## Chapter 1

# Calculating an Exceptional Machine

Graham Hutton and Joel Wright[1]

***Abstract:*** In previous work we showed how to verify a compiler for a small language with exceptions. In this article we show how to *calculate*, as opposed to verify, an abstract machine for this language. The key step is the use of Reynold's *defunctionalization*, an old program transformation technique that has recently been rejuvenated by the work of Danvy et al.

## 1.1 INTRODUCTION

Exceptions are an important feature of modern programming languages, but their compilation has traditionally been viewed as an advanced topic. In previous work we showed how the basic method of compiling exceptions using *stack unwinding* can be explained and verified using elementary functional programming techniques [HW04]. In particular, we developed a compiler for a small language with exceptions, together with a proof of its correctness.

In the formal reasoning community, however, one prefers *constructions* to verifications [Bac03]. That is, rather than first writing the compiler and then separately proving its correctness with respect to a semantics for the language, it would be preferable to try and calculate the compiler [Mei92] directly from the semantics, with the aim of giving a systematic *discovery* of the idea of compiling exceptions using stack unwinding, as opposed to a post-hoc verification.

In this article we take a step towards this goal, by showing how to calculate an abstract machine for evaluating expressions in our language with exceptions. The key step in the calculation is the use of *defunctionalization*, a program transformation technique that eliminates the use of higher-order functions, first introduced by Reynolds in his seminal work on definitional interpreters [Rey72].

[1]School of Computer Science and IT, University of Nottingham, Jubilee Campus, Wollaton Road, Nottingham NG8 1BB, UK. Email: {gmh,jjw}@cs.nott.ac.uk.

Despite being simple and powerful, defunctionalization seems to be somewhat neglected in recent years. For example, it features in few modern courses, textbooks, and research articles on program transformation, and does not seem to be as widely known and used as it should be. Recently, however, defunctionalization has been rejuvenated by the work of Danvy et al, who show how it can be applied in a variety of different areas, including the systematic design of abstract machines for functional languages [DN01, ABDM03b, ADM04].

In this article, we show how Danvy's approach can be used to calculate an abstract machine for our language with exceptions. Moreover, the calculation is *rabbit free*, in the sense that there are no Eureka steps in which one needs to metaphorically pull a rabbit out of a hat — all the required concepts arise naturally from the calculation process itself. The approach is based upon the work of Danvy et al, but the emphasis on calculation and the style of exposition are our own.

The language that we use comprises just integer values, an addition operator, a single exceptional value called throw, and a catch operator for this value [HW04]. This language does not provide features that are necessary for actual programming, but it *does* provide just what we need for expository purposes. 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 two steps, starting with the exception-free part of the language to introduce the basic techniques, to which support for exceptions is then added in the second step. All the programs are written in Haskell [Pey03], and all the calculations are presented using equational reasoning. An extended version of the article that includes the calculations omitted here for reasons of space is available from www.cs.nott.ac.uk/~gmh/machine-extended.pdf.

## 1.2 ABSTRACT MACHINES

An *abstract machine* can be defined as a term rewriting system for executing programs in a particular language, and is given by a set of rewrite rules that make explicit how each step of execution proceeds. Perhaps the best known example is Landin's SECD machine for the lambda calculus [Lan64], which comprises a set of rewrite rules that operate on tuples with four components that give the machine its name, called the stack, environment, control and dump.

For a simpler example, consider a language in which programs comprise a sequence of push and add operations on a stack of integers. In Haskell, such programs, operations and stacks can be represented by the following types:

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

An abstract machine for this language is given by defining two rewrite rules on pairs of programs and stacks from the set $\mathit{Prog} \times \mathit{Stack}$:

$$\begin{array}{lcl} \langle\, PUSH\,n \,:\, ops \,,\, s \,\rangle & \longrightarrow & \langle\, ops \,,\, n\!:\!s \,\rangle \\ \langle\, ADD\!:\!ops \,,\, n\!:\!m\!:\!s \,\rangle & \longrightarrow & \langle\, ops \,,\, n+m \,:\, s \,\rangle \end{array}$$

The first rule states that push places a new integer on top of the stack, while the second states that add replaces the top two integers on the stack by their sum. This machine can be implemented in Haskell by an execution function that repeatedly applies the two rules until this is no longer possible:

$$\begin{array}{lcl} exec & :: & (Prog, Stack) \rightarrow (Prog, Stack) \\ exec\ (PUSH\ n : ops, s) & = & exec\ (ops, n : s) \\ exec\ (ADD : ops, n : m : s) & = & exec\ (ops, n + m : s) \\ exec\ (p, s) & = & (p, s) \end{array}$$

For example, $exec\ ([PUSH\ 1, PUSH\ 2, ADD], [\,])$ gives the result $([\,], [3])$. In the remainder of this article, we will use the term abstract machine for such a functional implementation of an underlying set of rewrite rules.

## 1.3 ARITHMETIC EXPRESSIONS

As in our previous work [HW04], let us begin our development by considering a simple language of expressions comprising integers and addition, whose semantics is given by a function that evaluates an expression to its integer value:

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

We will now calculate an abstract machine for this language, by making a series of three transformations to the semantics.

### Step 1 - Add continuations

At present, the order in which addition evaluates its argument expressions is determined by the language in which the semantics is written, in this case Haskell. The first step in producing an abstract machine is to make the order of evaluation explicit in the semantics itself. A standard technique for achieving this aim is to rewrite the semantics in *continuation-passing* style [Rey72].

A *continuation* is a function that will be applied to the result of an evaluation. For example, in the equation $eval\ (Add\ x\ y) = eval\ x + eval\ y$ from our semantics, when the first recursive call, $eval\ x$, is being evaluated, the remainder of the right-hand side, $+\ eval\ y$, can be viewed as a continuation for this evaluation, in the sense that it is the function that will be applied to the result.

More formally, in the context of our semantics $eval :: Expr \rightarrow Int$, a continuation is a function of type $Int \rightarrow Int$ that will be applied to the result of type $Int$ to give a new result of type $Int$. (This type can be generalised to $Int \rightarrow a$, but we don't need the extra generality for our purposes here.) We capture the notion of such a continuation using the following type definition:

$$\textbf{type}\, \mathit{Cont} \quad = \quad \mathit{Int} \rightarrow \mathit{Int}$$

Our aim now is to define a new semantics, *eval′*, that takes an expression and returns an integer as previously, but also takes a continuation that will be applied to the resulting integer. That is, we seek to define a function

$$\mathit{eval}' \quad :: \quad \mathit{Expr} \rightarrow \mathit{Cont} \rightarrow \mathit{Int}$$

such that:

$$\mathit{eval}'\; e\; c \quad = \quad c\; (\mathit{eval}\; e)$$

At this point in most texts, a recursive definition for *eval′* would normally be written and then either proved to satisfy the above equation, or this be justified by appealing to the correctness of a general continuation-passing transformation. However, we prefer to *calculate* the definition for *eval′* directly from the above equation, by the use of structural induction on *Expr*.

Case: *Val n*

$$\begin{array}{ll}
 & \mathit{eval}'\; (\mathit{Val}\; n)\; c \\
= & \quad \{ \text{ specification of } \mathit{eval}' \;\} \\
 & c\; (\mathit{eval}\; (\mathit{Val}\; n)) \\
= & \quad \{ \text{ definition of } \mathit{eval} \;\} \\
 & c\; n
\end{array}$$

Case: *Add x y*

$$\begin{array}{ll}
 & \mathit{eval}'\; (\mathit{Add}\; x\; y)\; c \\
= & \quad \{ \text{ specification of } \mathit{eval}' \;\} \\
 & c\; (\mathit{eval}\; (\mathit{Add}\; x\; y)) \\
= & \quad \{ \text{ definition of } \mathit{eval} \;\} \\
 & c\; (\mathit{eval}\; x + \mathit{eval}\; y) \\
= & \quad \{ \text{ abstraction over } \mathit{eval}\; x \;\} \\
 & (\lambda n \rightarrow c\; (n + \mathit{eval}\; y))\; (\mathit{eval}\; x) \\
= & \quad \{ \text{ induction hypothesis for } x \;\} \\
 & \mathit{eval}'\; x\; (\lambda n \rightarrow c\; (n + \mathit{eval}\; y)) \\
= & \quad \{ \text{ abstraction over } \mathit{eval}\; y \;\} \\
 & \mathit{eval}'\; x\; (\lambda n \rightarrow (\lambda m \rightarrow c\; (n + m))\; (\mathit{eval}\; y)) \\
= & \quad \{ \text{ induction hypothesis for } y \;\} \\
 & \mathit{eval}'\; x\; (\lambda n \rightarrow \mathit{eval}'\; y\; (\lambda m \rightarrow c\; (n + m)))
\end{array}$$

In conclusion, we have calculated the following recursive definition:

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

That is, for an integer value we simply apply the continuation, while for an addition we evaluate the first argument and call the result $n$, then evaluate the second argument and call the result $m$, and finally apply the continuation to the sum of $n$ and $m$. In this manner, order of evaluation is now explicit in the semantics.

Note that we have ensured that addition evaluates its arguments from left-to-right by first abstracting over $\mathit{eval}\ x$ in the above calculation, and then abstracting over $\mathit{eval}\ y$. It would be perfectly valid to proceed in the other direction, which would result in right-to-left evaluation. Note also that our original semantics can be recovered from our new semantics, by substituting the identity continuation $\lambda n \rightarrow n$ into the equation from which $\mathit{eval}'$ was constructed. That is, our original semantics *eval* can now be redefined as follows:

$$\begin{array}{lcl} \mathit{eval} & :: & \mathit{Expr} \rightarrow \mathit{Int} \\ \mathit{eval}\ e & = & \mathit{eval}'\ e\ (\lambda n \rightarrow n) \end{array}$$

### Step 2 - Defunctionalize

We have now taken a step towards an abstract machine by making evaluation order explicit, but in so doing have also taken a step away from such a machine by making the semantics into a higher-order function. The next step is to regain the first-order nature of the original semantics by eliminating the use of continuations, but retaining the explicit order of evaluation that they introduced.

A standard technique for eliminating the use of functions as arguments is *defunctionalization* [Rey72]. This technique is based upon the observation that we don't usually need the entire function-space of possible argument functions, because only a few forms of such functions are actually used in practice. Hence, we can represent the argument functions that we actually need using a datatype, rather than using the actual functions themselves.

In our new semantics, there are only three forms of continuations that are actually used, namely one to invoke the semantics, and two in the case for evaluating an addition. We begin by separating out these three forms, by giving them names and abstracting over their free variables. That is, we define three combinators for constructing the required forms of continuations:

$$\begin{array}{lcl} \mathit{c1} & :: & \mathit{Cont} \\ \mathit{c1} & = & \lambda n \rightarrow n \\[4pt] \mathit{c2} & :: & \mathit{Expr} \rightarrow \mathit{Cont} \rightarrow \mathit{Cont} \\ \mathit{c2}\ y\ c & = & \lambda n \rightarrow \mathit{eval}'\ y\ (\mathit{c3}\ n\ c) \\[4pt] \mathit{c3} & :: & \mathit{Int} \rightarrow \mathit{Cont} \rightarrow \mathit{Cont} \\ \mathit{c3}\ n\ c & = & \lambda m \rightarrow c\ (n+m) \end{array}$$

At present we have just used anonymous names *c1*, *c2* and *c3* for the combinators, but these will be replaced by more suggestive names later on. Using these combinators, our semantics can now be rewritten as follows:

$$\begin{array}{lcl} \mathit{eval}' & :: & \mathit{Expr} \to \mathit{Cont} \to \mathit{Int} \\ \mathit{eval}'\ (\mathit{Val}\ n)\ c & = & c\ n \\ \mathit{eval}'\ (\mathit{Add}\ x\ y)\ c & = & \mathit{eval}'\ x\ (\mathit{c2}\ y\ c) \\[4pt] \mathit{eval} & :: & \mathit{Expr} \to \mathit{Int} \\ \mathit{eval}\ e & = & \mathit{eval}'\ e\ \mathit{c1} \end{array}$$

The next stage in applying defunctionalization is to define a datatype whose values represent the three combinators that we have isolated:

$$\mathbf{data}\ \mathit{CONT} \quad = \quad \mathit{C1} \mid \mathit{C2}\ \mathit{Expr}\ \mathit{CONT} \mid \mathit{C3}\ \mathit{Int}\ \mathit{CONT}$$

The constructors of this datatype have the same types as the corresponding combinators, except that the new type *CONT* plays the role of *Cont*:

$$\begin{array}{lcl} \mathit{C1} & :: & \mathit{CONT} \\ \mathit{C2} & :: & \mathit{Expr} \to \mathit{CONT} \to \mathit{CONT} \\ \mathit{C3} & :: & \mathit{Int} \to \mathit{CONT} \to \mathit{CONT} \end{array}$$

The fact that values of type *CONT* represent continuations of type *Cont* can be formalised by defining a function that maps from one to the other:

$$\begin{array}{lcl} \mathit{apply} & :: & \mathit{CONT} \to \mathit{Cont} \\ \mathit{apply}\ \mathit{C1} & = & \mathit{c1} \\ \mathit{apply}\ (\mathit{C2}\ y\ c) & = & \mathit{c2}\ y\ (\mathit{apply}\ c) \\ \mathit{apply}\ (\mathit{C3}\ n\ c) & = & \mathit{c3}\ n\ (\mathit{apply}\ c) \end{array}$$

The name of this function derives from the fact that when its type is expanded to $\mathit{apply} :: \mathit{CONT} \to \mathit{Int} \to \mathit{Int}$, it can be viewed as applying a representation of a continuation to an integer to give another integer.

Our aim now is to define a new semantics, $\mathit{eval}''$, that behaves in the same way as our previous semantics, $\mathit{eval}'$, except that it uses values of type *CONT* rather than continuations of type *Cont*. That is, we seek to define a function

$$\mathit{eval}'' \quad :: \quad \mathit{Expr} \to \mathit{CONT} \to \mathit{Int}$$

such that:

$$\mathit{eval}''\ e\ c \quad = \quad \mathit{eval}'\ e\ (\mathit{apply}\ c)$$

As previously, we calculate the definition for the function $\mathit{eval}''$ directly from this equation by the use of structural induction on *Expr*.

Case: *Val n*

$$\begin{array}{ll} & \mathit{eval}''\ (\mathit{Val}\ n)\ c \\ = & \quad \{ \text{ specification of } \mathit{eval}'' \ \} \\ & \mathit{eval}'\ (\mathit{Val}\ n)\ (\mathit{apply}\ c) \\ = & \quad \{ \text{ definition of } \mathit{eval}' \ \} \\ & \mathit{apply}\ c\ n \end{array}$$

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

$$
\begin{array}{ll}
 & \mathit{eval''}\ (\mathit{Add}\ x\ y)\ c \\
= & \quad \{\ \text{specification of } \mathit{eval''}\ \} \\
 & \mathit{eval'}\ (\mathit{Add}\ x\ y)\ (\mathit{apply}\ c) \\
= & \quad \{\ \text{definition of } \mathit{eval'}\ \} \\
 & \mathit{eval'}\ x\ (\mathit{c2}\ y\ (\mathit{apply}\ c)) \\
= & \quad \{\ \text{definition of } \mathit{apply}\ \} \\
 & \mathit{eval'}\ x\ (\mathit{apply}\ (\mathit{C2}\ y\ c)) \\
= & \quad \{\ \text{induction hypothesis for } x\ \} \\
 & \mathit{eval''}\ x\ (\mathit{C2}\ y\ c)
\end{array}
$$

In conclusion, we have calculated the following recursive definition:

$$
\begin{array}{lcl}
\mathit{eval''} & :: & \mathit{Expr} \to \mathit{CONT} \to \mathit{Int} \\
\mathit{eval''}\ (\mathit{Val}\ n)\ c & = & \mathit{apply}\ c\ n \\
\mathit{eval''}\ (\mathit{Add}\ x\ y)\ c & = & \mathit{eval''}\ x\ (\mathit{C2}\ y\ c)
\end{array}
$$

However, the definition for *apply* still refers to the previous semantics $\mathit{eval'}$, via its use of the combinator *c2*. We calculate a new definition for *apply* that refers to our new semantics instead by the use of case analysis on *CONT*.

Case: *C1*

$$
\begin{array}{ll}
 & \mathit{apply}\ \mathit{C1}\ n \\
= & \quad \{\ \text{definition of } \mathit{apply}\ \} \\
 & \mathit{c1}\ n \\
= & \quad \{\ \text{definition of } \mathit{c1}\ \} \\
 & n
\end{array}
$$

Case: $\mathit{C2}\ y\ c$

$$
\begin{array}{ll}
 & \mathit{apply}\ (\mathit{C2}\ y\ c)\ n \\
= & \quad \{\ \text{definition of } \mathit{apply}\ \} \\
 & \mathit{c2}\ y\ (\mathit{apply}\ c)\ n \\
= & \quad \{\ \text{definition of } \mathit{c2}\ \} \\
 & \mathit{eval'}\ y\ (\mathit{c3}\ n\ (\mathit{apply}\ c)) \\
= & \quad \{\ \text{definition of } \mathit{apply}\ \} \\
 & \mathit{eval'}\ y\ (\mathit{apply}\ (\mathit{C3}\ n\ c)) \\
= & \quad \{\ \text{specification of } \mathit{eval''}\ \} \\
 & \mathit{eval''}\ y\ (\mathit{C3}\ n\ c)
\end{array}
$$

Case: $\mathit{C3}\ n\ c$

$$
\begin{array}{ll}
 & \mathit{apply}\ (\mathit{C3}\ n\ c)\ m \\
= & \quad \{\ \text{definition of } \mathit{apply}\ \} \\
 & \mathit{c3}\ n\ (\mathit{apply}\ c)\ m \\
= & \quad \{\ \text{definition of } \mathit{c3}\ \} \\
 & \mathit{apply}\ c\ (n+m)
\end{array}
$$

In conclusion, we have calculated the following new definition:

$$\begin{array}{lcl}
\mathit{apply} & :: & \mathit{CONT} \rightarrow \mathit{Int} \rightarrow \mathit{Int} \\
\mathit{apply}\ \mathit{C1}\ n & = & n \\
\mathit{apply}\ (\mathit{C2}\ y\ c)\ n & = & \mathit{eval}''\ y\ (\mathit{C3}\ n\ c) \\
\mathit{apply}\ (\mathit{C3}\ n\ c)\ m & = & \mathit{apply}\ c\ (n+m)
\end{array}$$

We have now eliminated the use of functions as arguments, and hence made the semantics first-order again. But what about the fact that *eval″* and *apply* are curried functions, and hence return functions as results? As is common practice, we do not view the use of functions as results as being higher-order, as it is not essential and can easily be eliminated if required by uncurrying.

Finally, our original semantics can be recovered from our new semantics by redefining $\mathit{eval}\ e = \mathit{eval}''\ e\ \mathit{C1}$, as can be verified by a simple calculation:

$$\begin{array}{ll}
 & \mathit{eval}\ e \\
= & \quad \{ \text{ previous definition of } \mathit{eval} \ \} \\
 & \mathit{eval}'\ e\ (\lambda n \rightarrow n) \\
= & \quad \{ \text{ definition of } \mathit{c1} \ \} \\
 & \mathit{eval}'\ e\ \mathit{c1} \\
= & \quad \{ \text{ definition of } \mathit{apply} \ \} \\
 & \mathit{eval}'\ e\ (\mathit{apply}\ \mathit{C1}) \\
= & \quad \{ \text{ specification of } \mathit{eval}'' \ \} \\
 & \mathit{eval}''\ e\ \mathit{C1}
\end{array}$$

**Step 3 - Refactor**

At this point, after making two transformations to the original semantics, the reader may be wondering what we have actually produced? In fact, we now have an abstract machine for evaluating expressions, but this only becomes clear after we *refactor* the definitions, in this simple case by just renaming the components. In detail, we rename *CONT* as *Cont*, *C1* as *STOP*, *C2* as *EVAL*, *C3* as *ADD*, *eval″* as *eval*, *apply* as *exec*, and *eval* as *run* to give the following machine:

$$\begin{array}{lcl}
\textbf{data}\ \mathit{Cont} & = & \mathit{STOP} \mid \mathit{EVAL}\ \mathit{Expr}\ \mathit{Cont} \mid \mathit{ADD}\ \mathit{Int}\ \mathit{Cont} \\
\mathit{eval} & :: & \mathit{Expr} \rightarrow \mathit{Cont} \rightarrow \mathit{Int} \\
\mathit{eval}\ (\mathit{Val}\ n)\ c & = & \mathit{exec}\ c\ n \\
\mathit{eval}\ (\mathit{Add}\ x\ y)\ c & = & \mathit{eval}\ x\ (\mathit{EVAL}\ y\ c) \\
\mathit{exec} & :: & \mathit{Cont} \rightarrow \mathit{Int} \rightarrow \mathit{Int} \\
\mathit{exec}\ \mathit{STOP}\ n & = & n \\
\mathit{exec}\ (\mathit{EVAL}\ y\ c)\ n & = & \mathit{eval}\ y\ (\mathit{ADD}\ n\ c) \\
\mathit{exec}\ (\mathit{ADD}\ n\ c)\ m & = & \mathit{exec}\ c\ (n+m) \\
\mathit{run} & :: & \mathit{Expr} \rightarrow \mathit{Int} \\
\mathit{run}\ e & = & \mathit{eval}\ e\ \mathit{STOP}
\end{array}$$

We now explain the four parts of the abstract machine in turn:

- *Cont* is the type of *control stacks* for the machine, containing instructions that determine the behaviour of the machine after evaluating the current expression. The meaning of the three forms of instructions, *STOP*, *EVAL* and *ADD* will be explained shortly. Note that the type of control stacks could itself be refactored as an explicit list of instructions, as follows:

$$\begin{array}{lcl} \textbf{type}\ \mathit{Cont} & = & [\mathit{Inst}] \\ \textbf{data}\ \mathit{Inst} & = & \mathit{ADD\ Int} \mid \mathit{EVAL\ Expr} \end{array}$$

  However, we prefer the original definition above because it only requires the definition of a single type rather than a pair of types.

- *eval* evaluates an expression in the context of a control stack. If the expression is an integer value, it is already fully evaluated, and we simply execute the control stack using this integer as an argument. If the expression is an addition, we evaluate the first argument, $x$, placing the instruction *EVAL* $y$ on top of the current control stack to indicate that the second argument, $y$, should be evaluated once that of the first argument is completed.

- *exec* executes a control stack in the context of an integer argument. If the stack is empty, represented by the instruction *STOP*, we simply return the integer argument as the result of the execution. If the top of the stack is an instruction *EVAL* $y$, we evaluate the expression $y$, placing the instruction *ADD* $n$ on top of the remaining stack to indicate that the current integer argument, $n$, should be added together with the result of evaluating $y$ once this is completed. Finally, if the top of the stack is an instruction *ADD* $m$, evaluation of the two arguments of an addition is now complete, and we execute the remaining control stack in the context of the sum of the two resulting integers.

- *run* evaluates an expression to give an integer, by invoking *eval* with the given expression and the empty control stack as arguments.

The fact that our machine uses two mutually recursive functions, *eval* and *exec*, reflects the fact that it has two states, depending upon whether it is being driven by the structure of the expression (*eval*) or the control stack (*exec*). To illustrate the machine, here is how it evaluates $(2+3)+4$:

$$\begin{array}{cl} & \mathit{run}\ (\mathit{Add}\ (\mathit{Add}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 3))\ (\mathit{Val}\ 4)) \\ = & \mathit{eval}\ (\mathit{Add}\ (\mathit{Add}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 3))\ (\mathit{Val}\ 4))\ \mathit{STOP} \\ = & \mathit{eval}\ (\mathit{Add}\ (\mathit{Val}\ 2)\ (\mathit{Val}\ 3))\ (\mathit{EVAL}\ (\mathit{Val}\ 4)\ \mathit{STOP}) \\ = & \mathit{eval}\ (\mathit{Val}\ 2)\ (\mathit{EVAL}\ (\mathit{Val}\ 3)\ (\mathit{EVAL}\ (\mathit{Val}\ 4)\ \mathit{STOP})) \\ = & \mathit{exec}\ (\mathit{EVAL}\ (\mathit{Val}\ 3)\ (\mathit{EVAL}\ (\mathit{Val}\ 4)\ \mathit{STOP}))\ 2 \\ = & \mathit{eval}\ (\mathit{Val}\ 3)\ (\mathit{ADD}\ 2\ (\mathit{EVAL}\ (\mathit{Val}\ 4)\ \mathit{STOP})) \\ = & \mathit{exec}\ (\mathit{ADD}\ 2\ (\mathit{EVAL}\ (\mathit{Val}\ 4)\ \mathit{STOP}))\ 3 \\ = & \mathit{exec}\ (\mathit{EVAL}\ (\mathit{Val}\ 4)\ \mathit{STOP})\ 5 \\ = & \mathit{eval}\ (\mathit{Val}\ 4)\ (\mathit{ADD}\ 5\ \mathit{STOP}) \\ = & \mathit{exec}\ (\mathit{ADD}\ 5\ \mathit{STOP})\ 4 \\ = & \mathit{exec}\ \mathit{STOP}\ 9 \\ = & 9 \end{array}$$

Note how the function *eval* proceeds downwards to the leftmost integer in the expression, maintaining a trail of the pending right-hand expressions on the control stack. In turn, the function *exec* then proceeds upwards through the trail, transferring control back to *eval* and performing additions as appropriate.

Readers familiar with Huet's *zipper* data structure for navigating around expressions [Hue97] may find it useful to note that our type *Cont* is a zipper data structure for *Expr*, specialised to the purpose of evaluating expressions. Moreover, this specialised zipper arose naturally here by a process of systematic calculation, and did not require any prior knowledge of this structure.

## 1.4 ADDING EXCEPTIONS

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

```
data Expr  =  ... | Throw | Catch Expr Expr
```

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

```
data Maybe a  =  Nothing | 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 a normal value [Spi90]. Using this type, our original semantics for expressions can be rewritten to take account of exceptions as follows:

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

We will now calculate an abstract machine from this extended semantics, by following the same three-step process as previously. That is, we first add continuations, then defunctionalize, and finally refactor the definitions.

**Step 1 - Add continuations**

Because our semantics now returns a result of type *Maybe Int*, the type of continuations that we use must be modified accordingly:

$$\textbf{type}\ \mathit{Cont} \quad = \quad \mathit{Maybe\ Int} \rightarrow \mathit{Maybe\ Int}$$

Our aim now is to define a new semantics

$$\mathit{eval}' \quad :: \quad \mathit{Expr} \rightarrow \mathit{Cont} \rightarrow \mathit{Maybe\ Int}$$

such that:

$$\mathit{eval}'\ e\ c \quad = \quad c\ (\mathit{eval}\ e)$$

That is, the new semantics behaves in the same way as *eval*, except that it applies a continuation to the result. As previously, we can calculate a recursive definition for *eval′* directly from this equation by structural induction on *Expr*:

```
eval′                 ::  Expr → Cont → Maybe Int
eval′ (Val n) c       =   c (Just n)
eval′ (Throw) c       =   c Nothing
eval′ (Add x y) c     =   eval′ x (λx′ → case x′ of
                                Nothing → c Nothing
                                Just n → eval′ y (λy′ → case y′ of
                                     Nothing → c Nothing
                                     Just m → c (Just (n + m))))
eval′ (Catch x y) c   =   eval′ x (λx′ → case x′ of
                                Nothing → eval′ y c
                                Just n → c (Just n))
```

(The above and subsequent omitted calculations are included in the extended version of the article.) In turn, our original semantics can be recovered by invoking our new semantics with the identity continuation. That is, we have

```
eval     ::  Expr → Maybe Int
eval e   =   eval′ e (λx → x)
```

**Step 2 - Defunctionalize**

Our new semantics uses four forms of continuations, namely one to invoke the semantics, two in the case for addition, and one in the case for catch. We define four combinators for constructing these continuations:

$$
\begin{array}{lcl}
c1 & :: & Cont \\
c1 & = & \lambda x \rightarrow x \\[4pt]
c2 & :: & Expr \rightarrow Cont \rightarrow Cont \\
c2\ y\ c & = & \lambda x' \rightarrow \textbf{case}\ x'\ \textbf{of} \\
 & & \quad Nothing \rightarrow c\ Nothing \\
 & & \quad Just\ n \rightarrow eval'\ y\ (c3\ n\ c) \\[4pt]
c3 & :: & Int \rightarrow Cont \rightarrow Cont \\
c3\ n\ c & = & \lambda y' \rightarrow \textbf{case}\ y'\ \textbf{of} \\
 & & \quad Nothing \rightarrow c\ Nothing \\
 & & \quad Just\ m \rightarrow c\ (Just\ (n+m)) \\[4pt]
c4 & :: & Expr \rightarrow Cont \rightarrow Cont \\
c4\ y\ c & = & \lambda x' \rightarrow \textbf{case}\ x'\ \textbf{of} \\
 & & \quad Nothing \rightarrow eval'\ y\ c \\
 & & \quad Just\ n \rightarrow c\ (Just\ n)
\end{array}
$$

Using these combinators, our semantics can now be rewritten as follows:

$$
\begin{array}{lcl}
eval' & :: & Expr \rightarrow Cont \rightarrow Maybe\ Int \\
eval'\ (Val\ n)\ c & = & c\ (Just\ n) \\
eval'\ (Throw)\ c & = & c\ Nothing \\
eval'\ (Add\ x\ y)\ c & = & eval'\ x\ (c2\ y\ c) \\
eval'\ (Catch\ x\ y)\ c & = & eval'\ x\ (c4\ y\ c) \\[4pt]
eval & :: & Expr \rightarrow Maybe\ Int \\
eval\ e & = & eval'\ e\ c1
\end{array}
$$

We now define a datatype to represent the four combinators, together with an application function that formalises the representation:

$$
\begin{array}{lcl}
\textbf{data}\ CONT & = & C1 \mid C2\ Expr\ CONT \mid C3\ Int\ Cont \mid C4\ Expr\ CONT \\[4pt]
apply & :: & CONT \rightarrow Cont \\
apply\ C1 & = & c1 \\
apply\ (C2\ y\ c) & = & c2\ y\ (apply\ c) \\
apply\ (C3\ n\ c) & = & c3\ n\ (apply\ c) \\
apply\ (C4\ y\ c) & = & c4\ y\ (apply\ c)
\end{array}
$$

Our aim now is to define a new semantics

$$eval'' \quad :: \quad Expr \rightarrow CONT \rightarrow Maybe\ Int$$

such that:

$$eval''\ e\ c \quad = \quad eval'\ e\ (apply\ c)$$

That is, the new semantics behaves in the same way as *eval'*, except that it uses representations of continuations rather than actual continuations. We can calculate the definition for $eval''$ by structural induction on *Expr*:

$$\begin{array}{lcl}
eval'' & :: & Expr \to CONT \to Maybe\ Int \\
eval''\ (Val\ n)\ c & = & apply\ c\ (Just\ n) \\
eval''\ (Throw)\ c & = & apply\ c\ Nothing \\
eval''\ (Add\ x\ y)\ c & = & eval''\ x\ (C2\ y\ c) \\
eval''\ (Catch\ x\ y)\ c & = & eval''\ x\ (C4\ y\ c)
\end{array}$$

In turn, we can calculate a new definition for *apply* by case analysis:

$$\begin{array}{lcl}
apply & :: & CONT \to Maybe\ Int \to Maybe\ Int \\
apply\ C1\ x & = & x \\
apply\ (C2\ y\ c)\ Nothing & = & apply\ c\ Nothing \\
apply\ (C2\ y\ c)\ (Just\ n) & = & eval''\ y\ (C3\ n\ c) \\
apply\ (C3\ n\ c)\ Nothing & = & apply\ c\ Nothing \\
apply\ (C3\ n\ c)\ (Just\ m) & = & apply\ c\ (Just\ (n+m)) \\
apply\ (C4\ y\ c)\ Nothing & = & eval''\ y\ c \\
apply\ (C4\ y\ c)\ (Just\ n) & = & apply\ c\ (Just\ n)
\end{array}$$

Our original semantics can be recovered by invoking our new semantics with the representation of the identity continuation:

$$\begin{array}{lcl}
eval & :: & Expr \to Maybe\ Int \\
eval & = & eval''\ e\ C1
\end{array}$$

**Step 3 - Refactor**

We now rename the components in the same way as previously, and rename the new combinator *C4* as *HAND*. This time around, however, refactoring amounts to more than just renaming. In particular, we split the application function

$$apply \quad :: \quad Cont \to Maybe\ Int \to Maybe\ Int$$

into two separate application functions

$$\begin{array}{lcl}
exec & :: & Cont \to Int \to Maybe\ Int \\
unwind & :: & Cont \to Maybe\ Int
\end{array}$$

such that:

$$\begin{array}{lcl}
apply\ c\ (Just\ n) & = & exec\ c\ n \\
apply\ c\ Nothing & = & unwind\ c
\end{array}$$

That is, *exec* deals with normal arguments, and *unwind* with exceptional arguments. We can calculate the definitions for *exec* and *unwind* by structural induction on *Cont*, as a result of which we obtain the following machine:

```
data Cont             =  STOP | EVAL Expr Cont |
                         ADD Int Cont | HAND Expr Cont
eval                  :: Expr → Cont → Maybe Int
eval (Val n) c        =  exec c n
eval (Throw) c        =  unwind c
eval (Add x y) c      =  eval x (EVAL y c)
eval (Catch x y) c    =  eval x (HAND y c)
exec                  :: Cont → Int → Maybe Int
exec STOP n           =  Just n
exec (EVAL y c) n     =  eval y (ADD n c)
exec (ADD n c) m      =  exec c (n + m)
exec (HAND _ c) n     =  exec c n
unwind                :: Cont → Maybe Int
unwind STOP           =  Nothing
unwind (EVAL _ c)     =  unwind c
unwind (ADD _ c)      =  unwind c
unwind (HAND y c)     =  eval y c
run                   :: Expr → Maybe Int
run e                 =  eval e STOP
```

We now explain the three main functions of the abstract machine:

- *eval* evaluates an expression in the context of a control stack. The cases for integer values and addition are as previously. If the expression is a throw, we *unwind the stack* seeking a handler expression. If the expression is a catch, we evaluate its first argument, *x*, and *mark the stack* with the instruction *HAND y* to indicate that its second argument, the handler *y*, should be used if evaluation of its first produces an exceptional value.

- *exec* executes a control stack in the context of an integer argument. The first three cases are as previously, except that if the stack is empty the resulting integer is tagged as a normal result value. If the top of the stack is a handler instruction, there is no need for the associated handler expression because a normal integer result has already been produced, and we *unmark the stack* by popping the handler and then continue executing.

- *unwind* executes the control stack in the context of an exception. If the stack is empty, the exception is uncaught and we simply return the exceptional result value. If the top of the stack is an evaluation or an addition instruction, there is no need for their arguments because a handler is being sought, and we pop them from the stack and then continue unwinding. If the top of the stack is a handler instruction, we catch the exception by evaluating the associated handler expression in the context of the remaining stack.

Note that the idea of marking, unmarking, and unwinding the stack arose directly from the calculations, and did not require any prior knowledge of these concepts. It is also interesting to note that the above machine produced by calculation is both simpler and more efficient that those we had previously designed by hand. In particular, our previous machines did not make a clean separation between the three concepts of evaluating an expression (*eval*), executing the control stack (*exec*) and unwinding the control stack (*unwind*).

To illustrate our machine, here is how it evaluates $1 + (\mathit{catch}\ (2 + \mathit{throw})\ 3)$:

$$
\begin{array}{ll}
 & \mathit{run}\ (\mathit{Add}\ (\mathit{Val}\ 1)\ (\mathit{Catch}\ (\mathit{Add}\ (\mathit{Val}\ 2)\ \mathit{Throw})\ (\mathit{Val}\ 3))) \\
= & \mathit{eval}\ (\mathit{Add}\ (\mathit{Val}\ 1)\ (\mathit{Catch}\ (\mathit{Add}\ (\mathit{Val}\ 2)\ \mathit{Throw})\ (\mathit{Val}\ 3)))\ \mathit{STOP} \\
= & \mathit{eval}\ (\mathit{Val}\ 1)\ (\mathit{EVAL}\ (\mathit{Catch}\ (\mathit{Add}\ (\mathit{Val}\ 2)\ \mathit{Throw})\ (\mathit{Val}\ 3))\ \mathit{STOP}) \\
= & \mathit{exec}\ (\mathit{EVAL}\ (\mathit{Catch}\ (\mathit{Add}\ (\mathit{Val}\ 2)\ \mathit{Throw})\ (\mathit{Val}\ 3))\ \mathit{STOP})\ 1 \\
= & \mathit{eval}\ (\mathit{Catch}\ (\mathit{Add}\ (\mathit{Val}\ 2)\ \mathit{Throw})\ (\mathit{Val}\ 3))\ (\mathit{ADD}\ 1\ \mathit{STOP}) \\
= & \mathit{eval}\ (\mathit{Add}\ (\mathit{Val}\ 2)\ \mathit{Throw})\ (\mathit{HAND}\ (\mathit{Val}\ 3)\ (\mathit{ADD}\ 1\ \mathit{STOP})) \\
= & \mathit{eval}\ (\mathit{Val}\ 2)\ (\mathit{EVAL}\ \mathit{Throw}\ (\mathit{HAND}\ (\mathit{Val}\ 3)\ (\mathit{ADD}\ 1\ \mathit{STOP}))) \\
= & \mathit{exec}\ (\mathit{EVAL}\ \mathit{Throw}\ (\mathit{HAND}\ (\mathit{Val}\ 3)\ (\mathit{ADD}\ 1\ \mathit{STOP})))\ 2 \\
= & \mathit{eval}\ \mathit{Throw}\ (\mathit{ADD}\ 2\ (\mathit{HAND}\ (\mathit{Val}\ 3)\ (\mathit{ADD}\ 1\ \mathit{STOP}))) \\
= & \mathit{unwind}\ (\mathit{ADD}\ 2\ (\mathit{HAND}\ (\mathit{Val}\ 3)\ (\mathit{ADD}\ 1\ \mathit{STOP}))) \\
= & \mathit{unwind}\ (\mathit{HAND}\ (\mathit{Val}\ 3)\ (\mathit{ADD}\ 1\ \mathit{STOP})) \\
= & \mathit{eval}\ (\mathit{Val}\ 3)\ (\mathit{ADD}\ 1\ \mathit{STOP}) \\
= & \mathit{exec}\ (\mathit{ADD}\ 1\ \mathit{STOP})\ 3 \\
= & \mathit{exec}\ \mathit{STOP}\ 4 \\
= & 4
\end{array}
$$

That is, the machine first proceeds normally by transferring control back and forward between the functions *eval* and *exec*, until the exception is encountered, at which point the control stack is unwound to find the handler expression, and the machine then proceeds normally once again.

## 1.5 FURTHER WORK

We have shown how an abstract machine for a small language with exceptions can be calculated in a systematic way from a semantics for the language, using a three-step process of adding continuations, defunctionalizing, and refactoring. Moreover, the calculations themselves are straightforward, only requiring the basic concepts of structural induction and case analysis.

Possible directions for further work include exploring the impact of higher-level algebraic methods (such as monads [Wad92] and folds [Hut99]) on the calculations, mechanically checking the calculations using a theorem proving system (for example, see [Nip04]), factorising the abstract machine into the composition of a compiler and a virtual machine [ABDM03a], and generalising the underlying language (we are particularly interested in the addition of interrupts.)

**Acknowledgements**

Thanks to the referees, Thorsten Altenkirch, Olivier Danvy, Conor McBride, and the WG2.1 meeting in Nottingham for useful comments.

## REFERENCES

[ABDM03a] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. From Interpreter to Compiler and Virtual Machine: a Functional Derivation. Technical Report RS-03-14, BRICS, Aarhus, Denmark, March 2003.

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

[ADM04] Mads Sig Ager, Olivier Danvy, and Jan Midtgaard. A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. Research Series RS-04-28, BRICS, Department of Computer Science, University of Aarhus, December 2004.

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

[DN01] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *Proceedings of the Third ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, Firenze, September 2001.

[Hue97] Gerard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997.

[Hut99] Graham Hutton. A Tutorial on the Universality and Expressiveness of Fold. *Journal of Functional Programming*, 9(4):355–372, July 1999.

[HW04] Graham Hutton and Joel Wright. Compiling Exceptions Correctly. In *Proceedings of the 7th International Conference on Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, Stirling, Scotland, July 2004. Springer.

[Lan64] Peter Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[Mei92] Erik Meijer. *Calculating Compilers*. PhD thesis, Nijmegen, 1992.

[Nip04] Tobias Nipkow. Compiling Exceptions Correctly. In *Archive of Formal Proofs*. 2004. Available from `http://afp.sourceforge.net/`.

[Pey03] Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[Rey72] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM annual conference*, pages 717–740. ACM Press, 1972.

[Spi90] Mike Spivey. A Functional Theory of Exceptions. *Science of Computer Programming*, 14(1):25–43, 1990.

[Wad92] Philip Wadler. The Essence of Functional Programming. In *Proc. Principles of Programming Languages*, 1992.