UNMARK, JUMP\ b, LABEL\ a] \\
& && \quad\ \mathbin{+\!\!+} hs \mathbin{+\!\!+} [LABEL\ b], e) \\
& && \quad\ \textbf{where} \\
& && \qquad b = fresh\ a \\
& && \qquad c = fresh\ b \\
& && \qquad (xs, d) = comp'\ c\ x \\
& && \qquad (hs, e) = comp'\ d\ h
\end{aligned}$$

Our previous example, $Catch\ (Val\ 1)\ (Val\ 2)$, now produces the following code when compiled with the starting address 1:

$$[MARK\ 1, PUSH\ 1, UNMARK, JUMP\ 2, LABEL\ 1, PUSH\ 2, LABEL\ 2]$$

This new code can be executed using the following modified interpreter, which is similar to that demonstrated previously, with the appropriate extensions for the new instructions. Note that skipping code is no longer required, due to the explicit use of jumps.

$$
\begin{array}{ll}
\mathit{intpt} & :: \mathit{Stack} \rightarrow \mathit{Code} \rightarrow \mathit{Stack} \\
\mathit{intpt}\ s\ [\,] & = s \\
\mathit{intpt}\ s\ (\mathit{PUSH}\ x : \mathit{ops}) & = \mathit{intpt}\ (\mathit{VAL}\ x : s)\ \mathit{ops} \\
\mathit{intpt}\ (\mathit{VAL}\ x : s)\ (\mathit{POP} : \mathit{ops}) & = \mathit{intpt}\ s\ \mathit{ops} \\
\mathit{intpt}\ (\mathit{VAL}\ y : \mathit{VAL}\ x : s)\ (\mathit{ADD} : \mathit{ops}) & = \mathit{intpt}\ (\mathit{VAL}\ (x + y) : s)\ \mathit{ops} \\
\mathit{intpt}\ s\ (\mathit{THROW} : \mathit{ops}) & = \mathit{unwind}\ s\ \mathit{ops} \\
\mathit{intpt}\ s\ (\mathit{MARK}\ a : \mathit{ops}) & = \mathit{intpt}\ (\mathit{HAN}\ a : s)\ \mathit{ops} \\
\mathit{intpt}\ s\ (\mathit{UNMARK} : \mathit{ops}) & = \mathbf{case}\ s\ \mathbf{of} \\
 & \quad (x : \mathit{HAN}\ _ : s') \rightarrow \mathit{intpt}\ (x : s')\ \mathit{ops} \\
\mathit{intpt}\ s\ (\mathit{LABEL}\ _ : \mathit{ops}) & = \mathit{intpt}\ s\ \mathit{ops} \\
\mathit{intpt}\ s\ (\mathit{JUMP}\ a : \mathit{ops}) & = \mathit{intpt}\ s\ (\mathit{jump}\ a\ \mathit{ops})
\end{array}
$$

$$
\begin{array}{ll}
\mathit{unwind} & :: \mathit{Stack} \rightarrow \mathit{Code} \rightarrow \mathit{Stack} \\
\mathit{unwind}\ [\,]\ _ & = [\,] \\
\mathit{unwind}\ (\mathit{VAL}\ _ : s)\ \mathit{ops} & = \mathit{unwind}\ s\ \mathit{ops} \\
\mathit{unwind}\ (\mathit{HAN}\ a : s)\ \mathit{ops} & = \mathit{intpt}\ s\ (\mathit{jump}\ a\ \mathit{ops})
\end{array}
$$

$$
\begin{array}{ll}
\mathit{jump} & :: \mathit{Addr} \rightarrow \mathit{Code} \rightarrow \mathit{Code} \\
\mathit{jump}\ _\ [\,] & = [\,] \\
\mathit{jump}\ a\ (\mathit{LABEL}\ b : \mathit{ops}) & = \mathbf{if}\ a = b\ \mathbf{then}\ \mathit{ops}\ \mathbf{else}\ \mathit{jump}\ a\ \mathit{ops} \\
\mathit{jump}\ a\ (_ : \mathit{ops}) & = \mathit{jump}\ a\ \mathit{ops}
\end{array}
$$

We now give a proof of correctness of the jumping compiler. As we shall see, the introduction of jumps significantly increases the length and complexity of the proof.

**Theorem 5.3.1 (Jump Compiler Correctness).**

$$\mathit{isFresh}\ a\ s \Rightarrow \mathit{intpt}\ s\ (\mathit{comp}\ a\ e \mathbin{+\!\!+} \mathit{ops}) = \mathit{conv}\ s\ (\mathit{eval}\ e)\ \mathit{ops}$$

**where**

$$
\begin{array}{ll}
\mathit{isFresh} & :: \mathit{Addr} \rightarrow \mathit{Stack} \rightarrow \mathit{Bool} \\
\mathit{isFresh}\ _\ [\,] & = \mathit{True} \\
\mathit{isFresh}\ a\ (\mathit{VAL}\ _ : s) & = \mathit{isFresh}\ a\ s \\
\mathit{isFresh}\ a\ (\mathit{HAN}\ b : s) & = a > b \wedge \mathit{isFresh}\ a\ s
\end{array}
$$

$$
\begin{aligned}
&\mathit{conv} && :: \mathit{Stack} \to \mathit{Maybe\ Int} \to \mathit{Code} \to \mathit{Stack} \\
&\mathit{conv}\ s\ \mathit{Nothing}\ \mathit{ops} && = \mathit{unwind}\ s\ \mathit{ops} \\
&\mathit{conv}\ s\ (\mathit{Just}\ n)\ \mathit{ops} && = \mathit{intpt}\ (\mathit{VAL}\ n : s)\ \mathit{ops}
\end{aligned}
$$

The function *isFresh* checks that the initial label for compilation is greater than all those in the stack, and therefore that all future labels generated will be fresh. This property is essential to guarantee jumping to the correct handler address if an exception is raised.

*Proof.* By induction on $e :: \mathit{Expr}$

Case: $e = \mathit{Val}\ n$

$$
\begin{aligned}
& \quad \mathit{intpt}\ s\ (\mathit{comp}\ a\ (\mathit{Val}\ n) +\!\!+ \mathit{ops}) \\
= & \quad \{ \text{ definition of } \mathit{comp} \ \} \\
& \quad \mathit{intpt}\ s\ ([\mathit{PUSH}\ n] +\!\!+ \mathit{ops}) \\
= & \quad \{ \text{ definition of } \mathit{intpt} \ \} \\
& \quad \mathit{intpt}\ (\mathit{VAL}\ n : s)\ \mathit{ops} \\
= & \quad \{ \text{ definition of } \mathit{conv} \ \} \\
& \quad \mathit{conv}\ s\ (\mathit{Just}\ n)\ \mathit{ops} \\
= & \quad \{ \text{ definition of } \mathit{eval} \ \} \\
& \quad \mathit{conv}\ s\ (\mathit{eval}\ (\mathit{Val}\ n))\ \mathit{ops}
\end{aligned}
$$

Case: $e = \mathit{Throw}$

$$
\begin{aligned}
& \quad \mathit{intpt}\ s\ (\mathit{comp}\ a\ \mathit{Throw} +\!\!+ \mathit{ops}) \\
= & \quad \{ \text{ definition of } \mathit{comp} \ \} \\
& \quad \mathit{intpt}\ s\ ([\mathit{THROW}] +\!\!+ \mathit{ops}) \\
= & \quad \{ \text{ definition of } \mathit{intpt} \ \} \\
& \quad \mathit{unwind}\ s\ \mathit{ops} \\
= & \quad \{ \text{ definition of } \mathit{conv} \ \} \\
& \quad \mathit{conv}\ s\ \mathit{Nothing}\ \mathit{ops} \\
= & \quad \{ \text{ definition of } \mathit{eval} \ \} \\
& \quad \mathit{conv}\ s\ (\mathit{eval}\ \mathit{Throw})\ \mathit{ops}
\end{aligned}
$$

Case: $e = Add\ x\ y$

Let

$$(_, b) = comp'\ a\ x$$
$$(_, c) = comp'\ b\ y$$

in

$$
\begin{array}{ll}
 & intpt\ s\ (comp\ a\ (Add\ x\ y) +\!\!+\ ops) \\
= & \{\ \text{definition of } comp\ \} \\
 & intpt\ s\ (comp\ a\ x +\!\!+\ comp\ b\ y +\!\!+\ [ADD] +\!\!+\ ops) \\
= & \{\ \text{induction hypothesis}\ \} \\
 & conv\ s\ (eval\ x)\ (comp\ b\ y +\!\!+\ [ADD] +\!\!+\ ops) \\
= & \{\ \text{definition of } conv\ \} \\
 & \textbf{case}\ eval\ x\ \textbf{of} \\
 & \quad Nothing \rightarrow unwind\ s\ (comp\ b\ y +\!\!+\ [ADD] +\!\!+\ ops) \\
 & \quad Just\ n \rightarrow intpt\ (VAL\ n : s)\ (comp\ b\ y +\!\!+\ [ADD] +\!\!+\ ops)
\end{array}
$$

The two possible results from this expression are simplified below.

1:

$$
\begin{array}{ll}
 & unwind\ s\ (comp\ b\ y +\!\!+\ [ADD] +\!\!+\ ops) \\
= & \{\ \text{unwinding compiled code (lemma 5.3.3)}\ \} \\
 & unwind\ s\ ([ADD] +\!\!+\ ops) \\
= & \{\ \text{unwinding operators (lemma 5.3.2)}\ \} \\
 & unwind\ s\ ops
\end{array}
$$

2:

$$
\begin{array}{ll}
 & intpt\ (VAL\ n : s)\ (comp\ b\ y +\!\!+\ [ADD] +\!\!+\ ops) \\
= & \{\ \text{induction hypothesis}\ \} \\
 & conv\ (VAL\ n : s)\ (eval\ y)\ ([ADD] +\!\!+\ ops) \\
= & \{\ \text{definition of } conv\ \} \\
 & \textbf{case}\ eval\ y\ \textbf{of}
\end{array}
$$

$$
\begin{array}{ll}
& \quad \mathit{Nothing} \rightarrow \mathit{unwind}\ (\mathit{VAL}\ n : s)\ ([\mathit{ADD}] +\!\!+ \mathit{ops}) \\
& \quad \mathit{Just}\ m \rightarrow \mathit{intpt}\ (\mathit{VAL}\ m : \mathit{VAL}\ n : s)\ ([\mathit{ADD}] +\!\!+ \mathit{ops}) \\
= & \{ \text{ unwinding operators, definition of } \mathit{unwind} \text{ and } \mathit{intpt} \ \} \\
& \textbf{case}\ \mathit{eval}\ y\ \textbf{of} \\
& \quad \mathit{Nothing} \rightarrow \mathit{unwind}\ s\ \mathit{ops} \\
& \quad \mathit{Just}\ m \rightarrow \mathit{intpt}\ (\mathit{VAL}\ (n+m) : s)\ \mathit{ops}
\end{array}
$$

We now continue the calculation using the two simplified results.

$$
\begin{array}{ll}
& \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
& \quad \mathit{Nothing} \rightarrow \mathit{unwind}\ s\ \mathit{ops} \\
& \quad \mathit{Just}\ n \rightarrow \textbf{case}\ \mathit{eval}\ y\ \textbf{of} \\
& \qquad \mathit{Nothing} \rightarrow \mathit{unwind}\ s\ \mathit{ops} \\
& \qquad \mathit{Just}\ m \rightarrow \mathit{intpt}\ (\mathit{VAL}\ (n+m) : s)\ \mathit{ops} \\
= & \{ \text{ definition of } \mathit{conv} \ \} \\
& \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
& \quad \mathit{Nothing} \rightarrow \mathit{conv}\ s\ \mathit{Nothing}\ \mathit{ops} \\
& \quad \mathit{Just}\ n \rightarrow \textbf{case}\ \mathit{eval}\ y\ \textbf{of} \\
& \qquad \mathit{Nothing} \rightarrow \mathit{conv}\ s\ \mathit{Nothing}\ \mathit{ops} \\
& \qquad \mathit{Just}\ m \rightarrow \mathit{conv}\ s\ (\mathit{Just}\ (n+m))\ \mathit{ops} \\
= & \{ \text{ distribution over } \textbf{case} \ \} \\
& \mathit{conv}\ s\ (\textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
& \quad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
& \quad \mathit{Just}\ n \rightarrow \textbf{case}\ \mathit{eval}\ y\ \textbf{of} \\
& \qquad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
& \qquad \mathit{Just}\ m \rightarrow \mathit{Just}\ (n+m))\ \mathit{ops} \\
= & \{ \text{ definition of } \mathit{eval} \ \} \\
& \mathit{conv}\ s\ (\mathit{eval}\ (\mathit{Add}\ x\ y))\ \mathit{ops}
\end{array}
$$

Note that a number of steps in the above proof rely on freshness conditions. The first induction hypothesis step requires that *isFresh* $a$ $s$, which is true by assumption. The unwinding compiled code (lemma 5.3.3) step requires that *isFresh* $b$ $s$, which is verified by the following calculation:

$$
\begin{array}{ll}
& \mathit{isFresh}\ b\ s \\
\Leftarrow & \{\ \mathit{isFresh} \text{ is monotonic (lemma 5.3.4) } \} \\
& a \leqslant b \ \wedge\ \mathit{isFresh}\ a\ s \\
\Leftrightarrow & \{\ \text{definition of } b\ \} \\
& a \leqslant \mathit{snd}\ (\mathit{comp}'\ a\ x)\ \wedge\ \mathit{isFresh}\ a\ s \\
\Leftrightarrow & \{\ \mathit{comp}' \text{ is non-decreasing (lemma 5.3.5), assumption } \} \\
& \mathit{True}
\end{array}
$$

Finally, the second induction hypothesis step requires that $\mathit{isFresh}\ b\ (\mathit{Val}\ n : s)$, which is trivially true by the definition of *isFresh* and the previous condition.

Case: $e = \mathit{Catch}\ x\ h$

Let

$$
\begin{array}{ll}
b & = \mathit{fresh}\ a \\
c & = \mathit{fresh}\ b \\
(_, d) & = \mathit{comp}'\ c\ x \\
(_, e) & = \mathit{comp}'\ d\ h
\end{array}
$$

in

$$
\begin{array}{ll}
& \mathit{intpt}\ s\ (\mathit{comp}\ a\ (\mathit{Catch}\ x\ h) +\!\!+\ \mathit{ops}) \\
= & \{\ \text{definition of } \mathit{comp}\ \} \\
& \mathit{intpt}\ s\ ([\mathit{MARK}\ a] +\!\!+\ \mathit{comp}\ c\ x +\!\!+\ [\mathit{UNMARK}, \mathit{JUMP}\ b, \\
& \quad \mathit{LABEL}\ a] +\!\!+\ \mathit{comp}\ d\ h +\!\!+\ [\mathit{LABEL}\ b] +\!\!+\ \mathit{ops}) \\
= & \{\ \text{definition of } \mathit{intpt}\ \} \\
& \mathit{intpt}\ (\mathit{HAN}\ a : s)\ (\mathit{comp}\ c\ x +\!\!+\ (\mathit{UNMARK}, \mathit{JUMP}\ b, \\
& \quad \mathit{LABEL}\ a] +\!\!+\ \mathit{comp}\ d\ h +\!\!+\ [\mathit{LABEL}\ b] +\!\!+\ \mathit{ops}) \\
= & \{\ \text{induction hypothesis}\ \} \\
& \mathit{conv}\ (\mathit{HAN}\ a : s)\ (\mathit{eval}\ x)\ ([\mathit{UNMARK}, \mathit{JUMP}\ b, \\
& \quad \mathit{LABEL}\ a] +\!\!+\ \mathit{comp}\ d\ h +\!\!+\ [\mathit{LABEL}\ b] +\!\!+\ \mathit{ops}) \\
= & \{\ \text{definition of } \mathit{conv}\ \} \\
& \textbf{case}\ \mathit{eval}\ x\ \textbf{of} \\
& \quad \mathit{Nothing} \rightarrow \mathit{unwind}\ (\mathit{HAN}\ a : s)\ ([\mathit{UNMARK}, \mathit{JUMP}\ b,
\end{array}
$$

$$
\begin{array}{l}
\quad LABEL\ a] +\!\!+\ comp\ d\ h +\!\!+ [LABEL\ b] +\!\!+\ ops) \\
Just\ n \rightarrow intpt\ (VAL\ n : HAN\ a : s)\ ([UNMARK, JUMP\ b, \\
\quad LABEL\ a] +\!\!+\ comp\ d\ h +\!\!+ [LABEL\ b] +\!\!+\ ops)
\end{array}
$$

The two possible results from this expression are simplified below.

1:

$$
\begin{array}{ll}
& unwind\ (HAN\ a : s)\ ([UNMARK, JUMP\ b, \\
& \quad LABEL\ a] +\!\!+\ comp\ d\ h +\!\!+ [LABEL\ b] +\!\!+\ ops) \\
= & \{\ \text{definition of } unwind\ \} \\
& intpt\ s\ (jump\ a\ ([UNMARK, JUMP\ b, \\
& \quad LABEL\ a] +\!\!+\ comp\ d\ h +\!\!+ [LABEL\ b] +\!\!+\ ops)) \\
= & \{\ \text{definition of } jump\ \} \\
& intpt\ s\ (comp\ d\ h +\!\!+ [LABEL\ b] +\!\!+\ ops) \\
= & \{\ \text{induction hypothesis}\ \} \\
& conv\ s\ (eval\ h)\ ([LABEL\ b] +\!\!+\ ops) \\
= & \{\ \text{definition of } conv\ \} \\
& \textbf{case}\ eval\ h\ \textbf{of} \\
& \quad Nothing \rightarrow unwind\ s\ ([LABEL\ b] +\!\!+\ ops) \\
& \quad Just\ m \rightarrow intpt\ (VAL\ m : s)\ ([LABEL\ b] +\!\!+\ ops) \\
= & \{\ \text{unwinding operators (lemma 5.3.2), definition of } intpt\ \} \\
& \textbf{case}\ eval\ h\ \textbf{of} \\
& \quad Nothing \rightarrow unwind\ s\ ops \\
& \quad Just\ m \rightarrow intpt\ (VAL\ m : s)\ ops
\end{array}
$$

2:

$$
\begin{array}{ll}
& intpt\ (VAL\ n : HAN\ a : s)\ ([UNMARK, JUMP\ b, \\
& \quad LABEL\ a] +\!\!+\ comp\ d\ h +\!\!+ [LABEL\ b] +\!\!+\ ops) \\
= & \{\ \text{definition of } intpt\ \} \\
& intpt\ (VAL\ n : s)\ (jump\ b\ ([LABEL\ a] +\!\!+\ comp\ d\ h +\!\!+ [LABEL\ b] +\!\!+\ ops)) \\
= & \{\ \text{definition of } jump,\ a \not\equiv b\ \} \\
& intpt\ (VAL\ n : s)\ (jump\ b\ (comp\ d\ h +\!\!+ [LABEL\ b] +\!\!+\ ops))
\end{array}
$$

$=$ { jumping compiled code (lemma 5.3.6) }
$\quad intpt\ (VAL\ n : s)\ (jump\ b\ ([LABEL\ b] +\!\!+ ops))$
$=$ { definition of *jump* }
$\quad intpt\ (VAL\ n : s)\ ops$

We now continue the calculation using the two simplified results.

**case** *eval x* **of**
  *Nothing* → **case** *eval h* **of**
    *Nothing* → *unwind s ops*
    *Just m* → *intpt* (*VAL m* : *s*) *ops*
  *Just n* → *intpt* (*VAL n* : *s*) *ops*
$=$ { definition of *conv* }
**case** *eval x* **of**
  *Nothing* → **case** *eval h* **of**
    *Nothing* → *conv s Nothing ops*
    *Just m* → *conv s* (*Just m*) *ops*
  *Just n* → *conv s* (*Just n*) *ops*
$=$ { distribution over **case** }
*conv s* (**case** *eval x* **of**
  *Nothing* → **case** *eval h* **of**
    *Nothing* → *Nothing*
    *Just m* → *Just m*
  *Just n* → *Just n*) *ops*
$=$ { definition of *eval* }
*conv s* (*eval* (*Catch x h*)) *ops*

Again, a number of steps in the above proof rely on freshness conditions. The first induction hypothesis step requires that *isFresh c* (*HAN a* : *s*):

*isFresh c* (*HAN a* : *s*)
$\Leftrightarrow$ { definition of *isFresh* }
$c > a \ \wedge\ isFresh\ c\ s$

$$
\begin{aligned}
\Leftrightarrow \quad & \{ \text{ definition of } c \ \} \\
& a + 2 > a \ \ \wedge \ \ \mathit{isFresh}\ (a + 2)\ s \\
\Leftrightarrow \quad & \{ \text{ simplification } \} \\
& \mathit{isFresh}\ (a + 2)\ s \\
\Leftarrow \quad & \{ \ \mathit{isFresh} \text{ is monotonic (lemma 5.3.4) } \} \\
& \mathit{isFresh}\ a\ s \\
\Leftrightarrow \quad & \{ \text{ assumption } \} \\
& \mathit{True}
\end{aligned}
$$

The jumping compiled code (lemma 5.3.6) step requires that $b < d$:

$$
\begin{aligned}
& b \\
< \quad & \{ \text{ definition of } c \ \} \\
& c \\
\leqslant \quad & \{ \ \mathit{comp}' \text{ is non-decreasing (lemma 5.3.5) } \} \\
& \mathit{snd}\ (\mathit{comp}'\ c\ x) \\
= \quad & \{ \text{ definition of } d \ \} \\
& d
\end{aligned}
$$

The unwinding operators (lemma 5.3.2) step requires that $\mathit{isFresh}\ b\ s$:

$$
\begin{aligned}
& \mathit{isFresh}\ b\ s \\
\Leftarrow \quad & \{ \ \mathit{isFresh} \text{ is monotonic } \} \\
& a \leqslant b \ \ \wedge \ \ \mathit{isFresh}\ a\ s \\
\Leftrightarrow \quad & \{ \text{ definition of } b \text{, assumption } \} \\
& \mathit{True}
\end{aligned}
$$

And finally, the second induction hypothesis step requires that $\mathit{isFresh}\ d\ s$:

$$
\begin{aligned}
& \mathit{isFresh}\ d\ s \\
\Leftarrow \quad & \{ \ \mathit{isFresh} \text{ is monotonic } \} \\
& b \leqslant d \ \ \wedge \ \ \mathit{isFresh}\ b\ s \\
\Leftrightarrow \quad & \{ \text{ previous two conditions } \} \\
& \mathit{True}
\end{aligned}
$$

□

*Freshness properties*

This section presents the five properties concerning fresh addresses used in the previous section to prove the correctness of our final compiler.

**Lemma 5.3.2 (unwinding operators).**

$$(op = LABEL\ a \Rightarrow isFresh\ a\ s) \Rightarrow unwind\ s\ (op : ops) = unwind\ s\ ops$$

That is, when unwinding the stack the first operator in the code can be discarded, provided that it is not an address that may occur in the stack.

*Proof.* By induction on $s :: Stack$

Case: $s = [\,]$

$$\begin{aligned} & unwind\ [\,]\ (op : ops) \\ = \ & \{ \text{ definition of } unwind \ \} \\ & [\,] \\ = \ & \{ \text{ definition of } unwind \ \} \\ & unwind\ [\,]\ ops \end{aligned}$$

Case: $s = VAL\ n : s'$

$$\begin{aligned} & unwind\ (VAL\ n : s')\ (op : ops) \\ = \ & \{ \text{ definition of } unwind \ \} \\ & unwind\ s'\ (op : ops) \\ = \ & \{ \text{ induction hypothesis } \} \\ & unwind\ s'\ ops \\ = \ & \{ \text{ definition of } unwind \ \} \\ & unwind\ (VAL\ n : s')\ ops \end{aligned}$$

The induction hypothesis step in the above proof requires that $op = LABEL\ a \Rightarrow isFresh\ a\ s'$, which is verified by the following calculation:

$$
\begin{aligned}
& \quad op = LABEL\ a \\
\Rightarrow & \quad \{ \text{ assumption } \} \\
& \quad isFresh\ a\ (VAL\ n : s') \\
\Leftrightarrow & \quad \{ \text{ definition of } isFresh \text{ } \} \\
& \quad isFresh\ a\ s'
\end{aligned}
$$

Case: $s = HAN\ a : s'$

$$
\begin{aligned}
& \quad unwind\ (HAN\ a : s')\ (op : ops) \\
= & \quad \{ \text{ definition of } unwind \text{ } \} \\
& \quad intpt\ s'\ (jump\ a\ (op : ops)) \\
= & \quad \{ \text{ definition of } jump,\ op \not\equiv LABEL\ a \text{ } \} \\
& \quad intpt\ s'\ (jump\ a\ ops) \\
= & \quad \{ \text{ definition of } unwind \text{ } \} \\
& \quad unwind\ (HAN\ a : s')\ ops
\end{aligned}
$$

The condition $op \not\equiv LABEL\ a$ follows from the assumption of the lemma:

$$
\begin{aligned}
& \quad op = LABEL\ b \ \Rightarrow\ isFresh\ b\ (HAN\ a : s') \\
\Leftrightarrow & \quad \{ \text{ definition of } isFresh \text{ } \} \\
& \quad op = LABEL\ b \ \Rightarrow\ (b > a \ \wedge\ isFresh\ b\ s') \\
\Rightarrow & \quad \{ \text{ logic } \} \\
& \quad op = LABEL\ b \ \Rightarrow\ b > a \\
\Leftrightarrow & \quad \{ \text{ contraposition } \} \\
& \quad b \leqslant a \ \Rightarrow\ op \not\equiv LABEL\ b \\
\Rightarrow & \quad \{ \text{ taking } a = b \text{ } \} \\
& \quad op \not\equiv LABEL\ a
\end{aligned}
$$

□

**Lemma 5.3.3 (unwinding compiled code).**

$$isFresh\ a\ s \Rightarrow unwind\ s\ (comp\ a\ e +\!\!+\ ops) = unwind\ s\ ops$$

That is, unwinding the stack on compiled code followed by arbitrary additional code gives the same result as simply unwinding the stack on the additional code, provided that the initial address for the compiler is fresh for the stack.

*Proof.* By induction on $e :: Expr$, using lemma 5.3.2 above

Case: $e = Val\ n$

$$\begin{aligned}
& unwind\ s\ (comp\ a\ (Val\ n) +\!\!+\ ops) \\
= \ & \{ \text{ definition of } comp \ \} \\
& unwind\ s\ ([PUSH\ n] +\!\!+\ ops) \\
= \ & \{ \text{ unwinding operators (lemma 5.3.2) } \} \\
& unwind\ s\ ops
\end{aligned}$$

Case: $e = Throw$

$$\begin{aligned}
& unwind\ s\ (comp\ a\ Throw +\!\!+\ ops) \\
= \ & \{ \text{ definition of } comp \ \} \\
& unwind\ s\ ([THROW] +\!\!+\ ops) \\
= \ & \{ \text{ unwinding operators } \} \\
& unwind\ s\ ops
\end{aligned}$$

Case: $e = Add\ x\ y$

Let

$$(_, b) = comp'\ a\ x$$
$$(_, c) = comp'\ b\ y$$

in

$$unwind\ s\ (comp\ a\ (Add\ x\ y) +\!\!+\ ops)$$

$$
\begin{aligned}
= \; & \{ \text{ definition of } \mathit{comp} \; \} \\
& \mathit{unwind} \; s \; (\mathit{comp} \; a \; x \mathbin{+\!\!+} \mathit{comp} \; b \; y \mathbin{+\!\!+} [\mathit{ADD}] \mathbin{+\!\!+} \mathit{ops}) \\
= \; & \{ \text{ induction hypothesis } \} \\
& \mathit{unwind} \; s \; (\mathit{comp} \; b \; y \mathbin{+\!\!+} [\mathit{ADD}] \mathbin{+\!\!+} \mathit{ops}) \\
= \; & \{ \text{ induction hypothesis } \} \\
& \mathit{unwind} \; s \; ([\mathit{ADD}] \mathbin{+\!\!+} \mathit{ops}) \\
= \; & \{ \text{ unwinding operators } \} \\
& \mathit{unwind} \; s \; \mathit{ops}
\end{aligned}
$$

The two induction hypothesis steps in the above proof require that $\mathit{isFresh} \; a \; s$ and $\mathit{isFresh} \; b \; s$ respectively, the first of which is true by assumption, and the second of which was verified in the previous section.

Case: $e = \mathit{Catch} \; x \; h$

Let

$$
\begin{aligned}
b \; & = \mathit{fresh} \; a \\
c \; & = \mathit{fresh} \; b \\
(_, d) & = \mathit{comp}' \; c \; x \\
(_, e) & = \mathit{comp}' \; d \; h
\end{aligned}
$$

in

$$
\begin{aligned}
& \mathit{unwind} \; s \; (\mathit{comp} \; a \; (\mathit{Catch} \; x \; h) \mathbin{+\!\!+} \mathit{ops}) \\
= \; & \{ \text{ definition of } \mathit{comp} \; \} \\
& \mathit{unwind} \; s \; ([\mathit{MARK} \; a] \mathbin{+\!\!+} \mathit{comp} \; c \; x \mathbin{+\!\!+} [\mathit{UNMARK}, \mathit{JUMP} \; b, \\
& \quad \mathit{LABEL} \; a] \mathbin{+\!\!+} \mathit{comp} \; d \; h \mathbin{+\!\!+} [\mathit{LABEL} \; b] \mathbin{+\!\!+} \mathit{ops}) \\
= \; & \{ \text{ unwinding operators } \} \\
& \mathit{unwind} \; s \; (\mathit{comp} \; c \; x \mathbin{+\!\!+} [\mathit{UNMARK}, \mathit{JUMP} \; b, \\
& \quad \mathit{LABEL} \; a] \mathbin{+\!\!+} \mathit{comp} \; d \; h \mathbin{+\!\!+} [\mathit{LABEL} \; b] \mathbin{+\!\!+} \mathit{ops}) \\
= \; & \{ \text{ induction hypothesis } \} \\
& \mathit{unwind} \; s \; ([\mathit{UNMARK}, \mathit{JUMP} \; b, \mathit{LABEL} \; a] \mathbin{+\!\!+} \\
& \quad \mathit{comp} \; d \; h \mathbin{+\!\!+} [\mathit{LABEL} \; b] \mathbin{+\!\!+} \mathit{ops})
\end{aligned}
$$

$$
\begin{aligned}
= &\quad \{ \text{ unwinding operators } \} \\
&\quad \mathit{unwind}\ s\ (\mathit{comp}\ d\ h \mathbin{+\!\!+} [\mathit{LABEL}\ b] \mathbin{+\!\!+} \mathit{ops}) \\
= &\quad \{ \text{ induction hypothesis } \} \\
&\quad \mathit{unwind}\ s\ ([\mathit{LABEL}\ b] \mathbin{+\!\!+} \mathit{ops}) \\
= &\quad \{ \text{ unwinding operators } \} \\
&\quad \mathit{unwind}\ s\ \mathit{ops}
\end{aligned}
$$

The last four steps in the above proof require that $\mathit{isFresh}\ \alpha\ s$ for $\alpha = c, a, d, b$ respectively, all of which were verified in the previous section. □

**Lemma 5.3.4 (*isFresh* is monotonic).**

$$a \leqslant b \wedge \mathit{isFresh}\ a\ s \Rightarrow \mathit{isFresh}\ b\ s$$

That is, if one address is at most another, and the first is fresh with respect to a stack, then the second is also fresh with respect to this stack.

*Proof.* By induction on $s :: \mathit{Stack}$.

Case: $s = [\,]$

$$
\begin{aligned}
&\quad \mathit{isFresh}\ b\ [\,] \\
\Leftrightarrow &\quad \{ \text{ definition of } \mathit{isFresh} \ \} \\
&\quad \mathit{True}
\end{aligned}
$$

Case: $s = \mathit{VAL}\ n : s'$

$$
\begin{aligned}
&\quad \mathit{isFresh}\ b\ (\mathit{VAL}\ n : s') \\
\Leftrightarrow &\quad \{ \text{ definition of } \mathit{isFresh} \ \} \\
&\quad \mathit{isFresh}\ b\ s' \\
\Leftarrow &\quad \{ \text{ induction hypothesis } \} \\
&\quad a \leqslant b \ \wedge\ \mathit{isFresh}\ a\ s' \\
\Leftrightarrow &\quad \{ \text{ definition of } \mathit{isFresh} \ \} \\
&\quad a \leqslant b \ \wedge\ \mathit{isFresh}\ a\ (\mathit{VAL}\ n : s')
\end{aligned}
$$

Case: $s = HAN\ c : s'$

$$
\begin{array}{ll}
 & \mathit{isFresh}\ b\ (HAN\ c : s') \\
\Leftrightarrow & \{ \text{definition of } \mathit{isFresh} \} \\
 & b > c \ \wedge \ \mathit{isFresh}\ b\ s' \\
\Leftarrow & \{ \text{induction hypothesis} \} \\
 & b > c \ \wedge \ a \leqslant b \ \wedge \ \mathit{isFresh}\ a\ s' \\
\Leftarrow & \{ \text{logic} \} \\
 & a > c \ \wedge \ a \leqslant b \ \wedge \ \mathit{isFresh}\ a\ s' \\
\Leftrightarrow & \{ \text{commutativity of } \wedge \} \\
 & a \leqslant b \ \wedge \ a > c \ \wedge \ \mathit{isFresh}\ a\ s' \\
\Leftrightarrow & \{ \text{definition of } \mathit{isFresh} \} \\
 & a \leqslant b \ \wedge \ \mathit{isFresh}\ a\ (HAN\ c : s')
\end{array}
$$

□

**Lemma 5.3.5 (*compile* is non-decreasing).**

$$\mathit{snd}\ (\mathit{comp}'\ a\ e) \geqslant a$$

That is, the next address returned by the compiler will always be greater than or equal to the address supplied as an argument.

*Proof.* By induction on $e :: \mathit{Expr}$

Case: $e = \mathit{Val}\ n$

$$
\begin{array}{ll}
 & \mathit{snd}\ (\mathit{comp}'\ a\ (\mathit{Val}\ n)) \\
= & \{ \text{definition of } \mathit{comp}' \} \\
 & a
\end{array}
$$

Case: $e = \mathit{Throw}$

$$
\begin{array}{ll}
 & \mathit{snd}\ (\mathit{comp}'\ a\ \mathit{Throw}) \\
= & \{ \text{definition of } \mathit{comp}' \} \\
 & a
\end{array}
$$

Case: $e = Add\ x\ y$

Let

$$(_, b) = comp'\ a\ x$$
$$(_, c) = comp'\ b\ y$$

in

$$
\begin{array}{cl}
 & snd\ (comp'\ a\ (Add\ x\ y)) \\
= & \{\ \text{definition of } comp'\ \} \\
 & c \\
= & \{\ \text{definition of } c\ \} \\
 & snd\ (comp'\ b\ y) \\
\geqslant & \{\ \text{induction hypothesis}\ \} \\
 & b \\
= & \{\ \text{definition of } b\ \} \\
 & snd\ (comp'\ a\ x) \\
\geqslant & \{\ \text{induction hypothesis}\ \} \\
 & a
\end{array}
$$

Case: $e = Catch\ x\ h$

Let

$$
\begin{array}{ll}
b & = fresh\ a \\
c & = fresh\ b \\
(_, d) & = comp'\ c\ x \\
(_, e) & = comp'\ d\ h
\end{array}
$$

in

$$
\begin{array}{cl}
 & snd\ (comp'\ a\ (Catch\ x\ h)) \\
= & \{\ \text{definition of } comp'\ \} \\
 & e
\end{array}
$$

$$
\begin{aligned}
= \quad & \{ \text{definition of } e \} \\
& \mathit{snd}\ (\mathit{comp}'\ d\ h) \\
\geqslant \quad & \{ \text{induction hypothesis} \} \\
& d \\
= \quad & \{ \text{definition of } d \} \\
& \mathit{snd}\ (\mathit{comp}'\ c\ x) \\
\geqslant \quad & \{ \text{induction hypothesis} \} \\
& c \\
> \quad & \{ \text{definition of } c \} \\
& b \\
> \quad & \{ \text{definition of } b \} \\
& a
\end{aligned}
$$

□

**Lemma 5.3.6 (jumping compiled code).**

$$a < b \Rightarrow \mathit{jump}\ a\ (\mathit{comp}\ b\ e +\!\!+\ \mathit{ops}) = \mathit{jump}\ a\ \mathit{ops}$$

That is, jumping to an address in compiled code followed by arbitrary additional code gives the same result as simply jumping in the additional code, provided that the jump address is less than the initial address for the compiler.

*Proof.* By induction on $e :: \mathit{Expr}$, using lemma 5.3.5 above

Case: $e = \mathit{Val}\ n$

$$
\begin{aligned}
& \mathit{jump}\ a\ (\mathit{comp}\ b\ (\mathit{Val}\ n) +\!\!+\ \mathit{ops}) \\
= \quad & \{ \text{definition of } \mathit{comp} \} \\
& \mathit{jump}\ a\ ([\mathit{PUSH}\ n] +\!\!+\ \mathit{ops}) \\
= \quad & \{ \text{definition of } \mathit{jump} \} \\
& \mathit{jump}\ a\ \mathit{ops}
\end{aligned}
$$

Case: $e = \mathit{Throw}$

$$
\begin{array}{ll}
 & \mathit{jump}\ a\ (\mathit{comp}\ b\ \mathit{Throw} \mathbin{+\!\!+} \mathit{ops}) \\
= & \{\ \text{definition of } \mathit{comp}\ \} \\
 & \mathit{jump}\ a\ ([\mathit{THROW}] \mathbin{+\!\!+} \mathit{ops}) \\
= & \{\ \text{definition of } \mathit{jump}\ \} \\
 & \mathit{jump}\ a\ \mathit{ops}
\end{array}
$$

Case: $e = \mathit{Add}\ x\ y$

Let

$$
\begin{array}{l}
(_, c) = \mathit{comp}'\ b\ x \\
(_, _) = \mathit{comp}'\ c\ y
\end{array}
$$

in

$$
\begin{array}{ll}
 & \mathit{jump}\ a\ (\mathit{comp}\ b\ (\mathit{Add}\ x\ y) \mathbin{+\!\!+} \mathit{ops}) \\
= & \{\ \text{definition of } \mathit{comp}\ \} \\
 & \mathit{jump}\ a\ (\mathit{comp}\ b\ x \mathbin{+\!\!+} \mathit{comp}\ c\ y \mathbin{+\!\!+} [\mathit{ADD}] \mathbin{+\!\!+} \mathit{ops}) \\
= & \{\ \text{induction hypothesis}\ \} \\
 & \mathit{jump}\ a\ (\mathit{comp}\ c\ y \mathbin{+\!\!+} [\mathit{ADD}] \mathbin{+\!\!+} \mathit{ops}) \\
= & \{\ \text{induction hypothesis}\ \} \\
 & \mathit{jump}\ a\ ([\mathit{ADD}] \mathbin{+\!\!+} \mathit{ops}) \\
= & \{\ \text{definition of } \mathit{jump}\ \} \\
 & \mathit{jump}\ a\ \mathit{ops}
\end{array}
$$

The first induction hypothesis step in the above proof requires that $a < b$, which is true by assumption, and the second that $a < c$, which is verified as follows:

$$
\begin{array}{ll}
 & a \\
< & \{\ \text{assumption}\ \} \\
 & b \\
\leqslant & \{\ \mathit{comp}' \text{ is non-decreasing (lemma 5.3.5)}\ \} \\
 & \mathit{snd}\ (\mathit{comp}'\ b\ x)
\end{array}
$$

$=\quad \{$ definition of $c\ \}$
$\quad c$

Case: $e = \mathit{Catch}\ x\ h$

Let

$$
\begin{aligned}
c &= \mathit{fresh}\ b \\
d &= \mathit{fresh}\ c \\
(_, e) &= \mathit{comp}'\ d\ x \\
(_, _) &= \mathit{comp}'\ e\ h
\end{aligned}
$$

in

$$
\begin{aligned}
& \mathit{jump}\ a\ (\mathit{comp}\ b\ (\mathit{Catch}\ x\ h) +\!\!+ \mathit{ops}) \\
=\ & \{ \text{ definition of } \mathit{comp} \ \} \\
& \mathit{jump}\ a\ ([\mathit{MARK}\ b] +\!\!+ \mathit{comp}\ d\ x +\!\!+ [\mathit{UNMARK}, \mathit{JUMP}\ c, \\
& \quad \mathit{LABEL}\ b] +\!\!+ \mathit{comp}\ e\ h +\!\!+ [\mathit{LABEL}\ c] +\!\!+ \mathit{ops}) \\
=\ & \{ \text{ definition of } \mathit{jump} \ \} \\
& \mathit{jump}\ a\ (\mathit{comp}\ d\ x +\!\!+ [\mathit{UNMARK}, \mathit{JUMP}\ c, \\
& \quad \mathit{LABEL}\ b] +\!\!+ \mathit{comp}\ e\ h +\!\!+ [\mathit{LABEL}\ c] +\!\!+ \mathit{ops}) \\
=\ & \{ \text{ induction hypothesis } \} \\
& \mathit{jump}\ a\ ([\mathit{UNMARK}, \mathit{JUMP}\ c, \mathit{LABEL}\ b]\ +\!\!+ \\
& \quad \mathit{comp}\ e\ h +\!\!+ [\mathit{LABEL}\ c] +\!\!+ \mathit{ops}) \\
=\ & \{ \text{ definition of } \mathit{jump} \ \} \\
& \mathit{jump}\ a\ (\mathit{comp}\ e\ h +\!\!+ [\mathit{LABEL}\ c] +\!\!+ \mathit{ops}) \\
=\ & \{ \text{ induction hypothesis } \} \\
& \mathit{jump}\ a\ ([\mathit{LABEL}\ c] +\!\!+ \mathit{ops}) \\
=\ & \{ \text{ definition of } \mathit{jump} \ \} \\
& \mathit{jump}\ a\ \mathit{ops}
\end{aligned}
$$

The last four steps in the above proof require that $a < d$, $a \not\equiv b$, $a < e$ and $a \not\equiv c$ respectively, all of which are verified by the following calculation:

$$
\begin{array}{ll}
 & a \\
< & \{ \text{ assumption } \} \\
 & b \\
< & \{ \text{ definition of } c \ \} \\
 & c \\
< & \{ \text{ definition of } d \ \} \\
 & d \\
\leqslant & \{ \ comp' \text{ is non-decreasing (lemma 5.3.5) } \} \\
 & snd\ (comp'\ d\ x) \\
= & \{ \text{ definition of } e \ \} \\
 & e
\end{array}
$$

□

### 5.3.2 Continuation Passing Style Compiler

The previous section showed how to avoid the *skip* function by using explicit jumps. In this section we consider another approach. A Continuation Passing Style (CPS) [App92] compiler takes as its arguments not only the expression to be compiled, but also the code to be executed after that expression. Due to the fact that we have direct access to not only the handler, but the rest of the code to execute, we no longer need to skip code, and can simply push all of the code to execute if an exception is raised onto the stack. This has the effect that the compiler is considerably simpler to prove correct, whilst also modelling exception handling effectively. Because the proofs are simpler, we will continue with a modified version of this compiler throughout the remainder of this thesis.

For the purposes of defining a continuation passing version of the compiler, no operations are needed beyond those required for the original compiler:

$$
\begin{array}{l}
\mathbf{data}\ Op = PUSH\ Int \mid ADD \mid POP \\
\qquad\qquad \mid\ THROW \mid MARK\ Code \mid UNMARK
\end{array}
$$

The compiler itself is redefined as follows:

$$
\begin{array}{ll}
\mathit{comp} & :: \mathit{Expr} \rightarrow \mathit{Code} \\
\mathit{comp}\ e & = \mathit{comp}'\ e\ [\,] \\
\mathit{comp}' & :: \mathit{Expr} \rightarrow \mathit{Code} \rightarrow \mathit{Code} \\
\mathit{comp}'\ (\mathit{Val}\ n)\ \mathit{ops} & = \mathit{PUSH}\ n : \mathit{ops} \\
\mathit{comp}'\ (\mathit{Throw})\ \mathit{ops} & = \mathit{THROW} : \mathit{ops} \\
\mathit{comp}'\ (\mathit{Add}\ x\ y)\ \mathit{ops} & = \mathit{comp}'\ x\ (\mathit{comp}'\ y\ (\mathit{ADD} : \mathit{ops})) \\
\mathit{comp}'\ (\mathit{Catch}\ x\ h)\ \mathit{ops} & = \mathit{MARK}\ (\mathit{comp}'\ h\ \mathit{ops}) : \mathit{comp}'\ x\ (\mathit{UNMARK} : \mathit{ops}) \\
\mathit{comp}'\ (\mathit{Seq}\ x\ y)\ \mathit{ops} & = \mathit{comp}'\ x\ (\mathit{POP} : \mathit{comp}'\ y\ \mathit{ops})
\end{array}
$$

The interpreter is similar to our original version, except that *skip* is no longer required:

$$
\begin{array}{ll}
\mathit{intpt} & :: \mathit{Stack} \rightarrow \mathit{Code} \rightarrow \mathit{Stack} \\
\mathit{intpt}\ s\ [\,] & = s \\
\mathit{intpt}\ s\ (\mathit{PUSH}\ x : \mathit{ops}) & = \mathit{intpt}\ (\mathit{VAL}\ x : s)\ \mathit{ops} \\
\mathit{intpt}\ (\mathit{VAL}\ x : s)\ (\mathit{POP} : \mathit{ops}) & = \mathit{intpt}\ s\ \mathit{ops} \\
\mathit{intpt}\ (\mathit{VAL}\ y : \mathit{VAL}\ x : s)\ (\mathit{ADD} : \mathit{ops}) & = \mathit{intpt}\ (\mathit{VAL}\ (x + y) : s)\ \mathit{ops} \\
\mathit{intpt}\ s\ (\mathit{THROW} : \mathit{ops}) & = \mathit{unwind}\ s \\
\mathit{intpt}\ s\ (\mathit{MARK}\ \mathit{ops}' : \mathit{ops}) & = \mathit{intpt}\ (\mathit{HAN}\ \mathit{ops}' : s)\ \mathit{ops} \\
\mathit{intpt}\ s\ (\mathit{UNMARK} : \mathit{ops}) & = \mathbf{case}\ s\ \mathbf{of} \\
 & \quad (x : \mathit{HAN}\ _ : s') \rightarrow \mathit{intpt}\ (x : s')\ \mathit{ops} \\
\mathit{unwind} & :: \mathit{Stack} \rightarrow \mathit{Stack} \\
\mathit{unwind}\ [\,] & = [\,] \\
\mathit{unwind}\ (\mathit{VAL}\ _ : s) & = \mathit{unwind}\ s \\
\mathit{unwind}\ (\mathit{HAN}\ \mathit{ops}' : s) & = \mathit{intpt}\ s\ \mathit{ops}'
\end{array}
$$

This compiler avoids the need for skipping through code, and while it may seem odd to push the entire remaining program code onto the stack each time we compile a handler, this approach models the idea of pushing just the handler code and a pointer to the remainder of the program onto the stack. We do not provide a proof of correctness here, because we shall be using a modified version of this compiler for the rest of this thesis, and a proof of correctness for this style of compiler will follow in Chapter 7.

## 5.4 Summary

In this chapter we have demonstrated a variety of simple methods for compiling exception handling code, and have seen how these variations affect reasoning about the compilers. The proof of correctness for our simple compiler is no more complex than that seen in Chapter 3, however it does require us to consider many more cases at each stage, and occasionally simple supporting lemmas. We have also introduced two new forms of the compiler; the explicit jump compiler, which significantly increased the length and complexity of our proofs, and the continuation passing style compiler, which we shall be using throughout the rest of this thesis. In the next Chapter we consider an example of an interesting combinator, built from the primitives of our language, which we specify and prove correct using our semantics.

# Chapter 6

# *finally*, an Example

In this chapter we shall introduce an example of a higher-level exception handling combinator. We call this combinator *finally*, and it models a useful and common programming pattern. We shall implement this combinator in terms of the language primitives already described, and using the semantics defined in the previous chapters we shall give a proof of its correctness according a specification, which we define first.

## 6.1 What is *finally*?

The primitives of our language are rather low level, and in the context of a real programming language, using them is prone to errors. A useful, higher-level combinator is *finally* $x$ $y$, which is available in both the Haskell and Java programming languages. The behaviour of this combinator is described loosely in *Asynchronous Exception in Haskell* [MPMR01] as, "do $x$, then whatever happens do $y$". From the point of view of proving correctness this specification is rather vague: it does not specify what should happen to any exception raised by $x$, only that $y$ should be evaluated, and it does not specify how many times $x$ and $y$ should be evaluated. We now give a more precise specification for *finally*:

- If $x$ raises an exception, $y$ is evaluated and the exception is propagated.
- If $x$ does not raise an exception, $y$ is evaluated and the evaluation continues normally.
- Both $x$ and $y$ should be evaluated exactly once.

This combinator can be used for executing important *clean up* code, for example closing file handles and deallocating resources. These activities are essential for the correct functioning of a software system, because being unable to write results to disk and memory leaks caused by raised exceptions are unacceptable. We now proceed to produce a definition of *finally*.

## 6.2 *finally*, a Definition

In this section we explore a number of attempts to define the *finally* operator described above. We shall look at a number of definitions, and for each, give both an informal and a formal argument regarding its correctness.

- $\mathit{finally}\ x\ y\ =\ x; y$

  Our first attempt at implementing *finally* is simply $x$ and $y$ in sequence. This works for the case where $x$ evaluates successfully, but according to our semantics, $y$ will never be evaluated if $x$ raises an exception. Formally, the failure of this definition to work properly can be shown by the following evaluation tree, in which $x$ is evaluated and $y$ is not:

$$\dfrac{\dfrac{\nabla}{x \Downarrow \mathit{throw}}}{x; y \Downarrow \mathit{throw}}\ \textsc{Seq3}$$

  In the above, $\nabla$ denotes some possible evaluation tree.

- $\mathit{finally}\ x\ y\ =\ (\mathit{catch}\ x\ y); y$

  Here we fix the problem of $y$ not being evaluated when $x$ raises an exception, however we have introduced another problem. When an exception is raised by $x$, $y$ is evaluated, but is evaluated twice and the exception raised by $x$ is not propagated. Again we can demonstrate this problem by drawing the evaluation tree for this case:

$$\dfrac{\dfrac{\dfrac{\nabla}{x \Downarrow throw} \quad \dfrac{\nabla}{y \Downarrow \overline{n}}}{catch\ x\ y \Downarrow \overline{n}}\ \textsc{catch1} \qquad \dfrac{\nabla}{y \Downarrow \overline{n}}}{(catch\ x\ y); y \Downarrow \overline{n}}\ \textsc{Seq1}$$

Note in particular, two evaluation trees for $y$.

- $finally\ x\ y = (catch\ x\ (y; throw)); y$

  We again refine our previous definition to resolve the issue of $y$ being evaluated twice when $x$ raises an exception, by propagating an exception raised by $x$ after evaluating $y$. By inspection this definition would appear to be correct: if $x$ evaluates successfully then $y$ is run once, and if $x$ raises an exception $y$ is run once and the exception is propagated. This should mean that the second $y$ is never run. We now proceed to prove the correctness of this definition according to our specification.

## 6.3 Correctness of *finally*

We can demonstrate the correctness of *finally* using our big-step semantics because we can draw all the possible evaluation trees for an expression of the form *finally* $x$ $y$. We proved earlier that all expressions evaluate to either *throw* or a number, so using the big-step semantics we can show that whenever $x$ is evaluated $y$ is also evaluated, and that all the other criteria for a correct implementation of *finally* are met. For the trees, this amounts to showing that all possible evaluation trees have exactly one evaluation of $x$ and one evaluation of $y$:

At the top level the expression *finally* $x$ $y$ is a sequence of two expressions. First therefore, we need to consider every possible evaluation rule for sequencing, namely Seq1, Seq2 and Seq3 rules, and build up all the trees which can produce those results.

- Here we apply the Seq1 rule, meaning that both the *catch* and $y$ expression *must* evaluate successfully to numbers. Because the second element of the *catch* is a se-

quence with *throw* as its second element the only way we can produce a valid tree is if the $x$ also evaluates to a number:

$$\dfrac{\dfrac{\dfrac{\nabla}{x \Downarrow \overline{n}}}{catch\ x\ (y; throw) \Downarrow \overline{n}}\ \textsc{Catch1} \qquad \dfrac{\nabla}{y \Downarrow \overline{m}}}{catch\ x\ (y; throw); y \Downarrow \overline{m}}\ \textsc{Seq1}$$

- Here we apply the SEQ2 rule, which can only be applied if the *catch* expression evaluates to a number and $y$ evaluates to *throw*. Again this means that the expression $x$ must evaluate to a number, and the rule can be applied in only one way:

$$\dfrac{\dfrac{\dfrac{\nabla}{x \Downarrow \overline{n}}}{catch\ x\ (y; throw) \Downarrow \overline{n}}\ \textsc{Catch1} \qquad \dfrac{\nabla}{y \Downarrow throw}}{catch\ x\ (y; throw); y \Downarrow throw}\ \textsc{Seq2}$$

- Here we apply the SEQ3, which can be applied when the first element of the sequence of expressions (the *catch* expression) raises an exception. This means that the expression $x$ must evaluate to *throw*, however it does not force the evaluation of $y$. We therefore consider both cases (as $y$ can only evaluate to a number or *throw*):

  1. $y \Downarrow \overline{n}$

$$\dfrac{\dfrac{\dfrac{\nabla}{x \Downarrow throw} \qquad \textsc{Seq2}\ \dfrac{\dfrac{\nabla}{y \Downarrow \overline{n}} \qquad \textsc{Throw}\ \dfrac{}{throw \Downarrow throw}}{y; throw \Downarrow throw}}{catch\ x\ (y; throw) \Downarrow throw}\ \textsc{Catch3}}{catch\ x\ (y; throw); y \Downarrow throw}\ \textsc{Seq3}$$

2. $y \Downarrow throw$

$$
\dfrac{\dfrac{\nabla}{x \Downarrow throw} \quad \textsc{Seq3}\ \dfrac{\dfrac{\nabla}{y \Downarrow throw}}{y; throw \Downarrow throw}}{\dfrac{catch\ x\ (y; throw) \Downarrow throw}{catch\ x\ (y; throw); y \Downarrow throw}\ \textsc{Seq3}}\ \textsc{Catch3}
$$

By showing that all possible evaluation trees satisfy our specification we have shown the correctness of our definition of *finally*. In particular, note that each form of evaluation tree above satisfies our three requirements stated at the start of this chapter.

## 6.4 Summary

In this chapter we have defined a combinator using the primitives of our minimal language and produced a proof of its behaviour using the rules for our big-step semantics. Reasoning about combinators using the big-step semantics and some simple requirements for evaluation trees is very convenient, and easily checked. This contrasts strongly with proving behaviour for the low-level machine semantics. We now proceed to investigate the addition of interrupts to our language.

CHAPTER 7

# Interrupts

In this chapter we start to consider *interrupts*. As we noted in Chapter 1, interrupts are exceptions that do not arise as a direct result of a computation, and therefore can occur at any time. We wish to consider the worst-case scenario of interrupts, so we introduce a *demonic environment* in which our expressions are evaluated, and in which our virtual machine executes. At this point we also stop considering our small-step and Haskell definitions of the semantics and introduce a new relational definition of our virtual machine, due to the fact that evaluation and execution are now non-deterministic.

## 7.1 Big-Step Semantics and Interrupts

Interrupts can occur at any time, and are not caused by the evaluation of any particular expression. The delivery of an interrupt can be viewed as a special kind of raised exception. Even though interrupts are normally caused by some form of concurrently running process, either externally or even from another thread in the program, we need not consider concurrency explicitly, as no synchronisation or communication is necessary for delivering or recovering from interrupts. We attempt to introduce a simple extension to our big-step semantics which acts as a worst-case interrupt generator:

$$\frac{}{x \Downarrow \mathit{throw}} \ \textsc{Interrupt}_b$$

That is, any expression can be interrupted, and therefore raise an exception, at any time. The introduction of this rule, though simple, has considerable implications for our semantics, and we must now answer a number of important questions: What effect does this new rule have on our definition of *finally*? What effect does it have on reasoning about programs? and most importantly, is this the *right* rule for introducing interrupts?

We can answer the first question immediately, and unfortunately, our previous definition of *finally* now no longer fulfills our specification, as it is possible to show an evaluation of the expression *finally* $x$ $y$ that evaluates $x$ but never evaluates $y$:

$$\dfrac{\dfrac{\dfrac{\nabla}{x \Downarrow \mathit{throw}} \quad \dfrac{}{y;\mathit{throw} \Downarrow \mathit{throw}}\,\textsc{Interrupt}_b}{\mathit{catch}\ x\ (y : \mathit{throw}) \Downarrow \mathit{throw}}\,\textsc{Catch}_b2}{\mathit{catch}\ x\ (y;\mathit{throw}); y \Downarrow \mathit{throw}}\,\textsc{Seq}_b2$$

In fact, as the interrupt rule stands it is impossible to define an interrupt safe version of *finally*, simply because an interrupt may occur at any time during evaluation, which effectively answers our third question in the negative. In the presence of this rule, no guarantees can be given about the evaluation of an expression, and we must therefore attempt to modify it. We now attempt to define a worst-case interrupt generator, with which effective programming is still possible.

Inspired by the approach taken in *Asynchronous Exceptions in Haskell* [MPMR01] we now attempt to fix the problem of interrupts being delivered at any time by adding two scoping operators, *block* and *unblock*, to our language:

$$\begin{aligned} \mathbb{E} ::= &\ \ldots \\ |\ &\ \mathit{block}\ \mathbb{E} \\ |\ &\ \mathit{unblock}\ \mathbb{E} \end{aligned}$$

These new operators determine if an interrupt may be delivered during execution by changing the value of an *interrupt state* to either *blocked* or *unblocked*; interrupts may only be delivered if the current interrupt state is unblocked. We also extend our semantics, to deal

with both the new operators, and to provide support for the interrupt state. Our semantics is now an evaluation relation of the form $e \Downarrow^i e'$, where $e$ and $e'$ are expressions in our language, and $i \in \{b, u\}$ is an interrupt state, with $b$ meaning blocked, and $u$, unblocked. First of all, we define rules for *block* and *unblock*:

$$\frac{x \Downarrow^b v}{\mathit{block}\ x \Downarrow^i v}\ \textsc{Block}_b \qquad \frac{x \Downarrow^u v}{\mathit{unblock}\ x \Downarrow^i v}\ \textsc{Unblock}_b$$

That is, evaluating *block* $x$ in any interrupt state, $i$, is the result of evaluating $x$ in a blocked state, and conversely, evaluating *unblock* $x$ is the result of evaluating $x$ in an unblocked state. It should be noted that we do not perform any counting of states, e.g. the expression *block* (*block* (*unblock* $x$)) still evaluates $x$ in an unblocked state.

We also modify our previous INTERRUPT rule to only allow an interrupt to be delivered when the interrupt state is unblocked:

$$\frac{}{x \Downarrow^u \mathit{throw}}\ \textsc{Interrupt}_b$$

To complete this extension to the language we must also modify our original rules to propagate the current interrupt state:

$$\frac{}{\overline{n} \Downarrow^i \overline{n}}\ \textsc{Val}_b \qquad \frac{}{\mathit{throw} \Downarrow^i \mathit{throw}}\ \textsc{Throw}_b$$

$$\frac{x \Downarrow^i \overline{n} \qquad y \Downarrow^i \overline{m}}{x + y \Downarrow^i \overline{n+m}}\ \textsc{Add}_b1 \qquad \frac{x \Downarrow^i \mathit{throw}}{x + y \Downarrow^i \mathit{throw}}\ \textsc{Add}_b2 \qquad \frac{x \Downarrow^i \overline{n} \qquad y \Downarrow^i \mathit{throw}}{x + y \Downarrow^i \mathit{throw}}\ \textsc{Add}_b3$$

$$\frac{x \Downarrow^i \overline{n}}{\mathit{catch}\ x\ y \Downarrow^i \overline{n}}\ \textsc{Catch}_b 1 \qquad \frac{x \Downarrow^i \mathit{throw} \qquad y \Downarrow^i v}{\mathit{catch}\ x\ y \Downarrow^i v}\ \textsc{Catch}_b 2$$

$$\frac{x \Downarrow^i \overline{n} \qquad y \Downarrow^i v}{x; y \Downarrow^i v}\ \textsc{Seq}_b 1 \qquad \frac{x \Downarrow^i \mathit{throw}}{x; y \Downarrow^i \mathit{throw}}\ \textsc{Seq}_b 2$$

These changes have the effect that the semantics is now non-deterministic. This is demonstrated by the following valid evaluations of the expression $\overline{1} + \overline{2}$:

$$\frac{}{\overline{1} + \overline{2} \Downarrow^u \mathit{throw}}\ \textsc{Interrupt}_b \qquad \frac{\dfrac{}{\overline{1} \Downarrow^u \overline{1}}\ \textsc{Val}_b \qquad \dfrac{}{\overline{2} \Downarrow^u \overline{2}}\ \textsc{Val}_b}{\overline{1} + \overline{2} \Downarrow^u \overline{3}}\ \textsc{Add}_b 1$$

These changes to the language and semantics should allow us to write programs in the presence of interrupts; we can now limit the points at which interrupts may be delivered, allowing them to have useful effects, yet without them hindering our ability to write effective programs. We now move on to produce a new definition of our compiler and virtual machine which provide support for interrupts. We shall follow the same development process when adding interrupts to the machine that we have employed so far in this chapter.

## 7.2 Machine Interrupts

We adopt the same approach to introducing interrupts in the virtual machine as we did with the semantics. We would like to begin as before, by introducing a naive interrupt rule. This is immediately a problem, as our previous definition of the virtual machine does not allow for this easily, because by adding non-determinism our interpreter can no longer be clearly defined as a function. We therefore begin by introducing a relational definition of our previous virtual machine. Formally, we define the normal execution of the machine in terms of a step relation $\rightarrow$ on $<\mathit{Code}, \mathit{Stack}>$ pairs:

$$
\begin{array}{ll}
< \mathit{PUSH}\ n : \mathit{ops}, s > & \rightarrow\ < \mathit{ops}, \mathit{VAL}\ n : s > \\
< \mathit{ADD} : \mathit{ops}, \mathit{VAL}\ n : \mathit{VAL}\ m : s > & \rightarrow\ < \mathit{ops}, \mathit{VAL}\ (n+m) : s > \\
< \mathit{POP} : \mathit{ops}, \mathit{VAL}\ n : s > & \rightarrow\ < \mathit{ops}, s > \\
< \mathit{MARK}\ h : \mathit{ops}, s > & \rightarrow\ < \mathit{ops}, \mathit{HAN}\ h : s > \\
< \mathit{UNMARK} : \mathit{ops}, x : \mathit{HAN}\ y : s > & \rightarrow\ < \mathit{ops}, x : s > \\
< \mathit{THROW} : \mathit{ops}, s > & \rightarrow\ \ll s \gg
\end{array}
$$

The *unwind* function is also replaced by exceptional execution in our step relation $\rightarrow$, which operates only on a $\ll \mathit{Stack} \gg$:

$$
\begin{array}{ll}
\ll \mathit{VAL}\ n : s \gg & \rightarrow\ \ll s \gg \\
\ll \mathit{HAN}\ \mathit{ops} : s \gg & \rightarrow\ < \mathit{ops}, s >
\end{array}
$$

A first attempt to add support for interrupts to our new machine is similar to our first attempt to add interrupt support to the semantics:

$$
< \mathit{op} : \mathit{ops}, s > \ \rightarrow\ \ll s \gg
$$

That is, any operation can raise an exception, causing the virtual machine to begin unwinding the stack searching for an exception handler. However, this rule has the same deficiency as our first attempt at an INTERRUPT rule for the big-step semantics. We can now show a trace of *finally* $x\ y$ which never executes $y$, and a correct definition is now impossible because no operation is uninterruptible. An example of such a trace, given that $x \Downarrow \overline{n}$ and that $Cx$ and $Cy$ represent the compiled code for $x$ and $y$ respectively, is given below. Note that we abuse the list notation to include $Cx$ and $Cy$ in the list of operations.

$$
\begin{array}{ll}
 & < [\mathit{MARK}\ [Cy, \mathit{POP}, \mathit{THROW}, \mathit{POP}, Cy], Cx, \mathit{UNMARK}, \mathit{POP}, Cy], [-] > \\
\rightarrow & < [Cx, \mathit{UNMARK}, \mathit{POP}, Cy], [\mathit{HAN}\ [Cy, \mathit{POP}, \mathit{THROW}, \mathit{POP}, Cy]] > \\
\rightarrow^{*} & < [\mathit{UNMARK}, \mathit{POP}, Cy], [\mathit{VAL}\ n, \mathit{HAN}\ [Cy, \mathit{POP}, \mathit{THROW}, \mathit{POP}, Cy]] > \\
\rightarrow & \ll [\mathit{VAL}\ n, \mathit{HAN}\ [Cy, \mathit{POP}, \mathit{THROW}, \mathit{POP}, Cy]] \gg \\
\rightarrow & \ll [\mathit{HAN}\ [Cy, \mathit{POP}, \mathit{THROW}, \mathit{POP}, Cy]] \gg \\
\rightarrow & < [Cy, \mathit{POP}, \mathit{THROW}, \mathit{POP}, Cy], [-] > \\
\rightarrow & \ll [-] \gg
\end{array}
$$

We therefore extend our machine to include an interrupt state and a new interrupt rule. These additions behave similarly to the interrupt state and INTERRUPT rule for the semantics. We begin by extending the machine with two instructions which allow us to change

the interrupt state of an execution:

$$\begin{aligned}&\textbf{data}\ \mathit{Status} = \mathit{MASK} \mid \mathit{UNMASK}\\&\textbf{data}\ \mathit{Op} \quad\ \ = \ldots \mid \mathit{SET}\ \mathit{Status} \mid \mathit{RESET}\end{aligned}$$

These instructions allow us to set the interrupt state to either $\mathit{MASK}$, which corresponds to $b$ (or *blocked*) in the semantics, or to $\mathit{UNMASK}$, which corresponds to $u$ (or *unblocked*) in the semantics. We can now extend our continuation passing style compiler from Chapter 5 to include support for the *block* and *unblock* primitives described above:

$$\begin{aligned}&\mathit{comp}'\ (\mathit{block}\ x)\ \mathit{ops} \quad\ = \mathit{SET}\ \mathit{MASK} : \mathit{comp}'\ x\ (\mathit{RESET} : \mathit{ops})\\&\mathit{comp}'\ (\mathit{unblock}\ x)\ \mathit{ops} = \mathit{SET}\ \mathit{UNMASK} : \mathit{comp}'\ x\ (\mathit{RESET} : \mathit{ops})\end{aligned}$$

The two machine instructions, $\mathit{SET}$ and $\mathit{RESET}$, denote the start and end of the scope of a *block* or *unblock* primitive, similarly to the $\mathit{MARK}$ and $\mathit{UNMARK}$ instructions of a *catch* block. These new instructions change an interrupt state, and we therefore modify our virtual machine to keep track of this value. Formally, we extend the step relation $\rightarrow$ for normal execution to a set of rewrite rules on $<\mathit{Code}, \mathit{Status}, \mathit{Stack}>$ triples. In a similar way to the semantics, only the $\mathit{SET}$ and $\mathit{RESET}$ instructions modify the interrupt state, with the rest of the rules simply propagating the current state. We begin by introducing the new rules for $\mathit{SET}$ and $\mathit{RESET}$:

$$\begin{aligned}&< \mathit{SET}\ i' : \mathit{ops}, i, s > &&\rightarrow\ < \mathit{ops}, i', \mathit{INT}\ i : s >\\&< \mathit{RESET} : \mathit{ops}, i, x : \mathit{INT}\ i' : s > &&\rightarrow\ < \mathit{ops}, i', x : s >\end{aligned}$$

That is, the $\mathit{SET}$ instruction changes the interrupt state to either $\mathit{MASK}$ or $\mathit{UNMASK}$, indicated by its argument, and saves the current state on the stack. Conversely, the $\mathit{RESET}$ instruction retrieves a saved state from the stack and changes the current interrupt state accordingly. These two machine operations behave much like $\mathit{MARK}$ and $\mathit{UNMARK}$ for exceptions. The rest of the definition of the machine is extended in the obvious way, by simply propagating the interrupt status:

$$
\begin{array}{ll}
< \mathit{PUSH}\ n : \mathit{ops}, i, s > & \rightarrow\ < \mathit{ops}, i, \mathit{VAL}\ n : s > \\
< \mathit{ADDOP} : \mathit{ops}, i, \mathit{VAL}\ n : \mathit{VAL}\ m : s > & \rightarrow\ < \mathit{ops}, i, \mathit{VAL}\ n + m : s > \\
< \mathit{POP} : \mathit{ops}, i, \mathit{VAL}\ n : s > & \rightarrow\ < \mathit{ops}, i, s > \\
< \mathit{MARK}\ h : \mathit{ops}, i, s > & \rightarrow\ < \mathit{ops}, i, \mathit{HAN}\ h : s > \\
< \mathit{UNMARK} : \mathit{ops}, i, x : \mathit{HAN}\ y : s > & \rightarrow\ < \mathit{ops}, i, x : s > \\
< \mathit{THROW} : \mathit{ops}, i, s > & \rightarrow\ \ll i, s \gg
\end{array}
$$

We must also modify the unwinding portion of the virtual machine, which must now operate on $\ll \mathit{Status}, \mathit{Stack} \gg$ pairs. We add a rule so that when unwinding a stack, saved interrupt states are restored. This is important because proper recovery from a raised exception includes restoring the interrupt state in which the *catch* block was originally executed.

$$
\begin{array}{ll}
\ll i, \mathit{VAL}\ n : s \gg & \rightarrow\ \ll i, s \gg \\
\ll i, \mathit{INT}\ i' : s \gg & \rightarrow\ \ll i', s \gg \\
\ll i, \mathit{HAN}\ \mathit{ops} : s \gg & \rightarrow\ < \mathit{ops}, i, s >
\end{array}
$$

That is, if the top value of the stack is an integer it is simply discarded and we continue unwinding, if the top value is an interrupt state, that state is restored and we continue unwinding, and if the top stack element is an exception handler, we run the code contained in the handler. These changes to the definition of our compiler and virtual machine allow us to add a new interrupt rule in a similar way to the semantics:

$$< \mathit{op} : \mathit{ops}, \mathit{UNMASK}, s > \ \rightarrow\ \ll \mathit{UNMASK}, s \gg$$

That is, an interrupt may only be delivered if the currently executing interrupt state is *UNMASK*.

## 7.3 Compiler Correctness

Previously our language was deterministic, and we stated the correctness of the compiler as “the semantics and compiler always produce the same result”. Because we now express both the virtual machine and semantics as non-deterministic relations, and therefore must consider sets of possible results, we must rethink our approach to compiler correctness.

We now express the correctness of the compiler in two parts, the idea that a machine state *can reach* another machine state by applying the step relation, and the idea that a

machine state *will reach* another machine state by applying the step relation. By extending these ideas to cover sets of machine states we can produce a correctness result for our semantics which relates a machine starting state and the set of possible final states defined by the big-step semantics. We now proceed to formalise these two relations on machine states.

### 7.3.1 The *can reach* relation

As stated above we require two properties to define our correctness theorem, we therefore define the *can reach* relation, $\rightarrow^*$, on the step relation $\rightarrow$, which operates on both single machine states and sets of machine states:

- $x$ *can reach* $y$ in zero or more steps:

$$\frac{x = y}{x \rightarrow^* y}\ \textsc{CanEq} \qquad \frac{x \rightarrow x' \qquad x' \rightarrow^* y}{x \rightarrow^* y}\ \textsc{CanStep}$$

- $x$ *can reach* all of the set $Y$ in zero or more steps:

$$\frac{\forall y \in Y.\, x \rightarrow^* y}{x \rightarrow^* Y}\ \textsc{CanSet}$$

### 7.3.2 The *will reach* relation

The second property we require to state our compiler correctness theorem is the *will reach* relation. We define a relation, $\lhd$, on the step relation, $\rightarrow$, which operates on both single machine states and sets of machine states:

- $x$ *will reach* something in $Y$ in zero or more steps:

$$\frac{x \in Y}{x \lhd Y}\ \textsc{WillIn} \qquad \frac{\forall x'.\, x \rightarrow x' \ \Rightarrow\ x' \lhd Y}{x \lhd Y}\ \textsc{WillStep}$$

Note that $x \lhd Y$ cannot be defined simply by $\exists y \in Y.\, x \lhd y$, because this says that $x$ will always reach one specific $y$, not the weaker statement that $x$ will always reach something in $Y$.

- everything in $X$ *will reach* something in $Y$ in zero or more steps:

$$\frac{\forall x \in X.\, x \lhd Y}{X \lhd Y}\ \textsc{WillSet}$$

### 7.3.3 Stating Compiler Correctness

Using the two operators above, we can express the compiler correctness result we need. We define a relation, $\lll$, that is simply the combination of the two relations $\rightarrow^*$ and $\lhd$:

x can reach everything in Y and will reach something in Y:

$$\frac{x \rightarrow^* Y \qquad x \lhd Y}{x \lll Y}\ \textsc{BarStarred}$$

Compiler correctness can now be stated as follows:

**Theorem 7.3.1 (Interrupting Compiler Correctness).**

$$<\!\mathit{comp}\ e\ [\,], \mathit{UNMASK}, [\,]\!>$$
$$\lll$$
$$\{<\![\,], \mathit{UNMASK}, [\mathit{VAL}\ n]\!> \mid e \Downarrow^u \overline{n}\} \cup \{\ll \mathit{UNMASK}, [\,] \gg \mid e \Downarrow^u \mathit{throw}\}$$

That is, compiling an expression with no additional code, and then executing the resulting code with interrupts enabled using an empty stack can terminate with every number on the stack permitted by the semantics and can terminate with an uncaught exception if permitted by the semantics (the $\rightarrow^*$ part of the theorem), and moreover, will terminate with one of these possible outcomes (the $\lhd$ part of the theorem).

Note that if we simply used $\rightarrow^*$ the result would not be strong enough, as this would only say that from the start state we can reach everything in the resulting set, but does not rule out the case that there are paths that do not reach this set (i.e. the set of possible

results could be too small). Conversely, simply using $\lhd$ would only say that from the start state we will reach something in the resulting set, but does not ensure that we can reach everything (i.e. the set of possible results could be too big). Note also that we are using the fact that our semantics always produces a number or *throw* (lemma 7.3.2 in order to state our correctness theorem.

For the purposes of proofs, however, we generalise as follows:

$$<\mathit{comp}\ e\ \mathit{ops}, i, s> \\ \lhd\!\!\!\lhd \\ \{<\mathit{ops}, i, \mathit{VAL}\ n : s> \mid e \Downarrow^i \overline{n}\} \cup \{\ll i, s \gg \mid e \Downarrow^i \mathit{throw}\}$$

If we take $ops = s = [\,]$ and $i = \mathit{UNMASK}$, it is easy to show that this result simplifies to our original result above.

We prove the result in two parts, one of which proceeds using rule induction over the big-step semantics, and the other by structural induction over expressions. At present it is not clear whether the two parts can also be proved simultaneously.

**Lemma 7.3.2.**

$$x \Downarrow^i v \quad \Rightarrow \quad v \in \mathbb{Z} \cup \{\mathit{throw}\}$$

*Proof.* By induction on $x$ □

### 7.3.4 Proving the *can reach* relation

We begin by proving the first of the two relations which combine to make the $\lhd\!\!\!\lhd$ relation, the $\rightarrow^*$ relation:

$$<\mathit{comp}\ e\ \mathit{ops}, i, s> \\ \rightarrow^* \\ \{<\mathit{ops}, i, \mathit{VAL}\ n : s> \mid e \Downarrow^i \overline{n}\} \cup \{\ll i, s \gg \mid e \Downarrow^i \mathit{throw}\}$$

That is, an expression $e$ compiled with arbitrary continuing code *ops* and run in an interrupt status $i$ with an arbitrary initial stack $s$ can reach every element of the set comprising those

machine states corresponding to the possible evaluations in the big-step semantics with the same interrupt status $i$.

To begin the proof we rewrite our statement by applying lemma 7.3.7 to split the result into two parts:

$$<\!comp\ e\ ops, i, s\!> \rightarrow^* \{<\!ops, i, VAL\ n : s\!> \mid e \Downarrow^i \overline{n}\}$$
$$\wedge$$
$$<\!comp\ e\ ops, i, s\!> \rightarrow^* \{\ll i, s \gg \mid e \Downarrow^i throw\}$$

and then use the definition of $\rightarrow^*$ and logic to simplify these:

$$e \Downarrow^i \overline{n} \Rightarrow <\!comp\ e\ ops, i, s\!> \rightarrow^* <\!ops, i, VAL\ n : s\!>$$
$$\wedge$$
$$e \Downarrow^i throw \Rightarrow <\!comp\ e\ ops, i, s\!> \rightarrow^* \ll i, s \gg$$

Note that this now states precisely that whatever the semantics can do the machine can match. We might now expect to be able to verify both conjuncts separately using rule induction, but this doesn't work, because the proof of the first part requires the second part, and vice versa, and hence there would be a circularity in the proofs. The way around this is to prove both conjuncts at the same time using rule induction.

We now recall rule induction for $\Downarrow^i$. To show that some property $P\ (e, i, e')$ holds for all $e \Downarrow^i e'$, it suffices to show that $P\ (\overline{n}, i, \overline{n})$ holds, and that if $x \Downarrow^i \overline{n}$ and $y \Downarrow^i \overline{m}$ and $P\ (x, i, \overline{n})$ and $P\ (y, i, \overline{m})$ then $P\ (x + y, i, \overline{n + m})$, and similarly for all the other rules that define $\Downarrow^i$. In our case, the property $P\ (e, i, e')$ is defined as follows:

$$e' = \overline{n} \Rightarrow <\!comp\ e\ ops, i, s\!> \rightarrow^* <\!ops, i, VAL\ n : s\!>$$
$$\wedge$$
$$e' = throw \Rightarrow <\!comp\ e\ ops, i, s\!> \rightarrow^* \ll i, s \gg$$

We now verify that $P\ (e, i, e')$ holds for all $e \Downarrow^i e'$ by rule induction.

*Proof.* By rule induction

Case: $\text{VAL}_b$

$$
\begin{array}{ll}
 & <\!comp\ (\overline{n})\ ops, i, s\!> \\
= & \{\ \text{definition of } comp\ \} \\
 & <\!PUSH\ n : ops, i, s\!> \\
\rightarrow & \{\ \text{definition of} \rightarrow \} \\
 & <\!ops, i, VAL\ n : s\!>
\end{array}
$$

Case: $\text{THROW}_b$

$$
\begin{array}{ll}
 & <\!comp\ throw\ ops, i, s\!> \\
= & \{\ \text{definition of } comp\ \} \\
 & <\!THROW : ops, i, s\!> \\
\rightarrow & \{\ \text{definition of} \rightarrow \} \\
 & \ll i, s \gg
\end{array}
$$

Case: $\text{ADD}_b1$

$$
\begin{array}{ll}
 & <\!comp\ (x + y)\ ops, i, s\!> \\
= & \{\ \text{definition of } comp\ \} \\
 & <\!comp\ x\ (comp\ y\ (ADD : ops)), i, s\!> \\
\rightarrow^* & \{\ \text{induction hypothesis for } x\ \} \\
 & <\!comp\ y\ (ADD : ops), i, VAL\ n : s\!> \\
\rightarrow^* & \{\ \text{induction hypothesis for } y\ \} \\
 & <\!ADD : ops, i, VAL\ m : VAL\ n : s\!> \\
\rightarrow & \{\ \text{definition of} \rightarrow \} \\
 & <\!ops, i, VAL\ (n + m) : s\!>
\end{array}
$$

Case: $\text{ADD}_b 2$

$$
\begin{aligned}
& <\!comp\ (x+y)\ ops, i, s\!> \\
= \quad & \{ \text{ definition of } comp \ \} \\
& <\!comp\ x\ (comp\ y\ (ADD : ops)), i, s\!> \\
\to^* \quad & \{ \text{ induction hypothesis for } x \ \} \\
& \ll i, s \gg
\end{aligned}
$$

Case: $\text{ADD}_b 3$

$$
\begin{aligned}
& <\!comp\ (x+y)\ ops, i, s\!> \\
= \quad & \{ \text{ definition of } comp \ \} \\
& <\!comp\ x\ (comp\ y\ (ADD : ops)), i, s\!> \\
\to^* \quad & \{ \text{ induction hypothesis for } x \ \} \\
& <\!comp\ y\ (ADD : ops), i, VAL\ n : s\!> \\
\to^* \quad & \{ \text{ induction hypothesis for } y \ \} \\
& \ll i, VAL\ n : s \gg \\
\to \quad & \{ \text{ definition of } \to \ \} \\
& \ll i, s \gg
\end{aligned}
$$

Case: $\text{CATCH}_b 1$

$$
\begin{aligned}
& <\!comp\ (catch\ x\ y)\ ops, i, s\!> \\
= \quad & \{ \text{ definition of } comp \ \} \\
& <\!MARK\ (comp\ y\ ops) : comp\ x\ (UNMARK : ops), i, s\!> \\
\to \quad & \{ \text{ definition of } \to \ \} \\
& <\!comp\ x\ (UNMARK : ops), i, HAN\ (comp\ y\ ops) : s\!> \\
\to^* \quad & \{ \text{ induction hypothesis for } x \ \} \\
& <\!UNMARK : ops, i, VAL\ n : HAN\ (comp\ y\ ops) : s\!> \\
\to \quad & \{ \text{ definition of } \to \ \} \\
& <\!ops, i, VAL\ n : s\!>
\end{aligned}
$$

Case: $\text{C}\textsc{atch}_b 2$

From lemma 7.3.3 we have two subcases; $v$ is either a number or *throw*:

- $$<\!comp\ (catch\ x\ y)\ ops, i, s\!>$$
$$= \quad \{ \text{ definition of } comp \ \}$$
$$<\!MARK\ (comp\ y\ ops) : comp\ x\ (UNMARK : ops), i, s\!>$$
$$\rightarrow \quad \{ \text{ definition of } \rightarrow \ \}$$
$$<\!comp\ x\ (UNMARK : ops), i, HAN\ (comp\ y\ ops) : s\!>$$
$$\rightarrow^* \quad \{ \text{ induction hypothesis for } x \ \}$$
$$<\!i, HAN\ (comp\ y\ ops) : s \gg$$
$$\rightarrow \quad \{ \text{ definition of } \rightarrow \ \}$$
$$<\!comp\ y\ ops, i, s\!>$$
$$\rightarrow^* \quad \{ \text{ induction hypothesis for } y \ \}$$
$$<\!ops, i, VAL\ n : s\!>$$

- $$<\!comp\ (catch\ x\ y)\ ops, i, s\!>$$
$$= \quad \{ \text{ definition of } comp \ \}$$
$$<\!MARK\ (comp\ y\ ops) : comp\ x\ (UNMARK : ops), i, s\!>$$
$$\rightarrow \quad \{ \text{ definition of } \rightarrow \ \}$$
$$<\!comp\ x\ (UNMARK : ops), i, HAN\ (comp\ y\ ops) : s\!>$$
$$\rightarrow^* \quad \{ \text{ induction hypothesis for } x \ \}$$
$$\ll i, HAN\ (comp\ y\ ops) : s \gg$$
$$\rightarrow \quad \{ \text{ definition of } \rightarrow \ \}$$
$$<\!comp\ y\ ops, i, s\!>$$
$$\rightarrow^* \quad \{ \text{ induction hypothesis for } y \ \}$$
$$\ll i, s \gg$$

Case: $\textsc{Seq}_b 2$

$$
\begin{aligned}
& \langle comp\ (x; y)\ ops, i, s \rangle \\
= \quad & \{ \text{ definition of } comp \ \} \\
& \langle comp\ x\ (POP : comp\ y\ ops), i, s \rangle \\
\to^* \quad & \{ \text{ induction hypothesis for } x \ \} \\
& \ll i, s \gg
\end{aligned}
$$

Case: $\textsc{Seq}_b 1$

From lemma 7.3.3 we have two subcases; either $v$ is a number or *throw*:

- $$
\begin{aligned}
& \langle comp\ (x; y)\ ops, i, s \rangle \\
= \quad & \{ \text{ definition of } comp \ \} \\
& \langle comp\ x\ (POP : comp\ y\ ops), i, s \rangle \\
\to^* \quad & \{ \text{ induction hypothesis for } x \ \} \\
& \langle POP : comp\ y\ ops, i, VAL\ m : s \rangle \\
\to \quad & \{ \text{ definition of } \to \ \} \\
& \langle comp\ y\ ops, i, s \rangle \\
\to^* \quad & \{ \text{ induction hypothesis for } y \ \} \\
& \langle ops, i, VAL\ n : s \rangle
\end{aligned}
$$

- $$
\begin{aligned}
& \langle comp\ (x; y)\ ops, i, s \rangle \\
= \quad & \{ \text{ definition of } comp \ \} \\
& \langle comp\ x\ (POP : comp\ y\ ops), i, s \rangle \\
\to^* \quad & \{ \text{ induction hypothesis for } x \ \} \\
& \langle POP : comp\ y\ ops, i, VAL\ n : s \rangle \\
\to \quad & \{ \text{ definition of } \to \ \} \\
& \langle comp\ y\ ops, i, s \rangle \\
\to^* \quad & \{ \text{ induction hypothesis for } y \ \} \\
& \ll i, s \gg
\end{aligned}
$$

Case: $\textsc{Block}$

From lemma 7.3.3 we have two subcases; $v$ is either a number or *throw*:

- $$<comp\ (block\ x)\ ops, i, s>$$

$=\ \{$ definition of *comp* $\}$

$$<SET\ MASK : comp\ x\ (RESET : ops), i, s>$$

$\rightarrow\ \{$ definition of $\rightarrow\ \}$

$$<comp\ x\ (RESET : ops), MASK, INT\ i : s>$$

$\rightarrow^*\ \{$ induction hypothesis for $x\ \}$

$$<RESET : ops, MASK : VAL\ n : INT\ i : s>$$

$\rightarrow\ \{$ definition of $\rightarrow\ \}$

$$<ops, i, VAL\ n : s>$$

- $$<comp\ (block\ x)\ ops, i, s>$$

$=\ \{$ definition of *comp* $\}$

$$<SET\ MASK : comp\ x\ (RESET : ops), i, s>$$

$\rightarrow\ \{$ definition of $\rightarrow\ \}$

$$<comp\ x\ (RESET : ops), MASK, INT\ i : s>$$

$\rightarrow^*\ \{$ induction hypothesis for $x\ \}$

$$\ll MASK, INT\ i : s \gg$$

$\rightarrow\ \{$ definition of $\rightarrow\ \}$

$$\ll i, s \gg$$

Case: UNBLOCK

From lemma 7.3.3 we have two subcases; $v$ is either a number or *throw*:

- $$<comp\ (unblock\ x)\ ops, i, s>$$

$=\ \{$ definition of *comp* $\}$

$$<SET\ UNMASK : comp\ x\ (RESET : ops), i, s>$$

$\rightarrow\ \{$ definition of $\rightarrow\ \}$

$$<comp\ x\ (RESET : ops), UNMASK, INT\ i : s>$$

$\rightarrow^*\ \{$ induction hypothesis for $x\ \}$

$$<RESET : ops, UNMASK : VAL\ n : INT\ i : s>$$

$\rightarrow\ \{$ definition of $\rightarrow\ \}$

$$<ops, i, VAL\ n : s>$$

- $$<comp\ (unblock\ x)\ ops, i, s>$$
  $=$ { definition of $comp$ }
  $$<SET\ UNMASK : comp\ x\ (RESET : ops), i, s>$$
  $\rightarrow$ { definition of $\rightarrow$ }
  $$<comp\ x\ (RESET : ops), UNMASK, INT\ i : s>$$
  $\rightarrow^*$ { induction hypothesis for $x$ }
  $$\ll UNMASK, INT\ i : s \gg$$
  $\rightarrow$ { definition of $\rightarrow$ }
  $$\ll i, s \gg$$

Case: INTERRUPT

$$<comp\ x\ ops, UNMASK, s>$$
$\rightarrow$ { definition of $\rightarrow$, lemma 7.3.4 }
$$\ll UNMASK, s \gg$$

□

The above proof requires a number of lemmas, which we state and prove below.

**Lemma 7.3.3 (evaluation values).**

$$x \Downarrow^i v \Rightarrow v \in \mathbb{Z} \cup \{throw\}$$

*Proof.* By rule induction. □

**Lemma 7.3.4 (non-empty code).**

$$isCons\ (comp\ e\ ops)$$

**where**
$isCons\ [\,] = False$
$isCons\ (x : xs) = True$

*Proof.* by induction on $e :: Expr$ □

**Lemma 7.3.5 (can reach reflexivity).**

$$x \to^* x$$

*Proof.* by definition of $\to^*$ □

**Lemma 7.3.6 (can reach transitivity).**

$$x \to^* y \land y \to^* z \Rightarrow x \to^* z$$

*Proof.*

$$\begin{array}{ll}
& x \to^* y \land y \to^* z \Rightarrow x \to^* z \\
= & \{ \text{ currying } \} \\
& x \to^* y \Rightarrow (y \to^* z \Rightarrow x \to^* z) \\
= & \{ \text{ logic } \} \\
& x \to^* y \Rightarrow \forall z.\, (y \to^* z \Rightarrow x \to^* z) \\
= & \{ \text{ define } P\ (x, y) = \forall z.\, (y \to^* z \Rightarrow x \to^* z) \ \} \\
& x \to^* y \Rightarrow P\ (x, y) \\
\Leftarrow & \{ \text{ rule induction } \} \\
& (1)\ P\ (x, x) \\
& (2)\ x \to x' \land x' \to^* y \land P\ (x', y) \Rightarrow P\ (x, y)
\end{array}$$

Condition (1) expands to $x \to^* z \Rightarrow x \to^* z$, which is trivially true.

By expanding $P$, condition (2) is equivalent to

$$\frac{x \to x' \qquad x' \to^* y \qquad \forall z.\, (y \to^* z \Rightarrow x' \to^* z)}{\forall z'.\, (y \to^* z' \Rightarrow x \to^* z')}$$

We now verify this rule as follows. Assume $y \to^* z'$, then by the third assumption, we have $x' \to^* z'$. Then by the first assumption and the definition of $\to^*$, we have $x \to^* z'$ as required.

□

**Lemma 7.3.7 (can reach union property).**

$$x \rightarrow^* Y \cup Z \Leftrightarrow x \rightarrow^* Y \wedge x \rightarrow^* Z$$

*Proof.*

$$
\begin{array}{cl}
 & x \rightarrow^* Y \wedge x \rightarrow^* Z \Leftrightarrow x \rightarrow^* Y \cup Z \\
= & \{ \text{definition of } \rightarrow^* \} \\
 & (\forall y \in Y.\, x \rightarrow^* y) \wedge (\forall z \in Z.\, x \rightarrow^* z) \Leftrightarrow (\forall w \in Y \cup Z.\, x \rightarrow^* w) \\
\Leftarrow & \{ \text{generalise on } x \rightarrow^* \} \\
 & \forall y \in Y.\, P\ (y) \wedge \forall z \in Z.\, P\ (z) \Leftrightarrow \forall w \in Y \cup Z.\, P\ (w) \\
= & \{ \text{view predicate as set} \} \\
 & Y \subseteq P \wedge Z \subseteq P = (Y \cup Z) \subseteq P \\
= & \{ \text{set theory} \} \\
 & \mathit{true}
\end{array}
$$

□

### 7.3.5 Proving the *will reach* relation

We conclude by proving the second of the two relations which combine to make the $\trianglelefteq$ relation, the $\triangleleft$ relation:

$$
\begin{array}{c}
<\!\mathit{comp}\ e\ \mathit{ops}, i, s\!> \\
\triangleleft \\
\{<\!\mathit{ops}, i, n : s\!> \mid e \Downarrow^i \overline{n}\} \cup \{\ll i, s \gg \mid e \Downarrow^i \mathit{throw}\}
\end{array}
$$

That is, an expression $e$ compiled with arbitrary continuing code *ops* and run in an interrupt status $i$ with an arbitrary initial stack $s$ will reach an element of the set comprising those machine states corresponding to the possible evaluations in the big-step semantics with the same interrupt status $i$. We prove this statement directly by induction on $e$:

*Proof.* By induction on $e$

Case: $\overline{n}$

$$
\begin{array}{ll}
 & <\!comp\ (\overline{n})\ ops, i, s\!> \\
= & \{ \text{ definition of comp } \} \\
 & <\!PUSH\ n : ops, i, s\!> \\
\lhd & \{ \text{ definition of } \rightarrow \} \\
 & \{<\!ops, i, VAL\ n : s\!>\} \cup \{\ll i, s \gg \mid i = UNMASK\} \\
= & \{ \text{ definition of } \Downarrow \} \\
 & \{<\!ops, i, VAL\ n : s\!> \mid \overline{n} \Downarrow^i \overline{n}\} \cup \\
 & \{\ll i, s \gg \mid \overline{n} \Downarrow^i throw\}
\end{array}
$$

Note that in the second step, there are two possible outcomes permitted by the machine: either the push is performed, or an exception is raised if interrupts are permitted. Moreover, in the last step, the predicate $\overline{n} \Downarrow^i \overline{n}$ is true by the rule for integers, and $\overline{n} \Downarrow^i throw$ is equivalent to $i = UNMASK$ by the rule for interrupts.

Case: *throw*

$$
\begin{array}{ll}
 & <\!comp\ throw\ ops, i, s\!> \\
= & \{ \text{ definition of } comp \} \\
 & <\!THROW : ops, i, s\!> \\
\lhd & \{ \text{ definition of } \rightarrow \} \\
 & \{\ll i, s \gg\} \\
= & \{ \text{ definition of } \Downarrow \} \\
 & \{\ll i, s \gg \mid throw \Downarrow^i throw\} \\
\subseteq & \{ \text{ property of } \cup :\ X \subseteq X \cup Y \} \\
 & \{<\!ops, i, VAL\ n : s\!> \mid throw \Downarrow^i \overline{n}\} \cup \\
 & \{\ll i, s \gg \mid throw \Downarrow^i throw\}
\end{array}
$$

The result now follows because $X \subseteq Y \Rightarrow X \lhd Y$ (lemma 7.3.11), and hence the $\subseteq$ in the last step can safely be replaced by $\lhd$.

The following case is somewhat more complex, we shall therefore explain the process in more detail after we give the proof.

Case: $x + y$

$$
\begin{array}{ll}
 & <\!comp\ (x+y), i, s\!> \\
= & \{\ \text{definition of } comp\ \} \\
 & <\!comp\ x\ (comp\ y\ (ADD : ops)), i, s\!> \\
\lhd & \{\ \text{induction hypothesis for } x\ \} \\
 & \{<\!comp\ y\ (ADD : ops), i, VAL\ n : s\!> \mid x \Downarrow^i \overline{n}\} \cup \\
 & \{\ll i, s \gg \mid x \Downarrow^i throw\} \\
\lhd & \{\ (1)\ \text{induction hypothesis for } y,\ \cup \text{ preserves } \lhd \text{ (lemma 7.3.12)}\ \} \\
 & \{<\!ADD : ops, i, VAL\ m : VAL\ n : s\!> \mid x \Downarrow^i \overline{n} \wedge y \Downarrow^i \overline{m}\} \cup \\
 & \{\ll i, VAL\ n : s \gg \mid x \Downarrow^i \overline{n} \wedge y \Downarrow^i throw\} \cup \\
 & \{\ll i, s \gg \mid x \Downarrow^i throw\} \\
\lhd & \{\ (2)\ \text{definition of } \rightarrow,\ \cup \text{ preserves } \lhd \text{ (lemma 7.3.12)}\ \} \\
 & \{<\!ops, i, VAL\ (n+m) : s\!> \mid x \Downarrow^i \overline{n} \wedge y \Downarrow^i \overline{m}\} \cup \\
 & \{\ll i, s \gg \mid x \Downarrow^i \overline{n} \wedge y \Downarrow^i \overline{m} \wedge i = UNMASK\} \cup \\
 & \{\ll i, s \gg \mid x \Downarrow^i \overline{n} \wedge y \Downarrow^i throw\} \cup \\
 & \{\ll i, s \gg \mid x \Downarrow^i throw\} \\
\subseteq & \{\ (3)\ \text{definition of } \Downarrow,\ \cup \text{ preserves } \subseteq\ \} \\
 & \{<\!ops, i, VAL\ (n+m) : s\!> \mid x + y \Downarrow^i \overline{n+m}\} \cup \\
 & \{\ll i, s \gg \mid x + y \Downarrow^i throw\} \cup \\
 & \{\ll i, s \gg \mid x + y \Downarrow^i throw\} \cup \\
 & \{\ll i, s \gg \mid x + y \Downarrow^i throw\} \\
= & \{\ (4)\ \text{idempotence of } \cup\ \} \\
 & \{<\!ops, i, VAL\ (n+m) : s\!> \mid x + y \Downarrow^i \overline{n+m}\} \cup \\
 & \{\ll i, s \gg \mid x + y \Downarrow^i throw\}
\end{array}
$$

The result now follows because $X \subseteq Y \ \Rightarrow\ X \lhd Y$ (lemma 7.3.11).

1. In this step, we apply the induction hypothesis, and the properties that $A \lhd B \wedge C \lhd D \Rightarrow A \cup C \lhd B \cup D$ (lemma 7.3.12) and $X \lhd X$ (lemma 7.3.8) in conjunction.

2. In this step, all of the step rules for $\rightarrow$ which may apply for each set of machine states, and the properties that $A \lhd B \wedge C \lhd D \Rightarrow A \cup C \lhd B \cup D$ (lemma 7.3.12) and $X \lhd X$ (lemma 7.3.8) in conjunction.

3. In this step:
   - We apply the definition of $\Downarrow$ to gain $x + y \Downarrow^i \overline{n+m}$ from $x \Downarrow^i \overline{n} \wedge y \Downarrow^i \overline{m}$ (rule ADD1)
   - We apply the definition of $\Downarrow$ to gain $x+y \Downarrow^i \mathit{throw}$ from $(x \Downarrow^i \overline{n}) \wedge (y \Downarrow^i \overline{m}) \wedge (i = \mathit{UNMASK})$ (rule ADD3)
   - We apply the properties that $A \lhd B \wedge C \lhd D \Rightarrow A \cup C \lhd B \cup D$ (lemma 7.3.12) and $X \lhd X$ (lemma 7.3.8) in conjunction.
4. In this step, we apply $A \cup A = A$ to achieve the result we require.

Case: $x; y$

$$
\begin{array}{ll}
 & <\mathit{comp}\ (x;y), i, s> \\
= & \{ \text{ definition of } \mathit{comp} \ \} \\
 & <\mathit{comp}\ x\ (\mathit{POP} : \mathit{comp}\ y\ \mathit{ops}), i, s> \\
\lhd & \{ \text{ induction hypothesis for } x \ \} \\
 & \{<\mathit{POP} : \mathit{comp}\ y\ \mathit{ops}, i, \mathit{VAL}\ n : s> \mid x \Downarrow^i \overline{n}\} \cup \\
 & \{\ll i, s \gg \mid x \Downarrow^i \mathit{throw}\} \\
\lhd & \{ \text{ definition of } \rightarrow, \cup \text{ preserves } \lhd \text{ (lemma 7.3.12) } \} \\
 & \{<\mathit{comp}\ y\ \mathit{ops}, i, s> \mid x \Downarrow^i \overline{n}\} \cup \\
 & \{\ll i, \mathit{VAL}\ n : s \gg \mid x \Downarrow^i \overline{n} \wedge i = \mathit{UNMASK}\} \cup \\
 & \{\ll i, s \gg \mid x \Downarrow^i \mathit{throw}\} \\
\lhd & \{ \text{ induction hypothesis for } y, \cup \text{ preserves } \lhd \text{ (lemma 7.3.12) } \} \\
 & \{<\mathit{ops}, i, \mathit{VAL}\ m : s> \mid x \Downarrow^i \overline{n} \wedge y \Downarrow^i \overline{m}\} \cup \\
 & \{\ll i, s \gg \mid x \Downarrow^i \overline{n} \wedge y \Downarrow^i \mathit{throw}\} \cup \\
 & \{\ll i, \mathit{VAL}\ n : s \gg \mid x \Downarrow^i \overline{n} \wedge i = \mathit{UNMASK}\} \cup \\
 & \{\ll i, s \gg \mid x \Downarrow^i \mathit{throw}\} \\
\lhd & \{ \text{ definition of } \rightarrow, \cup \text{ preserves } \lhd \text{ (lemma 7.3.12) } \} \\
 & \{<\mathit{ops}, i, \mathit{VAL}\ m : s> \mid x \Downarrow^i \overline{n} \wedge y \Downarrow^i \overline{m}\} \cup \\
 & \{\ll i, s \gg \mid x \Downarrow^i \overline{n} \wedge y \Downarrow^i \mathit{throw}\} \cup \\
 & \{\ll i, s \gg \mid x \Downarrow^i \overline{n} \wedge i = \mathit{UNMASK}\} \cup \\
 & \{\ll i, s \gg \mid x \Downarrow^i \mathit{throw}\} \\
\subseteq & \{ \text{ definition of } \Downarrow, \cup \text{ preserves } \subseteq \ \}
\end{array}
$$

$$
\begin{array}{ll}
& \{<ops, i, VAL\ m : s> \mid x; y \Downarrow^i \overline{m}\} \cup \\
& \{\ll i, s \gg \mid x; y \Downarrow^i \mathit{throw}\} \cup \\
& \{\ll i, s \gg \mid x; y \Downarrow^i \mathit{throw}\} \cup \\
& \{\ll i, s \gg \mid x; y \Downarrow^i \mathit{throw}\} \\
= & \{ \text{ idempotence of } \cup \ \} \\
& \{<ops, i, VAL\ m : s> \mid x; y \Downarrow^i \overline{m}\} \cup \\
& \{\ll i, s \gg \mid x; y \Downarrow^i \mathit{throw}\}
\end{array}
$$

The result now follows because $X \subseteq Y \Rightarrow X \lhd Y$ (lemma 7.3.11).

Case: *catch x y*

$$
\begin{array}{ll}
& <comp\ (catch\ x\ y), i, s> \\
= & \{ \text{ definition of } comp \ \} \\
& <MARK\ (comp\ y\ ops) : comp\ x\ (UNMARK : ops), i, s> \\
\lhd & \{ \text{ definition of } \rightarrow \} \\
& \{<comp\ x\ (UNMARK : ops), i, HAN\ (comp\ y\ ops) : s>\} \cup \\
& \{\ll i, s \gg \mid i = UNMASK\} \\
\lhd & \{ \text{ induction hypothesis for } x, \cup \text{ preserves } \lhd \text{ (lemma 7.3.12) } \} \\
& \{<UNMARK : ops, i, VAL\ n : HAN\ (comp\ y\ ops) : s \mid x \Downarrow^i \overline{n}\} \cup \\
& \{\ll i, HAN\ (comp\ y\ ops) : s \gg \mid x \Downarrow^i \mathit{throw}\} \cup \\
& \{\ll i, s \gg \mid i = UNMASK\} \\
\lhd & \{ \text{ definition of } \rightarrow, \cup \text{ preserves } \lhd \text{ (lemma 7.3.12) } \} \\
& \{<ops, i, VAL\ n : s> \mid x \Downarrow^i \overline{n}\} \cup \\
& \{<comp\ y\ ops, i, s> \mid x \Downarrow^i \overline{n} \wedge i = UNMASK\} \cup \\
& \{<comp\ y\ ops, i, s> \mid x \Downarrow^i \mathit{throw}\} \cup \\
& \{\ll i, s \gg \mid i = UNMASK\} \\
\lhd & \{ \text{ induction hypothesis for } y, \cup \text{ preserves } \lhd \text{ (lemma 7.3.12) } \} \\
& \{<ops, i, VAL\ n : s> \mid x \Downarrow^i \overline{n}\} \cup \\
& \{<ops, i, VAL\ m : s> \mid x \Downarrow^i \overline{n} \wedge i = UNMASK \wedge y \Downarrow^i \overline{m}\} \cup \\
& \{\ll i, s \gg \mid x \Downarrow^i \overline{n} \wedge i = UNMASK \wedge y \Downarrow^i \mathit{throw}\} \cup \\
& \{<ops, i, VAL\ m : s> \mid x \Downarrow^i \mathit{throw} \wedge y \Downarrow^i \overline{m}\} \cup \\
& \{\ll i, s \gg \mid x \Downarrow^i \mathit{throw} \wedge y \Downarrow^i \mathit{throw}\} \cup
\end{array}
$$

$$
\begin{array}{ll}
 & \{\ll i, s \gg \mid i = \mathit{UNMASK}\} \\
\subseteq & \{ \text{ definition of } \Downarrow, \ \cup \text{ preserves } \subseteq \ \} \\
 & \{<\mathit{ops}, i, \mathit{VAL}\ n : s> \mid \mathit{catch}\ x\ y \Downarrow^i \overline{n}\} \cup \\
 & \{<\mathit{ops}, i, \mathit{VAL}\ m : s> \mid \mathit{catch}\ x\ y \Downarrow^i \overline{m}\} \cup \\
 & \{\ll i, s \gg \mid \mathit{catch}\ x\ y \Downarrow^i \mathit{throw}\} \cup \\
 & \{<\mathit{ops}, i, \mathit{VAL}\ m : s> \mid \mathit{catch}\ x\ y \Downarrow^i \overline{m}\} \cup \\
 & \{\ll i, s \gg \mid \mathit{catch}\ x\ y \Downarrow^i \mathit{throw}\} \cup \\
 & \{\ll i, s \gg \mid \mathit{catch}\ x\ y \Downarrow^i \mathit{throw}\} \\
= & \{ \text{ idempotence of } \cup \ \} \\
 & \{<\mathit{ops}, i, \mathit{VAL}\ n : s> \mid \mathit{catch}\ x\ y \Downarrow^i \overline{n}\} \cup \\
 & \{\ll i, s \gg \mid \mathit{catch}\ x\ y \Downarrow^i \mathit{throw}\}
\end{array}
$$

The result now follows because $X \subseteq Y \;\Rightarrow\; X \lhd Y$ (lemma 7.3.11). Note in the second last step that $x \Downarrow^i \overline{n} \wedge i = \mathit{UNMASK} \wedge y \Downarrow^i \overline{m} \;\Rightarrow\; \mathit{catch}\ x\ y \Downarrow^i \overline{m}$ because the right hand side is always true by using interrupt rule (and $i = \mathit{UNMASK}$) to conclude that $x \Downarrow^i \mathit{throw}$ and then applying the second catch rule (and $y \Downarrow^i \overline{m}$) to conclude that $\mathit{catch}\ x\ y \Downarrow^i \overline{m}$, i.e. we don't need the fact that $x \Downarrow^i \overline{n}$.

Case: $\mathit{block}\ x$

$$
\begin{array}{ll}
 & <\mathit{comp}\ (\mathit{block}\ x), i, s> \\
= & \{ \text{ definition of } \mathit{comp} \ \} \\
 & <\mathit{SET}\ \mathit{MASK} : \mathit{comp}\ x\ (\mathit{RESET} : \mathit{ops}), i, s> \\
\lhd & \{ \text{ definition of } \rightarrow \ \} \\
 & \{<\mathit{comp}\ x\ (\mathit{RESET} : \mathit{ops}), \mathit{MASK}, \mathit{INT}\ i : s>\} \cup \\
 & \{\ll i, s \gg \mid i = \mathit{UNMASK}\} \\
\lhd & \{ \text{ induction hypothesis for } x, \ \cup \text{ preserves } \lhd \text{ (lemma 7.3.12) } \} \\
 & \{<\mathit{RESET} : \mathit{ops}, \mathit{MASK}, \mathit{VAL}\ n : \mathit{INT}\ i : s> \mid x \Downarrow^b \overline{n}\} \cup \\
 & \{\ll \mathit{MASK}, \mathit{INT}\ i : s \gg \mid x \Downarrow^b \mathit{throw}\} \cup \\
 & \{\ll i, s \gg \mid i = \mathit{UNMASK}\} \\
\lhd & \{ \text{ definition of } \rightarrow, \ \cup \text{ preserves } \lhd \text{ (lemma 7.3.12) } \} \\
 & \{<\mathit{ops}, i, \mathit{VAL}\ n : s> \mid x \Downarrow^b \overline{n}\} \cup \\
 & \{\ll i, s \gg \mid x \Downarrow^b \mathit{throw}\} \cup
\end{array}
$$

$$
\begin{array}{ll}
& \{\ll i, s \gg \mid i = \mathit{UNMASK}\} \\
\subseteq & \{ \text{ definition of } \Downarrow,\ \cup \text{ preserves } \subseteq \} \\
& \{<\mathit{ops}, i, \mathit{VAL}\ n : s> \mid \mathit{block}\ x \Downarrow^i \overline{n}\} \cup \\
& \{\ll i, s \gg \mid \mathit{block}\ x \Downarrow^i \mathit{throw}\} \cup \\
& \{\ll i, s \gg \mid \mathit{block}\ x \Downarrow^i \mathit{throw}\} \\
= & \{ \text{ idempotence of } \cup \} \\
& \{<\mathit{ops}, i, \mathit{VAL}\ n : s> \mid \mathit{block}\ x \Downarrow^i \overline{n}\} \cup \\
& \{\ll i, s \gg \mid \mathit{block}\ x \Downarrow^i \mathit{throw}\}
\end{array}
$$

The result now follows because $X \subseteq Y \;\Rightarrow\; X \lhd Y$ (lemma 7.3.11).

Case: *unblock y*

$$
\begin{array}{ll}
& <\mathit{comp}\ (\mathit{unblock}\ x), i, s> \\
= & \{ \text{ definition of } \mathit{comp} \} \\
& <\mathit{SET\ UNMASK} : \mathit{comp}\ x\ (\mathit{RESET} : \mathit{ops}), i, s> \\
\lhd & \{ \text{ definition of } \rightarrow \} \\
& \{<\mathit{comp}\ x\ (\mathit{RESET} : \mathit{ops}), \mathit{UNMASK}, \mathit{INT}\ i : s>\} \cup \\
& \{\ll i, s \gg \mid i = \mathit{UNMASK}\} \\
\lhd & \{ \text{ induction hypothesis for } x,\ \cup \text{ preserves } \lhd \text{ (lemma 7.3.12) } \} \\
& \{<\mathit{RESET} : \mathit{ops}, \mathit{UNMASK}, \mathit{VAL}\ n : \mathit{INT}\ i : s> \mid x \Downarrow^u \overline{n}\} \cup \\
& \{\ll \mathit{UNMASK}, \mathit{INT}\ i : s \gg \mid x \Downarrow^u \mathit{throw}\} \cup \\
& \{\ll i, s \gg \mid i = \mathit{UNMASK}\} \\
\lhd & \{ \text{ definition of } \rightarrow,\ \cup \text{ preserves } \lhd \text{ (lemma 7.3.12) } \} \\
& \{<\mathit{ops}, i, \mathit{VAL}\ n : s> \mid x \Downarrow^u \overline{n}\} \cup \\
& \{\ll i, s \gg \mid x \Downarrow^u \overline{n}\} \cup \\
& \{\ll i, s \gg \mid x \Downarrow^u \mathit{throw}\} \cup \\
& \{\ll i, s \gg \mid i = \mathit{UNMASK}\} \\
\subseteq & \{ \text{ definition of } \Downarrow,\ \cup \text{ preserves } \subseteq \} \\
& \{<\mathit{ops}, i, \mathit{VAL}\ n : s> \mid \mathit{unblock}\ x \Downarrow^i \overline{n}\} \cup \\
& \{\ll i, s \gg \mid \mathit{unblock}\ x \Downarrow^i \mathit{throw}\} \cup \\
& \{\ll i, s \gg \mid \mathit{unblock}\ x \Downarrow^i \mathit{throw}\} \cup \\
& \{\ll i, s \gg \mid \mathit{unblock}\ x \Downarrow^i \mathit{throw}\}
\end{array}
$$

$= \quad \{ \text{ idempotence of } \cup \ \}$
$\quad \{<ops, i, VAL\ n : s> \mid unblock\ x \Downarrow^i \overline{n}\} \cup$
$\quad \{\ll i, s \gg \mid unblock\ x \Downarrow^i throw\}$

The result now follows because $X \subseteq Y \ \Rightarrow \ X \lhd Y$ (lemma 7.3.11). Note in the second last step that $x \Downarrow^u \overline{n} \ \Rightarrow \ unblock\ x \Downarrow^i throw$, because the right hand side is always true by using the rule for *unblock* and then the interrupt rule, i.e. we don't actually need the $x \Downarrow^u \overline{n}$.

□

The above proof makes use of a number of lemmas, which we state and prove below.

**Lemma 7.3.8 (Reflexivity of $\lhd$).**

$$X \lhd X$$

*Proof.*

$\quad X \lhd X$
$= \quad \{ \text{ definition of } \lhd \ \}$
$\quad \forall x \in X.\ x \lhd X$
$= \quad \{ \text{ definition of } \lhd \ \}$
$\quad \forall x \in X.\ x \in X \vee \forall x'.\ x \rightarrow x' \Rightarrow x' \lhd X$
$\Leftarrow \quad \{ \vee \text{ Introduction } \}$
$\quad \forall x \in X.\ x \in X$
$= \quad \{ \text{ definition of } \subseteq \}$
$\quad True$

□

**Lemma 7.3.9 (Transitivity of $\lhd$).**

$$X \lhd Y \wedge Y \lhd Z \Rightarrow X \lhd Z$$

*Proof.*

$$
\begin{array}{ll}
 & X \lhd Y \wedge Y \lhd Z \\
= & \{ \text{ definition of } \lhd \ \} \\
 & (\forall x \in X.\, x \lhd Y) \wedge Y \lhd Z \\
= & \{ \text{ logic } \} \\
 & \forall x \in X.\, (x \lhd Y \wedge Y \lhd Z) \\
\Rightarrow & \{ \text{ lemma 7.3.10 } \} \\
 & \forall x \in X.\, x \lhd Z \\
= & \{ \text{ definition of } \lhd \ \} \\
 & X \lhd Z
\end{array}
$$

□

**Lemma 7.3.10.**

$$x \lhd Y \wedge Y \lhd Z \Rightarrow x \lhd Z$$

*Proof.*

$$
\begin{array}{ll}
 & x \lhd Y \wedge Y \lhd Z \Rightarrow x \lhd Z \\
= & \{ \text{ curry } \} \\
 & x \lhd Y \Rightarrow (Y \lhd Z \Rightarrow x \lhd Z) \\
= & \{ \text{ logic } \} \\
 & x \lhd Y \Rightarrow (\forall Z.\ Y \lhd Z \Rightarrow x \lhd Z) \\
= & \{ \text{ define } P\ (x, Y) = \forall Z.\ Y \lhd Z \Rightarrow x \lhd Z \ \} \\
 & x \lhd Y \Rightarrow P\ (x, Y) \\
\Leftarrow & \{ \text{ rule induction } \}
\end{array}
$$

We have the following rule inductive principle:

$$\frac{\begin{array}{c}\forall x, Y.\ (x \in Y \Rightarrow P\ (x, Y)) \\ \forall x, Y.\ ((\forall x'.\ x \to x' \Rightarrow x' \lhd Y) \wedge (\forall x'.\ x \to x' \Rightarrow P\ (x', Y)) \Rightarrow P\ (x, Y))\end{array}}{\forall x, Y.\ x \lhd Y \Rightarrow P\ (x, Y)}$$

Base Case : $\forall x, Y.\ (x \in Y \Rightarrow P\ (x, Y))$

Given

$x, Y$

$x \in Y$ (A)

Show

$P\ (x, Y)$, which is equivalent to: $\forall Z.\ Y \lhd Z \Rightarrow x \lhd Z$

Given

$Z$

$Y \lhd Z$ $(= \forall y \in Y.\ y \lhd Z)$ (B)

Show

$x \lhd Z$

This follows directly from (A) and (B) because we know $x \in Y$ and $Y \lhd Z$.

Step Case :

Given

$x$, $Y$

$\forall x'.\ x \to x' \Rightarrow x' \lhd Y$

$\forall x'.\ x \to x' \Rightarrow P\ (x', Y)\ (= \forall Z.\ Y \lhd Z \Rightarrow x' \lhd Z)$ (IH)

Show

$P\ (x, Y)$, which is equivalent to: $\forall Z.\ Y \lhd Z \Rightarrow x \lhd Z$

Given

$Z$

$Y \lhd Z$ (A)

Show

$x \lhd Z$

which follows from the step rule if

$\forall x'.\ x \to x' \lhd Z$

Given

$x'$

$x \to x'$ (B)

Show

$x' \lhd Z$

which follows from (A) (B) and (IH)

This follows because we know from assumptions that $x \to x'$, and therefore that $\forall Z.\ Y \lhd Z \Rightarrow x' \lhd Z$. We also have the assumption that $Y \lhd Z$, which leads directly to $x' \lhd Z$.

□

**Lemma 7.3.11 (Larger than inclusion).**

$$X \subseteq Y \Rightarrow X \lhd Y$$

*Proof.*

$X \lhd Y$

$=$ { definition of $\lhd$ }

$$
\begin{array}{ll}
& \forall x \in X.\, x \lhd Y \\
= & \{ \text{ definition of } \lhd \text{ } \} \\
& \forall x \in X.\, (x \in Y \vee \forall x'.\, x \rightarrow x' \Rightarrow x' \lhd Y) \\
\Leftarrow & \{ \vee \text{ Introduction } \} \\
& \forall x \in X.\, x \in Y \\
= & \{ \text{ definition of } \subseteq \text{ } \} \\
& X \subseteq Y
\end{array}
$$

□

**Lemma 7.3.12 ($\cup$ preserves $\lhd$).**

$$A \lhd B \wedge C \lhd D \;\Rightarrow\; (A \cup C) \lhd (B \cup D)$$

*Proof.*

$$
\begin{array}{ll}
& (A \cup C) \lhd (B \cup D) \\
= & \{ \text{ definition of } \lhd \text{ } \} \\
& \forall x \in A \cup C.\, x \lhd (B \cup D) \\
= & \{ \text{ set theory, logic } \} \\
& \forall a \in A.\, a \lhd (B \cup D) \;\wedge\; \forall c \in C.\, c \lhd (B \cup D) \\
= & \{ \text{ definition of } \lhd \text{ } \} \\
& A \lhd (B \cup D) \;\wedge\; C \lhd (B \cup D) \\
\Leftarrow & \{ \text{ transitivity } \} \\
& A \lhd B \wedge B \lhd (B \cup D) \;\wedge\; C \lhd D \wedge D \lhd (B \cup D) \\
\Leftarrow & \{ \text{ lemma 7.3.11 } \} \\
& A \lhd B \wedge B \subseteq (B \cup D) \;\wedge\; C \lhd D \wedge D \subseteq (B \cup D) \\
= & \{ \text{ set theory } \} \\
& A \lhd B \wedge C \lhd D
\end{array}
$$

□

The conclusion of these lemmas concludes our proof of equivalence between the semantics and the compiler.

## 7.4 Summary

In this chapter we introduced interrupts to our language by adding a worst-case interrupt generator rule to both our semantics and virtual machine, and by adding new language features to allow effective programming in their presence. Adding these features had the effect of making evaluations and computations non-deterministic, which not only forced us to state our virtual machine in terms of a step relation rather than a function, but also had a considerable effect on reasoning about the correctness of the compiler. We now state correctness in terms of possible sets of *final states* which an execution must pass through, and two relations, which when combined, capture this idea. This had the effect of making our proofs both more complex and somewhat more difficult, but how they affect reasoning about our combinator, *finally*, we shall find out in the next chapter.

CHAPTER 8

# Interrupts and *finally*

In this chapter we wish to develop, using our new language features, a version of the *finally* operator, which we introduced in Chapter 7, that is safe in the presence of interrupts. Again, we wish to define this operator in terms of the low level primitives of our interrupting language and prove its correctness using our big-step semantics.

## 8.1 A New Specification of *finally*

As we saw in the previous chapter, the definition of *finally* that we introduced in Chapter 6 is no longer correct in the presence of interrupts. Taking that definition as a starting point, and using our new language primitives, we wish to develop a new version that is safe in the presence of interrupts. The first stage in this development is to extend our specification of *finally* to take into account not only exceptions, but also interrupts. The vague definition of "do $x$, then whatever happens do $y$" now needs even more clarification, as the combinator may be interrupted at any time, or even before it begins evaluate. We update our specification as follows:

- If $x$ raises an exception, $y$ is evaluated and the exception is propagated.
- If $x$ is interrupted, $y$ is evaluated and the exception is propagated.
- If $x$ is not interrupted and does not raise an exception, $y$ is evaluated and the program continues normally.

- If $x$ evaluates, $y$ should be evaluated, and should be evaluated exactly once.

These extensions to the specification give us the precision we need to test possible definitions for correctness.

## 8.2 Another *finally* Definition

We now attempt to redefine our *finally* combinator using the new *block* and *unblock* language features we added in the last chapter:

- A naive approach is simply to evaluate our original definition of finally whilst interrupts are blocked. While this solution meets all of our definition requirements it rather defeats the object of adding interrupts to our language in the first place.

  $$\textit{finally}\ x\ y = \textit{block}\ (\textit{catch}\ x\ (y; \textit{throw}); y)$$

- Here we modify our naive approach by unblocking the evaluation of $x$. This has the effect that the *work* part of the combinator is evaluated whilst interrupts are permitted and the structure and *clean up* work of the combinator is left unaffected by interrupts.

  $$\textit{finally}\ x\ y = \textit{block}\ (\textit{catch}\ (\textit{unblock}\ x)\ (y; \textit{throw}); y)$$

- Here we also allow the *clean up* expression, $y$, to be interrupted. This also fulfils the specification, however this also seems to go against the intuition of the combinator, because we would expect that *clean up* code should be fully evaluated.

  $$\textit{finally}\ x\ y = \textit{block}\ (\textit{catch}\ (\textit{unblock}\ x)\ ((\textit{unblock}\ y); \textit{throw}); (\textit{unblock}\ y))$$

Here we discover a new problem introduced along with interrupts — we are able to give a number of possible definitions for *finally*, all of which appear to fulfill our specifications. We must now introduce a level of judgement, and how we expect the combinator to be used in order to pick the *right* definition. We shall choose the second attempt as our preferred definition of *finally*, as it appears to fulfill our specifications, and also seems to match the intended use of the combinator more closely than the final definition.

## 8.3 Safety in the Presence of Interrupts

With our new definition of *finally*, only $x$ is now evaluated in an unblocked context, the framework of the operator being no longer interruptible once it has started evaluating. The correctness of this new finally operator can again be proved by inspecting all possible evaluation trees of the expression, showing that whenever $x$ is evaluated, $y$ is also evaluated exactly once:

- Here we apply the INTERRUPT rule, which can only be applied if the combinator was in an unblocked interrupt state before it started evaluation. The evaluation is interrupted before it enters a blocked state and shows neither $x$ nor $y$ evaluating, which is allowed by our correctness statement.

$$\frac{}{\mathit{block}\ (\mathit{catch}\ (\mathit{unblock}\ x)\ (y; \mathit{throw}); y) \Downarrow^{u} \mathit{throw}}\ \textsc{Interrupt}$$

- Here we apply the SEQ1 rule which can be applied when the first element of a sequence does not raise an exception and is not interrupted. The only way for the *catch* to evaluate without raising an exception is for the $x$ to evaluate successfully to a number. Only the $x$ need be considered because the rest of the *catch* cannot be interrupted.

$$\frac{\frac{\textsc{Catch1}\ \frac{\frac{\frac{\nabla}{x \Downarrow^{u} \overline{n}}}{\mathit{unblock}\ x \Downarrow^{b} \overline{n}}\ \textsc{Unblock}}{\mathit{catch}\ (\mathit{unblock}\ x)\ (y; \mathit{throw}) \Downarrow^{b} \overline{n}} \qquad \frac{\nabla}{y \Downarrow^{b} \overline{m}}}{\mathit{catch}\ (\mathit{unblock}\ x)\ (y; \mathit{throw}); y \Downarrow^{b} \overline{m}}\ \textsc{Seq1}}{\mathit{block}\ (\mathit{catch}\ (\mathit{unblock}\ x)\ (y; \mathit{throw}); y) \Downarrow^{i} \overline{m}}\ \textsc{Block}$$

- Here we apply the SEQ1 rule again, with $y$ raising an exception. Note that the evaluation of $y$ cannot be interrupted as it is evaluated in a blocked state.

$$\dfrac{\dfrac{\textsc{Catch1}\;\dfrac{\dfrac{\dfrac{\nabla}{x \Downarrow^{u} \overline{n}}}{unblock\ x \Downarrow^{b} \overline{n}}\,\textsc{Unblock}}{catch\ (unblock\ x)\ (y; throw) \Downarrow^{b} \overline{n}} \qquad \dfrac{\nabla}{y \Downarrow^{b} throw}}{catch\ (unblock\ x)\ (y; throw); y \Downarrow^{b} throw}\,\textsc{Seq1}}{block\ (catch\ (unblock\ x)\ (y; throw); y) \Downarrow^{i} throw}\,\textsc{Block}$$

- Here we apply the SEQ2 rule. The expression $x$ must raise an exception because the *catch* expression must raise an exception. Note that it is entirely possible for the evaluation of $x$ to consist of a single application of the INTERRUPT rule. The expression $y$ evaluates to a number, however the handler always raises an exception due to the second element of the sequence being *throw*.

$$\dfrac{\dfrac{\dfrac{\textsc{Unblock}\;\dfrac{\dfrac{\nabla}{x \Downarrow^{u} throw}}{unblock\ x \Downarrow^{b} throw} \qquad \dfrac{\dfrac{\nabla}{y \Downarrow^{b} \overline{n}} \quad \dfrac{}{throw \Downarrow^{b} throw}\,\textsc{Throw}}{y; throw \Downarrow^{b} throw}\,\textsc{Seq1}}{catch\ (unblock\ x)\ (y; throw) \Downarrow^{b} throw}\,\textsc{Catch2}}{catch\ (unblock\ x)\ (y; throw); y \Downarrow^{b} throw}\,\textsc{Seq2}}{block\ (catch\ (unblock\ x)\ (y; throw); y) \Downarrow^{i} throw}\,\textsc{Block}$$

- Here we again apply the SEQ2 rule, however in this evaluation the expression $y$ raises an exception. The result is again that the *catch* expression raises an exception.

$$\dfrac{\dfrac{\dfrac{\dfrac{\nabla}{x \Downarrow^u \mathit{throw}}}{\mathit{unblock}\ x \Downarrow^b \mathit{throw}}\,\textsc{Unblock} \quad \dfrac{\dfrac{\nabla}{y \Downarrow^b \mathit{throw}}}{y; \mathit{throw} \Downarrow^b \mathit{throw}}\,\textsc{Seq2}}{\dfrac{\mathit{catch}\ (\mathit{unblock}\ x)\ (y; \mathit{throw}) \Downarrow^b \mathit{throw}}{\mathit{catch}\ (\mathit{unblock}\ x)\ (y; \mathit{throw}); y \Downarrow^b \mathit{throw}}\,\textsc{Seq2}}\,\textsc{Catch2}}{\mathit{block}\ (\mathit{catch}\ (\mathit{unblock}\ x)\ (y; \mathit{throw}); y) \Downarrow^i \mathit{throw}}\,\textsc{Block}$$

These are all the possible evaluations of *finally* $x\ y$, because combined with lemma 7.3.2, which states that an expression always evaluates to a number or *throw*, no other evaluation trees can be drawn using the big-step semantics. We have therefore proven that our definition of *finally* behaves according to our specifications.

## 8.4 Summary

In this chapter we have developed a new specification and definition of *finally*, which is safe in the presence of interrupts, and proved its correctness using our big-step semantics. The interesting point to note is that even though the proof of correctness for the interrupting compiler is far more complex than that seen previously, and our language is now non-deterministic, the proof of correctness of our new definition is no more complex that we saw in Chapter 6. We now proceed to the final chapter of this thesis in which we give a final summary, and detail possible direction for future research.

Chapter 9

# Summary and Further Work

When the work began for this thesis the intention was to reason about the semantics of the exception and interrupt handling extensions to the Glasgow Haskell Compiler, as presented in *Asynchronous Exceptions in Haskell* [MPMR01]. A full semantics for both the exception and interrupt handling mechanisms of GHC is given, but without any kind of formal justification for the correctness of the semantics, or any examples of how they could be used to reason about programs. Reasoning about this semantics turned out to be more complex and problematic than was originally anticipated, so the emphasis of the work turned to understanding exceptions and interrupts in a simpler programming language.

With this in mind I decided upon two clear goals: to reason about the correctness of a compiler for a simple langauge including exceptions and interrupts; and to reason about combinators defined using the primitives of this language. These goals had a profound implication on the language I defined — it should be as simple as possible, focusing on the details of exception and interrupt handling, rather than useful language features. This approach would not only make any proofs clearer, but also highlight the effects of adding exceptions and interrupts to a language.

In this thesis I produced a rational reconstruction of the core features of exceptions and interrupts which is both formally justified, and used for reasoning about programs. I took a first principles approach, beginning with a language consisting only of integers and addition, and produced a high-level semantics, a compiler, and a proof of their equivalence. I repeated these steps whilst extending the language, first with exceptions, and then with interrupts, in order to study the effects of reasoning about both the compiler and semantics as the

language became more complex, as well as investigating various compilation techniques for exceptions.

This minimal approach led me to discover an error in the semantics presented in *Asynchronous Exceptions in Haskell* [MPMR01], which had remained undetected in the four years since this paper was published. The error leads to a mismatch in the possible IO actions performed by a program in the semantics, and in its implementation. The semantics does not allow an expression of the form *block* $(x)$ to be interrupted immediately before its execution begins, leading to a mismatch in the possible *IO* actions performed by compiled code. Due to the necessarily complex nature of the original semantics, this error was far from obvious.

Another benefit of a minimal approach applies to the proofs of compiler correctness. To the author's knowledge, the proofs of correctness of the compilers given in Chapters 5 and 7 are the first proofs of correctness of a compiler for languages with exceptions and interrupts respectively. This should indicate that the work has in some way achieved the first of the revised goals for the project, however the second goal of reasoning about combinators is a little under-developed.

Whilst extending the language I also studied the effects of exceptions and interrupts on specifying, and proving properties of, a higher-level combinator, *finally*. For each potential definition of the *finally* combinator I was able to give clear and simple evidence for its correctness according to a formal specification, purely by appealing to the high level semantics. However, due to the simplicity of the language, some of the arguments put forward could be viewed as somewhat unconvincing. The description of the machine given in Chapter 7 is a necessary step towards fixing this problem, as it sets the ground work for reasoning about a rather more expressive interrupt language including concurrency.

Further possible approaches to extending our ability to reason about the behaviour of combinators are discussed later in this chapter.

## 9.1 Further Work

The perceived lack of work on reasoning about exceptions and interrupts, combined with the increasing use of concurrency (and hence interrupts) in programming, suggests that a variety of future research is possible. This section details possible directions for further work which have a basis in the work carried out in this thesis.

### 9.1.1 Richer Languages

So far we have focussed on a minimal language, which has allowed us to study some of the effects that exceptions and interrupts have on programming languages. However, in order to study more realistic examples, such as those involving thread communication and synchronisation, it is important to extend the work started in this thesis to include such language features as input/output, concurrency and communication. This thesis was inspired by work on concurrency and exceptions in Haskell, so it would also seem natural to extend our work to some realistic subset of Concurrent Haskell [PGF96]. A new model of concurrency for Haskell, based on Software Transactional Memory [HMPH05], has recently been implemented by Peyton Jones et al. This new form of *lockless* concurrency has a simpler semantics that the previous model, whilst still providing support for both exceptions and interrupts, and would seem to be an ideal candidate for an extension of our work on compiler correctness and reasoning about programs.

### 9.1.2 Reasoning about Programs

By way of example, this thesis considers reasoning about a single combinator, *finally*, in terms of a specification of its intended behaviour. We could consider the correctness of a number of other useful combinators, such as *timeout*, *either* and *both* [MPMR01], first in the context of an appropriate minimal language, and then in terms of a subset of Concurrent Haskell. Is our reasoning about *finally* still valid in the presence of a richer language? How would the introduction of communication and synchronisation affect reasoning about programs? An extended language would enable us to tackle these questions, and we could also consider the correctness of more complex programs, such as a simple concurrent server.

### 9.1.3 Machine Verification

In addition to random testing using QuickCheck [CH00], a number of the lemmas and theorems in this thesis have been verified mechanically, in a variety of proof checking systems. In particular, Theorem 5.2.1 was verified in Lego by McBride, Theorem 5.3.1 was verified in Isabelle by Nipkow, and Lemmas 7.3.9 and 7.3.12 were verified in Epigram. An interesting aspect of both the Lego and Epigram verifications is their use of dependent types to precisely capture some of the stack properties of the virtual machine, such as the fact that an $ADD$ operation requires two values on the top of the stack. This use of the type system leads to a further simplification of our correctness proofs, and certainly warrants further investigation.

### 9.1.4 Bisimulation

In this thesis we have shown that our semantics and compiler are equivalent in terms of producing the same results, and we have provided a simple example of reasoning about the behaviour of programs, using the big-step semantics. It would be interesting to show an equivalence of behaviour between the high level semantics and abstract machine, by proving that a bisimulation relation exists between them. Such a relation between the compiler and the big-step semantics would allow us to show that the proofs of *behaviour* using the big-step semantics apply equally well to the compiler.

### 9.1.5 Calculating the Compiler

Towards the end of this thesis we developed a compiler for a simple language with support for both exceptions and interrupts, and gave a proof of its correctness. The formal reasoning community, however, tends to prefer *construction* [Bac03] to verification. In previous work [HW05] we have shown how to calculate an abstract machine for evaluating expressions for our simple language with exceptions. The key to the calculation is a program transformation technique called defunctionalization, first introduced by Reynolds in his work on *definitional interpreters* [Rey72], and recently repopularized by Danvy et al [DN01, ABDM03b, ABDM03a]. This work is a first step towards calculating a compiler with support for exceptions directly from the semantics, and it would be useful to investi-

gate whether these techniques could be applied to our interrupting language, and possibly further towards giving a systematic discovery of a compiler for a subset of “real” Haskell, which includes both exceptions and interrupts.

# References

[ABDM03a] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From interpreter to compiler and virtual machine: a functional derivation. Technical Report RS-03-14, University of Aarhus, 2003.

[ABDM03b] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the Fifth ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*. ACM Press, 2003.

[ALZ01] D. Ancona, G. Lagorio, and E. Zucca. A Core Calculus for Java Exceptions. *SIGPLAN Notices*, 36(11):16–30, 2001.

[App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[AWW90] Alexander Aiken, Edward L. Wimmers, and John H. Williams. Program transformation in the presence of errors. In *POPL 1990: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 210–217. ACM Press, 1990.

[Bac03] Roland Backhouse. *Program Construction: Calculating Implementations from Specifications*. Wiley, 2003.

[Bir98] Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 1998.

[BS00] Egon Borger and Wolfram Schulte. A Practical Method for Specification and Analysis of Exception Handling: A Java/JVM Case Study. *IEEE Transactions on Software Engineering*, 26(9):872–887, 2000.

[CH00] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *ACM SIGPLAN Notices*, 35(9):268–279, 2000.

[CLN01] Rance Cleaveland, Gerald Luttgen, and V. Natarajan. Priority in process algebra. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 711–765. Elsevier, 2001.

[DGL95] S. Drew, K. Gough, and J. Ledermann. Implementing zero overhead exception handling. Technical Report Technical Report 95-12, Faculty of Information Technology, Queensland University of Technology, 1995.

[DN01] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *Proceedings of the 3rd ACM SIGPLAN international conference on Principles and practice of declarative programming*. ACM Press, 2001.

[DV00] Sophia Drossopoulou and Tanya Valkevych. Java exceptions throw no surprises. Technical report, Department of Computing, Imperial College of Science, Technology and Medicine, March 2000.

[GF96] V. Gulias and J. Freire. Concurrent Programming in Haskell, 1996.

[Ghc] Ghc. Glasgow Haskell Compiler. www.haskell.org/ghc.

[GJSB00] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages, Structures and Techniques*. MIT Press, 1992.

[HMPH05] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. Submitted to the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2005.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[Hug] Hugs98. www.haskell.org/hugs.

[HW04] Graham Hutton and Joel Wright. Compiling exceptions correctly. In *Proceedings of the 7th International Conference on Mathematics of Program Construction*. Springer, 2004.

[HW05] Graham Hutton and Joel Wright. Calculating an exceptional machine. In *Proceedings of the Fifth Symposium on Trends in Functional Programming*, 2005.

[Jac01] Bart Jacobs. A Formalisation of Java's Exception Mechanism. In *ESOP 2001: Proceedings of the 10th European Symposium on Programming Languages and Systems*, pages 284–301. Springer-Verlag, 2001.

[Jon03] Simon Peyton Jones. *Haskell 98 Language and Libraries.* Cambridge University Press, 2003.

[JP03] B. Jacobs and E. Poll. Java Program Verification at Nijmegen: Developments and Perspective. Technical Report NIII-R0318, Nijmegen Institute for Computing and Information Sciences, September 2003.

[KN05] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 2005. To appear.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, 1988.

[Lai01] Jim Laird. A fully abstract game semantics of local exceptions. In *LICS 2001: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 2001.

[Led81] H. Ledgard. *Ada: An Introduction / Ada Reference Manual*. Springer Verlag, 1981.

[Lio96] Jacques-Louis Lions. ARIANE 5 Flight 501 Failure: Report by the Enquiry Board. Technical report, European Space Agency, Paris, July 1996.

[LP95] John Launchbury and Simon L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.

[LvdS94] K. Rustan M. Leino and Jan L. A. van de Snepscheut. Semantics of exceptions. In *PROCOMET '94: Proceedings of the IFIP TC2/WG2.1/WG2.2/WG2.3 Working Conference on Programming Concepts, Methods and Calculi*, pages 447–466. North-Holland, 1994.

[Mil82] Robin Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1982.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the* $\pi$*-calculus.* Cambridge University Press, 1999.

[MLP99] Andrew Moran, Søren B. Lassen, and Simon L. Peyton Jones. Imprecise exceptions, co-inductively. *Electronic Notes in Theoretical Computer Science*, 26, 1999.

[MPMR01] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. Asynchronous Exceptions In Haskell. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and Dave MacQueen. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[Pey01] Simon Peyton Jones. Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In Tony Hoare, Manfred Broy, and Ralf Steinbruggen, editors, *Engineering Theories of Software Construction*, pages 47–96. IOS Press, 2001. Presented at the 2000 Marktoberdorf Summer School.

[PGF96] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 1996.

[PL00] François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems*, 22(2):340–377, 2000.

[Plo81] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[PM02] Jens Palsberg and Di Ma. A typed interrupt calculus. In *FTRTFT '02: Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 291–310. Springer-Verlag, 2002.

[PRH+99] Simon Peyton Jones, Alastair Reid, Tony Hoare, Simon Marlow, and Fergus Henderson. A Semantics for Imprecise Exceptions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, Georgia, May 1999.

[Rei98] A. Reid. Handling exceptions in Haskell, 1998.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference*, 1972.

[Sib65] R. A. Sibley. A New Programming Language: PL/1. In *Proceedings of the 1965 20th National ACM Conference, Cleveland, Ohio, United States*, 1965.

[Spi90] Mike Spivey. A Functional Theory of Exceptions. *Science of Computer Programming*, 14(1):25–43, 1990.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[Sut05] Herb Sutter. The free lunch is over: A fundamental turn towards concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005.

[Wad92] Philip Wadler. The Essence of Functional Programming. In *Proceedings of the 19th Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 19 – 22, 1992. ACM Press.

[YR02] Kwangkeun Yi and Sukyoung Ryu. A cost-effective estimation of uncaught exceptions in standard ml programs. *Theoretical Computer Science*, 277(1-2):185–217, 2002.

# Appendix A

# Rule Induction

Rule induction is a basic technique for reasoning about operational semantics, but unfortunately most textbooks on semantics gloss over the details. This thesis makes extensive use of rule induction, so we now formally define it, and give a simple example of its use.

Suppose that we recursively define a set $\mathbb{X}$ using one axiom and one rule:

$$\frac{}{a \in \mathbb{X}} \qquad \frac{x \in \mathbb{X}}{f(x) \in \mathbb{X}}$$

That is, the value $a$ is in the set $\mathbb{X}$ (the base case), for any value $x \in \mathbb{X}$ then we have $f(x) \in \mathbb{X}$ for some function $f$ (the inductive case). Note that unlike the special case when $\mathbb{X}$ is a free datatype, there is no restriction that $f$ must be injective, or that $a$ is not in the range of $f$. This scheme can easily be generalised to multiple base cases, multiple inductive cases, and also to rules with multiple preconditions.

Then in order to prove that $\forall x \in \mathbb{X}.P(x)$ for some predicate $P$, rule induction states that it is sufficient to show that $P$ holds for $a$ (the base case), and that if $P$ holds for any $x \in \mathbb{X}$, then it also holds for $f(x)$ (the inductive case). That is, we have:

$$\frac{P(a) \qquad \forall x \in \mathbb{X}.[P(x) \Rightarrow P(f(x))]}{\forall x \in \mathbb{X}.P(x)}$$

## A.1 An Example

Consider a simple language of expressions, built up from integers with an addition operator, described by the following BNF grammar:

$$\begin{aligned}\mathbb{E} &::= \mathbb{Z} \\ &\quad\mid\ \mathbb{E} + \mathbb{E}\end{aligned}$$

Such expressions can be evaluated using a function $[\![\text{-}]\!] : \mathbb{E} \to \mathbb{Z}$, which constitutes a denotational semantics for $\mathbb{E}$, defined as follows:

$$\begin{aligned}[\![n]\!] &= n \\ [\![x + y]\!] &= [\![x]\!] + [\![y]\!]\end{aligned}$$

Evaluation of expressions can also be described using a small-step operational semantics, which we now define:

$$\frac{}{\overline{n} + \overline{m} \to \overline{n + m}}\ \textsc{Add1}$$

$$\frac{x \to x'}{x + y \to x' + y}\ \textsc{Add2} \qquad \frac{y \to y'}{x + y \to x + y'}\ \textsc{Add3}$$

The relation $\to$ is defined by a single axiom and two rules, which means that properties of $\to$ can be proved using rule induction. A simple such property is that transitions do not affect the denotation of an expression:

**Theorem A.1.1 (denotation preserved).**

$$x \to x' \Rightarrow [\![x]\!] = [\![x']\!]$$

*Proof.*

$$\begin{aligned}& x \to x' \Rightarrow [\![x]\!] = [\![x']\!] \\ \Leftrightarrow\quad & \{\ \text{define } P(x, x') \Leftrightarrow [\![x]\!] = [\![x']\!]\ \}\end{aligned}$$

$$
\begin{array}{ll}
& \forall (x, x') \in \rightarrow . P(x, x') \\
\Leftarrow & \{ \text{ rule induction for } \rightarrow \} \\
& P(\overline{n} + \overline{m}, \overline{n+m}) \\
& \quad x \rightarrow x' \wedge P(x, x') \Rightarrow P(x+y, x'+y) \\
& \quad y \rightarrow y' \wedge P(y, y') \Rightarrow P(x+y, x+y') \\
\Leftrightarrow & \{ \text{ definition of } P \} \\
& [\![\overline{n} + \overline{m}]\!] = [\![\overline{n+m}]\!] \\
& \quad x \rightarrow x' \wedge [\![x]\!] = [\![x']\!] \Rightarrow [\![x+y]\!] = [\![x'+y]\!] \\
& \quad y \rightarrow y' \wedge [\![y]\!] = [\![y']\!] \Rightarrow [\![x+y]\!] = [\![x+y']\!]
\end{array}
$$

We now verify each conjunct in turn:

$$
\begin{array}{ll}
& [\![\overline{n} + \overline{m}]\!] \\
= & \{ \text{ definition of } [\![-]\!] \} \\
& [\![\overline{n}]\!] + [\![\overline{m}]\!] \\
= & \{ \text{ definition of } [\![-]\!] \} \\
& \overline{n+m} \\
= & \{ \text{ definition of } [\![-]\!] \} \\
& [\![\overline{n+m}]\!]
\end{array}
$$

$$
\begin{array}{ll}
& [\![x+y]\!] \\
= & \{ \text{ definition of } [\![-]\!] \} \\
& [\![x]\!] + [\![y]\!] \\
= & \{ \text{ assumption that } [\![x]\!] = [\![x']\!] \} \\
& [\![x']\!] + [\![y]\!] \\
= & \{ \text{ definition of } [\![-]\!] \} \\
& [\![x'+y]\!]
\end{array}
$$

$$
\begin{array}{ll}
& [\![x+y]\!] \\
= & \{ \text{ definition of } [\![-]\!] \} \\
& [\![x]\!] + [\![y]\!] \\
= & \{ \text{ assumption that } [\![y]\!] = [\![y']\!] \} \\
& [\![x]\!] + [\![y']\!]
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \text{ definition of } [\![-]\!] \ \} \\
 & [\![x + y']\!]
\end{array}
$$

□

Just as proofs using structural induction do not normally proceed in full detail by explicitly defining a predicate and stating the induction principle being used, so that same is true with rule induction. In practice, then, the above proof would normally be written along the following lines:

*Proof.* by rule induction on $x \rightarrow x'$.

Case: ADD1

$$
\begin{array}{ll}
 & [\![\overline{n} + \overline{m}]\!] \\
= & \{ \text{ definition of } [\![-]\!] \ \} \\
 & [\![\overline{n}]\!] + [\![\overline{m}]\!] \\
= & \{ \text{ definition of } [\![-]\!] \ \} \\
 & \overline{n + m} \\
= & \{ \text{ definition of } [\![-]\!] \ \} \\
 & [\![\overline{n + m}]\!]
\end{array}
$$

Case: ADD2

We can assume $x \rightarrow x'$ by rule ADD2 for $\rightarrow$, and that $[\![x]\!] = [\![x']\!]$ as our induction hypothesis, and verify $[\![x + y]\!] = [\![x' + y]\!]$ as follows:

$$
\begin{array}{ll}
 & [\![x + y]\!] \\
= & \{ \text{ definition of } [\![-]\!] \ \} \\
 & [\![x]\!] + [\![y]\!] \\
= & \{ \text{ assumption that } [\![x]\!] = [\![x']\!] \ \} \\
 & [\![x']\!] + [\![y]\!] \\
= & \{ \text{ definition of } [\![-]\!] \ \} \\
 & [\![x' + y]\!]
\end{array}
$$

Case: ADD3

We can assume $y \rightarrow y'$ by rule ADD3 for $\rightarrow$, and that $[\![y]\!] = [\![y']\!]$ as our induction hypothesis, and verify $[\![x + y]\!] = [\![x + y']\!]$ as follows:

$$
\begin{array}{ll}
 & [\![x + y]\!] \\
= & \{ \text{ definition of } [\![-]\!] \ \} \\
 & [\![x]\!] + [\![y]\!] \\
= & \{ \text{ assumption that } [\![y]\!] = [\![y']\!] \ \} \\
 & [\![x]\!] + [\![y']\!] \\
= & \{ \text{ definition of } [\![-]\!] \ \} \\
 & [\![x + y']\!]
\end{array}
$$

□