Compiling Exceptions Correctly

Graham Hutton and Joel Wright

School of Computer Science and IT
University of Nottingham, United Kingdom

Abstract. Exceptions are an important feature of modern programming languages, but their compilation has traditionally been viewed as an advanced topic. In this article we show that the basic method of compiling exceptions using stack unwinding can be explained and verified both simply and precisely, using elementary functional programming techniques. In particular, we develop a compiler for a small language with exceptions, together with a proof of its correctness.

1 Introduction

Most modern programming languages support some form of programming with *exceptions*, typically based upon a primitive that abandons the current computation and *throws* an exception, together with a primitive that *catches* an exception and continues with another computation [3, 16, 11, 8]. In this article we consider the problem of compiling such exception primitives.

Exceptions have traditionally been viewed as an advanced topic in compilation, usually being discussed only briefly in courses, textbooks, and research articles, and in many cases not at all. In this article, we show that the basic method of compiling exceptions using *stack unwinding* can in fact be explained and verified both simply and precisely, using elementary functional programming techniques. In particular, we develop a compiler for a small language with exceptions, together with a proof of its correctness with respect to a formal semantics for this language. Surprisingly, this appears to be the first time that a compiler for exceptions has been proved to be correct.

In order to focus on the essence of the problem and avoid getting bogged down in other details, we adopt a particularly simple language comprising just four components, namely integer values, an addition operator, a single exceptional value called throw, and a catch operator for this value. This language does not provide features that are necessary for actual programming, but it *does* provide just what we need for our expository purposes in this article. In particular, integers and addition constitute a minimal language in which to consider computation using a stack, and throw and catch constitute a minimal extension in which such computations can involve exceptions.

Our development proceeds in three steps, starting with the language of integer values and addition, then adding throw and catch to this language, and finally adding explicit jumps to the virtual machine. Starting with a simpler language allows us to introduce our approach to compilation and its correctness

without the extra complexity of exceptions. In turn, deferring the introduction of jumps allows us to introduce our approach to the compilation of exceptions without the extra complexity of dealing with jump addresses.

All the programs in the article are written in Haskell [11], but we only use the basic concepts of recursive types, recursive functions, and inductive proofs, what might be termed the "holy trinity" of functional programming. An extended version of the article that includes the proofs omitted in this conference version for reasons of space is available from the authors' home pages.

2 Arithmetic expressions

Let us begin by considering a simple language of arithmetic expressions, built up from integers using an addition operator. In Haskell, the language of such expressions can be represented by the following type:

$$\textbf{data}\ \mathit{Expr} \quad = \quad \mathit{Val}\ \mathit{Int} \mid \mathit{Add}\ \mathit{Expr}\ \mathit{Expr}$$

The semantics of such expressions is most naturally given denotationally [14], by defining a function that evaluates an expression to its integer value:

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

Now let us consider compiling arithmetic expressions into code for execution using a stack machine, in which the stack is represented as a list of integers, and code comprises a list of push and add operations on the stack:

$$\begin{array}{lcl} \textbf{type}\ \mathit{Stack} & = & [\mathit{Int}] \\ \textbf{type}\ \mathit{Code} & = & [\mathit{Op}] \\ \textbf{data}\ \mathit{Op} & = & \mathit{PUSH}\ \mathit{Int} \mid \mathit{ADD} \end{array}$$

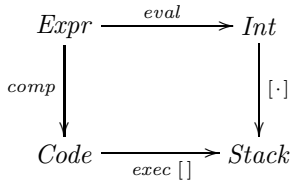
For ease of identification, we always use upper-case names for machine operations. Functions that compile an expression into code, and execute code using an initial stack to give a final stack, can now be defined as follows:

$$\begin{array}{lcl} \mathit{comp} & :: & \mathit{Expr} \rightarrow \mathit{Code} \\ \mathit{comp}\ (\mathit{Val}\ n) & = & [\mathit{PUSH}\ n] \\ \mathit{comp}\ (\mathit{Add}\ x\ y) & = & \mathit{comp}\ x +\!\!+ \mathit{comp}\ y +\!\!+ [\mathit{ADD}] \\ \mathit{exec} & :: & \mathit{Stack} \rightarrow \mathit{Code} \rightarrow \mathit{Stack} \\ \mathit{exec}\ s\ [\,] & = & s \\ \mathit{exec}\ s\ (\mathit{PUSH}\ n : \mathit{ops}) & = & \mathit{exec}\ (n : s)\ \mathit{ops} \\ \mathit{exec}\ (m : n : s)\ (\mathit{ADD} : \mathit{ops}) & = & \mathit{exec}\ (n + m : s)\ \mathit{ops} \end{array}$$

For simplicity, the function *exec* does not consider the case of *stack underflow*, which arises if the stack contains less than two integers when executing an add operation. We will return to this issue in the next section.

3 Compiler correctness

The correctness of our compiler for expressions with respect to our semantics can be expressed as the commutativity of the following diagram:



That is, compiling an expression and then executing the resulting code using an empty initial stack gives the same final stack as evaluating the expression and then converting the resulting integer into a singleton stack:

$$exec\ [\,]\ (comp\ e) \quad = \quad [eval\ e]$$

In order to prove this result, however, it is necessary to generalise from the empty initial stack to an arbitrary initial stack.

Theorem 1 (compiler correctness).

$$exec\ s\ (comp\ e) \quad = \quad eval\ e\ :\ s$$

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

Case: $e = Val\ n$

$$\begin{array}{ll} & exec\ s\ (comp\ (Val\ n)) \\ = & \qquad \{ \text{ definition of } comp \ \} \\ & exec\ s\ [PUSH\ n] \\ = & \qquad \{ \text{ definition of } exec \ \} \\ & n\ :\ s \\ = & \qquad \{ \text{ definition of } eval \ \} \\ & eval\ (Val\ n)\ :\ s \end{array}$$

Case: $e = Add\ x\ y$

$$\begin{array}{ll} & exec\ s\ (comp\ (Add\ x\ y)) \\ = & \qquad \{ \text{ definition of } comp \ \} \\ & exec\ s\ (comp\ x +\!\!+ comp\ y +\!\!+ [ADD]) \\ = & \qquad \{ \text{ execution distributivity (lemma 1) } \} \\ & exec\ (exec\ s\ (comp\ x))\ (comp\ y +\!\!+ [ADD]) \\ = & \qquad \{ \text{ induction hypothesis } \} \\ & exec\ (eval\ x\ :\ s)\ (comp\ y +\!\!+ [ADD]) \\ = & \qquad \{ \text{ execution distributivity } \} \\ & exec\ (exec\ (eval\ x\ :\ s)\ (comp\ y))\ [ADD] \\ = & \qquad \{ \text{ induction hypothesis } \} \end{array}$$

$$
\begin{array}{ll}
 & exec\ (eval\ y\ :\ eval\ x\ :\ s)\ [ADD] \\
= & \quad \{\ \text{definition of } exec\ \} \\
 & eval\ x + eval\ y\ :\ s \\
= & \quad \{\ \text{definition of } eval\ \} \\
 & eval\ (Add\ x\ y)\ :\ s
\end{array}
$$

□

Note that without first generalising the result, the second induction hypothesis step above would be invalid. The distribution lemma is as follows.

Lemma 1 (execution distributivity).

$$exec\ s\ (xs +\!\!+ ys) \quad = \quad exec\ (exec\ s\ xs)\ ys$$

That is, executing two pieces of code appended together gives the same result as executing the two pieces of code separately in sequence.

Proof. By induction on $xs :: Code$.

When performing an addition in this proof, the stack not containing at least two integers corresponds to a stack underflow error. In this case, the equation to be proved is trivially true, because the result of both sides is undefined ($\bot$), provided that we assume that *exec* is strict in its stack argument ($exec\ \bot\ ops = \bot$.) This extra strictness assumption could be avoided by representing and managing stack underflow explicitly, rather than implicitly using $\bot$. In fact, however, both lemma 1 and its consequent underflow issue can be avoided altogether by further generalising our correctness theorem to also consider additional code.

Theorem 2 (generalised compiler correctness).

$$exec\ s\ (comp\ e +\!\!+ ops) \quad = \quad exec\ (eval\ e\ :\ s)\ ops$$

That is, compiling an expression and then executing the resulting code appended together with arbitrary additional code gives the same result as pushing the value of the expression to give a new stack, which is then used to execute the additional code. Note that with $s = ops = [\,]$, theorem 2 simplifies to $exec\ [\,]\ (comp\ e) = [eval\ e]$, our original statement of compiler correctness.

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

Case: $e = Val\ n$

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

Case: $e = Add\ x\ y$

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

□

In addition to avoiding the problem of stack underflow, the above proof also has the important benefit of being approximately one third of the combined length of our previous two proofs. As is often the case in mathematics, generalising a theorem in the appropriate manner can considerably simplify its proof.

4 Adding exceptions

Now let us extend our language of arithmetic expressions with simple primitives for throwing and catching an exception:

$$\textbf{data}\ Expr \quad = \quad \ldots \mid Throw \mid Catch\ Expr\ Expr$$

Informally, *Throw* abandons the current computation and throws an exception, while $Catch\ x\ h$ behaves as the expression x unless it throws an exception, in which case the catch behaves as the *handler* expression h. To formalise the meaning of these new primitives, we first recall the *Maybe* type:

$$\textbf{data}\ Maybe\ a \quad = \quad Nothing \mid Just\ a$$

That is, a value of type $Maybe\ a$ is either *Nothing*, which we think of as an exceptional value, or has the form $Just\ x$ for some x of type a, which we think of as normal value [15]. Using this type, our denotational semantics for expressions can now be rewritten to take account of exceptions as follows:

$$
\begin{array}{lcl}
eval & :: & Expr \rightarrow Maybe\ Int \\
eval\ (Val\ n) & = & Just\ n \\
eval\ (Add\ x\ y) & = & \textbf{case}\ eval\ x\ \textbf{of} \\
 & & \quad Nothing \rightarrow Nothing \\
 & & \quad Just\ n \rightarrow \textbf{case}\ eval\ y\ \textbf{of} \\
 & & \quad\quad Nothing \rightarrow Nothing \\
 & & \quad\quad Just\ m \rightarrow Just\ (n + m) \\
eval\ (Throw) & = & Nothing \\
eval\ (Catch\ x\ h) & = & \textbf{case}\ eval\ x\ \textbf{of} \\
 & & \quad Nothing \rightarrow eval\ h \\
 & & \quad Just\ n \rightarrow Just\ n
\end{array}
$$

Note that addition propagates an exception thrown in either argument. By exploiting the fact that *Maybe* forms a *monad* [17], the above definition can be expressed more abstractly and concisely using monadic syntax [12]:

$$
\begin{array}{lcl}
\mathit{eval} & :: & \mathit{Expr} \to \mathit{Maybe}\ \mathit{Int} \\
\mathit{eval}\ (\mathit{Val}\ n) & = & \mathit{return}\ n \\
\mathit{eval}\ (\mathit{Add}\ x\ y) & = & \mathbf{do}\ n \leftarrow \mathit{eval}\ x \\
 & & \quad\ m \leftarrow \mathit{eval}\ y \\
 & & \quad\ \mathit{return}\ (n + m) \\
\mathit{eval}\ (\mathit{Throw}) & = & \mathit{mzero} \\
\mathit{eval}\ (\mathit{Catch}\ x\ h) & = & \mathit{eval}\ x\ \text{`}\mathit{mplus}\text{`}\ \mathit{eval}\ h
\end{array}
$$

For the purposes of proofs, however, we use our non-monadic definition for *eval*. To illustrate our new semantics, here are a few simple examples:

$$
\begin{array}{lcll}
\mathit{eval}\ (\mathit{Add}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 3)) & = & \mathit{Just}\ 5 & \text{– no exceptions} \\
\mathit{eval}\ (\mathit{Add}\ \mathit{Throw}\ (\mathit{Val}\ 3)) & = & \mathit{Nothing} & \text{– uncaught exception} \\
\mathit{eval}\ (\mathit{Catch}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 3)) & = & \mathit{Just}\ 2 & \text{– unused handler} \\
\mathit{eval}\ (\mathit{Catch}\ \mathit{Throw}\ (\mathit{Val}\ 3)) & = & \mathit{Just}\ 3 & \text{– caught exception}
\end{array}
$$

Now let us consider how the exception primitives can be compiled. First of all, we introduce three new machine operations:

$$\mathbf{data}\ \mathit{Op} \;=\; \ldots \mid \mathit{THROW} \mid \mathit{MARK}\ \mathit{Code} \mid \mathit{UNMARK}$$

Informally, *THROW* throws an exception, *MARK* pushes a piece of code onto the stack, while *UNMARK* pops such code from the stack. Using these operations, our compiler for expressions can now be extended as follows:

$$
\begin{array}{lcl}
\mathit{comp}\ (\mathit{Throw}) & = & [\,\mathit{THROW}\,] \\
\mathit{comp}\ (\mathit{Catch}\ x\ h) & = & [\,\mathit{MARK}\ (\mathit{comp}\ h)\,] +\!\!+ \mathit{comp}\ x +\!\!+ [\,\mathit{UNMARK}\,]
\end{array}
$$

That is, *Throw* is compiled directly to the corresponding machine operation, while *Catch x h* is compiled by first *marking the stack* with the compiled code for the handler h, then compiling the expression to be evaluated x, and finally *unmarking the stack* by removing the handler. In this way, the mark and unmark operations delimit the scope of the handler h to the expression x, in the sense that the handler is only present on the stack during execution of the expression. Note that the stack is marked with the actual compiled code for the handler, rather than some form of *pointer* to the code as would be used in a real implementation. We will return to this issue later on in the article.

Because the stack can now contain handler code as well as integer values, the type for stacks must itself be rewritten:

$$
\begin{array}{lcl}
\mathbf{type}\ \mathit{Stack} & = & [\mathit{Item}] \\
\mathbf{data}\ \mathit{Item} & = & \mathit{VAL}\ \mathit{Int} \mid \mathit{HAN}\ \mathit{Code}
\end{array}
$$

In turn, our function that executes code is now rewritten as follows:

$$
\begin{array}{lcl}
\mathit{exec} & :: & \mathit{Stack} \to \mathit{Code} \to \mathit{Stack} \\
\mathit{exec}\ s\ [\,] & = & s \\
\mathit{exec}\ s\ (\mathit{PUSH}\ n : \mathit{ops}) & = & \mathit{exec}\ (\mathit{VAL}\ n : s)\ \mathit{ops} \\
\mathit{exec}\ s\ (\mathit{ADD} : \mathit{ops}) & = & \textbf{case}\ s\ \textbf{of} \\
\quad (\mathit{VAL}\ m : \mathit{VAL}\ n : s') \to \mathit{exec}\ (\mathit{VAL}\ (n + m) : s')\ \mathit{ops} & & \\
\mathit{exec}\ s\ (\mathit{THROW} : \mathit{ops}) & = & \mathit{unwind}\ s\ (\mathit{skip}\ \mathit{ops}) \\
\mathit{exec}\ s\ (\mathit{MARK}\ \mathit{ops}' : \mathit{ops}) & = & \mathit{exec}\ (\mathit{HAN}\ \mathit{ops}' : s)\ \mathit{ops} \\
\mathit{exec}\ s\ (\mathit{UNMARK} : \mathit{ops}) & = & \textbf{case}\ s\ \textbf{of} \\
\quad (x : \mathit{HAN}\ \_ : s') \to \mathit{exec}\ (x : s')\ \mathit{ops} & &
\end{array}
$$

That is, push and add are executed as previously, except that we must now take account of the fact that values on the stack are tagged. For execution of a throw, there are a number of issues to consider. First of all, the current computation needs to be abandoned, which means removing any intermediate values that have been pushed onto the stack by this computation, as well as skipping any remaining code for this computation. And secondly, the current handler code needs to be executed, if there is any, followed by any code that remains after the abandoned computation. The function *exec* implements these ideas using an auxiliary function *unwind* that pops items from the stack until a handler is found, at which point the handler is executed followed by the remaining code, which is itself produced using a function *skip* that skips to the next unmark:

$$
\begin{array}{lcl}
\mathit{unwind} & :: & \mathit{Stack} \to \mathit{Code} \to \mathit{Stack} \\
\mathit{unwind}\ [\,]\ \_ & = & [\,] \\
\mathit{unwind}\ (\mathit{VAL}\ \_ : s)\ \mathit{ops} & = & \mathit{unwind}\ s\ \mathit{ops} \\
\mathit{unwind}\ (\mathit{HAN}\ \mathit{ops}' : s)\ \mathit{ops} & = & \mathit{exec}\ s\ (\mathit{ops}' +\!\!+ \mathit{ops}) \\
\mathit{skip} & :: & \mathit{Code} \to \mathit{Code} \\
\mathit{skip}\ [\,] & = & [\,] \\
\mathit{skip}\ (\mathit{UNMARK} : \mathit{ops}) & = & \mathit{ops} \\
\mathit{skip}\ (\mathit{MARK}\ \_ : \mathit{ops}) & = & \mathit{skip}\ (\mathit{skip}\ \mathit{ops}) \\
\mathit{skip}\ (\_ : \mathit{ops}) & = & \mathit{skip}\ \mathit{ops}
\end{array}
$$

Note that *unwind* has the same type as *exec*, and can be viewed as an alternative mode of this function for the case when the virtual machine is in the process of handling an exception. For simplicity, *unwind* returns the empty stack in the case of an uncaught exception. For a language in which the empty stack was a valid result, a separate representation for an uncaught exception would be required. Note also the double recursion when skipping a mark, which reflects the fact that there may be nested mark/unmark pairs in the remaining code.

Returning to the remaining cases in the definition of *exec* above, a mark is executed simply by pushing the given handler code onto the stack, and dually, an unmark by popping this code from the stack. Between executing a mark and its corresponding unmark, however, the code delimited by these two operations will have pushed its result value onto the stack, and hence when the handler code is popped it will actually be the second-top item.

To illustrate our new compiler and virtual machine, their behaviour on the four example expressions from earlier in this section is shown below, in which the symbol \$\$ denotes the result of the last compilation:

$$
\begin{array}{lcl}
comp\ (Add\ (Val\ 2)\ (Val\ 3)) & = & [PUSH\ 2, PUSH\ 3, ADD] \\
exec\ [\,]\ \$\$ & = & [VAL\ 5]
\end{array}
$$

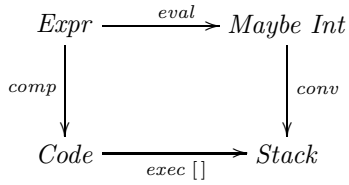
$$
\begin{array}{lcl}
comp\ (Add\ Throw\ (Val\ 3)) & = & [THROW, PUSH\ 3, ADD] \\
exec\ [\,]\ \$\$ & = & [\,]
\end{array}
$$

$$
\begin{array}{lcl}
comp\ (Catch\ (Val\ 2)\ (Val\ 3)) & = & [MARK\ [PUSH\ 3], PUSH\ 2, UNMARK] \\
exec\ [\,]\ \$\$ & = & [VAL\ 2]
\end{array}
$$

$$
\begin{array}{lcl}
comp\ (Catch\ Throw\ (Val\ 3)) & = & [MARK\ [PUSH\ 3], THROW, UNMARK] \\
exec\ [\,]\ \$\$ & = & [VAL\ 3]
\end{array}
$$

5 Compiler correctness

Generalising from the examples in the previous section, the correctness of our new compiler is expressed by the commutativity of the following diagram:



That is, compiling an expression and then executing the resulting code using an empty initial stack gives the same final stack as evaluating the expression and then converting the resulting semantic value into the corresponding stack, using an auxiliary function *conv* that is defined as follows:

$$
\begin{array}{lcl}
conv & :: & Maybe\ Int \to Stack \\
conv\ Nothing & = & [\,] \\
conv\ (Just\ n) & = & [VAL\ n]
\end{array}
$$

As previously, however, in order to prove this result we generalise to an arbitrary initial stack and also consider additional code, and in turn rewrite the function *conv* to take account of these two extra arguments.

Theorem 3 (compiler correctness).

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

where

$$
\begin{array}{lcl}
conv & :: & Stack \to Maybe\ Int \to Code \to Stack \\
conv\ s\ Nothing\ ops & = & unwind\ s\ (skip\ ops) \\
conv\ s\ (Just\ n)\ ops & = & exec\ (VAL\ n : s)\ ops
\end{array}
$$

Note that with $s = ops = [\,]$, this theorem simplifies to our original statement of correctness above. The right-hand side of theorem 3 could also be written as $exec\ s\ (conv\ (eval\ e)\ :\ ops)$ using a simpler version of *conv* with type $Maybe\ Int \to Op$, but the above formulation leads to simpler proofs.

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

Case: $e = Val\ n$

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

Case: $e = Throw$

$$
\begin{array}{ll}
 & exec\ s\ (comp\ Throw +\!\!+ ops) \\
= & \quad \{ \text{ definition of } comp \ \} \\
 & exec\ s\ ([THROW] +\!\!+ ops) \\
= & \quad \{ \text{ definition of } exec \ \} \\
 & unwind\ s\ (skip\ ops) \\
= & \quad \{ \text{ definition of } conv \ \} \\
 & conv\ s\ Nothing\ ops \\
= & \quad \{ \text{ definition of } eval \ \} \\
 & conv\ s\ (eval\ Throw)\ ops
\end{array}
$$

Case: $e = Add\ x\ y$

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

The two possible results from this expression are simplified below.

1:

$$
\begin{array}{ll}
 & \mathit{unwind}\ s\ (\mathit{skip}\ (\mathit{comp}\ y +\!\!+ [\mathit{ADD}] +\!\!+ \mathit{ops})) \\
= & \quad \{ \text{ skipping compiled code (lemma 2) } \} \\
 & \mathit{unwind}\ s\ (\mathit{skip}\ ([\mathit{ADD}] +\!\!+ \mathit{ops})) \\
= & \quad \{ \text{ definition of } \mathit{skip} \ \} \\
 & \mathit{unwind}\ s\ (\mathit{skip}\ \mathit{ops})
\end{array}
$$

2:

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

We now continue the calculation using the two simplified results.

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

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

$$
\begin{array}{ll}
 & \mathit{exec}\ s\ (\mathit{comp}\ (\mathit{Catch}\ x\ h) +\!\!+ \mathit{ops}) \\
= & \quad \{ \text{ definition of } \mathit{comp} \ \} \\
 & \mathit{exec}\ s\ ([\mathit{MARK}\ (\mathit{comp}\ h)] +\!\!+ \mathit{comp}\ x +\!\!+ [\mathit{UNMARK}] +\!\!+ \mathit{ops}) \\
= & \quad \{ \text{ definition of } \mathit{exec} \ \}
\end{array}
$$

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

□

The two distribution over **case** steps in the above proof rely on the fact that *conv* is strict in its semantic value argument ($\mathit{conv}\ s\ \bot\ \mathit{ops} = \bot$), which is indeed the case because *conv* is defined by pattern matching on this argument. The skipping lemma used in the above proof is as follows.

Lemma 2 (skipping compiled code).

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

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 in compiled code are matched by preceding marks.

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

6 Adding jumps

Now let us extend our virtual machine with primitives that allow exceptions to be compiled using explicit jumps, rather than by pushing handler code onto the

stack. First of all, we introduce three new machine operations:

$$\textbf{data}\ Op \;=\; \ldots \mid MARK\ Addr \mid LABEL\ Addr \mid JUMP\ Addr$$

Informally, *MARK* pushes the address of a piece of code onto the stack (replacing our previous mark operator that pushed code itself), *LABEL* declares an address, and *JUMP* transfers control to an address. Addresses themselves are represented simply as integers, and we ensure that each address is declared at most once by generating addresses in sequential order using a function *fresh*:

$$\begin{array}{lcl} \textbf{type}\ Addr & = & Int \\ fresh & :: & Addr \to Addr \\ fresh\ a & = & a+1 \end{array}$$

Our compiler for expressions is now extended to take the next fresh address as an additional argument, and is rewritten in terms of another function *compile* that also returns the next fresh address as an additional result:

$$\begin{array}{lcl} comp & :: & Addr \to Expr \to Code \\ comp\ a\ e & = & fst\ (compile\ a\ e) \end{array}$$

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

Note that integer values, addition, and throw are compiled as previously, except that the next fresh address is threaded through, while *Catch x h* is now compiled to the following sequence, in which *a*: abbreviates *LABEL a*:

$$\begin{array}{l} \quad MARK\ a \\ \quad compiled\ code\ for\ x \\ \quad UNMARK \\ \quad JUMP\ b \\ a{:}\ compiled\ code\ for\ h \\ b{:}\ rest\ of\ the\ code \end{array}$$

That is, *Catch x h* is now compiled by first marking the stack with the address of the compiled code for the handler h, compiling the expression to be evaluated x, then unmarking the stack by removing the address of the handler, and finally jumping over the handler code to the rest of the code.

By exploiting the fact that the type for *compile* can be expressed using a *state monad* [17], the above definition can also be expressed more abstractly and concisely using monadic syntax. As with the function *eval*, however, for the purposes of proofs we use our non-monadic definition for *compile*.

Because the stack can now contain handler addresses rather than handler code, the type for stack items must be rewritten:

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

In turn, our function that executes code requires four modifications:

$$\begin{array}{lcl} \mathit{exec}\ s\ (\mathit{THROW} : \mathit{ops}) & = & \mathit{unwind}\ s\ \mathit{ops} \\ \mathit{exec}\ s\ (\mathit{MARK}\ a : \mathit{ops}) & = & \mathit{exec}\ (\mathit{HAN}\ a : s)\ \mathit{ops} \\ \mathit{exec}\ s\ (\mathit{LABEL}\ \_ : \mathit{ops}) & = & \mathit{exec}\ s\ \mathit{ops} \\ \mathit{exec}\ s\ (\mathit{JUMP}\ a : \mathit{ops}) & = & \mathit{exec}\ s\ (\mathit{jump}\ a\ \mathit{ops}) \end{array}$$

For execution of a throw, the use of explicit jumps means that the function *skip* is no longer required, and there are now only two issues to consider. First of all, the current computation needs to be abandoned, by removing any intermediate values that have been pushed onto the stack by this computation. And secondly, the current handler needs to be executed, if there is any. Implementing these ideas requires modifying one line in the definition of *unwind*:

$$\mathit{unwind}\ (\mathit{HAN}\ a : s)\ \mathit{ops} \quad = \quad \mathit{exec}\ s\ (\mathit{jump}\ a\ \mathit{ops})$$

That is, once the address of a handler is found, the handler is executed using a function *jump* that transfers control to a given address:

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

Note that our language only requires forward jumps. If backward jumps were also possible, a slightly generalised virtual machine would be required.

Returning to the remaining modified cases in the definition of *exec* above, a mark is executed simply by pushing the given address onto the stack, a label is executed by skipping the label, and a jump is executed by transferring control to the given address using the function *jump* defined above.

The behaviour of our new compiler and virtual machine on the four example expressions from earlier in the article is shown below:

$$
\begin{array}{lcl}
comp\ 0\ (Add\ (Val\ 2)\ (Val\ 3)) & = & [PUSH\ 2, PUSH\ 3, ADD] \\
exec\ [\,]\ \$\$ & = & [\,VAL\ 5\,]
\end{array}
$$

$$
\begin{array}{lcl}
comp\ 0\ (Add\ Throw\ (Val\ 3)) & = & [THROW, PUSH\ 3, ADD] \\
exec\ [\,]\ \$\$ & = & [\,]
\end{array}
$$

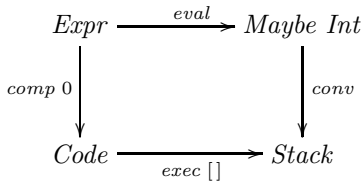
$$
\begin{array}{lcl}
comp\ 0\ (Catch\ (Val\ 2)\ (Val\ 3)) & = & [MARK\ 0, PUSH\ 2, UNMARK, JUMP\ 1, \\
 & & \quad LABEL\ 0, PUSH\ 3, LABEL\ 1] \\
exec\ [\,]\ \$\$ & = & [\,VAL\ 2\,]
\end{array}
$$

$$
\begin{array}{lcl}
comp\ 0\ (Catch\ Throw\ (Val\ 3)) & = & [MARK\ 0, THROW, UNMARK, JUMP\ 1, \\
 & & \quad LABEL\ 0, PUSH\ 3, LABEL\ 1] \\
exec\ [\,]\ \$\$ & = & [\,VAL\ 3\,]
\end{array}
$$

Note that our compiler now once again produces "flat" code, in contrast to our previous version, which produced tree-structured code.

7 Compiler correctness

The correctness of our new compiler is expressed by the same diagram as the previous version, except that new compiler takes the next fresh address as an additional argument, for which we supply zero as the initial value:



For the purposes of proofs we once again generalise this result to an arbitrary initial stack and additional code, and extend the function *conv* accordingly. In turn, we also generalise to an arbitrary initial address that is fresh with respect to the initial stack, using a predicate *isFresh* that decides if a given address is greater than every address that occurs in a stack.

Theorem 4 (compiler correctness).

$$If\ isFresh\ a\ s\ \ then\ \ exec\ s\ (comp\ a\ e +\!\!+ ops)\ \ =\ \ conv\ s\ (eval\ e)\ ops$$

where

$$
\begin{array}{lcl}
isFresh & :: & Addr \to Stack \to Bool \\
isFresh\ \_\ [\,] & = & True \\
isFresh\ a\ (VAL\ \_ : s) & = & isFresh\ a\ s \\
isFresh\ a\ (HAN\ b : s) & = & a > b\ \ \wedge\ \ isFresh\ a\ s \\
conv & :: & Stack \to Maybe\ Int \to Code \to Stack \\
conv\ s\ Nothing\ ops & = & unwind\ s\ ops \\
conv\ s\ (Just\ n)\ ops & = & exec\ (VAL\ n : s)\ ops
\end{array}
$$

Proof. By induction on $e :: Expr$ in a similar manner to theorem 3, except that five lemmas concerning fresh addresses are required:

Lemma 3 (unwinding operators).

$$If\ op = LABEL\ a\ \Rightarrow\ isFresh\ a\ s\ \ then$$

$$unwind\ s\ (op : ops)\ =\ unwind\ s\ ops$$

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 $s :: Stack$.

Lemma 4 (unwinding compiled code).

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

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 3 above.

Lemma 5 (*isFresh* is monotonic).

$$If\ a \leqslant b \wedge isFresh\ a\ s\ \ then\ \ isFresh\ b\ s$$

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 $s :: Stack$.

Lemma 6 (*compile* is non-decreasing).

$$snd\ (compile\ a\ e)\ \geqslant\ a$$

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 $e :: Expr$.

Lemma 7 (jumping compiled code).

$$If\ a < b\ \ then\ \ jump\ a\ (comp\ b\ e +\!\!+ ops)\ =\ jump\ a\ ops$$

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 $e :: Expr$, using lemma 6 above.

8 Further work

We have shown how the compilation of exceptions using stack unwinding can be explained and verified in a simple manner. In this final section we briefly describe a number of possible directions for further work.

- *Mechanical verification.* The correctness of our two compilers for exceptions has also been verified mechanically. In particular, theorem 3 was verified in Lego by McBride [6], and theorem 4 in Isabelle by Nipkow [10]. A novel aspect of the Lego verification that merits further investigation is the use of dependent types to precisely capture the stack demands of the virtual machine operations (e.g. add requires a stack with two integers on the top), which leads to a further simplification of our correctness proof, at the expense of requiring a more powerful type system.

- *Modular compilers.* Inspired by the success of using monads to define the denotational semantics of languages in terms of the semantics of individual features [9], similar techniques are now being applied to build compilers in a modular manner [4]. To date, however, this work has not considered the compilation of exceptions, so there is scope for trying to incorporate the present work into this modular framework.

- *Calculating the compiler.* Rather than first defining the compiler and virtual machine and then proving their correctness with respect to the semantics, another approach would be to try and calculate the definition of these functions starting from the compiler correctness theorem itself [7, 1], with the aim of giving a systematic *discovery* of the idea of compiling exceptions using stack unwinding, as opposed to a post-hoc verification.

- *Generalising the language.* Arithmetic expressions with exceptions served as a suitable language for the purposes of this article, but it is important to explore how our approach can be scaled to more expressive languages, such as a simple functional or imperative language, to languages with more than one kind of exception and user-defined exceptions, and to other notions of exception, such as imprecise [13] and asynchronous [5] exceptions.

- *Compiler optimisations.* The basic compilation method presented in this article can be optimised in a number of ways. For example, we might rearrange the compiled code to avoid the need to jump over handlers in the case of no exceptions being thrown, use a separate stack of handler addresses to make the process of stack unwinding more efficient, or use a separate table of handler scopes to avoid an execution-time cost for installing a handler. It would be interesting to consider how such optimisations can be incorporated into our compiler and its correctness proof.

Acknowledgements

This work was jointly funded by the University of Nottingham and Microsoft Research Ltd in Cambridge. Thanks to Simon Peyton Jones and Simon Marlow at Microsoft for answering many questions about exceptions and their semantics, to Thorsten Altenkirch, Olivier Danvy, Conor McBride, Simon Peyton Jones and the anonymous referees for useful comments and suggestions, and to Ralf Hinze for the `lhs2TeX` system for typesetting Haskell code.

QuickCheck [2] was used extensively in the production of this article, and proved invaluable as an aid to getting the definitions and results correct before proceeding to formal proofs. A number of (often subtle) mistakes in our definitions and results were discovered in this way.

References

1. R. Backhouse. *Program Construction: Calculating Implementations from Specifications*. John Wiley, 2003.
2. K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, Montreal, Canada, Sept. 2000.
3. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.
4. W. Harrison. *Modular Compilers and Their Correctness Proofs*. PhD thesis, University of Illinois, 2001.
5. S. Marlow, S. Peyton Jones, A. Moran, and J. Reppy. Asynchronous Exceptions In Haskell. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001.
6. C. McBride. Personal communication, 2003.
7. E. Meijer. *Calculating Compilers*. PhD thesis, Nijmegen University, 1992.
8. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
9. E. Moggi. An Abstract View of Programming Languages. Technical Report ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, University of Edinburgh, 1990.
10. T. Nipkow. Personal communication, 2004.
11. S. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
12. S. Peyton Jones and J. Launchbury. State in Haskell. University of Glasgow, 1994.
13. S. Peyton Jones, A. Reid, T. Hoare, S. Marlow, and F. Henderson. A Semantics For Imprecise Exceptions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, Georgia, May 1999.
14. D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.
15. M. Spivey. A Functional Theory of Exceptions. *Science of Computer Programming*, 14(1):25–43, 1990.
16. B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley, 1997.
17. P. Wadler. The Essence of Functional Programming. In *Proc. Principles of Programming Languages*, 1992.