

What is the Meaning of These Constant Interruptions?

GRAHAM HUTTON and JOEL WRIGHT

*School of Computer Science and IT
University of Nottingham, UK*

Abstract

Asynchronous exceptions, or *interrupts*, are important for writing robust, modular programs, but are traditionally viewed as being difficult from a semantic perspective. In this article we present a simple, formally justified, semantics for interrupts. Our approach is to show how a high-level semantics for interrupts can be justified with respect to a low-level implementation, by means of a compiler and its correctness theorem. In this manner we obtain two different perspectives on the problem, formally shown to be equivalent, which gives greater confidence in the correctness of our semantics.

1 Introduction

Exceptions are an important feature of modern programming languages, supporting the development of modular programs that are robust to various kinds of unexpected events. There are two basic kinds of exceptions: those that arise from within a program itself, such as a division by zero, and those that arise from its external environment, such as a timeout. The former are termed synchronous exceptions, because they can only arise at specific points, while the latter are termed asynchronous exceptions, because they can potentially arise at any point. For simplicity, however, in this article we follow the common practice of referring to synchronous exceptions as *exceptions*, and to asynchronous exceptions as *interrupts*¹.

To allow programs to recover from exceptions and interrupts, some form of language support for these features is required. For exceptions, this is usually based upon a primitive that abandons the current computation and *throws* an exception, together with a primitive that *catches* an exception thrown in one computation and handles it using another computation. In turn, support for interrupts is typically based upon the view of interrupts as special kinds of exceptions, together with a primitive that allows one computation to throw an interrupt exception in another (concurrently executing) computation, which can then be handled as any other kind of exception. To allow some control over when interrupts are received, additional primitives are usually provided for blocking and unblocking interrupts.

¹ Not to be confused with the hardware notion of an interrupt, which is more akin to an asynchronous subroutine call. Our concern is with asynchronous exceptions.

Exceptions are well studied and understood from a semantic point of view, whereas interrupts are still viewed as being difficult. Indeed, Haskell is currently the only language to provide both full support for interrupts and a formal semantics for this feature (Marlow *et al.*, 2001; Peyton Jones, 2001). But this semantics is subtle, and relies on considerable technical machinery, including the π -calculus, two-level semantics, two-level evaluation contexts, and various forms of annotations. How do we know that the semantics for interrupts is correct?

In order to focus on the essence of this problem, we abstract from the details of a real language such as Haskell, and consider a minimal language in which to understand, clarify and verify the basic semantics of interrupts. In particular, we consider a simple expression language comprising integers, addition, a single exceptional value called *throw*, a catch operator for this value, and operators for blocking and unblocking interrupts. This language does not provide features that are necessary for actual programming, but it does provide just what we need to consider the basic semantics of exceptions and interrupts. In particular, integers and addition constitute a minimal language in which to consider normal (non-exceptional) computation, *throw* and *catch* constitute a minimal extension in which computations can involve exceptions, and finally, *block* and *unblock* allow us to consider interrupts. In this context, the article makes the following contributions:

- We show how a high-level semantics for interrupts (section 5) can be justified with respect to a low-level implementation (section 7), by means of a compiler (section 8) and its correctness theorem (section 9).
- We show how the notion of a *bar* operator for transition paths (section 9) can be used to formulate and prove the compiler correctness theorem.
- We identify an error in the semantics for interrupts in Haskell (section 6), concerning the notion of when interrupts can be received.
- As a motivating example, we define and verify a *finally* operator (sections 3 and 6), a useful construct for programming with interrupts.

To the best of our knowledge, this article is the first to give a formally justified semantics for interrupts, and to verify a compiler for a language with interrupts. A summary of related work, and a discussion of the limitations of our present work and how it may be further developed, is provided in the concluding sections. Throughout the article Haskell is used as meta-language for defining types and functions. The associated code, and an extended version of the article that includes all the proofs, are available from the authors' web pages.

2 An exceptional language

We begin by considering a simple language with exceptions, comprising integer values, *throw*, addition, sequencing, and *catch*. The sequencing operator is provided for the purposes of our running example. In Haskell, the language of such expressions can be represented by the following recursive type:

$$\mathbf{data} \textit{Expr} = \textit{Val Int} \mid \textit{Throw} \mid \textit{Add Expr Expr} \mid \textit{Seqn Expr Expr} \mid \textit{Catch Expr Expr}$$

We specify the meaning of expressions in this language using a big-step operational semantics (or natural semantics), writing $e \Downarrow v$ to mean that the expression e can evaluate to the value v . Formally, the evaluation relation $\Downarrow \subseteq \text{Expr} \times \text{Value}$, where Value is the subtype of Expr comprising expressions of the form $\text{Val } n$ or Throw , is defined by the following set of inference rules:

$$\begin{array}{c}
\frac{}{\text{Val } n \Downarrow \text{Val } n} \text{VAL} \qquad \frac{}{\text{Throw} \Downarrow \text{Throw}} \text{THROW} \\
\\
\frac{x \Downarrow \text{Val } n \quad y \Downarrow \text{Val } m}{\text{Add } x \ y \Downarrow \text{Val } (n + m)} \text{ADD1} \qquad \frac{x \Downarrow \text{Throw}}{\text{Add } x \ y \Downarrow \text{Throw}} \text{ADD2} \\
\\
\frac{x \Downarrow \text{Val } n \quad y \Downarrow \text{Throw}}{\text{Add } x \ y \Downarrow \text{Throw}} \text{ADD3} \qquad \frac{x \Downarrow \text{Val } n \quad y \Downarrow v}{\text{Seqn } x \ y \Downarrow v} \text{SEQN1} \\
\\
\frac{x \Downarrow \text{Throw}}{\text{Seqn } x \ y \Downarrow \text{Throw}} \text{SEQN2} \qquad \frac{x \Downarrow \text{Val } n}{\text{Catch } x \ y \Downarrow \text{Val } n} \text{CATCH1} \\
\\
\frac{x \Downarrow \text{Throw} \quad y \Downarrow v}{\text{Catch } x \ y \Downarrow v} \text{CATCH2}
\end{array}$$

These rules specify that addition propagates an exception thrown in either argument, that sequencing propagates an exception thrown in its first argument, and otherwise discards the resulting integer value and behaves as its second argument, and that catch behaves as its first argument unless it throws an exception, in which case the exception is handled by behaving as its second argument.

Our evaluation relation \Downarrow can readily be shown to be a total function from expressions to values. Hence, we could have defined our semantics directly as a function, rather than as a relation. However, when we extend our language with interrupts later on, evaluation becomes non-deterministic, at which point the relational approach to semantics used above is more natural.

3 Finally, an example

Many programming languages with exceptions provide some form of *finally* operator that ensures that whatever happens during the execution of one computation, in particular, whether it terminates normally or with an exception, another computation will always be executed afterwards. For example, both Java and Haskell provide such an operator, which is typically used to “clean up” after a computation to ensure that important invariants are maintained.

We will use *finally* as a running example in the next few sections, as it can be defined in terms of the basic exception primitives, and illustrates a number of important issues about programming with exceptions. To define a *finally* operator

within our language, a first attempt might be to simply use sequencing:

$$\textit{finally } x \ y = \textit{Seqn } x \ y$$

This definition works fine if x produces an integer, but not if it produces an exception, in which case the semantics of sequencing means that y is not evaluated. To address this problem, we refine the definition by introducing a catch:

$$\textit{finally } x \ y = \textit{Seqn } (\textit{Catch } x \ y) \ y$$

Now if x produces an exception then y is evaluated, but if this evaluation terminates normally then the semantics of sequencing means that the second y in the definition will also be evaluated, and hence y is evaluated twice. To solve this problem, we propagate the exception within the catch by using throw:

$$\textit{finally } x \ y = \textit{Seqn } (\textit{Catch } x \ (\textit{Seqn } y \ \textit{Throw})) \ y$$

This definition now has the correct behaviour: if x produces an integer then y is evaluated, and if x produces an exception then y is evaluated and the exception is propagated. The fact that *finally* $x \ y$ ensures that evaluation of x is always succeeded by evaluation of y can be captured as follows:

Proposition 1 (behaviour of finally)

Every proof tree for an evaluation of *finally* $x \ y$ contains a single tree for the evaluation of x to the left of a single tree for the evaluation of y .

4 Adding interrupts

We now consider how to extend our language with interrupts. Rather than first extending our language with concurrency, which would be a significant undertaking in itself, we continue with our minimalist approach. In particular, we avoid the need for concurrency by considering the evaluation of a single expression within a *worst-case scenario* in which evaluation can be interrupted at any point. This behaviour is achieved by adding the following interrupt rule to our semantics:

$$\frac{}{x \Downarrow \textit{Throw}} \text{INT}$$

This rule specifies that evaluation may be interrupted at any time by simply replacing the current expression by an exception. Many readers may, quite reasonably, have concerns about this rule. For example, is it appropriate that it can be applied without any preconditions, and that any form of expression can be interrupted? We will return to such concerns shortly. Prior to this, however, we consider two immediate consequences of the above interrupt rule.

First of all, evaluation is no longer deterministic, because an expression may now produce more than one possible value. For example, the expression *Val* 1 can evaluate to either *Val* 1 or *Throw*, depending upon whether we apply the VAL rule or the INT rule. More generally, an expression may now produce any number of possible values. For example, *Catch* (*Val* 1) (*Val* 2) can evaluate to either *Val* 1, *Val* 2, or *Throw*, depending upon the rules applied.

Secondly, our *finally* operator is no longer correct, because an interrupt may interfere with its operation. For example, the definition for *finally* $x\ y$ could be interrupted just as the handler expression *Seqn* $y\ Throw$ is about to be evaluated, thereby failing to ensure that the expression y is evaluated.

5 Controlling interrupts

What is needed to address the problem with *finally* is a means to control when interrupts can be received. For this purpose, we extend our language with two new primitives, which block and unblock interrupts within an expression:

data $Expr = \dots \mid Block\ Expr \mid Unblock\ Expr$

For example, assuming that interrupts are initially unblocked, the intention is that the expression $Block\ (Add\ (Unblock\ x)\ y)$ can only be interrupted at the very start prior to entering the scope of the block, and during the evaluation of x . In particular, the addition itself cannot be interrupted, nor can the expression y , unless this expression itself explicitly unblocks interrupts.

Note that there is no intended counting of scopes with the new primitives. Rather, a single use of block or unblock immediately subsumes any number of previous uses of these primitives. For example, two nested blocks only require a single unblock to admit interrupts, rather than two nested unblocks.

To formalise the meaning of the new primitives, we first define a status type that specifies whether interrupts are currently blocked or unblocked:

data $Status = B \mid U$

We now refine our semantics to take account of this notion, writing $e \Downarrow^i v$ to mean that the expression e in interrupt status i can evaluate to the value v . That is, we define a new evaluation relation $\Downarrow \subseteq Expr \times Status \times Value$. The semantics for block and unblock are given by new rules that change the current status,

$$\frac{x \Downarrow^B v}{Block\ x \Downarrow^i v} \text{ BLOCK} \qquad \frac{x \Downarrow^U v}{Unblock\ x \Downarrow^i v} \text{ UNBLOCK}$$

while our previous rule for interrupts is modified to ensure that it can only be applied when interrupts are unblocked:

$$\frac{}{x \Downarrow^U Throw} \text{ INT}$$

In turn, the inference rules for the other primitives are simply modified to propagate the current interrupt status to their argument expressions:

$$\frac{}{Val\ n \Downarrow^i Val\ n} \text{ VAL} \qquad \frac{}{Throw \Downarrow^i Throw} \text{ THROW}$$

$$\frac{x \Downarrow^i Val\ n \quad y \Downarrow^i Val\ m}{Add\ x\ y \Downarrow^i Val\ (n + m)} \text{ ADD1} \qquad \frac{x \Downarrow^i Throw}{Add\ x\ y \Downarrow^i Throw} \text{ ADD2}$$

$$\begin{array}{c}
\frac{x \Downarrow^i \text{Val } n \quad y \Downarrow^i \text{Throw}}{\text{Add } x \ y \ \Downarrow^i \ \text{Throw}} \text{ADD3} \qquad \frac{x \Downarrow^i \text{Val } n \quad y \Downarrow^i v}{\text{Seqn } x \ y \ \Downarrow^i \ v} \text{SEQN1} \\
\\
\frac{x \Downarrow^i \text{Throw}}{\text{Seqn } x \ y \ \Downarrow^i \ \text{Throw}} \text{SEQN2} \qquad \frac{x \Downarrow^i \text{Val } n}{\text{Catch } x \ y \ \Downarrow^i \ \text{Val } n} \text{CATCH1} \\
\\
\frac{x \Downarrow^i \text{Throw} \quad y \Downarrow^i v}{\text{Catch } x \ y \ \Downarrow^i \ v} \text{CATCH2}
\end{array}$$

6 Finally revisited

Using the new primitives, we now refine our definition for *finally* $x \ y$ by first blocking interrupts, which are then unblocked prior to evaluating x :

$$\text{finally } x \ y \ = \ \text{Block } (\text{Seqn } (\text{Catch } (\text{Unblock } x) (\text{Seqn } y \ \text{Throw})) \ y)$$

Modulo differences in syntax, the *finally* operator in Haskell is defined in precisely the same way (Marlow *et al.*, 2001). Prior to specifying the behaviour of this new definition, we consider a number of semantic issues:

- (1) The new definition ensures that evaluation of x is started with interrupts unblocked, but other options are possible, such as removing the use of `unblock`, or replacing it by some means of restoring the previous status.
- (2) The new definition ensures that evaluation of y is started with interrupts blocked, but this does not guarantee that y cannot be interrupted, in particular because y itself may contain an explicit use of `unblock`.
- (3) If interrupts are initially unblocked, the new definition can be interrupted at the very start prior to entering the scope of the block. In this case, y is not evaluated, and hence *finally* does not behave as originally expected.

The first issue concerns a design choice, but for our purposes the above definition for *finally* as used in Haskell will suffice, so we do not explore this further here. In turn, the second issue is addressed simply by taking account of the fact that y may be interruptible when modifying our behavioural proposition.

The third issue requires more careful thought. One solution to (3) would be to modify the interrupt rule to prevent an interrupt being received if the expression being evaluated is just about to block interrupts, in the following manner:

$$\frac{x \neq \text{Block } y}{x \Downarrow^U \text{Throw}}$$

This rule would fix the problem with *finally*, but there are other situations in which the same problem occurs. For example, in the expression $\text{Seqn } (\text{Block } x) \ y$, the top-level primitive is not a block and hence the interrupt rule above could be applied.

However, when this expression is compiled to low-level code, the first operation that will actually be performed is to block interrupts. Hence, the notion of “just about to block interrupts” is more subtle than might first be expected.

A second solution for issue (3) above is to retain our existing interrupt rule, but modify our proposition for *finally* x y . In particular, we could specify that evaluation of y is only required if evaluation of x actually starts, which won't be the case if an interrupt is received prior to entering the scope of the block. For simplicity, and for consistency with the intended semantics of interrupts in Haskell upon which our definition for *finally* is based, we adopt this solution here.

The use of the word *intended* above refers to the fact that our work on this article identified an error in the semantics for interrupts in Haskell. In particular, the semantics (Marlow *et al.*, 2001) defines a rule for interrupts that corresponds to our first solution for (3), whereas in the actual implementation (Peyton Jones & Marlow, 2004), which the semantics is intended to formalise, there is no such restriction on an interrupt being received if interrupts are just about to be blocked, which corresponds to our second solution.

In conclusion, our behavioral proposition for the *finally* operator can now be revised to take account of the possibility of interrupts as follows:

Proposition 2 (revised behaviour of finally)

Every proof tree for an evaluation of *finally* x y contains at most one tree for the evaluation of x (with interrupts unblocked), which if it exists occurs to the left of a single tree for the evaluation of y (with interrupts blocked.)

7 Virtual machine

How do we know that our semantics for interrupts is correct? In particular, how does our high-level semantics reflect our low-level intuition about interrupts? As a first step towards addressing this issue, in this section we present a simple virtual machine that can be used to implement our language of expressions. The machine operates by means of a stack represented as a list of items, where each item is either an integer value, a piece of handler code, or an interrupt status:

```
type Stack = [Item]
data Item  = VAL Int | HAN Code | INT Status
```

In turn, code for the machine comprises a list of operations on the stack:

```
type Code = [Op]
data Op   = PUSH Int | THROW | ADD | POP |
            MARK Code | UNMARK |
            SET Status | RESET
```

We specify the meaning of such code using a machine with two modes of operation. During *normal* execution, the machine operates on a state that comprises a piece of code, an interrupt status and a stack. Formally, the transition relation $\longrightarrow \subseteq$

$State \times State$ for normal execution, where $State = Code \times Status \times Stack$, is defined by the following set of rewrite rules:

$$\begin{array}{ll}
\langle PUSH\ n : ops, i, s \rangle & \longrightarrow \langle ops, i, VAL\ n : s \rangle \\
\langle THROW : ops, i, s \rangle & \longrightarrow \langle\langle i, s \rangle\rangle \\
\langle ADD : ops, i, VAL\ m : VAL\ n : s \rangle & \longrightarrow \langle ops, i, VAL\ (n + m) : s \rangle \\
\langle POP : ops, i, VAL\ _ : s \rangle & \longrightarrow \langle ops, i, s \rangle \\
\langle MARK\ ops' : ops, i, s \rangle & \longrightarrow \langle ops, i, HAN\ ops' : s \rangle \\
\langle UNMARK : ops, i, x : HAN\ _ : s \rangle & \longrightarrow \langle ops, i, x : s \rangle \\
\langle SET\ i' : ops, i, s \rangle & \longrightarrow \langle ops, i', INT\ i : s \rangle \\
\langle RESET : ops, _, x : INT\ i' : s \rangle & \longrightarrow \langle ops, i', x : s \rangle
\end{array}$$

These rules specify that *PUSH* places an integer value onto the stack, *THROW* changes the machine into exceptional execution mode (indicated by the use of double parentheses, and to be defined shortly), *ADD* replaces two integer values on the stack with their sum, and *POP* simply removes an integer value from the stack. In turn, the *MARK* operation places handler code onto the stack, and dually, *UNMARK* removes such code from the stack. Finally, *SET* changes the interrupt status to a given value and saves the previous status on the stack, while *RESET* removes such a status and restores the interrupt status to this value. We will consider the behaviour of these operations in more detail in the next section, when we present a compiler that produces code for this machine.

Our transition relation \longrightarrow is currently deterministic, but as with our evaluation relation \Downarrow it becomes non-deterministic when we consider interrupts, which is achieved by adding the following rewrite rule:

$$\langle _ : _, U, s \rangle \longrightarrow \langle\langle U, s \rangle\rangle$$

This rule specifies that, provided interrupts are unblocked, normal execution of any operation may be interrupted by simply changing the machine into exceptional execution mode. Note that the empty list of code is not interruptible, as this represents a state in which the machine has terminated.

In the other mode of operation, *exceptional* execution, the machine operates on a reduced state comprising just an interrupt status and a stack. Formally, the transition relation $\longrightarrow \subseteq State' \times State'$ for exceptional execution, where $State' = Status \times Stack$, is defined by the following set of rewrite rules:

$$\begin{array}{ll}
\langle\langle i, VAL\ _ : s \rangle\rangle & \longrightarrow \langle\langle i, s \rangle\rangle \\
\langle\langle _, INT\ i' : s \rangle\rangle & \longrightarrow \langle\langle i', s \rangle\rangle \\
\langle\langle i, HAN\ ops : s \rangle\rangle & \longrightarrow \langle ops, i, s \rangle
\end{array}$$

That is, if the top of the stack is an integer value it is removed, if the top is a saved interrupt status it is removed and restored as the current status, and finally, if the top of the stack is handler code it is used to change the machine back to normal execution mode. These rules formalise the basic method of implementing exceptions known as stack unwinding, in which an exception being thrown results in items being popped from the stack seeking a handler.

8 Compiler

We now define a function *comp* that compiles an expression into code for our virtual machine. In fact, we define a more general function that takes an additional argument, in the form of a piece of code to be appended to the compiled code. Using such an accumulator simplifies the process of proving the compiler correct (Hutton, 2007, section 13.7), and also admits a simpler implementation of exceptions than in our previous work (Hutton & Wright, 2004).

$$\begin{aligned}
\mathit{comp} &:: \mathit{Expr} \rightarrow \mathit{Code} \rightarrow \mathit{Code} \\
\mathit{comp} (\mathit{Val} \ n) \ \mathit{ops} &= \mathit{PUSH} \ n : \ \mathit{ops} \\
\mathit{comp} (\mathit{Throw}) \ \mathit{ops} &= \mathit{THROW} : \ \mathit{ops} \\
\mathit{comp} (\mathit{Add} \ x \ y) \ \mathit{ops} &= \mathit{comp} \ x \ (\mathit{comp} \ y \ (\mathit{ADD} : \ \mathit{ops})) \\
\mathit{comp} (\mathit{Seqn} \ x \ y) \ \mathit{ops} &= \mathit{comp} \ x \ (\mathit{POP} : \ \mathit{comp} \ y \ \mathit{ops}) \\
\mathit{comp} (\mathit{Catch} \ x \ y) \ \mathit{ops} &= \mathit{MARK} \ (\mathit{comp} \ y \ \mathit{ops}) : \ \mathit{comp} \ x \ (\mathit{UNMARK} : \ \mathit{ops}) \\
\mathit{comp} (\mathit{Block} \ x) \ \mathit{ops} &= \mathit{SET} \ B : \ \mathit{comp} \ x \ (\mathit{RESET} : \ \mathit{ops}) \\
\mathit{comp} (\mathit{Unblock} \ x) \ \mathit{ops} &= \mathit{SET} \ U : \ \mathit{comp} \ x \ (\mathit{RESET} : \ \mathit{ops})
\end{aligned}$$

These equations specify that *Val* and *Throw* are compiled directly to the corresponding machine operations; in turn, *Add* is compiled by producing code for the two argument expressions in turn, and then adding the resulting two integers on the stack; *Seqn* is compiled by producing code for the first argument, removing the resulting integer from the stack, and producing code for the second argument; *Catch* is compiled in the manner explained below; and finally, *Block* and *Unblock* are compiled by changing the interrupt status, producing code for the argument expression, and then restoring the previous interrupt status.

Returning to the case for *Catch*, this primitive is compiled by marking the stack with the compiled code for the handler, compiling the expression to be evaluated, and then unmarking the stack by removing the handler code. In this way, the *MARK* and *UNMARK* operations delimit the extent of the handler to the particular expression being evaluated, in the sense that the handler is only present on the stack during evaluation of this expression. Note that the stack is marked with actual code, rather than the address of the code as would be used in a real implementation. Moreover, two copies of the additional code *ops* are used in the compilation of *catch*, rather than a single copy and two jumps. However, our concern in this article is with basic semantic issues, rather than practicality or efficiency.

9 Compiler correctness

To specify the correctness of our compiler, we will exploit two notions of reachability for a transition relation \longrightarrow on states. First of all, we write $x \xrightarrow{*} Y$ if the state x can reach everything in the set of states Y , defined by $\forall y \in Y. x \xrightarrow{*} y$. Secondly, we write $x \triangleleft Y$ if the state x will reach something in Y , or equivalently, x is *barred* by Y (Troelstra & van Dalen, 1988), defined by the following two rules:

$$\frac{x \in Y}{x \triangleleft Y} \qquad \frac{\forall x'. x \longrightarrow x' \Rightarrow x' \triangleleft Y}{x \triangleleft Y}$$

Writing $x \triangleleft Y$ to mean that both $x \xrightarrow{*} Y$ and $x \triangleleft Y$, the correctness of our compiler can now be captured as follows: for all expressions e and integers n ,

$$\begin{aligned} & \langle \text{comp } e [], U, [] \rangle \\ \triangleleft & \{ \langle [], U, [VAL\ n] \rangle \mid e \Downarrow^U Val\ n \} \cup \{ \langle U, [] \rangle \mid e \Downarrow^U Throw \} \end{aligned}$$

This property specifies that if the compiled code for an expression is executed with interrupts unblocked and an empty initial stack, this process can terminate with any number on the stack that is permitted by the semantics of expressions, can terminate with an uncaught exception if this is permitted by the semantics, and moreover, will terminate with one of these possible outcomes.

Note that both parts of the compiler correctness property are necessary. Just using $\xrightarrow{*}$ would not be strong enough, as this would only specify that from the start state we can reach everything in the set of expected results, but does not preclude execution paths that do not reach this set; that is, the set of expected results could be too small. Conversely, just using \triangleleft would only specify that from the start state we will reach something in the resulting set, but does not ensure that we can reach everything; that is, the set could be too big.

For the purposes of proof, however, we generalise the above property to arbitrary additional code, interrupt statuses and initial stacks.

Theorem 1 (compiler correctness)

For all expressions e , code ops , statuses i , stacks s and integers n :

$$\begin{aligned} & \langle \text{comp } e ops, i, s \rangle \\ \triangleleft & \{ \langle ops, i, VAL\ n : s \rangle \mid e \Downarrow^i Val\ n \} \cup \{ \langle i, s \rangle \mid e \Downarrow^i Throw \} \end{aligned}$$

10 Related work

Exceptions in various forms have been the subject of research for more than three decades, so not surprisingly there is a large body of related work. In this section we survey a selection of this work, under the general headings of design, implementation, semantics, reasoning and expressiveness.

Design. Exceptions because a subject of research in their own right with the publication of Goodenough's seminal article (1975), which led to the first proper language support for exceptions, in PL/I (Auwaerter, 1976). A summary of early developments is given in (Cristian, 1989), while the design space for exception handling mechanisms is explored in (Drew & Gough, 1994). A modern account of the fundamental concepts, design issues, and the exception handling mechanisms provided in a number of languages, is given by (Sebesta, 2006).

Implementation. The basic technique of stack unwinding can be optimised in a number of ways, such as using a separate stack of handler addresses to make the process more efficient (Ramsey & Jones, 2000), or using a separate table of handler extents to avoid a run-time cost for installing a handler (Drew *et al.*, 1995).

Exceptions can also readily be implemented in a compiler that is based upon the use of continuations (Appel, 1992). A detailed survey of the techniques used to implement exceptions and interrupts is given in (Chase, 1994a; Chase, 1994b).

Semantics. A range of approaches to the semantics of exceptions have been explored, including the use of weakest preconditions (Leino & van de Snepscheut, 1994), continuations (Reynolds, 1972), special return values (Spivey, 1990), and games (Laird, 2001). Modern languages in which the semantics of exceptions have been studied include Java (Jacobs, 2001; Ancona *et al.*, 2001; Klein & Nipkow, 2005; Drossopoulou & Valkevych, 2000; Borger & Schulte, 2000), ML (Milner *et al.*, 1997), and Haskell (Peyton Jones, 2001; Peyton Jones *et al.*, 1999; Moran *et al.*, 1999).

In the case of interrupts, most previous work on semantics has been within process calculi such as CSP (Hoare, 1985), CCS (Milner, 1989) and the π -calculus (Milner, 1999), often within the more general setting of prioritized processes (Cleaveland *et al.*, 2001). The notion of engines explored in Scheme provides an implementation-oriented semantics for the special case of timeouts (Haynes & Friedman, 1984; Dybvig & Hieb, 1989). As noted earlier on, Haskell is currently the only language with both full support for asynchronous interrupts and a formal semantics for this feature (Marlow *et al.*, 2001; Peyton Jones, 2001).

Reasoning. Early work on reasoning about exceptions focused on their impact on program transformation (Aiken *et al.*, 1990), and the verification of basic properties of exception primitives (Spivey, 1990; Leino & van de Snepscheut, 1994). In the context of exceptions in Java, researchers have considered the verification of a compiler (Klein & Nipkow, 2005; Borger & Schulte, 2000), verification of simple programs (Jacobs & Poll, 2003), and soundness for a type system that covers both normal and exceptional behaviour (Drossopoulou & Valkevych, 2000). Research in ML has focused on the use of static analysis to detect potentially uncaught exceptions (Yi & Ryu, 2002; Pessaux & Leroy, 2000) and on type soundness in the presence of exceptions (Wright & Felleisen, 1994), while in Haskell a number of basic program equivalences and transformations have been verified (Moran *et al.*, 1999).

As a consequence of the lack of languages with formally-based support for interrupts, reasoning about the correctness of programs that use interrupts is largely unexplored, although the use of types to detect potentially unbounded stack usage in interrupt systems has been considered (Palsberg & Ma, 2002).

Expressiveness. Exceptions and continuations are both powerful mechanisms for altering control flow, so it is natural to ask how they compare. Early work showed how exceptions could be used as an alternative to continuations for defining the semantics of jumps (Allison, 1989). More recently, a range of results have been established concerning the relative expressiveness of exceptions and continuations in the context of other features such as recursion and state (Lillibridge, 1999; Riecke & Thielecke, 1999; Thielecke, 2000; Laird, 2002).

11 Further work

In this final section we discuss the limitations of our present work on the semantics of interrupts, and how it may be further developed in the future.

Generalisation. Our key weapon in this article has been simplicity, by abstracting from the details of a real language to focus on a minimal language with interrupts. This approach has allowed us to establish two important milestones: a formally justified semantics for interrupts, and the verification of a compiler for interrupts. The most important step now is to consider how our work can be scaled-up to more realistic languages, both to investigate how interrupts interact with other language features, and to study more advanced programming examples.

In this regard, we are particularly interested in generalising our work to the core language of Concurrent Haskell, an extension of the λ -calculus that supports recursion (and hence non-termination), input/output, concurrency, communication, exceptions and interrupts. The introduction of non-termination and concurrency add considerably to the complexity of the compiler correctness problem, in particular by requiring a shift from an inductive to a co-inductive notion of equality, based upon bisimilarity (Gordon, 1995). Current approaches to compiler correctness for concurrent languages utilise a process calculus as an intermediate language (Wand, 1995), but we are in the process of developing a simpler approach that avoids the need for an intermediate language, by establishing a bisimulation directly between a high-level semantics and its low-level implementation.

Mechanisation. Even for such a simple language, our compiler correctness proof was non-trivial, and required great attention to detail. In our previous work on the correctness of a compiler for exceptions (Hutton & Wright, 2004), our proof was mechanically verified using Isabelle (Nipkow, 2004). It would be interesting to attempt a similar verification of our proof for interrupts, for which purposes we are exploring the use of Epigram (McBride *et al.*, 2006). Preliminary work has shown how the use of dependent types in Epigram can be used to simplify the process of proving a compiler correct, by shifting some of the burden of proof from the user to the type checker (McKinna & Wright, 2006). Once our work is generalised to more realistic languages, we anticipate that mechanical support is no longer just useful, but becomes an essential part of the verification task.

Strength. An important question about compiler correctness in the presence of interrupts concerns the strength of the theorem. In this article, we have established that compiled code can produce every result that is permitted by the semantics for the language (completeness), and dually, that compiled code will always produce a result that is permitted by the semantics (soundness.) However, weaker or stronger notions of correctness may sometimes be appropriate.

First of all, some language implementations are by design not complete with respect to the semantics for the language, because implementing every behaviour that is permitted by the semantics may not be practical. For example, an implementation may only deliver interrupts at *safe points*, and not insert safe points as often as the semantics requires, because doing so would be prohibitively expensive. In such circumstances, a weaker notion of completeness would be appropriate, which would thereby result in a weaker compiler correctness theorem.

Secondly, a language with observable effects, such as input/output, will require a stronger notion of correctness. In particular, our present theorem is based upon the extensional notion of comparing final results, whereas the presence of effects

demands the intentional notion of comparing intermediate actions. For example, for an interactive program we are not just interested in its final result, but any input/output that takes place in the process of producing this result. Interrupts themselves can also be viewed as a form of effect, and one can consider strengthening our correctness theorem to say that if an expression and its compiled code receive an interrupt “at the same time”, they should behave “in the same way”. Formalising these ideas will require the development of a suitable notion of bisimulation between a semantics and its implementation, as noted above.

Reasoning. Dealing with interrupts is an important, and increasingly inevitable, aspect of modern software production. Despite their importance, however, the issue of provable correctness for programs in the presence of interrupts has received little attention, due to the lack of languages with a formal semantics for interrupts. In this article, we verified a simple *finally* operator in the presence of interrupts, but this is only a first step. For example, one can consider a number of useful operators that can be defined within Concurrent Haskell, such as *timeout*, *either* and *both*, together with programs written using them. As with scaling our present work up to more realistic languages, we anticipate the mechanical support will be required for the purposes of reasoning about more realistic programs.

Calculation. In formal reasoning, we often seek to replace verification by construction (Backhouse, 2003). For example, rather than first writing a compiler and then proving its correctness, it would be preferable to calculate the compiler directly from a semantics for the language, with the aim of giving a systematic discovery of the basic implementation techniques, as opposed to a post-hoc verification.

In our previous work, we have taken a step towards this goal in the context of exceptions, by showing how to calculate an abstract machine in which the key ideas of marking, unmarking, and unwinding the stack arise directly from the calculation itself, without requiring any prior knowledge of these concepts (Hutton & Wright, 2006). It would, however, be preferable to calculate a compiler, which differs from an abstract machine in that it generates code for execution using a lower-level virtual machine, rather than executing programs directly. Building upon recent work on compiler construction (Ager *et al.*, 2003), it would be interesting to try and calculate a compiler for exceptions (and interrupts if this feasible) by factorising our abstract machine into a compiler and a virtual machine.

Acknowledgements

We would like to thank Kathleen Fisher and the referees for comments that greatly improved the paper, Simon Peyton Jones and Simon Marlow for useful feedback, Thorsten Altenkirch for suggesting the use of \triangleleft , Conor McBride for showing how to prove its transitivity, and Microsoft Research Ltd in Cambridge for providing funding. The Haskell code was typeset using lhs2TeX (Hinze & Löh, 2006), and QuickCheck (Claessen & Hughes, 2000) was used for testing purposes.

References

- Ager, Mads Sig, Biernacki, Dariusz, Danvy, Olivier, & Midtgaard, Jan. (2003). *From Interpreter to Compiler and Virtual Machine: a Functional Derivation*. Technical Report RS-03-14. BRICS, Aarhus, Denmark.
- Aiken, Alexander, Wimmers, Edward L., & Williams, John H. (1990). Program Transformation in the Presence of Errors. *Pages 210–217 of: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press.
- Allison, Lloyd. (1989). Direct Semantics and Exceptions Define Jumps and Coroutines. *Information Processing Letters*, **31**, 327–330.
- Ancona, D., Lagorio, G., & Zucca, E. (2001). A Core Calculus for Java Exceptions. *SIGPLAN Notices*, **36**(11), 16–30.
- Appel, Andrew. (1992). *Compiling With Continuations*. Cambridge University Press.
- Auwaerter, J. F. (1976). *American National Standard – Programming Language PL/I*. ANSI X3.53-1976. American National Standards Institute, New York.
- Backhouse, Roland. (2003). *Program Construction: Calculating Implementations from Specifications*. John Wiley.
- Borger, Egon, & Schulte, Wolfram. (2000). A Practical Method for Specification and Analysis of Exception Handling: A Java/JVM Case Study. *IEEE Transactions on Software Engineering*, **26**(9), 872–887.
- Chase, David. (1994a). Implementation of Exception Handling, Part I. *The Journal of C Language Translation*, **5**(4), 229–240.
- Chase, David. (1994b). Implementation of Exception Handling, Part II. *The Journal of C Language Translation*, **6**(1), 20–32.
- Claessen, Koen, & Hughes, John. (2000). QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*.
- Cleaveland, Rance, Luttmann, Gerald, & Natarajan, V. (2001). Priority in Process Algebra. *Pages 711–765 of: Bergstra, J.A., Ponse, A., & Smolka, S.A. (eds), Handbook of Process Algebra*. Elsevier.
- Cristian, Flaviu. (1989). Exception Handling. *Pages 68–97 of: Anderson, T. (ed), Dependability of Resilient Computers*. Blackwell Scientific Publications.
- Drew, S., Gough, K. J., & Ledermann, J. (1995). *Implementing Zero Overhead Exception Handling*. Tech. rept. 95-12. Faculty of Information Technology, Queensland University of Technology.
- Drew, Steven J., & Gough, K. John. (1994). Exception Handling: Expecting The Unexpected. *Computer languages*, **20**(2), 69–87.
- Drossopoulou, Sophia, & Valkevych, Tanya. (2000). *Java Exceptions Throw No Surprises*. Technical report. Department of Computing, Imperial College of Science, Technology and Medicine.
- Dybvig, R. Kent, & Hieb, Robert. (1989). Engines from Continuations. *Journal of computer languages*, **14**(2), 109–123.
- Goodenough, John B. (1975). Exception Handling: Issues and a Proposed Notation. *Communications of the ACM*, **18**(12), 683–696.
- Gordon, Andrew. (1995). *Bisimilarity as a Theory of Functional Programming*. BRICS Notes Series NS-95-3. Aarhus University.
- Haynes, Christopher T., & Friedman, Daniel P. (1984). Engines Build Process Abstractions. *Pages 18–24 of: Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*. ACM Press.

- Hinze, Ralf, & Löh, Andres. (2006). *The lhs2TeX System for Typesetting Haskell*. Available from: <http://www.cs.uu.nl/~andres/lhs2tex/>.
- Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice Hall.
- Hutton, Graham. (2007). *Programming in Haskell*. Cambridge University Press.
- Hutton, Graham, & Wright, Joel. (2004). Compiling Exceptions Correctly. *Proceedings of the 7th International Conference on Mathematics of Program Construction*. Lecture Notes in Computer Science, vol. 3125. Stirling, Scotland: Springer.
- Hutton, Graham, & Wright, Joel. (2006). Calculating an Exceptional Machine. Loidl, Hans-Wolfgang (ed), *Trends in Functional Programming volume 5*. Intellect. Selected papers from the Fifth Symposium on Trends in Functional Programming, Munich, November 2004.
- Jacobs, B., & Poll, E. (2003). *Java Program Verification at Nijmegen: Developments and Perspective*. Technical Report NIII-R0318. Nijmegen Institute for Computing and Information Sciences.
- Jacobs, Bart. (2001). A Formalisation of Java's Exception Mechanism. *Pages 284–301 of: ESOP 2001: Proceedings of the 10th European Symposium on Programming Languages and Systems*. Springer-Verlag.
- Klein, Gerwin, & Nipkow, Tobias. (2005). A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler. *ACM Transactions on Programming Languages and Systems*. To appear.
- Laird, Jim. (2001). A Fully Abstract Game Semantics of Local Exceptions. *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society.
- Laird, Jim. (2002). Exceptions, Continuations and Macro-expressiveness. *Proceedings of the 11th European Symposium on Programming Languages and Systems*. Springer Verlag.
- Leino, K. Rustan M., & van de Snepscheut, Jan L. A. (1994). Semantics of Exceptions. *Pages 447–466 of: Proceedings of the IFIP Working Conference on Programming Concepts, Methods and Calculi*. North-Holland.
- Lillibridge, Mark. (1999). Unchecked Exceptions Can Be Strictly More Powerful Than Call/CC. *Higher-Order and Symbolic Computation*, **12**(1), 75–104.
- Marlow, Simon, Peyton Jones, Simon, Moran, Andrew, & Reppy, John. (2001). Asynchronous Exceptions In Haskell. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.
- McBride, Conor, *et al.* . (2006). *The Epigram System*. Available on the web from: <http://www.e-pig.org/>.
- McKinna, James, & Wright, Joel. 2006 (July). *Towards Type-Correct, Provably Correct, Compilers: A Case Study in Epigram*. Accepted for publication in the Journal of Functional Programming.
- Milner, Robin. (1989). *Communication and Concurrency*. Prentice Hall.
- Milner, Robin. (1999). *Communicating and Mobile Systems: The π -Calculus*. Cambridge University Press.
- Milner, Robin, Tofte, Mads, Harper, Robert, & MacQueen, Dave. (1997). *The Definition of Standard ML (Revised)*. MIT Press.
- Moran, Andrew, Lassen, Søren B., & Jones, Simon L. Peyton. (1999). Imprecise Exceptions, Co-Inductively. *Electronic Notes in Theoretical Computer Science*, **26**.
- Nipkow, Tobias. (2004). Compiling Exceptions Correctly. *Archive of Formal Proofs*. Available from <http://afp.sourceforge.net/>.
- Palsberg, Jens, & Ma, Di. (2002). A Typed Interrupt Calculus. *Pages 291–310 of: Pro-*

- ceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer-Verlag.
- Pessaux, François, & Leroy, Xavier. (2000). Type-Based Analysis of Uncaught Exceptions. *ACM Transactions on Programming Languages and Systems*, **22**(2), 340–377.
- Peyton Jones, Simon. (2001). Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-Language Calls in Haskell. *Pages 47–96 of: Hoare, Tony, Broy, Manfred, & Steinbruggen, Ralf (eds), Engineering Theories of Software Construction*. IOS Press. Presented at the 2000 Marktoberdorf Summer School.
- Peyton Jones, Simon, & Marlow, Simon. (2004). Personal communication.
- Peyton Jones, Simon, Reid, Alastair, Hoare, Tony, Marlow, Simon, & Henderson, Fergus. (1999). A Semantics For Imprecise Exceptions. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.
- Ramsey, Norman, & Jones, Simon Peyton. (2000). A Single Intermediate Language That Supports Multiple Implementations Of Exceptions. *Pages 285–298 of: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. ACM Press.
- Reynolds, John C. (1972). Definitional Interpreters for Higher-Order Programming Languages. *Pages 717–740 of: Proceedings of the ACM Annual Conference*. ACM Press.
- Riecke, Jon G., & Thielecke, Hayo. (1999). Typed Exeptions and Continuations Cannot Macro-Express Each Other. *Proceedings of the 26th International Colloquium on Automata, Languages and Programming*. Springer Verlag.
- Sebesta, Robert W. (2006). *Concepts of Programming Languages (7th edition)*. Pearson Addison Wesley.
- Spivey, Mike. (1990). A Functional Theory of Exceptions. *Science of Computer Programming*, **14**(1), 25–43.
- Thielecke, Hayo. (2000). On Exceptions Versus Continuations in the Presence of State. *Proceedings of the 9th European Symposium on Programming Languages and Systems*. Springer Verlag.
- Troelstra, A. S., & van Dalen, D. (1988). *Constructivism in Mathematics: An Introduction*. Vol. 1. Elsevier.
- Wand, Mitchell. (1995). Compiler Correctness for Parallel Languages. *Proceedings of the 7th International Conference on Functional Programming and Computer Architecture*. ACM Press, La Jolla, California.
- Wright, Andrew K., & Felleisen, Matthias. (1994). A Syntactic Approach to Type Soundness. *Information and Computation*, **115**(1), 38–94.
- Yi, Kwangkeun, & Ryu, Sukyoung. (2002). A Cost-Effective Estimation of Uncaught Exceptions in Standard ML Programs. *Theoretical Computer Science*, **277**(1-2), 185–217.