J. Functional Programming **11** (4): 395–410, July 2001. Printed in the United Kingdom
© 2001 Cambridge University Press

Exceptional syntax

NICK BENTON and ANDREW KENNEDY
Microsoft Research, St. George House, 1 Guildhall Street, Cambridge CB2 3NH, UK
(*e-mail:* {nick,akenn}@microsoft.com)

Abstract

From the points of view of programming pragmatics, rewriting and operational semantics, the syntactic construct used for exception handling in ML-like programming languages, and in much theoretical work on exceptions, has subtly undesirable features. We propose and discuss a more well-behaved construct.

Capsule Review

The propositions-as-types principle has often been cited as an influence on the design of functional programming languages. Often the influence is seen only indirectly. In this short note the authors draw lessons from the proof theory of disjunction to suggest changes to the syntax of exception constructs that not only improve their utility in programming, but also admit simpler expression of common program transformations.

1 Introduction

Many programming languages (from Mesa and PL/I to SML, Java and C#) include *exceptions* to provide a structured, but non-local, way of signalling and recovering from error conditions. Programmers often also use exceptions as convenient, and sometimes more efficient, way of varying control flow in code which has nothing to do with what most people would consider error-handling (for example, the parser combinators in Paulson (1991)).

The basic idea of exceptions is simple and familiar: the evaluation of an expression may, instead of completing normally by returning a value or diverging, terminate abnormally by *raising* a named exception. The evaluation of any expression may be wrapped in an *exception handler*, which provides an alternative expression to be evaluated in the case that the wrapped expression raises a particular exception. The way in which a raised exception unwinds the evaluation stack until the closest matching handler is found is syntactically implicit, so the handler may be dynamically far from the point at which the exception is raised without the intervening calls having explicitly to test for, and propagate, an error value.

There are many differences between exception mechanisms in different programming languages, but for the purposes of this paper we shall take a simplified version of the constructs provided in Standard ML (Milner *et al*., 1997) as paradigmatic of those used in modern expression-based languages. To the usual simply-typed lambda calculus we add a set $\mathbb{E}$ of exception names, a new base type `exn`, and new constructs with the typing rules

$$\frac{}{\Gamma \vdash E : \texttt{exn}}\; E \in \mathbb{E} \qquad \frac{\Gamma \vdash M : \texttt{exn}}{\Gamma \vdash \text{raise}\, M : A}$$

$$\frac{\Gamma \vdash M : A \qquad \{\Gamma \vdash N_i : A\}_{i=1\ldots n}}{\Gamma \vdash M \text{ handle } E_1 \Rightarrow N_1 \mid \cdots \mid E_n \Rightarrow N_n : A}\; \{E_i\}_{i=1\ldots n} \subseteq \mathbb{E}$$

where, in the last rule, the E_i are required to be distinct. We take as basic a form of *handle* in which multiple handlers may cover the evaluation of a single expression, as this is strictly more expressive than the simpler form in which only one named exception may be caught at once. We will sometimes use an abbreviated notation, using H to range over finite sets $\{E_i \Rightarrow N_i\}$ of handlers, and writing $E \in H$ for $\exists N. (E \Rightarrow N) \in H$ and $H(E)$ for the (unique) N such that $(E \Rightarrow N) \in H$ if that exists. We write $\Gamma \vdash H : A$ for $H = \{E_i \Rightarrow N_i\}$ and $\forall i.\, \Gamma \vdash N_i : A$.

One way of explaining the intended behavior of these constructs is to give a big-step operational semantics in which there are two (mutually inductive) forms of judgement: $M \Downarrow V$ means that the closed expression M evaluates to the value V, whereas $M \uparrow E$ means that the expression M raises the exception E. The rules for deriving these judgements comprise the usual evaluation rules for a call by value lambda calculus[1] together with at least the following:

$$\frac{}{E \Downarrow E}\; E \in \mathbb{E} \qquad \frac{M \Downarrow E}{(\text{raise}\, M) \uparrow E}$$

$$\frac{M \Downarrow V}{(M \text{ handle } H) \Downarrow V} \qquad \frac{M \uparrow E}{(M \text{ handle } H) \uparrow E}\; E \notin H$$

$$\frac{M \uparrow E \quad N \Downarrow V}{(M \text{ handle } H) \Downarrow V}\; H(E) = N \qquad \frac{M \uparrow E \quad N \uparrow E'}{(M \text{ handle } H) \uparrow E'}\; H(E) = N$$

$$\frac{M \uparrow E}{(M\, N) \uparrow E} \qquad \frac{M \Downarrow \lambda x. M' \quad N \uparrow E}{(M\, N) \uparrow E}$$

There will be further rules, similar to the last two above, which express the way in which thrown exceptions propagate through whatever other constructs we choose to add to our language.

We became aware of essentially the same shortcoming of the *handle* construct in three different ways whilst working on our Standard ML compiler, MLj (Benton *et al.*, 1998a). First, when coding in SML to implement both the compiler itself and its libraries, we occasionally came across situations in which exception-handling behaviour could only be expressed clumsily. Secondly, when performing rewriting on the compiler intermediate language, we found that some rewrites were inexpressible if

[1] We restrict attention to call by value, as the naïve addition of exceptions to a language with call by name semantics wrecks the equational theory to the extent that the resulting language is essentially unusable. The ingenious addition of *imprecise* exceptions to Haskell does, however, sidestep some of the problems; see Peyton Jones *et al.* (1999) for details.

$$\dfrac{\Gamma \vdash M : A \qquad \Gamma, x : A \vdash P : B \qquad \{\Gamma \vdash N_i : B\}_{i=1\ldots n\geqslant 0}}{\Gamma \vdash \mathrm{try}\ x \Leftarrow M\ \mathrm{in}\ P\ \mathrm{unless}\ E_1 \Rightarrow N_1 \mid \cdots \mid E_n \Rightarrow N_n : B}\ \{E_i\}_{i=1\ldots n} \subseteq \mathbb{E}$$

$$\dfrac{M \Downarrow V \qquad P[V/x] \Downarrow V'}{\mathrm{try}\ x \Leftarrow M\ \mathrm{in}\ P\ \mathrm{unless}\ H \Downarrow V'} \qquad \dfrac{M \uparrow E \qquad N \Downarrow V}{\mathrm{try}\ x \Leftarrow M\ \mathrm{in}\ P\ \mathrm{unless}\ H \Downarrow V}\ H(E) = N$$

$$\dfrac{M \Downarrow V \qquad P[V/x] \uparrow E}{\mathrm{try}\ x \Leftarrow M\ \mathrm{in}\ P\ \mathrm{unless}\ H \uparrow E} \qquad \dfrac{M \uparrow E \qquad N \uparrow E'}{\mathrm{try}\ x \Leftarrow M\ \mathrm{in}\ P\ \mathrm{unless}\ H \uparrow E'}\ H(E) = N$$

$$\dfrac{M \uparrow E}{\mathrm{try}\ x \Leftarrow M\ \mathrm{in}\ P\ \mathrm{unless}\ H \uparrow E}\ E \notin H$$

Fig. 1. Typing rule and natural semantics for *try*.

the intermediate language contained the usual exception handling construct. Thirdly, when formalising the intermediate language in order to prove some theorems about the validity of optimising transformations (Benton & Kennedy, 1999), we found that the alternative syntax we had chosen (for the previous reason) allowed a neat and tractable presentation of the operational semantics in terms of a structurally inductive termination predicate, which would not otherwise have been possible.

2 The new construct

Since the fix for the problems we observed is actually rather simple, and to avoid building unnecessary suspense in the reader, we will reverse the usual order of presentation by giving our solution straight away and then going into the more technical explanations of the problems it solves.

We replace the ML-style *handle* construct with a new one, which builds in a continuation to be applied only in the case that no exception is raised:

$$\boxed{\mathrm{try}\ x \Leftarrow M\ \mathrm{in}\ P\ \mathrm{unless}\ E_1 \Rightarrow N_1 \mid \cdots \mid E_n \Rightarrow N_n}$$

This first evaluates M and, if it returns a value, binds that to x and evaluates P. If M raises the exception E_i, however, N_i is evaluated instead (x is bound in P but *not* in any of the N_i). If M raises an exception distinct from all the E_i, then so does the whole expression.

More formally, figure 2 presents a typing rule[2] for *try* along with its natural semantics rules. Note that we find it convenient to allow empty handlers in this construct and that the type of the expressions N_i in a handler is the same as that of the continuation P, *not* the same as that of the expression M being covered, as is the case with the traditional *handle*.

[2] The typing rule for *try* in our intermediate language is actually a little more complex since it involves computation types (Benton & Kennedy, 1999).

3 So what was wrong with *handle*?

We now describe the problem with the traditional *handle* construct in each of the three contexts in which we observed it. To avoid dragging in too much extraneous material concerning, for example, our compiler intermediate language, we will often gloss over the non-exceptional details of the various languages mentioned: this should not (we hope!) obscure our main point.

3.1 The programming problem

Suppose one has a library of ML functions to open, read and close files, all of which raise the `Io` exception if something goes wrong. The problem is to write a function which runs down a list of filenames, concatenating the results of applying some string-valued function to each file whilst skipping those files which cannot be opened successfully. One's first thought might be that the following will suffice:

```
fun catpartial [] = ""
  | catpartial (n::ns) =
      let val s = readIt (openIn n)
                  handle Io => ""
      in s ^ catpartial ns
      end
```

However, this does not quite do what we want, as the function `readIt` might also raise the `Io` exception: when that happens then we want the exception to be passed up to the caller of `catpartial`, but the above code will handle the exception and move on to the next name in the list irrespective of whether the error occured in `openIn` or `readIt`.

There are, of course, various straightforward ways of programming around this problem. For example, we might use the `option` datatype:

```
fun catpartial1 [] = ""
  | catpartial1 (n::ns) =
      case SOME(openIn n) handle Io => NONE
      of NONE => catpartial1 ns
       | SOME f => readIt f ^ catpartial1 ns
```

Or use abstraction to delay the call to `readIt` so that the handler does not cover it:

```
fun catpartial2 [] = ""
  | catpartial2 (n::ns) =
      (let val f = openIn n
       in fn () => readIt f ^ catpartial2 ns
       end handle Io => fn () => catpartial2 ns
      ) ()
```

Or use another exception:

```
exception OpenFailed
fun catpartial3 [] = ""
  | catpartial3 (n::ns) =
       let val f = openIn n handle Io => raise OpenFailed
       in readIt f ^ catpartial3 ns
       end handle OpenFailed => catpartial3 ns
```

But none of these seems entirely satisfactory as they all introduce a new value (sum, closure or exception) only to eliminate it straight away – it is just there to express some control flow which the `handle` construct is too weak to express directly.

The fix: programming with *try*

The *try-in-unless* syntax nicely solves our programming problem:

```
fun catpartial [] = ""
  | catpartial (n::ns) = try val f = openIn n
                         in readIt f ^ catpartial ns
                         unless Io => catpartial ns
                         end
```

and also generalises both *let* and *handle*:

$$\begin{aligned} \text{let } x \Leftarrow M \text{ in } N &= \text{try } x \Leftarrow M \text{ in } N \text{ unless } \{\} \\ M \text{ handle } H &= \text{try } x \Leftarrow M \text{ in } x \text{ unless } H \end{aligned}$$

3.2 The transformation problem

Like many compilers for functional languages, MLj performs fairly extensive rewriting in order to optimise programs. The design of MLj's intermediate language, MIL, and its rewrites is motivated by a somewhat informal belief in 'taking the proof theory seriously'. One instance of this prejudice is that the compiler transforms programs into a 'cc-normal form', in which all of the *commuting conversions* have been applied.

In natural deduction presentations of logics (and hence, via the Curry-Howard correspondence, in typed lambda calculi), commuting conversions occur when logical rules (usually eliminations) have what Girard (1989) calls a 'parasitic formula', a typical case being that of the sum. The elimination rule for sums is

$$\frac{\Gamma \vdash M : A + B \quad \Gamma, x_1 : A \vdash N_1 : C \quad \Gamma, x_2 : B \vdash N_2 : C}{\Gamma \vdash \text{case } M \text{ of in}_1 x_1.N_1 \mid \text{in}_2 x_2.N_2 : C}$$

in which the formula/type C has no connection with that being eliminated. The presence of such rules introduces undesirable distinctions between proofs and also, for example, causes the subformula property of normal deductions to fail. These problems are addressed by adding commuting conversions to the more familiar β

and η rules. Commuting conversions typically have the general form

$$\dfrac{\dfrac{\begin{matrix}\vdots\\ A\end{matrix}\qquad \begin{matrix}\vdots\\ C\end{matrix}\ \cdots\ \begin{matrix}\vdots\\ C\end{matrix}}{C}\,\mathscr{E}_1 \qquad \cdots}{D}\,\mathscr{E}_2 \quad\rightsquigarrow\quad \dfrac{\begin{matrix}\vdots\\ A\end{matrix}\qquad \dfrac{\begin{matrix}\vdots\\ C\end{matrix}\quad \cdots}{D}\,\mathscr{E}_2\ \cdots\ \dfrac{\begin{matrix}\vdots\\ C\end{matrix}\quad \cdots}{D}\,\mathscr{E}_2}{D}\,\mathscr{E}_1$$

where $\mathscr{E}_1$ is the 'bad' elimination rule for the top-level connective in A, with parasitic formula C (which may occur one or more times as a premiss, according to the connective being eliminated), and $\mathscr{E}_2$ is the elimination rule for the top-level connective in C. For example, if $\mathscr{E}_1$ is $\vee$-elimination and $\mathscr{E}_2$ is $\rightarrow$-elimination, we get the following commuting conversion on terms:[3]

$$\begin{aligned} &(\text{case } M \text{ of } \text{in}_1 x_1.N_1 \mid \text{in}_2 x_2.N_2)\ P \\ \rightsquigarrow\ \ &\text{case } M \text{ of } \text{in}_1 x_1.(N_1\ P) \mid \text{in}_2 x_2.(N_2\ P). \end{aligned}$$

(Here and elsewhere, we adopt the 'variable convention': sufficient α-conversion to avoid unwanted variable capture is assumed. In the above, this implies that neither x_1 nor x_2 is free in P.) Commuting conversions often enable further reductions which would otherwise be blocked, as in

$$\begin{aligned} &(\text{case } M \text{ of } \text{in}_1 x_1.\lambda y.\, y + x_1 \mid \text{in}_2 x_2.\lambda y.\, y)\ 2 \\ \rightsquigarrow\ \ &\text{case } M \text{ of } \text{in}_1 x_1.((\lambda y.\, y + x_1)\ 2) \mid \text{in}_2 x_2.((\lambda y.\, y)\ 2) \\ \rightsquigarrow\ \ &\text{case } M \text{ of } \text{in}_1 x_1.(2 + x_1) \mid \text{in}_2 x_2.2 \end{aligned}$$

and we also find generating code from cc-normal forms considerably more straightforward than for arbitrary terms. Other compilers perform similar rewrites (for example, the *case-of-case* and *let-floating* transformations in Jones & Santos (1998)), though we are unusually dogmatic in recognising them as instances of a common pattern and peforming *all* of them.

Interestingly, cc-normal form for our intermediate language, which is based on Moggi's computational metalanguage (Moggi, 1991), turns out to be almost the same thing as Sabry and Felleisen's *A-normal form* (Sabry & Felleisen, 1993; Flanagan *et al.*, 1993), which was derived from an analysis of CPS-based compilation. A nice discussion of the connection between CPS and Moggi's metalanguage may be found in Hatcliff & Danvy (1994).

For most of the type constructors of our intermediate language, MIL, we have well-behaved introduction and elimination rules for which it is clear how to derive the commuting conversions. For the exception-related constructs, the situation is messier (since part of the point of exceptions is that they are not explicitly visible in source-language types), but it is nevertheless obvious that there are some cc-like rewrites which we would like to perform. For example

$$(M \text{ handle } E \Rightarrow N)\ P$$

[3] Applied naïvely, of course, the duplication of terms in conversions like this one could lead to an unacceptable blowup in code size. MLj avoids this by selective use of a special abstraction construct which compiles to a block of code accessed by jumps.

looks as though it should convert to something like

$$(M\ P)\ \text{handle}\ E \Rightarrow (N\ P)$$

so that if, for example, N is a λ-abstraction, we get to perform a compile-time β-reduction. But this transformation is not generally sound if either P or the application of the value of M to the value of P might raise the exception E. Furthermore, there is no correct transformation which we can use instead. It should be remarked at this point that the limited expressibility of an intermediate language based on a λ-calculus with *handle* is not shared by lower-level target languages. Using Java bytecodes, for example, a code sequence corresponding to a correct version of the above transformation is easily written:

```
L1: Code to evaluate M
L2: Code to evaluate P
    invokevirtual <Method resultype apply(argtype)>
L3: Code for rest of computation

L4: pop  // throw away the actual exception object
    Code to evaluate N
    Code to evaluate P
    invokevirtual <Method resultype apply(argtype)>
    jmp L3

Exception table:
   from   to  target type
   L1     L2   L4    <Class E>
```

and the same is true of target languages in which exception handlers are explicitly pushed onto and popped from a stack.

In fact, because of the separation of computations from values in MIL, we would have to express the first term above as

$$\text{let}\ f \Leftarrow (M\ \text{handle}\ E \Rightarrow N)\ \text{in let}\ v \Leftarrow P\ \text{in}\ (f\ v)$$

but the essential point remains unchanged: there is simply no correct way to write the transformation which we feel we should be able to perform.

Of course, one could simply accept the inexpressibility of such transformations and generate slightly lower quality code. Alternatively, one can observe that the commuting conversions are not in themselves generally optimisations; they are reorganisations of the code which enable more computationally significant β redexes to be exposed. Hence the same optimisations might well be obtained by using non-local rewrites which look for larger patterns in the term. This would, however, significantly increase the complexity of the rewriting function and, we believe, would make it less efficient (despite the fact that the non-local steps would combine the effect of more than one local rewrite).

$$\begin{array}{lr}
\pi_j(\mathrm{try}\ x \Leftarrow M\ \mathrm{in}\ P\ \mathrm{unless}\ \{E_i \Rightarrow N_i\}) & \textit{proj-try} \\
\rightsquigarrow\ \mathrm{try}\ x \Leftarrow M\ \mathrm{in}\ \pi_j(P)\ \mathrm{unless}\ \{E_i \Rightarrow \pi_j(N_i)\} & \\
 & \\
(\mathrm{try}\ x \Leftarrow M\ \mathrm{in}\ P\ \mathrm{unless}\ \{E_i \Rightarrow N_i\})\ Q & \textit{app-try} \\
\rightsquigarrow\ \mathrm{try}\ x \Leftarrow M\ \mathrm{in}\ (P\ Q)\ \mathrm{unless}\ \{E_i \Rightarrow (N_i\ Q)\} & \\
 & \\
\mathrm{case}\ (\mathrm{try}\ x \Leftarrow M\ \mathrm{in}\ P\ \mathrm{unless}\ \{E_i \Rightarrow N_i\})\ \mathrm{of}\ \mathrm{in}_1 y_1.Q_1 \mid \mathrm{in}_2 y_2.Q_2 & \textit{case-try} \\
\rightsquigarrow\ \mathrm{try}\ x \Leftarrow M\ \mathrm{in}\ \mathrm{case}\ P\ \mathrm{of}\ \mathrm{in}_1 y_1.Q_1 \mid \mathrm{in}_2 y_2.Q_2 & \\
\quad\quad \mathrm{unless}\ \{E_i \Rightarrow \mathrm{case}\ N_i\ \mathrm{of}\ \mathrm{in}_1 y_1.Q_1 \mid \mathrm{in}_2 y_2.Q_2\} & \\
 & \\
\mathrm{try}\ x \Leftarrow (\mathrm{case}\ M\ \mathrm{of}\ \mathrm{in}_1 y_1.N_1 \mid \mathrm{in}_2 y_2.N_2)\ \mathrm{in}\ Q\ \mathrm{unless}\ \{E_i \Rightarrow P_i\} & \textit{try-case} \\
\rightsquigarrow\ \mathrm{try}\ z \Leftarrow M\ \mathrm{in} & \\
\quad\quad \mathrm{case}\ z\ \mathrm{of}\ \mathrm{in}_1 y_1.\mathrm{try}\ x \Leftarrow N_1\ \mathrm{in}\ Q\ \mathrm{unless}\ \{E_i \Rightarrow P_i\} \mid & \\
\quad\quad\quad\quad \mathrm{in}_2 y_2.\mathrm{try}\ x \Leftarrow N_2\ \mathrm{in}\ Q\ \mathrm{unless}\ \{E_i \Rightarrow P_i\}\ \mathrm{unless}\ \{E_i \Rightarrow P_i\} & \\
 & \\
\mathrm{try}\ x \Leftarrow (\mathrm{try}\ y \Leftarrow M\ \mathrm{in}\ P\ \mathrm{unless}\ \{E_i \Rightarrow N_i\}_{i\in I})\ \mathrm{in}\ Q\ \mathrm{unless}\ \{E'_j \Rightarrow N'_j\}_{j\in J} & \textit{try-try} \\
\rightsquigarrow\ \mathrm{try}\ y \Leftarrow M\ \mathrm{in} & \\
\quad\quad\quad \mathrm{try}\ x \Leftarrow P\ \mathrm{in}\ Q & \\
\quad\quad\quad \mathrm{unless}\ \{E'_j \Rightarrow N'_j\}_{j\in J} & \\
\quad\quad \mathrm{unless}\ \{E_i \Rightarrow \mathrm{try}\ x \Leftarrow N_i\ \mathrm{in}\ Q & \\
\quad\quad\quad\quad\quad \mathrm{unless}\ \{E'_j \Rightarrow N'_j\}_{j\in J}\}_{i\in I} \cup \{E'_j \Rightarrow N'_j\}_{E'_j \notin \{E_i\}_{i\in I}} &
\end{array}$$

Fig. 2. Conversions.

The fix: rewriting with *try*

The *try-in-unless* syntax comes with unsurprising β-like reductions, similar to those for *handle* and *let*

$$\begin{array}{rcll}
\mathrm{try}\ x \Leftarrow \mathrm{raise}\ E\ \mathrm{in}\ P\ \mathrm{unless}\ H & \rightsquigarrow & N & (N = H(E)) \\
\mathrm{try}\ x \Leftarrow \mathrm{raise}\ E\ \mathrm{in}\ P\ \mathrm{unless}\ H & \rightsquigarrow & \mathrm{raise}\ E & (N \notin H) \\
\mathrm{try}\ x \Leftarrow V\ \mathrm{in}\ P\ \mathrm{unless}\ H & \rightsquigarrow & P[V/x] & (V\ \text{a value})
\end{array}$$

but, unlike *handle*, also has well-behaved commuting conversions, which allow us to express useful compiler transformations. We present in figure 2 a general list of conversions for *try-in-unless* against itself and the eliminations for sums, products and functions. Although these look complex, it should be noted that in a language like MIL (which separates values from computations at both the type and term levels) or that of Pitts (1997) (which has term-level restrictions on the places where non-values may occur), most of these cases either do not occur or only occur in a simplified form. In MIL, for example, only *try-case* and *try-try* are well typed, because projection, application and *case* can only be applied to values, whereas a *try* is always a computation. Furthermore, the restriction that M in the *try-case* rewrite be a value simplifies it to

(try-case'):

$$\begin{array}{l}
\mathrm{try}\ x \Leftarrow (\mathrm{case}\ V\ \mathrm{of}\ \mathrm{in}_1 y_1.N_1 \mid \mathrm{in}_2 y_2.N_2)\ \mathrm{in}\ Q\ \mathrm{unless}\ \{E_i \Rightarrow P_i\} \\
\rightsquigarrow\ \mathrm{case}\ V\ \mathrm{of}\ \mathrm{in}_1 y_1.\mathrm{try}\ x \Leftarrow N_1\ \mathrm{in}\ Q\ \mathrm{unless}\ \{E_i \Rightarrow P_i\} \mid \\
\quad\quad\quad\quad \mathrm{in}_2 y_2.\mathrm{try}\ x \Leftarrow N_2\ \mathrm{in}\ Q\ \mathrm{unless}\ \{E_i \Rightarrow P_i\}
\end{array}$$

The *try-in-unless* construct is the one which we use in MIL, and the MLj compiler actually does perform the *try-try* and *try-case'* rewrites.

As an interesting example of MIL rewriting, showing the *try* construct working with our monadic effect analysis (Benton & Kennedy, 1999), consider the following ML function for summing all the elements of an array:

```
fun sumarray a =
  let fun s(n,sofar) = let val v = Array.sub(a,n)
                       in s(n+1, sofar+v)
                       end handle Subscript => sofar
  in s(0,0)
  end
```

Because the SML source language does not have *try*, the programmer has made the handler cover both the array access and the recursive call to the inner function `s`. But this would prevent a naïve compiler from recognising that call as tail-recursive. In MLj, the intermediate code for `s` looks like (in MLish, rather than MIL, syntax):

```
fun s(n,sofar) =
      try val x = try val v = Array.sub(a,n)
                  in s(n+1, sofar+v)
                  unless {}
                  end
      in x
      unless Subscript => sofar
      end
```

The *try-try* rewrite turns this into

```
fun s(n,sofar) = try val v = Array.sub(a,n)
                 in try val x = s(n+1, sofar+v)
                    in x
                    unless Subscript => sofar
                    end
                 unless Subscript => sofar
                 end
```

(The two identical handlers are actually abstracted as a shared local block.) The effect analysis detects that the recursive call to `s` cannot, in fact, ever throw the `Subscript` exception, so the function is rewritten again to

```
fun s(n,sofar) = try val v = Array.sub(a,n)
                 in s(n+1, sofar+v)
                 unless Subscript => sofar
                 end
```

which *is* tail recursive, and so gets compiled as a loop in the final code for `sumarray`.

3.3 The semantics problem

There are several different styles in which one can specify the operational semantics of ML-like languages. We have already seen (in section 1) a big-step, natural semantics presentation, but this is not always the most convenient formulation with which to work when proving results about observational equivalences. A popular alternative is to use a small-step semantics presented using Felleisen's notion of *evaluation context* (Felleisen & Hieb, 1992). In this style, one first defines axioms for the primitive transitions $R \to M$, saying that redex R reduces to term M, and then gives an inductive definition of evaluation contexts as terms $E[\cdot]$ containing a single 'hole' in the place where the next reduction will take place. A simple lemma that every non-value is uniquely of the form $E[R]$ then allows the one-step transition relation to be defined as $E[R] \to E[M]$ for every evaluation context $E[\cdot]$ and primitive transition $R \to M$ (and the evaluation relation to be defined in terms of the reflexive transitive closure of the transition relation). Wright & Felleisen (1994) give an evaluation context semantics for ML with exceptions which uses a second kind of context for propagating exceptions.

Pitts (1997) has argued that for reasoning about contextual equivalences it is convenient to reify the notion of evaluation context and give a small-step operational semantics in which a configuration is a pair of a term and an explicit context (continuation). The advantages of this approach include the fact that the right-hand sides of transitions are all defined by structural induction over the left-hand side and that there is a Galois connection between relations on terms and relations on contexts which has proved useful in reasoning about, for example, equivalence of polymorphic functions. This style of presentation is also particularly natural if the language includes first-class continuations, in the style of Scheme or SML/NJ (see Harper *et al.* (1993), for example).

Pitts formalises contexts by introducing new syntactic categories for defining *continuation stacks*: a configuration looks like

$$\langle (x_1).N_1 \circ \cdots \circ (x_n).N_n\,,\ M\rangle$$

where M is the term being evaluated (in a λ-calculus with a strict `let` construct and a restriction that only values and variables may occur in eliminations) and $(x_1).N_1 \circ \cdots \circ (x_n).N_n$ is a sequence of (closed) abstractions representing the context in which the evaluation takes place. The rules defining the transition relation include

$$\begin{array}{rcl}
\langle K \circ (x).N\,,\ V\rangle & \to & \langle K\,,\ N[V/x]\rangle \\
\langle K\,,\ \text{let}\, x \Leftarrow M\, \text{in}\, N\rangle & \to & \langle K \circ (x).N\,,\ M\rangle \\
\langle K\,,\ (\lambda x.M)\, V\rangle & \to & \langle K\,,\ M[V/x]\rangle
\end{array}$$

which, it should be apparent, amounts to defining a kind of abstract machine.[4]

[4] Actually, since Pitts is interested in which configurations lead to termination, for reasoning about contextual equivalence, the one-step transitions are implicit in inference rules defining the termination predicate $\searrow$ directly, such as

$$\frac{\langle K\,,\ N[V/x]\rangle \searrow}{\langle K \circ (x).N\,,\ V\rangle \searrow}.$$

This style of semantics has been applied by Pitts (2000), Pitts & Stark (1998) and Bierman (1998), and the relational operators it induces are further discussed in Abadi (2000). Coincidentally, the current implementation of MLj uses essentially the same representation internally for efficient rewriting of terms in context.

Pitts gives the relationship between the stack-based semantics and a natural semantics using the following lemma: For all appropriately-typed, closed K, M and V

$$\langle K\,,\ M\rangle \rightarrow^* \langle \cdot\,,\ V\rangle \quad\Longleftrightarrow\quad K@M \Downarrow V$$

where $\cdot$ is the empty continuation stack and the 'unwinding' operator @ is defined by

$$\begin{array}{rcl} \cdot\, @M & = & M \\ (K \circ (x).N)@M & = & K@(\text{let } x \Leftarrow M \text{ in } N). \end{array}$$

Note how the place where the action (reduction) happens is at the root of the syntax tree of a stack configuration but buried deep in that of its unwinding, as

$$\begin{array}{rl} & ((x_1).N_1 \circ \cdots \circ (x_n).N_n) \ @\ M \\ = & \begin{array}{l} \text{let } x_1 \Leftarrow (\\ \quad \text{let } x_2 \Leftarrow \\ \qquad (\ldots(\text{let } x_n \Leftarrow M \text{ in } N)\ldots) \\ \quad \text{in } N_2) \\ \text{in } N_1 \end{array} \end{array}$$

It is straightfoward to extend Pitts's semantics to a language with exceptions: one simply allows (closed) handlers H (which we previously introduced as an abbreviation for part of the syntax of the *handle* construct and are now making slightly more first-class) to appear as a new kind of element in continuation stacks, with the new transitions

$$\begin{array}{rcll} \langle K \circ H\,,\ V\rangle & \rightarrow & \langle K\,,\ V\rangle & \\ \langle K \circ H\,,\ \text{raise } E\rangle & \rightarrow & \langle K\,,\ N\rangle & \text{if } H(E) = N \\ \langle K \circ H\,,\ \text{raise } E\rangle & \rightarrow & \langle K\,,\ \text{raise } E\rangle & \text{if } E \notin H \\ \langle K \circ (x).N\,,\ \text{raise } E\rangle & \rightarrow & \langle K\,,\ \text{raise } E\rangle & \\ \langle K\,,\ M \text{ handle } H\rangle & \rightarrow & \langle K \circ H\,,\ M\rangle & \end{array}$$

The connection with the natural semantics extends to

$$\langle K\,,\ M\rangle \rightarrow^* \langle \cdot\,,\ \text{raise } E\rangle \quad\Longleftrightarrow\quad K@M \uparrow E$$

where the definition of @ is extended by

$$(K \circ H)@M \quad = \quad K@(M \text{ handle } H)$$

and this is the formulation we initially used when working on the equational theory of MIL. However, there is a certain amount of clutter involved in using stacks (extra syntax, type rules, etc.), and we noticed that if one's syntax is sufficiently well-behaved then it is possible to obtain an equally tractable presentation of the transition relation just using terms of the original language. For Pitts's language *without* exceptions, the idea is to axiomatise directly transitions between terms of the form $\text{let}\ x \Leftarrow M\ \text{in}\ N$ by using commuting conversion transitions to 'bubble up' the next redex in M until it is at the top (and its surrounding context within M has been pushed into N). For example:

$$\begin{array}{rcl} \text{let}\ x \Leftarrow V\ \text{in}\ N & \to & \text{let}\ y \Leftarrow N[V/x]\ \text{in}\ y \quad (N \neq x) \\ \text{let}\ x \Leftarrow (\text{let}\ y \Leftarrow M\ \text{in}\ N)\ \text{in}\ P & \to & \text{let}\ y \Leftarrow M\ \text{in let}\ x \Leftarrow N\ \text{in}\ P \\ \text{let}\ x \Leftarrow (\lambda y.M)\ V\ \text{in}\ N & \to & \text{let}\ x \Leftarrow M[V/y]\ \text{in}\ N \end{array}$$

Using this style of presentation, the relationship between the big-step and small-step semantics becomes

$$(\text{let}\ x \Leftarrow M\ \text{in}\ x) \to^* (\text{let}\ x \Leftarrow V\ \text{in}\ x) \quad \Longleftrightarrow \quad M \Downarrow V.$$

Intuitively, the stack-free transition relation is defined directly on a variant of Pitts's 'unwound' terms, in which the *let*s associate the other way around from the original definition:

$$(K \circ (x).N)@M \quad = \quad \text{let}\ x \Leftarrow M\ \text{in}\ (K@N).$$

The equivalence of the two definitions of @ depends on the validity of the associativity of *let* (which, as discussed in Benton *et al.* (1998b), *is* a commuting conversion in the logic corresponding to Moggi's computational metalanguage).

However, if we add exceptions and the *handle* construct, the definition of the stack-free transition relation fails to extend. Once again, the problem is the lack of commuting conversions which would allow an exception handler to be pushed into a surrounding context so that the evaluation of the expression convered by the handler 'bubbles' to the top. More concretely, consider the following putative transition:

$$\text{let}\ x \Leftarrow (M\ \text{handle}\ E \Rightarrow N)\ \text{in}\ P \quad \to \quad ?$$

We would like to put something on the right-hand side in which the evaluation of M is at the top of the syntax tree, but there is no rewrite to anything of the form $\text{let}\, x \Leftarrow M\, \text{in} \ldots$. Nor can we extend the collection of top-level forms to include *handle* as well as *let* constructs: there's no rewrite to something of the form $M\,\text{handle}\,E \Rightarrow \ldots$ either.

The fix: operational semantics with *try*

If our language includes *try-in-unless*, then there is no difficulty in giving a stack-free presentation of a structurally inductive transition semantics. Figure 3 presents transitions between terms of the form $\text{try}\ x \Leftarrow M\ \text{in}\ P\ \text{unless}\ H$ (recall that *try-in-unless* generalises *let*). The syntax (H catch H' in $x.Q$) is an abbreviation for the covering of one handler by the other handler and continuation used in the *try-try* conversion (as in figure 2).

The connection between the transition semantics and the big-step semantics is

$$\begin{array}{rcll}
\text{try } x \Leftarrow V \text{ in } P \text{ unless } H & \rightarrow & \text{try } y \Leftarrow P[V/x] \text{ in } y \text{ unless } \{\} & (P \neq x) \\
\text{try } x \Leftarrow \text{raise } E \text{ in } P \text{ unless } H & \rightarrow & \text{try } y \Leftarrow H(E) \text{ in } y \text{ unless } \{\} & \\
\text{try } x \Leftarrow (\lambda y.M)\ V \text{ in } P \text{ unless } H & \rightarrow & \text{try } x \Leftarrow M[V/y] \text{ in } P \text{ unless } H & \\
\multicolumn{4}{l}{\text{try } x \Leftarrow (\text{try } y \Leftarrow M \text{ in } P \text{ unless } H) \text{ in } Q \text{ unless } H'} \\
\multicolumn{4}{r}{\rightarrow\ \text{try } y \Leftarrow M \text{ in } (\text{try } x \Leftarrow P \text{ in } Q \text{ unless } H') \text{ unless } (H \text{ catch } H' \text{ in } x.Q)}
\end{array}$$

$$\begin{array}{rcl}
\{E_i \Rightarrow N_i\} \text{ catch } \{E'_j \Rightarrow N'_j\} \text{ in } x.Q & \stackrel{\text{def}}{=} & \{E_i \Rightarrow \text{try } x \Leftarrow N_i \text{ in } Q \text{ unless } \{E'_j \Rightarrow N'_j\}\} \\
& & \cup\{E'_j \Rightarrow N'_j \mid \nexists i.E_i = E'_j\}
\end{array}$$

Fig. 3. Transition semantics.

then expressed by

$$\begin{array}{rcl}
M \Downarrow V & \Longleftrightarrow & \text{try } x \Leftarrow M \text{ in } x \text{ unless } \{\} \\
& & \rightarrow^{*}\ \text{try } x \Leftarrow V \text{ in } x \text{ unless } H \\
M \uparrow E & \Longleftrightarrow & \text{try } x \Leftarrow M \text{ in } x \text{ unless } \{\} \\
& & \rightarrow^{*}\ \text{try } x \Leftarrow \text{raise } E \text{ in } P \text{ unless } H\ \ (E \notin H)
\end{array}$$

This formulation of the transition semantics is the one which we have used when reasoning about observational congruence for MIL in order to validate effect-based transformations (Benton & Kennedy, 1999).[5]

4 Remarks on concrete syntax

Using *try-in-unless* in theoretical work or in a compiler intermediate language is straightforward. But adding the construct to a programming language requires a human-friendly concrete syntax to be chosen and, annoyingly, there does not seem to be an obviously ‘right’ choice here. The main problem is choosing whether the handlers or the continuation expression should come first, i.e. between

```
try x = M                           try x = M
in N                                unless E=>P
unless E=>P          and            in N
end                                 end
```

Neither of these is entirely satisfactory. In the first case the fact that the handler only covers `M` and not `N` is obscured; this is particularly bad if `N` is large. In the second, that `x` is bound in `N` but not in `P` is certainly not what one would expect.

Our own preference for SML is firstly to retain the *handle* construct in the source syntax, since it is simpler and suffices for most situations, and then either to add the first alternative above or (more radically) to allow both of them. Since *try-in-unless* generalises *let*, it also seems sensible to do without the `try` keyword and just allow `unless` to be an optional part of `let`-expressions. We have tweaked MLj so that it will accept syntax like the following:

[5] Though, embarassingly, the HOOTS paper gives an incorrect shorthand for one handler covering another in the operational semantics.

```
fun f ((n1,n2)::rest) =
  let val s1 = openIn n1
      val s2 = openIn n2
  in combine(s1,s2)
  unless Io => f rest
  end
```

Note that SML allows multiple sequential declarations in a single `let` expression. None of the variables in the left-hand sides are bound in the handler, which is evaluated if *any* of the right-hand sides raise a matching exception.

5 Remarks on try-finally

Some imperative languages have a *try* statement which allows execution of a command to be covered not only by a set of handlers, but also by an optional *finally* clause. This specifies a command which is to be executed once control has left the covered command (and any of the handlers), irrespective of whether the exit was normal or by raising an exception. The *try-(catch)-finally* construct is typically used for imperative 'cleanup' code which needs to be executed whether or not an error occurs, the usual example being closing open files.

One way to extend ML with a similar feature would be by new syntax `M finally N`, the typing rule for which requires `N` to be of type `unit`, and whose behaviour may be specified by the translation:

```
                        let val x = M handle e => (N ; raise e)
M finally N    =        in N ; x
                        end
```

where `x` is not free in `N`. In the absence of special syntax, one has to program directly in terms of the translation, which involves the unpleasant duplication of `N`. This duplication may be minimised by abstracting (thunking) `N`, and by doing the same to `M` one can write a higher-order function `finally` of type `(unit -> 'a) * (unit -> unit) -> 'a`. One might (and the referees did) wonder whether our alternative syntax for exception handling allows the behaviour of `finally` to be obtained in a more convenient first-order way. Unfortunately, the answer is no. The translation of `M finally N` in our syntax is

```
let val x = M
in N ; x
unless e => N ; raise e
end
```

which, although it arguably makes the control flow a little clearer, is not really any better than the translation in terms of `handle` – it still duplicates `N`.[6]

[6] This duplication differs from those introduced by commuting conversions in that the continuations of the two copies of `N` are different, so the sharing cannot be captured by MIL's special 'local block' abstractions. This problem with `finally` is the reason that the JVM includes a very restricted and *ad hoc* form of subroutine.

6 Conclusions

Although the point is undeniably a small one, we hope we have convinced the reader that the *try-in-unless* syntax for exception handling really is more well-behaved than the traditional *handle* construct. It is also probably worth noting that if one translates a language with exceptions into one without them, by using sums to encode the exceptions monad (if the set of exceptions is infinite then this requires either infinite syntax or defaults in pattern matching), then the derived elimination construct for computations is essentially *try-in-unless*. (The difference is that *all* exceptions are always caught, though all but a finite number are then rethrown.)

As far as we know, MIL is the first language to use *try-in-unless*, though we are not the only people to have spotted that it might be a useful programming construct – whilst we were writing this Judicael Courant (1999) suggested the essentially same thing on the CAML mailing list.

From a methodological perspective, we feel that this is another small piece of evidence for the benefits of taking insights from proof-theory seriously when doing language design. Although the solution seems obvious in retrospect, and other people might have reached it by a different route, we personally would not have recognised that there was an identifiable problem in the first place (as opposed to some ugly bits of code and slightly messy proofs) had we not been thinking in terms of proof-theoretic normal forms.

References

Abadi, M. (2000) ⊤⊤-closed relations and admissibility. *Math. Struc. in Comput. Sci.* **10**(3), 313–320.

Benton, N. and Kennedy, A. (1999) Monads, effects and transformations. *Third International Workshop on Higher Order Operational Techniques in Semantics (HOOTS)*, Paris. *Electronic Notes in Theoretical Computer Science 26*. Elsevier.

Benton, N., Kennedy, A. and Russell, G. (1998) Compiling Standard ML to Java bytecodes. *3rd ACM SIGPLAN Conference on Functional Programming*.

Benton, P. N., Bierman, G. M. and de Paiva, V. C. V. (1998b) Computational types from a logical perspective. *J. Functional Programming*, **8**(2), 177–193. (Preliminary version appeared as Technical Report 365, University of Cambridge Computer Laboratory, May 1995.)

Bierman, G. M. (1998) A computational interpretation of the $\lambda\mu$-calculus. In: Brim, L., Gruska, J. and Zlatuška, J. (eds.), *Proceedings of the Symposium on Mathematical Foundations of Computer Science: Lecture Notes in Computer Science 1450*, pp. 336–345. Springer-Verlag.

Courant, J. (1999) *A common use of try...with*. Message to the CAML mailing list 16 December 1999 `http://pauillac.inria.fr/caml/caml-list/2121.html`.

Felleisen, M. and Hieb, R. (1992) The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.* **103**, 235–271.

Flanagan, C., Sabry, A., Duba, B. F. and Felleisen, M. (1993) The essence of compiling with continuations. *Proceedings of the 1993 Conference on Programming Language Design and Implementation*. ACM.

Girard, J.-Y., Lafont, Y. and Taylor, P. (1989) *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science, no. 7. Cambridge University Press.

Harper, R., Duba, B. and MacQueen, D. (1993) Typing first-class continuations in ML. *J. Functional Programming*, **4**(3), 465–484.

Hatcliff, J. and Danvy, O. (1994) A generic account of continuation-passing styles. *Proceedings of the 21st Annual Symposium on Principles of Programming Languages*. ACM.

Milner, R., Tofte, M., Harper, R. and MacQueen, D. (1997) *The Definition of Standard ML (revised)*. MIT Press.

Moggi, E. (1991) Notions of computation and monads. *Infor. & Computation*, **93**, 55–92.

Paulson, L. C. (1991) *ML for the Working Programmer*. Cambridge University Press.

Peyton Jones, S. and Santos, A. (1998) A transformation-based optimiser for Haskell. *Science of Computer Programming*, **32**(1-3), 3–47.

Peyton Jones, S., Reid, A., Hoare, T., Marlow, S. and Henderson, F. (1999) A semantics for imprecise exceptions. *Proceedings of the 1999 Conference on Programming Language Design and Implementation.*

Pitts, A. M. (1997) *Operational semantics for program equivalence*. Invited talk at MFPS XIII, CMU, Pittsburgh. (See `http://www.cl.cam.ac.uk/users/amp12/talks/`.)

Pitts, A. M. (2000) Parametric polymorphism and operational equivalence. *Math. Struc. in Comput. Sci.* **10**, 1–39. (A preliminary version appeared in *Proceedings, Second Workshop on Higher Order Operational Techniques in Semantics (HOOTS II)*, Stanford CA, December 1997.)

Pitts, A. M. and Stark, I. D. B. (1998) Operational reasoning for functions with local state. In: Gordon, A. D. and Pitts, A. M. (eds.), *Higher Order Operational Techniques in Semantics*, pp. 227–273. Publications of the Newton Institute. Cambridge University Press.

Sabry, A. and Felleisen, M. (1993) Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, **6**(3/4), 289–360.

Wright, A. K. and Felleisen, M. (1994) A syntactic approach to type soundness. *Infor. & Computation*, **115**(1), 38–94.