Assessing the Overhead of ML Exceptions by Selective CPS Transformation

Jung-taek Kim, Kwangkeun Yi
Programming Languages Lab. *
Department of Computer Science
KAIST †

Olivier Danvy
BRICS ‡
Department of Computer Science
University of Aarhus §

Abstract

ML's exception handling makes it possible to describe exceptional execution flows conveniently, but it also forms a performance bottleneck. Our goal is to reduce this overhead by source-level transformation.

To this end, we transform source programs into continuation-passing style (CPS), replacing handle and raise expressions by continuation-catching and throwing expressions, respectively. CPS-transforming every expression, however, introduces a new cost. We therefore use an exception analysis to transform expressions selectively: if an expression is statically determined to involve exceptions then it is CPS-transformed; otherwise, it is left in direct style.

In this article, we formalize this selective CPS transformation, prove its correctness, and present early experimental data indicating its effect on ML programs.

1 Introduction

Programming with exceptions is an expensive affair in ML, since it involves installing and uninstalling exceptions handlers at run time. This dynamic nesting of handlers can be a performance bottleneck if it is frequent, e.g., when the handler exists in the body of a recursive function.

1.1 An example

For example, let us consider the problem of substituting a closed expression for a variable in another expression. The expression is either a variable, a lambda abstraction, or an application (see Figure 1). In a naïve implementation of substitution, the source expression is entirely copied. In an economical implementation, the subexpressions unaffected by the substitution (i.e., those where the variable to substitute does not occur free) are shared between the source expression and the resulting expression. This economical implementation is an instance of what Gérard Huet, in the mid-80's, called "sharing transducers."

```
structure Exp
= struct
    datatype exp = VAR of string
                 | LAM of string * exp
                 | APP of exp * exp
  end
```

Figure 1: A sample definition of expressions

We measured the performances of two sharing transducers: one using exceptions (Figure 2) and the other using continuations (Figure 3). Figure 4 displays two extreme cases: (a) one where the source expression yields much exception raising and handling; and (b) one where the source expression yields no exception raising and handling at all. In Case (a), the continuation-based version is about twice as fast as the exception-based version.[1] In Case (b), the continuation-based version runs in about the same time as the exception-based version.

We also implemented a third sharing transducer, using a disjoint sum instead of an exception or two continuations. In performance, it matches the continuation-based one, thus showing that at least for sharing transducers, continuations form a viable alternative solution to disjoint sums.

Figure 4 shows that there is considerable room for improvement, particularly when the input program uses exceptions frequently. Our goal is to achieve this optimization by source-level transformation(s). Since exceptions affect the control flow of programs, we choose to translate the `raise` and `handle` constructs into continuation-based idioms.

1.2 This work

We remove `raise` and `handle` from ML programs by transforming them into continuation-passing style (CPS). Specifically, we pass *two* continuations: one for the normal course of execution, and one for exceptional situations.

* `http://pllab.kaist.ac.kr`
This work is supported in part by Korea Science and Engineering Foundation grant KOSEF 961-0100-001-2 and by Korea Ministry of Information and Communication grant 96151-IT2-12.
† Department of Computer Science (`http://cs.kaist.ac.kr`), Korea Advanced Institute of Science & Technology, Taejon 305-701, Korea. E-mail: {`judaigi,kwang`}`@cs.kaist.ac.kr`

‡ Basic Research in Computer Science (`http://www.brics.dk`), Centre of the Danish National Research Foundation.

§ Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark. E-mail: `danvy@brics.dk`

[1] The peaks in the graph reflect the situations where the programs hit the memory thresholds that make the garbage collector increase its heap size (which causes some overhead).

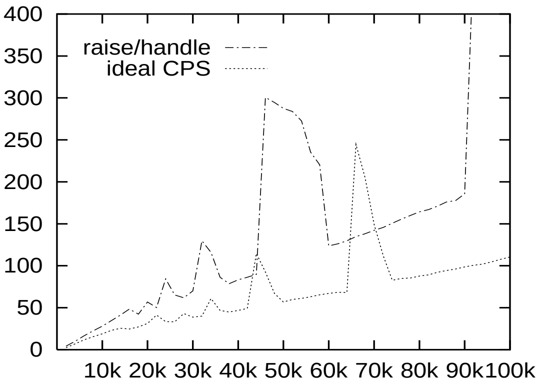
```
structure Subst_exn
= struct
    local open Exp
          exception same
    in fun subst (x, e, b)
           = let fun walk (VAR x')
                       = if x' = x then e else raise same
                     | walk (LAM (x', b'))
                       = if x' = x then raise same else LAM (x', walk b')
                     | walk (APP (e0, e1))
                       = let val e0' = walk e0
                         in APP (e0', walk e1
                                      handle same => e1)
                         end
                         handle same => APP (e0, walk e1)
             in walk b
                handle same => b
             end
    end
  end
```

Figure 2: Economical substitution using exceptions

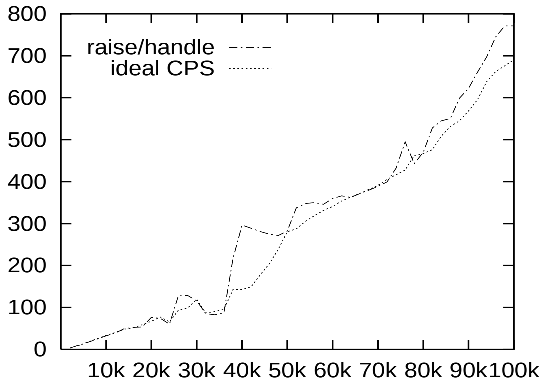
```
structure Subst_cps =
struct
  local open Exp
  in fun subst (x, e, b)
         = let fun walk (VAR x') k h
                   = if x' = x then k e else h ()
                 | walk (LAM (x', b')) k h
                   = if x' = x then h () else walk b' (fn b'' => k (LAM (x', b''))) h
                 | walk (APP (e0, e1)) k h
                   = walk e0
                          (fn e0' => walk e1 (fn e1' => k (APP (e0', e1'))) (fn () => k (APP (e0', e1))))
                          (fn () => walk e1 (fn e1' => k (APP (e0, e1'))) h)
           in walk b (fn b' => b') (fn () => b)
           end
  end
end
```

Figure 3: Economical substitution using continuations

X coordinate: the input expression size.
Y coordinate: execution time in ms, using SML/NJ 110 on a Sun UltraSPARC 2 (user time + system time + gc time).



(a) Exceptions are frequently raised.



(b) No exceptions are raised.

Figure 4: Performance difference: exception-based (Figure 2) versus continuation-based (Figure 3) versions

Passing two continuations to process exceptions is not a new idea: for example, Appel mentions it in his book on compiling with continuations [App92]. However, and even though his ML compiler uses a CPS transformation, it does not remove `raise` and `handle`: these two constructs remain as primitive operators (`sethdlr` and `gethdlr`). We conjecture that it was not cost-effective to pass two continuations to every function as specified, e.g., by Biagioni et al. [BCL+98, Figure 2].

The new idea here is a cost-effective CPS transformation using two continuations: we exploit the static information from Yi and Ryu's exception analysis [YR97] to reduce the continuation-passing traffic to where it is actually needed. To this end, we present a *selective CPS transformation* that only introduces continuations where they are needed, based on the static information gathered by the exception analysis. Our selective CPS transformation thus generalizes both the CPS transformation with two continuations (for expressions using exceptions) and the identity transformation (for exception-free expressions).

We prove that this selective CPS transformation is correct, and present some early experimental data that show its effectiveness for exception-intensive programs.

1.3 Related work

We know of three earlier works using the idea of exploiting static information to perform a selective CPS transformation. This has been done for strictness analysis [DH93a], for totality analysis [DH93b], and for binding-time analysis [DD95]. None of these selective CPS transformations, however, has been integrated in an actual compiler. Only the latter transformation has been integrated in a partial evaluator [Dus97]. In their work on "selective thunkification," Steckler and Wand use a similar idea to reduce the number of thunks when converting a call-by-name program into a call-by-value program, after strictness analysis [HD97, SW94].

1.4 Overview

The rest of this article is organized as follows. Section 2 defines the syntax and semantics of our source language, a subset of the ML's core. Section 3 presents a naïve CPS transformation that blindly passes continuations to every expression. Section 4 defines the annotation process of selecting candidate expressions for our selective transformation. Section 5 formalizes our selective transformation and proves its correctness. Section 6 shows our preliminary experimental data and discusses some more selective transformations which we will consider if the ongoing experiments with the selective transformation turn out to be not economical enough.

2 Source Language

2.1 Abstract syntax

We consider ML's core language, which is call-by-value and higher-order. Its abstract syntax reads as follows. (κ denotes a constructor.)

$$
\begin{array}{rcll}
e & ::= & \mathtt{1} & \text{unit} \\
 & | & x & \text{variable} \\
 & | & \lambda x.\, e & \text{function} \\
 & | & \mathtt{fix}\, f\ \lambda x.\, e & \text{recursive function} \\
 & | & e_1\ e_2 & \text{application} \\
 & | & \mathtt{con}\, \kappa\ e & \text{exception construction} \\
 & | & \mathtt{decon}\ e & \text{deconstruction} \\
 & | & \mathtt{case}\ e_1\ \kappa\ e_2\ e_3 & \text{switch} \\
 & | & \mathtt{handle}\ e_1\ \kappa\ \lambda x.\, e_2 & \text{handle expression} \\
 & | & \mathtt{raise}\ e & \text{raise exception}
\end{array}
$$

An exception value $\kappa \cdot v$ is constructed by "$\mathtt{con}\, \kappa\, e$" where evaluating e yields v. Symmetrically, an exception value $\kappa \cdot v$ denoted by e is deconstructed into v by evaluating "$\mathtt{decon}\ e$." Evaluating the case expression "$\mathtt{case}\ e_1\ \kappa\ e_2\ e_3$" yields the value of e_2 if the value of e_1 is $\kappa \cdot v$; otherwise, it yields the value of e_3. When evaluating the raise expression "$\mathtt{raise}\ e$" e is first evaluated, yielding an exception value κv. The exception associated to κ is then raised. The handle expression "$\mathtt{handle}\ e_1\ \kappa\ \lambda x.e_2$" evaluates e_1 first. If e_1 yields an exception value $\kappa \cdot v$, this exception value is passed to $\lambda x.e_2$. Otherwise, the value of the handle expression is the value yielded by e_1.

For brevity, we omit datatype values, strings, and memory operations (assignment, reference, and dereference) here. In reality, we work on the complete core language of Standard ML.

2.2 Operational semantics

We define the semantics of expressions with a structural operational semantics [Plo81] using Felleisen's evaluation contexts [Fel87]. In doing so, we need to extend the expressions to contain a set of values v and raised exceptions p that represent terminated computations:

$$
\begin{array}{rcll}
v & ::= & \mathtt{1} & \text{unit} \\
 & | & \lambda x.\, e & \text{function} \\
 & | & \mathtt{fix}\, f\ \lambda x.\, e & \text{recursive function} \\
 & | & \kappa \cdot v & \text{exception value with argument } v \\
p & ::= & \underline{\kappa \cdot v} & \text{raised exception}
\end{array}
$$

The evaluation context C is defined by

$$
\begin{array}{rcll}
C & ::= & [\,] & \text{hole} \\
 & | & \mathtt{con}\, \kappa\ C & \\
 & | & \mathtt{decon}\ C & \\
 & | & C\ e & \\
 & | & v\ C & \\
 & | & \mathtt{case}\ C\ \kappa\ e_1\ e_2 & \\
 & | & \mathtt{handle}\ C\ \kappa\ \lambda x.\, e & \\
 & | & \mathtt{raise}\ C &
\end{array}
$$

This context defines a left-to-right, call-by-value reduction. As usual, we write $C[e]$ if the hole in context C is filled with e. We use this context to define the reduction rule for arbitrary expressions:

$$\frac{e \to e'}{C[e] \to C[e']}$$

The single reduction step $e \to e'$ for a redex e is defined in Figure 5. As usual, the notation $[v/x]e$ denotes the new expression that results from substituting v for every free occurrence of x in e.

Normal reduction steps:

$$
\begin{array}{rcl}
\mathtt{con}\,\kappa\,v & \rightarrow & \kappa\cdot v \\
\mathtt{decon}\,\kappa\cdot v & \rightarrow & v \\
(\lambda x.\,e)\,v & \rightarrow & [v/x]e \\
(\mathtt{fix}\,f\,\lambda x.\,e)\,v & \rightarrow & [v/x][\mathtt{fix}\,f\,\lambda x.\,e/f]e \\
\mathtt{case}\,\kappa\cdot v\,\kappa\,e_1\,e_2 & \rightarrow & e_1 \\
\mathtt{case}\,\kappa\cdot v\,\kappa'\,e_1\,e_2 & \rightarrow & e_2 \quad (\kappa' \neq \kappa) \\
\mathtt{handle}\,v\,\kappa\,\lambda x.\,e & \rightarrow & v
\end{array}
$$

Exceptional reduction steps:

$$
\begin{array}{rcl}
\mathtt{raise}\,\kappa\cdot v & \rightarrow & \underline{\kappa\cdot v} \\
\mathtt{raise}\,\underline{\kappa\cdot v} & \rightarrow & \underline{\kappa\cdot v} \\
\mathtt{handle}\,\underline{\kappa\cdot v}\,\kappa\,\lambda x.\,e & \rightarrow & (\lambda x.\,e)\,\kappa\cdot v \\
\mathtt{handle}\,\underline{\kappa\cdot v}\,\kappa'\,\lambda x.\,e & \rightarrow & \underline{\kappa\cdot v}\ (\kappa' \neq \kappa) \\
\mathtt{con}\,\kappa\,\underline{\kappa\cdot v} & \rightarrow & \underline{\kappa\cdot v} \\
\mathtt{decon}\,\underline{\kappa\cdot v} & \rightarrow & \underline{\kappa\cdot v} \\
\mathtt{case}\,\underline{\kappa\cdot v}\,\kappa'\,e_1\,e_2 & \rightarrow & \underline{\kappa\cdot v} \\
\underline{\kappa\cdot v}\,e & \rightarrow & \underline{\kappa\cdot v} \\
(\lambda x.\,e)\,\underline{\kappa\cdot v} & \rightarrow & \underline{\kappa\cdot v} \\
(\mathtt{fix}\,f\,\lambda x.\,e)\,\underline{\kappa\cdot v} & \rightarrow & \underline{\kappa\cdot v}
\end{array}
$$

Figure 5: Reduction steps

$$
\begin{array}{rcl}
T(\mathtt{1}) & = & \lambda\langle K,H\rangle.\,K(\mathtt{1}) \\
T(x) & = & \lambda\langle K,H\rangle.\,K(x) \\
T(\mathtt{con}\,\kappa\,e) & = & \lambda\langle K,H\rangle.\,T(e)\,\langle\lambda v.\,K\,(\mathtt{con}\,\kappa\,v),H\rangle \\
T(\mathtt{decon}\,e) & = & \lambda\langle K,H\rangle.\,T(e)\,\langle\lambda v.\,K\,(\mathtt{decon}\,v),H\rangle \\
T(\lambda x.\,e) & = & \lambda\langle K,H\rangle.\,K(\lambda x.\,T(e)) \\
T(\mathtt{fix}\,f\,\lambda x.\,e) & = & \lambda\langle K,H\rangle.\,K(\mathtt{fix}\,f\,\lambda x.\,T(e)) \\
T(e_1\,e_2) & = & \lambda\langle K,H\rangle.\,T(e_1)\,\langle\lambda f.\,T(e_2)\,\langle\lambda v.\,f\,v\,\langle K,H\rangle,H\rangle,H\rangle \\
T(\mathtt{case}\,e_1\,\kappa\,e_2\,e_3) & = & \lambda\langle K,H\rangle.\,T(e_1)\,\langle\lambda v.\,\mathtt{case}\,v\,\kappa\,(T(e_2)\langle K,H\rangle)\,(T(e_3)\langle K,H\rangle),H\rangle \\
T(\mathtt{handle}\,e_1\,\kappa\,\lambda x.\,e_2) & = & \lambda\langle K,H\rangle.\,T(e_1)\,\langle K,\lambda v.\,\mathtt{case}\,v\,\kappa\,((\lambda x.\,T(e_2))\,v\,\langle K,H\rangle)\,(H\,v)\rangle \\
T(\mathtt{raise}\,e) & = & \lambda\langle K,H\rangle.\,T(e)\,\langle H,H\rangle
\end{array}
$$

Figure 6: Naïve CPS transformation T

Definition 1 *The semantics of a closed expression e is defined to be the sequence of reduction steps*

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \cdots.$$

If the sequence terminates with a value v (with an uncaught exception $\underline{\kappa\cdot v}$, respectively) after zero or more reductions, we write

$$e \xrightarrow{*} v \quad (e \xrightarrow{*} \underline{\kappa\cdot v}, \textit{respectively})$$

3 Naïve CPS Transformation

We can remove the raise and handle expressions by passing two continuations to each expression. Consider, for example, the following handle expression

$$\mathtt{handle}\,e_1\,\kappa\,\lambda x.\,e_2.$$

The handler $\lambda x.\,e_2$ is installed prior to evaluating e_1. Therefore, if an exception $\underline{\kappa\cdot v}$ is raised during this evaluation, then it is caught by the handler and $\lambda x.\,e_2$ is applied to $\kappa\cdot v$, yielding the value of the handle expression.

The CPS transformation encodes both how to handle a raised exception and how to proceed thereafter, with a *handler continuation.* This handler continuation is passed to the sub-expressions of e_1. Accordingly, a raise expression is CPS-transformed with the current handler continuation in place of the current continuation. Note that, because a handler continuation needs to encode how to continue after the handling, we also have to make the normal continuation ready to be captured by a handler continuation. Thus we keep passing two continuations (normal and handler continuations) to every expression.

Figure 6 shows the definition of this CPS-transformation function T. The T function transforms expressions of type τ to expressions of type $(\tau \rightarrow Ans) \times (Exn \rightarrow Ans) \rightarrow Ans$.

Theorem 1 (Correctness of T) *For any program $\wp$*

$$
\begin{array}{rcl}
\wp \xrightarrow{*} v & \Longrightarrow & T(\wp)\,\langle K,H\rangle \xrightarrow{*} K(\Psi(v)) \\
\wp \xrightarrow{*} \underline{\kappa\cdot v} & \Longrightarrow & T(\wp)\,\langle K,H\rangle \xrightarrow{*} H(\Psi(\kappa\cdot v))
\end{array}
$$

where the auxiliary function Ψ coerces direct-style values to CPS values:

$$
\begin{array}{rcl}
\Psi(\mathtt{1}) & = & \mathtt{1} \\
\Psi(\lambda x.\,e) & = & \lambda x.\,T(e) \\
\Psi(\mathtt{fix}\,f\,\lambda x.\,e) & = & \mathtt{fix}\,f\,\lambda x.\,T(e) \\
\Psi(\kappa\cdot v) & = & \kappa\cdot\Psi(v)
\end{array}
$$

Proof. Analogous to the proof of Plotkin's simulation theorem [Plo75]. In the proof we need to extend T to be defined for values and raised exceptions:

$$
\begin{array}{rcl}
T(v) & = & \lambda\langle K,H\rangle.\,K(\Psi(v)) \\
T(\underline{\kappa\cdot v}) & = & \lambda\langle K,H\rangle.\,H(\Psi(\kappa\cdot v))
\end{array}
$$

□

Transforming every expression into this CPS-form, as done by T, however, offsets the benefit of removing raise-handle expressions. Indeed, every expression becomes a higher-order function application that receives two functions (continuations).

At this stage, two orthogonal optimizations can take place:

- We can perform administrative reductions, e.g., during the CPS transformation [DF92].

- We can *selectively* apply the CPS transformation by exploiting the results of Yi and Ryu's exception analysis [YR97]. The point is that an exception analysis tells us a conservative approximation of which expressions may raise an uncaught exception when they are evaluated and of which functions may raise an uncaught exception when they are applied. Therefore, such an analysis also tells us a safe approximation of which expressions do not raise any uncaught exceptions when they are evaluated and of which functions do not raise any uncaught exceptions when they are applied. These latter expressions and functions need not be CPS-transformed.

4 Annotation

We pass two continuations to an expression e to cater for the case where uncaught exceptions raised while evaluating e (in the future) are caught by a surrounding handler (that was installed in the past). The raise expression will call a handler continuation that was passed to it from the outer context. If we know that evaluating an expression will not yield an uncaught exception, there is no need to pass it any continuations since we know statically that these continuations would be of no use. Three cases occur:

- If evaluating e cannot yield any uncaught exceptions (i.e., it can only yield normal values), then e needs no continuations.

- If evaluating e may yield an uncaught exception which is not handled in the complete program, then e needs no continuations either: we can just call an abort function instead of raising this uncaught exception.

- If evaluating e may yield an uncaught exception which is handled in the complete program, then e needs two continuations.

The expressions of the third kind are candidates to our selective CPS transformation. The other ones can be kept in direct style.

Our selective CPS transformation therefore needs two pieces of information per source expression: whether the expression is protected by a handler and which exceptions, if any, may be raised when the expression is evaluated. The first condition $\lceil e \rceil^{\kappa}$ is checked by scanning the sub-expressions in the scope of a handler (i.e., sub-expressions of e in "$\texttt{handle}\ e\ \kappa\ \lambda x.\, e'$"), referencing a safe closure analysis [YR97].[2] The second condition $\lfloor e \rfloor_{\kappa}$ checks whether evaluating e raises an exception κ. It is based on Yi and Ryu's exception analysis [YR97].

If both the conditions $\lceil e \rceil^{\kappa}$ and $\lfloor e \rfloor_{\kappa}$, for the same exception κ, hold for an expression e, the *exceptional* expression e becomes the candidate $\breve{e}$ of our CPS transformation. Otherwise, the expression e is marked $\dot{e}$ as *normal*.

Figure 7 displays the rules

$$\mathcal{R}(\mathit{Exn\_analysis}_{\wp}, \mathit{Closure\_analysis}_{\wp})$$

[2]For languages with exceptions that can carry functions, existing works [Shi91, HM97, JW96, Ses89] cannot be used because exception analysis and closure analysis are interdependent in ML. We therefore had to devise our own closure analysis.

$$\frac{\kappa \in \mathit{Exn\_analysis}_{\wp}(e)}{\lfloor e \rfloor_{\kappa}}$$

$$\frac{\texttt{handle}\ e_1\ \kappa\ \lambda x.\, e_2}{\lceil e_1 \rceil^{\kappa}}$$

$$\frac{\lceil \texttt{con}\ \kappa'\ e \rceil^{\kappa}}{\lceil e \rceil^{\kappa}} \qquad \frac{\lceil \texttt{decon}\ e \rceil^{\kappa}}{\lceil e \rceil^{\kappa}} \qquad \frac{\lceil \texttt{raise}\ e \rceil^{\kappa}}{\lceil e \rceil^{\kappa}}$$

$$\frac{\lceil e_1\ e_2 \rceil^{\kappa} \quad \lambda x.\, e \in \mathit{Closure\_analysis}_{\wp}(e_1)}{\lceil e_1 \rceil^{\kappa} \quad \lceil e_2 \rceil^{\kappa} \quad \lceil e \rceil^{\kappa}}$$

$$\frac{\lceil \texttt{case}\ e_1\ \kappa\ e_2\ e_3 \rceil^{\kappa}}{\lceil e_1 \rceil^{\kappa} \quad \lceil e_2 \rceil^{\kappa} \quad \lceil e_3 \rceil^{\kappa}}$$

$$\frac{\lceil e \rceil^{\kappa} \quad \lfloor e \rfloor_{\kappa}}{\breve{e}}$$

$\lfloor e \rfloor_{\kappa}$ is read as "evaluating e may raise a κ-exception."
$\lceil e \rceil^{\kappa}$ is read as "a κ-handler protects e."
Once the analysis is completed,
any expression e that is not marked as exceptional $\breve{e}$
is marked as normal $\dot{e}$.

Figure 7: Rules $\mathcal{R}(\mathit{Exn\_analysis}_{\wp}, \mathit{Closure\_analysis}_{\wp})$ for deciding whether an expression e in a program $\wp$ is exceptional ($\breve{e}$) or normal ($\dot{e}$).

for determining which expressions are exceptional in program $\wp$. The first analysis, $\mathit{Exn\_analysis}_{\wp}$, maps each expression of the program $\wp$ to the set of exceptions that may be uncaught in the expression. The second analysis, $\mathit{Closure\_analysis}_{\wp}$, maps each function expression to the set of lambda-abstractions that can flow into it. These two auxiliary analyses are assumed correct. In our implementation we use the ones developed by Yi and Ryu [YR97].

Definition 2 *The annotated version $\mathit{Annotate}(\wp)$ of a program $\wp$ is the one resulting from marking each sub-expression e either as exceptional ($\breve{e}$) or as normal ($\dot{e}$), based on the rules $\mathcal{R}(\mathit{Exn\_analysis}_{\wp}, \mathit{Closure\_analysis}_{\wp})$ in Figure 7.*

Theorem 2 (Safety of Annotation) *If an expression e of a program is marked normal $\dot{e}$, then evaluating it yields either a normal value or an uncaught exception that aborts the program.*

Proof. Assume for contradiction that an expression e evaluates into an uncaught exception that will be handled inside the program. Then it must be marked exceptional $\breve{e}$ because $\lfloor e \rfloor_{\kappa}$ and $\lceil e \rceil^{\kappa}$ hold and are based on annotation rules that use $\mathit{Exn\_analysis}_{\wp}$ and $\mathit{Closure\_analysis}_{\wp}$, both of which are safe. □

Figure 8 shows the syntax of well-formed sets of normal expressions $\dot{e}$ and exceptional expressions $\breve{e}$, respectively. Let us list some noteworthy cases:

- An application expression $\dot{e}_1\ \dot{e}_2$ with all normal sub-expressions can be exceptional if a function that flows into $\dot{e}_1$ has an exceptional body.

$$\text{Annotated expressions: } e \quad ::= \quad \dot{e} \mid \breve{e}$$

$$\begin{array}{lcl}
\text{Normal expressions: } \dot{e} & ::= & \mathtt{1} \mid x \\
& \mid & \lambda x.\, \dot{e} \mid \lambda x.\, \breve{e} \\
& \mid & \mathtt{fix}\, f\, \lambda x.\, e \\
& \mid & \dot{e}_1\, \dot{e}_2 \\
& \mid & \mathtt{con}\, \kappa\, \dot{e} \mid \mathtt{decon}\, \dot{e} \\
& \mid & \mathtt{case}\, \dot{e}_1\, \kappa\, \dot{e}_2\, \dot{e}_3 \\
& \mid & \mathtt{handle}\, e_1\, \kappa\, \lambda x.\, \dot{e}_2 \\
& \mid & \mathtt{handle}\, \dot{e}_1\, \kappa\, \lambda x.\, \breve{e}_2 \\
& \mid & \mathtt{raise}\, \dot{e}
\end{array}
\qquad
\begin{array}{lcl}
\text{Exceptional expressions: } \breve{e} & ::= & e_1\, e_2 \\
& \mid & \mathtt{con}\, \kappa\, \breve{e} \\
& \mid & \mathtt{decon}\, \breve{e} \\
& \mid & \mathtt{case}\, \breve{e}_1\, \kappa\, e_2\, e_3 \\
& \mid & \mathtt{case}\, e_1\, \kappa\, \breve{e}_2\, e_3 \\
& \mid & \mathtt{case}\, e_1\, \kappa\, e_2\, \breve{e}_3 \\
& \mid & \mathtt{handle}\, \breve{e}_1\, \kappa\, \lambda x.\, e_2 \\
& \mid & \mathtt{raise}\, e
\end{array}$$

Figure 8: Two classes of expressions: normal or exceptional

Define $\bot_K = \lambda x.x$ and $\bot_H = \lambda x.\mathtt{raise}\ x$.

Transformation $\dot{T}(\dot{e})$ for normal expressions $\dot{e}$:

$$\begin{array}{rcl}
\dot{T}(\mathtt{1}) & = & \mathtt{1} \\
\dot{T}(x) & = & x \\
\dot{T}(\lambda x.\, \dot{e}) & = & \lambda x.\, (\lambda \langle K, H\rangle.\, K(\dot{T}(\dot{e}))) \\
\dot{T}(\lambda x.\, \breve{e}) & = & \lambda x.\, \breve{T}(\breve{e}) \\
\dot{T}(\mathtt{fix}\, f\, \lambda x.\, \dot{e}) & = & \mathtt{fix}\, f\, \lambda x.\, \lambda \langle K, H\rangle.\, K(\dot{T}(\dot{e})) \\
\dot{T}(\mathtt{fix}\, f\, \lambda x.\, \breve{e}) & = & \mathtt{fix}\, f\, \lambda x.\, \breve{T}(\breve{e}) \\
\dot{T}(\dot{e}_1\, \dot{e}_2) & = & \dot{T}(\dot{e}_1)\, \dot{T}(\dot{e}_2)\, \langle \bot_K, \bot_H \rangle \\
\dot{T}(\mathtt{con}\, \kappa\, \dot{e}) & = & \mathtt{con}\, \kappa\, \dot{T}(\dot{e}) \\
\dot{T}(\mathtt{decon}\, \dot{e}) & = & \mathtt{decon}\, \dot{T}(\dot{e}) \\
\dot{T}(\mathtt{case}\, \dot{e}_1\, \kappa\, \dot{e}_2\, \dot{e}_3) & = & \mathtt{case}\, \dot{T}(\dot{e}_1)\, \kappa\, \dot{T}(\dot{e}_2)\, \dot{T}(\dot{e}_3) \\
\dot{T}(\mathtt{handle}\, \dot{e}_1\, \kappa\, \lambda x.\, \dot{e}_2) & = & \dot{T}(\dot{e}_1) \\
\dot{T}(\mathtt{handle}\, \dot{e}_1\, \kappa\, \lambda x.\, \breve{e}_2) & = & \dot{T}(\dot{e}_1) \\
\dot{T}(\mathtt{handle}\, \breve{e}_1\, \kappa\, \lambda x.\, \dot{e}_2) & = & \breve{T}(\breve{e}_1)\, \langle \bot_K, \lambda v.\, \mathtt{case}\, v\, \kappa\, ((\lambda x.\, \dot{T}(\dot{e}_2))\, v)\, (\bot_H\, v) \rangle \\
\dot{T}(\mathtt{raise}\, \dot{e}) & = & \bot_H(\dot{T}(\dot{e}))
\end{array}$$

Transformation $\breve{T}(\breve{e})$ for exceptional expressions $\breve{e}$:

$$\begin{array}{rcl}
\breve{T}(\dot{e}_1\, \dot{e}_2) & = & \lambda \langle K, H\rangle.\, \dot{T}(\dot{e}_1)\, \dot{T}(\dot{e}_2)\, \langle K, H\rangle \\
\breve{T}(\breve{e}_1\, \dot{e}_2) & = & \lambda \langle K, H\rangle.\, \breve{T}(\breve{e}_1)\, \langle \lambda f.\, (f\, \dot{T}(\dot{e}_2)\, \langle K, H\rangle), H\rangle \\
\breve{T}(\dot{e}_1\, \breve{e}_2) & = & \lambda \langle K, H\rangle.\, (\lambda f.\, \breve{T}(\breve{e}_2)\, \langle \lambda v.\, (f\, v\, \langle K, H\rangle), H\rangle)\, \dot{T}(\dot{e}_1) \\
\breve{T}(\breve{e}_1\, \breve{e}_2) & = & \lambda \langle K, H\rangle.\, \breve{T}(\breve{e}_1)\, \langle \lambda f.\, \breve{T}(\breve{e}_2)\, \langle \lambda v.\, (f\, v\, \langle K, H\rangle), H\rangle, H\rangle \\
\breve{T}(\mathtt{con}\, \kappa\, \breve{e}) & = & \lambda \langle K, H\rangle.\, \breve{T}(\breve{e})\langle \lambda v.\, K(\kappa{\cdot}v), H\rangle \\
\breve{T}(\mathtt{decon}\, \breve{e}) & = & \lambda \langle K, H\rangle.\, \breve{T}(\breve{e})\langle \lambda \kappa{\cdot}v.\, K(v), H\rangle \\
\breve{T}(\mathtt{case}\, \breve{e}_1\, \kappa\, \dot{e}_2\, \dot{e}_3) & = & \lambda \langle K, H\rangle.\, \breve{T}(\breve{e}_1)\, \langle \lambda v.\, \mathtt{case}\, v\, \kappa\, K(\dot{T}(\dot{e}_2))\, K(\dot{T}(\dot{e}_3)), H\rangle \\
\breve{T}(\mathtt{case}\, \dot{e}_1\, \kappa\, \breve{e}_2\, \dot{e}_3) & = & \lambda \langle K, H\rangle.\, \mathtt{case}\, \dot{T}(\dot{e}_1)\, \kappa\, (\breve{T}(\breve{e}_2)\langle K, H\rangle)\, K(\dot{T}(\dot{e}_3)) \\
\breve{T}(\mathtt{case}\, \dot{e}_1\, \kappa\, \dot{e}_2\, \breve{e}_3) & = & \lambda \langle K, H\rangle.\, \mathtt{case}\, \dot{T}(\dot{e}_1)\, \kappa\, K(\dot{T}(\dot{e}_2))\, (\breve{T}(\breve{e}_3)\, \langle K, H\rangle) \\
\breve{T}(\mathtt{case}\, \breve{e}_1\, \kappa\, \breve{e}_2\, \dot{e}_3) & = & \lambda \langle K, H\rangle.\, \breve{T}(\breve{e}_1)\, \langle \lambda v.\, \mathtt{case}\, v\, \kappa\, (\breve{T}(\breve{e}_2)\, \langle K, H\rangle)\, K(\dot{T}(\dot{e}_3)), H\rangle \\
\breve{T}(\mathtt{case}\, \breve{e}_1\, \kappa\, \dot{e}_2\, \breve{e}_3) & = & \lambda \langle K, H\rangle.\, \breve{T}(\breve{e}_1)\, \langle \lambda v.\, \mathtt{case}\, v\, \kappa\, K(\dot{T}(\dot{e}_2))\, (\breve{T}(\breve{e}_3)\, \langle K, H\rangle), H\rangle \\
\breve{T}(\mathtt{case}\, \dot{e}_1\, \kappa\, \breve{e}_2\, \breve{e}_3) & = & \lambda \langle K, H\rangle.\, \mathtt{case}\, \dot{T}(\dot{e}_1)\, \kappa\, (\breve{T}(\breve{e}_2)\, \langle K, H\rangle)\, (\breve{T}(\breve{e}_3)\, \langle K, H\rangle) \\
\breve{T}(\mathtt{case}\, \breve{e}_1\, \kappa\, \breve{e}_2\, \breve{e}_3) & = & \lambda \langle K, H\rangle.\, \breve{T}(\breve{e}_1)\, \langle \lambda v.\, \mathtt{case}\, v\, \kappa\, (\breve{T}(\breve{e}_2)\langle K, H\rangle)\, (\breve{T}(\breve{e}_3)\langle K, H\rangle), H\rangle \\
\breve{T}(\mathtt{handle}\, \breve{e}_1\, \kappa\, \lambda x.\, \dot{e}_2) & = & \lambda \langle K, H\rangle.\, \breve{T}(\breve{e}_1)\, \langle K, \lambda v.\, \mathtt{case}\, v\, \kappa\, K((\lambda x.\, \dot{T}(\dot{e}_2))\, v)\, (H\, v)\rangle \\
\breve{T}(\mathtt{handle}\, \breve{e}_1\, \kappa\, \lambda x.\, \breve{e}_2) & = & \lambda \langle K, H\rangle.\, \breve{T}(\breve{e}_1)\, \langle K, \lambda v.\, \mathtt{case}\, v\, \kappa\, ((\lambda x.\, \breve{T}(\breve{e}_2))\, v\, \langle K, H\rangle)\, (H\, v)\rangle \\
\breve{T}(\mathtt{raise}\, \dot{e}) & = & \lambda \langle K, H\rangle.\, H(\dot{T}(\dot{e})) \\
\breve{T}(\mathtt{raise}\, \breve{e}) & = & \lambda \langle K, H\rangle.\, \breve{T}(\breve{e})\, \langle H, H\rangle
\end{array}$$

Figure 9: Selective CPS transformation

- $\mathtt{raise}\,\dot{e}$ can be normal[3] if no handler exists that catches the exception $\dot{e}$.
- $\mathtt{handle}\ \breve{e}_1\ \kappa\ \lambda x.\,\dot{e}_2$ can be normal or exceptional depending on whether an uncaught exception from $\breve{e}_1$ is handled by $\lambda x.\,\dot{e}_2$ or not.
- The toplevel expression of the program is always marked normal because it evaluates either into a normal value or into an uncaught exception.

5 Selective CPS Transformation

Our selective CPS transformation is defined by two transformation functions $\dot{T}$ and $\breve{T}$, which transform normal and exceptional expressions, respectively. We transform a program $\wp$ by applying $\dot{T}$ to its annotated version $Annotate(\wp)$:

$$\dot{T}(Annotate(\wp)).$$

Each sub-expression of $\wp$ is transformed by either $\dot{T}$ or $\breve{T}$, depending on its annotation. Figure 9 displays the definitions.

$\breve{T}$ transforms exceptional expressions of type τ into expressions of type $(\tau \to \alpha) \times (Exn \to \alpha) \to \alpha$. The type α is determined by the context where the transformation $\breve{T}$ occurs. If $\breve{T}(\breve{e}_1)$ occurs where $\breve{e}_1$ is a sub-expression of a normal expression $\dot{e}$, then the result type α of the continuations becomes the type of $\dot{e}$. If $\breve{e}_1$ is a sub-expression of an exceptional expression $\breve{e}$, then the α is determined by the context where $\breve{T}(\breve{e})$ occurs, and so on.

$\dot{T}$ transforms normal expressions of type τ into expressions of the same type, except for one case: $\dot{T}$ always transform lambda-abstractions ($\lambda x.\,e$ and $\mathtt{fix}\ f\ \lambda x.\,e$) to receive continuations. This coercion is necessary when an exceptional application can execute a normal body, which is possible as follows. For an application $\dot{e}_1\,\dot{e}_2$, let us assume that two lambda-abstraction $\lambda x.\breve{e}$ and $\lambda y.\dot{e}$ can flow into $\dot{e}_1$. Because of the exceptional body $\breve{e}$ the application expression can be marked exceptional, hence is transformed by $\breve{T}$ to

$$\lambda\langle K,H\rangle.\,\dot{T}(\dot{e}_1)\,\dot{T}(\dot{e}_2)\,\langle K,H\rangle.$$

Then it is necessary to coerce the other function $\lambda y.\,\dot{e}$ to expect continuations:

$$\lambda y.\,\lambda\langle K,H\rangle.\,K(\dot{T}(\dot{e})).$$

Because of this coercion, if an application $e_1\,e_2$ is transformed we must always pass two continuations. In case of the normal transformation $\dot{T}(e_1\,e_2)$ we pass a dummy pair $\langle\perp_K,\perp_H\rangle$. In case of the exceptional transformation $\breve{T}(e_1\,e_2)$ we pass the continuation pair $\langle K,H\rangle$ received from the context.

Note that even though the transformed program can still have raise expressions because of $(\lambda x.\,\mathtt{raise}\ x)$, which is our $\perp_H$ function, such raise expressions are evaluated if and only if the input program raises exceptions that abort the program execution.

[3] The word "normal" is slightly misleading here because such an expression aborts the whole execution of the program—in fact, we replace it by an abort operation. But since this operation does not need to be passed any continuation, we can mark the expression as normal.

Let us consider several examples. Their basic theme is that an exceptional expression $\breve{e}$ is transformed with $\breve{T}$ to receive a pair of continuations, whereas a normal expression $\dot{e}$ is transformed with $\dot{T}$ to yield its value in the usual direct style.

- Case of normal handle-expression whose first sub-expression is exceptional:

  $$\dot{T}(\mathtt{handle}\ \breve{e}_1\ \kappa\ \lambda x.\,\dot{e}_2) = \breve{T}(\breve{e}_1)\ \langle\perp_K, \lambda v.\,\mathtt{case}\ v\ \kappa\ ((\lambda x.\,\dot{T}(\dot{e}_2))\ v)\ (\perp_H v)\rangle$$

  The exceptional sub-expression $\breve{e}_1$ is transformed with $\breve{T}$ to receive two continuations. The normal continuation is the identity function $\perp_K$, because the normal value of $\breve{e}_1$ is the value of the handle expression. The handler continuation is a case expression that, if the exception v is unmatched, aborts the program by raising the exception $(\perp_H v)$, and otherwise, handles it normally $((\lambda x.\,\dot{T}(\dot{e}_2))\,v)$. The abortion case takes place when $\breve{e}_1$ raises an exception the program cannot handle.

- Similar case, but when the handle-expression is exceptional:

  $$\breve{T}(\mathtt{handle}\ \breve{e}_1\ \kappa\ \lambda x.\,\dot{e}_2) = \lambda\langle K,H\rangle.\,\breve{T}(\breve{e}_1)\ \langle K, \lambda v.\,\mathtt{case}\ v\ \kappa\ K((\lambda x.\,\dot{T}(\dot{e}_2))\ v)\ (H\ v)\rangle$$

  The difference from the previous case is that the transformed expression expects continuations $\langle K,H\rangle$ from the context and uses them in defining continuations to pass to subexpression $\breve{T}(\breve{e}_1)$. Continuations to pass to $\breve{T}(\breve{e}_1)$ are defined using the continuations $\langle K,H\rangle$ passed from the context.

- Case of exceptional application-expression whose second expression is normal.

  $$\breve{T}(\breve{e}_1\,\dot{e}_2) = \lambda\langle K,H\rangle.\,\breve{T}(\breve{e}_1)\ \langle\lambda f.\,(f\ \dot{T}(\dot{e}_2)\ \langle K,H\rangle), H\rangle$$

  The exceptional sub-expression $\breve{e}_1$ is transformed with $\breve{T}$ to receive two continuations. We pass two continuations to it: the normal continuation part $\lambda f.\,(f\,...)$ is to apply the resulting function f to argument $\dot{T}(\dot{e}_2)$ with the current continuations $\langle K,H\rangle$, and the handler continuation part is the H from the context.

Theorem 3 (Correctness of $\dot{T}$ and $\breve{T}$) *For any program $\wp$,*

$$\begin{array}{lcl} \wp \xrightarrow{*} v & \Longrightarrow & \dot{T}(Annotate(\wp)) \xrightarrow{*} \dot{\Psi}(v) \\ \wp \xrightarrow{*} \underline{\kappa{\cdot}v} & \Longrightarrow & \dot{T}(Annotate(\wp)) \xrightarrow{*} \underline{\dot{\Psi}(\kappa{\cdot}v)} \end{array}$$

where the auxiliary function $\dot{\Psi}$ coerces direct-style values to selective-CPS values:

$$\begin{array}{rcl} \dot{\Psi}(1) & = & 1 \\ \dot{\Psi}(\kappa{\cdot}v) & = & \kappa{\cdot}\dot{\Psi}(v) \\ \dot{\Psi}(\lambda x.\,\dot{e}) & = & \lambda x.\,\lambda\langle K,H\rangle.\,K(\dot{T}(e)) \\ \dot{\Psi}(\lambda x.\,\breve{e}) & = & \lambda x.\,\breve{T}(e) \\ \dot{\Psi}(\mathtt{fix}\ f\ \lambda x.\,\dot{e}) & = & \mathtt{fix}\ f\ \lambda x.\,\lambda\langle K,H\rangle.\,K(\dot{T}(e)) \\ \dot{\Psi}(\mathtt{fix}\ f\ \lambda x.\,\breve{e}) & = & \mathtt{fix}\ f\ \lambda x.\,\lambda\langle K,H\rangle.\,K(\breve{T}(e)) \end{array}$$

The proof uses the following three lemmas, which are analogous to those used in Plotkin's simulation theorem [Plo75].

Lemma 1

$$\begin{cases} [\dot{\Psi}(v)/x]\dot{T}(\dot{e}) & = & \dot{T}([v/x]\dot{e}) \\ [\dot{\Psi}(v)/x]\breve{T}(\breve{e}) & = & \breve{T}([v/x]\breve{e}) \end{cases}$$

Proof. By induction on size of e. □

Now the following two lemmas prove two vital properties about the infix operator ":", which is defined analogously to Plotkin's one. (See Figure 12, in appendix, for the definition.) Particular to our case is the definition of e:`Return`, which denotes that the term e needs no continuations to finish.

For the proofs, we need to extend the $\dot{T}$ and $\breve{T}$ functions, which were defined only for a program, to be defined also for all terms in the reduction sequence of the program:

$$\begin{array}{lcl} \dot{T}(\kappa\cdot v) & = & \dot{\Psi}(\kappa\cdot v) \\ \dot{T}(\underline{\kappa\cdot v}) & = & \underline{\dot{\Psi}(\kappa\cdot v)} \\ \breve{T}(v) & = & \lambda\langle K, H\rangle.\, K(\dot{\Psi}(v)) \\ \breve{T}(\underline{\kappa\cdot v}) & = & \lambda\langle K, H\rangle.\, H(\dot{\Psi}(\kappa\cdot v)) \end{array}$$

Although our annotation is defined only for programs not for the terms that occur during reductions, we find it convenient in the proofs that the terms also have annotations inherited from their original expressions:

$$\frac{\dot{e}_1, \quad e_1 \to e_2}{\dot{e}_2} \qquad \frac{\breve{e}_1, \quad e_1 \to e_2}{\breve{e}_2}$$

Lemma 2

$$\begin{cases} \dot{T}(\dot{e}) & \xrightarrow{+} & \dot{e} : \mathtt{Return} \\ \breve{T}(\breve{e})\,\langle K, H\rangle & \xrightarrow{+} & \breve{e} : \langle K, H\rangle \end{cases}$$

Proof. By induction on size of e. □

Lemma 3

$$\begin{cases} \dot{e} \to \dot{e}' & \Longrightarrow & e : \mathtt{Return} \xrightarrow{+} e' : \mathtt{Return} \\ \breve{e} \to \breve{e}' & \Longrightarrow & e : \langle K, H\rangle \xrightarrow{+} e' : \langle K, H\rangle \end{cases}$$

Proof.

- $\dot{e} = \mathtt{handle}\, \underline{\kappa\cdot\breve{v}_1}\, \kappa\ \dot{e}_2 \ \to\ [\kappa\cdot\breve{v}_1/x]\, \dot{e}_2$

  $\mathtt{handle}\, \underline{\kappa\cdot\breve{v}_1}\, \kappa\ \dot{e}_2 : \mathtt{Return}$
  $= \mathtt{case}\, \dot{\Psi}(\kappa\cdot\breve{v}_1)\, \kappa\ (\perp_K (\lambda x.\, \dot{T}(\dot{e}_2)\, \dot{\Psi}(\kappa\cdot\breve{v}_1)))\ (\perp_H \dot{\Psi}(\kappa\cdot\breve{v}_1))$
  (by def. of :)
  $\xrightarrow{+} \perp_K (\lambda x.\, \dot{T}(\dot{e}_2)\, \dot{\Psi}(\kappa\cdot\breve{v}_1))$
  $\xrightarrow{+} [\dot{\Psi}(\kappa\cdot\breve{v}_1)/x]\, \dot{T}(\dot{e}_2)$(by reduction rule)
  $= \dot{T}([\kappa\cdot\breve{v}_1/x]\, \dot{e}_2)$(by Lemma1)
  $= [\kappa\cdot\breve{v}_1/x]\, \dot{e}_2 : \mathtt{Return}$(by Lemma2)

- $\breve{e} = \mathtt{raise}\, \dot{e}_1 \ \to\ \mathtt{raise}\, \underline{\dot{v}_1}$

  $\mathtt{raise}\, \dot{e}_1 : \langle K, H\rangle$
  $= H\ (\dot{e}_1 : \mathtt{Return})$(by def. of :)
  $\xrightarrow{+} H\ (\underline{\dot{v}_1} : \mathtt{Return})$(by I.H.)
  $= H\ \underline{\dot{\Psi}(\dot{v}_1)}$(by def. of :)
  $\xrightarrow{+} \underline{\dot{\Psi}(\dot{v}_1)}$(by reduction rule)
  $= \mathtt{raise}\, \underline{\dot{v}_1} : \langle K, H\rangle$(by def. of :)

- $\dot{e} = \mathtt{handle}\, \dot{e}_1\, \kappa\, \lambda x.\, \dot{e}_2 \ \to\ \mathtt{handle}\, \underline{\kappa\cdot\dot{v}_1}\, \kappa\, \lambda x.\, \dot{e}_2$
  Impossible by annotation rule.

Other cases are similarly treated. □

It is now straightforward to prove Theorem 3:

Proof. If $\wp \xrightarrow{*} v$, then

$$\begin{array}{lcll} \dot{T}(\dot{\wp}) & \xrightarrow{+} & \wp : \mathtt{Return} & \text{(by Lemma 2)} \\ & \xrightarrow{*} & v & \text{(by Lemma 3)} \\ & \to & \dot{\Psi}(v) & \text{(by definition)} \end{array}$$

Similarly when $\wp \xrightarrow{*} \underline{\kappa\cdot v}$. □

6 Preliminary Experiments and Discussion

6.1 Sharing transducers

We tested our transformation for the `subst` function of Figure 2, i.e., a sharing transducer implementing substitution with exceptions. Figure 10 displays the transformed program, which results from our one-pass selective CPS transformer (à la Danvy and Filinski [DF92]).

Figure 11 shows (in solid line) that the transformed program can run with almost twice the speed of the original exception-based substitution program. As Graph (b) shows, if the program does not raise exceptions (hence handlers are not used) then the transformed program and the exception-based code run with about the same speed. Overall, the performance of the transformed program is actually nearing the performance of the CPS version (in dotted line) as specified in Figure 3, and which we consider as ideal.

6.2 Non exception-intensive programs

Our transformation's assumptions that all exceptions are explicitly raised inside the program does not hold in reality because some exceptions (e.g., Overflow) can be raised from pre-defined functions (e.g., `+`). While the program can handle such primitive exceptions, the transformed version misses them because our transformer doesn't see any raise expression which is transformed to activate the handler continuation. Thus our transformation must transform only such programs where uncaught exceptions from primitive functions are uncaught also inside the program. Meanwhile, if the input program has neither `raise` nor `handle` constructs, our transformation yields the same program.

6.3 More selective CPS transformations

We can tune our CPS transformation further. Currently, we transform every function (and application, respectively) to receive (to pass, respectively) two continuations. This blind transformation provides a simple solution to the "untypeful" flow situation where both normal and exceptional functions may flow into exceptional applications. Because of this situation, we need to coerce such normal functions to receive continuations. Since the coerced functions are also called at normal applications, we also coerce the normal applications to pass dummy continuations.

By classifying functions and applications more finely, we can reduce the traffic of such dummy continuations. Two cases occur for functions:

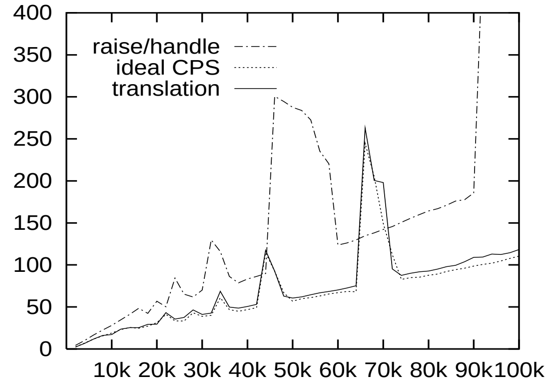
- Pure normal functions: normal functions which are always called in normal applications.

- Impure normal functions: normal functions which are called in exceptional applications.

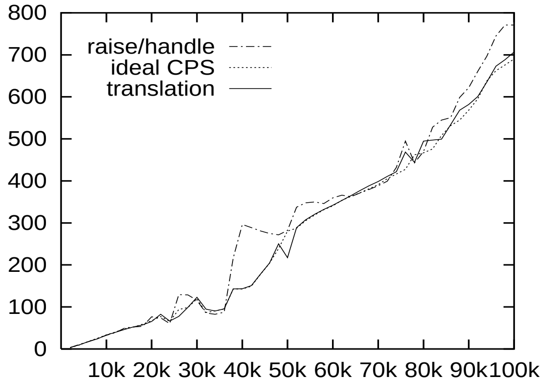
```
exception same

fun subst_trans {1=x,2=e,3=body}
    = let fun walk (VAR x') (k_var1,h_var2)
               = (case x'=x
                    of true => k_var1 e
                     | false => h_var2 same)
             | walk (LAM {1=x',2=body'}) (k_var1,h_var2)
               = (case x'=x
                    of true => h_var2 same
                     | false => walk body' ((fn a_var5 => k_var1 (LAM {2=a_var5,1=x'})),
                                            (fn b_var6 => h_var2 b_var6)))
             | walk (APP {1=e0,2=e1}) (k_var1,h_var2)
               = walk e0
                      ((fn a_var14 => walk e1
                                            ((fn a_var17 => k_var1 (APP {2=a_var17,1=a_var14})),
                                             (fn b_var18 => ((fn same => k_var1 (APP {2=e1,1=a_var14})
                                                                | a_var16 => raise a_var16)
                                                              b_var18)))),
                       (fn b_var15 => ((fn same => walk e1
                                                          ((fn a_var10 => k_var1 (APP {2=a_var10,1=e0})),
                                                           (fn b_var11 => h_var2 b_var11))
                                          | a_var7 => h_var2 a_var7)
                                        b_var15)))
      in walk body ((fn a_var20 => a_var20),
                     (fn b_var21 => ((fn same => body
                                        | a_var19 => raise a_var19)
                                      b_var21)))
      end
```

Figure 10: The exception-based substitution of Figure 2 after selective CPS transformation

X coordinate: the input expression size.
Y coordinate: execution time in ms, using SML/NJ 110 on a Sun UltraSPARC 2 (user time + system time + gc time).



(a) Exceptions are frequently raised.



(b) No exceptions are raised.

Figure 11: Experimental results for Figure 2 (—·—), Figure 3 (···), and Figure 10 (—)

Dually, two cases also occur for applications:

- Pure normal applications: all functions to call are pure normal functions.
- Impure normal applications: some functions to call are coerced normal functions.

Only impure normal functions must be coerced to receive continuations and only impure normal applications must be coerced to send dummy continuations.

The above finer classification is possible by adding the following two annotation rules to Figure 7: let $\bar{\dot{e}}$ denote that $\dot{e}$ is an impure normal expression.

$$\frac{\lambda x.\dot{e} \in \mathit{Closure\_analysis}_{\wp}(e_1) \quad \breve{e} = e_1\, e_2}{\overline{\lambda x.\dot{e}}}$$

$$\frac{\overline{\lambda x.\dot{e}} \in \mathit{Closure\_analysis}_{\wp}(e_1) \quad \dot{e} = \dot{e}_1\, \dot{e}_2}{\overline{\dot{e}_1\, \dot{e}_2}}$$

A new transformation function can easily be defined to exploit this finer annotation.

Depending on the source programs, however, this finer transformation can offset the benefit of removing raise and handle expressions. In some cases, it may actually be preferable to keep some of them. Indeed, if a handler is installed long before a matching raise expression occurs, the CPS-transformed intermediate expressions have the burden of carrying the two continuations for a long time. For such cases, we would better leave the handle and raise expressions intact. To this end, we are currently designing a static analysis that estimates the interval between a handler installation and its exception raise.

All such tunings of our selective CPS transformation are to be applied based on our experiments with realistic application programs (e.g., Isabelle, HOL, Knuth-bendix, etc.). Currently, our transformation is not compatible with separate compilation.

7 Conclusion

Processing ML exceptions forms an overhead. In SML/NJ, the overhead amounts to installing and uninstalling exception handlers in a global resource. In this work, we are exploring an alternative implementation based on a selective source-level CPS transformation. Instead of relying on one global stack of exception handlers, we pass to each function both its conventional continuation and a handler continuation accounting for exceptions. Furthermore, we reduce this continuation traffic using the static information provided by Yi and Ryu's exception analysis [YR97]. We have implemented this selective CPS transformation for Standard ML's core language and have integrated it as a separate phase in the SML/NJ compiler. (Hence our transformed programs are CPS-transformed yet again when they are compiled.) Our selective CPS transformation has been observed to improve the run times of exception-intensive programs by a factor of 2. As for programs where no exceptions are raised, it leaves them in direct style and their performance is thus the same as with SML/NJ. The more usual programs where exceptions are occasionally raised are the most challenging ones, since carrying continuations de facto forms a new overhead which is sometimes bigger than SML/NJ's. We are currently investigating this last issue.

Acknowledgment

In a personal communication to the second author, Andrew Appel suggested that the results of the exception analysis could be exploited in reducing the overhead of raise/handle expressions. The present work originates from this suggestion. Thanks are also due to the anonymous referees for perceptive comments.

References

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

[BCL+98] Edoardo Biagioni, Ken Cline, Peter Lee, Chris Okasaki, and Chris Stone. Safe-for-space threads in Standard ML. *Higher-Order and Symbolic Computation (née Lisp and Symbolic Computation)*, 11(2), 1998.

[DD95] Olivier Danvy and Dirk Dussart. CPS transformation after binding-time analysis. Unpublished note, Department of Computer Science, University of Aarhus, April 1995.

[DF92] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[DH93a] Olivier Danvy and John Hatcliff. CPS transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, 1(3):195–212, 1993.

[DH93b] Olivier Danvy and John Hatcliff. On the transformation between direct and continuation semantics. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Proceedings of the 9th Conference on Mathematical Foundations of Programming Semantics*, number 802 in Lecture Notes in Computer Science, pages 627–648, New Orleans, Louisiana, April 1993. Springer-Verlag.

[Dus97] Dirk Dussart. *Topics in program specialization and analysis for statically typed functional languages.* PhD thesis, Katholieke Universiteit Leuven, Leuven, Belgium, May 1997.

[Fel87] Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* PhD thesis, Department of Computer Science, Indiana University, Bloomington, Indiana, August 1987.

[HD97] John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. *Journal of Functional Programming*, 7(2):303–319, 1997.

[HM97] Nevin Heintze and David McAllester. Linear-time subtransitive control flow analysis. In Ron K. Cytron, editor, *Proceedings of the ACM SIGPLAN'97 Conference on Programming Languages Design and Implementation*, SIGPLAN

$$
\begin{array}{lcl}
v : \mathtt{Return} & = & \dot{\Psi}(v) \\
v : \langle K, H\rangle & = & K\ (\ \dot{\Psi}(v)\ ) \\
\underline{v} : \mathtt{Return} & = & \underline{\dot{\Psi}(v)} \\
\underline{v} : \langle K, H\rangle & = & H\ (\ \dot{\Psi}(v)\ ) \\
\dot{e}_1\ \dot{e}_2 : \mathtt{Return} & = & (e_1 : \mathtt{Return})\ \dot{T}(e_2)\ \langle \bot_K, \bot_H\rangle \\
\dot{e}_1\ \dot{e}_2 : \langle K, H\rangle & = & (e_1 : \mathtt{Return})\ \dot{T}(e_2)\ \langle K, H\rangle \\
\breve{e}_1\ \dot{e}_2 : \langle K, H\rangle & = & e_1 : \langle \lambda f.\ f\ \dot{T}(e_2)\ \langle K, H\rangle, H\rangle \\
\dot{e}_1\ \breve{e}_2 : \langle K, H\rangle & = & \\
\multicolumn{3}{l}{\quad (\lambda f.\ \breve{T}(e_2)\ \langle \lambda v.\ f\ v\ \langle K, H\rangle, H\rangle)(e_1 : \mathtt{Return})} \\
\breve{e}_1\ \breve{e}_2 : \langle K, H\rangle & = & \\
\multicolumn{3}{l}{\quad e_1 : \langle \lambda f.\ \breve{T}(e_2)\ \langle \lambda v.\ f\ v\ \langle K, H\rangle, H\rangle, H\rangle} \\
v_1\ \dot{e}_2 : \mathtt{Return} & = & \dot{\Psi}(v_1)\ (e_2 : \mathtt{Return})\ \langle \bot_K, \bot_H\rangle \\
v_1\ \dot{e}_2 : \langle K, H\rangle & = & \dot{\Psi}(v_1)\ (e_2 : \mathtt{Return})\ \langle K, H\rangle \\
v_1\ \breve{e}_2 : \langle K, H\rangle & = & e_2 : \langle \lambda v.\ \dot{\Psi}(v_1)\ v\ \langle K, H\rangle, H\rangle \\
\underline{v_1}\ \dot{e}_2 : \mathtt{Return} & = & \underline{\dot{\Psi}(v_1)} \\
\underline{v_1}\ \dot{e}_2 : \langle K, H\rangle & = & H\ (\ \dot{\Psi}(v_1)\ ) \\
\underline{v_1}\ \breve{e}_2 : \langle K, H\rangle & = & H\ (\ \dot{\Psi}(v_1)\ ) \\
v_1\ v_2 : \mathtt{Return} & = & \dot{\Psi}(v_1)\dot{\Psi}(v_2)\ \langle \bot_K, \bot_H\rangle \\
v_1\ v_2 : \langle K, H\rangle & = & \dot{\Psi}(v_1)\dot{\Psi}(v_2)\ \langle K, H\rangle \\
v_1\ \underline{v_2} : \mathtt{Return} & = & \underline{\dot{\Psi}(v_2)} \\
v_1\ \underline{v_2} : \langle K, H\rangle & = & H\ (\ \dot{\Psi}(v_2)\ ) \\
\mathtt{con}\,\kappa\ \ \dot{e} : \mathtt{Return} & = & \mathtt{con}\,\kappa\ \ (e : \mathtt{Return}) \\
\mathtt{con}\,\kappa\ \ \breve{e} : \langle K, H\rangle & = & e : \langle \lambda v.\ K\ (\ \kappa{\cdot}v\ ), H\rangle \\
\mathtt{con}\,\kappa\ \ v : \mathtt{Return} & = & \mathtt{con}\,\kappa\ \ \dot{\Psi}(v) \\
\mathtt{con}\,\kappa\ \ v : \langle K, H\rangle & = & K\ (\ \dot{\Psi}(\kappa{\cdot}v)\ ) \\
\mathtt{con}\,\kappa\ \ \underline{v} : \mathtt{Return} & = & \underline{\dot{\Psi}(v)} \\
\mathtt{con}\,\kappa\ \ \underline{v} : \langle K, H\rangle & = & H\ (\ \dot{\Psi}(v)\ ) \\
\mathtt{decon}\ \dot{e} : \mathtt{Return} & = & \mathtt{decon}\ (e : \mathtt{Return}) \\
\mathtt{decon}\ \breve{e} : \langle K, H\rangle & = & e : \langle \lambda \kappa{\cdot}v.\ K\ (\ \dot{\Psi}(v)\ ), H\rangle \\
\multicolumn{3}{l}{\mathtt{decon}\ \kappa{\cdot}v : \mathtt{Return} = \dot{\Psi}(v)} \\
\multicolumn{3}{l}{\mathtt{decon}\ \kappa{\cdot}v : \langle K, H\rangle = K\ (\ \dot{\Psi}(v)\ )} \\
\mathtt{decon}\ \underline{v} : \mathtt{Return} & = & \underline{\dot{\Psi}(v)} \\
\mathtt{decon}\ \underline{v} : \langle K, H\rangle & = & H\ (\ \dot{\Psi}(v)\ ) \\
\mathtt{raise}\ \dot{e} : \mathtt{Return} & = & \bot_H\ (e : \mathtt{Return}) \\
\mathtt{raise}\ \dot{e} : \langle K, H\rangle & = & H\ (\ e : \mathtt{Return}\ ) \\
\mathtt{raise}\ \breve{e} : \langle K, H\rangle & = & e : \langle H, H\rangle \\
\mathtt{raise}\ v : \mathtt{Return} & = & \bot_H\ \dot{\Psi}(v) \\
\mathtt{raise}\ v : \langle K, H\rangle & = & H\ (\ \dot{\Psi}(v)\ ) \\
\mathtt{raise}\ \underline{v} : \mathtt{Return} & = & \underline{\dot{\Psi}(v)} \\
\mathtt{raise}\ \underline{\breve{v}} : \langle K, H\rangle & = & H\ (\ \dot{\Psi}(v)\ ) \\
\mathtt{raise}\ \underline{\dot{v}} : \langle K, H\rangle & = & \underline{\dot{\Psi}(v)}
\end{array}
$$

$$
\begin{array}{lcl}
\mathtt{handle}\ \dot{e}_1\ \kappa\ \ \lambda x.\ \dot{e}_2 : \mathtt{Return} & = & e_1 : \mathtt{Return} \\
\mathtt{handle}\ \breve{e}_1\ \kappa\ \ \lambda x.\ \dot{e}_2 : \mathtt{Return} & = & \\
\multicolumn{3}{l}{\quad e_1 : \langle \bot_K, \lambda v.\ \mathtt{case}\ v\ \kappa\ \ (\bot_K\ (\lambda x.\ \dot{T}(e_2)\ v))\ (\bot_H\ v)\rangle} \\
\mathtt{handle}\ \breve{e}_1\ \kappa\ \ \lambda x.\ \dot{e}_2 : \langle K, H\rangle & = & \\
\multicolumn{3}{l}{\quad e_1 : \langle K, \lambda v.\ \mathtt{case}\ v\ \kappa\ \ (K\ (\lambda x.\ \dot{T}(e_2)\ v))\ (H\ v)\rangle} \\
\mathtt{handle}\ \breve{e}_1\ \kappa\ \ \lambda x.\ \breve{e}_2 : \langle K, H\rangle & = & \\
\multicolumn{3}{l}{\quad e_1 : \langle K, \lambda v.\ \mathtt{case}\ v\ \kappa\ \ (\lambda x.\ \breve{T}(e_2)\ v\ \langle K, H\rangle)\ (H\ v)\rangle} \\
\mathtt{handle}\ v_1\ \kappa\ \ \lambda x.\ \dot{e}_2 : \mathtt{Return} & = & \dot{\Psi}(v_1) \\
\mathtt{handle}\ v_1\ \kappa\ \ \lambda x.\ \dot{e}_2 : \langle K, H\rangle & = & K\ (\ \dot{\Psi}(v_1)\ ) \\
\multicolumn{3}{l}{\mathtt{handle}\ \underline{\kappa{\cdot}v_1}\ \kappa\ \ \lambda x.\ \dot{e}_2 : \mathtt{Return} = [\dot{\Psi}(v_1)/x]e_2 : \mathtt{Return}} \\
\multicolumn{3}{l}{\mathtt{handle}\ \underline{\kappa'{\cdot}v_1}\ \kappa\ \ \lambda x.\ \dot{e}_2 : \mathtt{Return} = \bot_H(\dot{\Psi}(v_1))} \\
\multicolumn{3}{l}{\mathtt{handle}\ \underline{\kappa{\cdot}v_1}\ \kappa\ \ \lambda x.\ \dot{e}_2 : \langle K, H\rangle = K([\dot{\Psi}(v_1)/x]e_2 : \mathtt{Return})} \\
\multicolumn{3}{l}{\mathtt{handle}\ \underline{\kappa'{\cdot}v_1}\ \kappa\ \ \lambda x.\ \dot{e}_2 : \langle K, H\rangle = H(\dot{\Psi}(v_1))} \\
\multicolumn{3}{l}{\mathtt{handle}\ \underline{\kappa{\cdot}v_1}\ \kappa\ \ \lambda x.\ \breve{e}_2 : \langle K, H\rangle = K([\dot{\Psi}(v_1)/x]\breve{T}(e_2)\ \langle K, H\rangle)} \\
\multicolumn{3}{l}{\mathtt{handle}\ \underline{\kappa'{\cdot}v_1}\ \kappa\ \ \lambda x.\ \breve{e}_2 : \mathtt{Return} = H(\dot{\Psi}(v_1))} \\
\mathtt{case}\ \dot{e}_1\ \kappa\ \dot{e}_2\ \dot{e}_3 : \mathtt{Return} & = & \\
\multicolumn{3}{l}{\quad \mathtt{case}\ (\dot{e}_1 : \mathtt{Return})\ \kappa\ \dot{T}(\dot{e}_2)\ \dot{T}(\dot{e}_3)} \\
\mathtt{case}\ \dot{e}_1\ \kappa\ \dot{e}_2\ \breve{e}_3 : \langle K, H\rangle & = & \\
\multicolumn{3}{l}{\quad \mathtt{case}\ (\dot{e}_1 : \mathtt{Return})\ \kappa\ \dot{T}(\dot{e}_2)\ \breve{T}(\dot{e}_3)\ \langle K, H\rangle} \\
\mathtt{case}\ \dot{e}_1\ \kappa\ \breve{e}_2\ \dot{e}_3 : \langle K, H\rangle & = & \\
\multicolumn{3}{l}{\quad \mathtt{case}\ (\dot{e}_1 : \mathtt{Return})\ \kappa\ \breve{T}(\dot{e}_2)\ \langle K, H\rangle\ \dot{T}(\dot{e}_3)} \\
\mathtt{case}\ \breve{e}_1\ \kappa\ \dot{e}_2\ \dot{e}_3 : \langle K, H\rangle & = & \\
\multicolumn{3}{l}{\quad \breve{e}_1 : \langle \lambda v.\ \mathtt{case}\ v\ \kappa\ \dot{T}(\dot{e}_2)\ \dot{T}(\dot{e}_3), H\rangle} \\
\mathtt{case}\ \dot{e}_1\ \kappa\ \breve{e}_2\ \breve{e}_3 : \langle K, H\rangle & = & \\
\multicolumn{3}{l}{\quad \mathtt{case}\ (\dot{e}_1 : \mathtt{Return})\ \kappa\ \breve{T}(\dot{e}_2)\ \langle K, H\rangle\ \breve{T}(\dot{e}_3)\ \langle K, H\rangle} \\
\mathtt{case}\ \breve{e}_1\ \kappa\ \dot{e}_2\ \breve{e}_3 : \langle K, H\rangle & = & \\
\multicolumn{3}{l}{\quad \breve{e}_1 : \langle \lambda v.\ \mathtt{case}\ v\ \kappa\ \dot{T}(\dot{e}_2)\ \breve{T}(\dot{e}_3)\ \langle K, H\rangle, H\rangle} \\
\mathtt{case}\ \breve{e}_1\ \kappa\ \breve{e}_2\ \dot{e}_3 : \langle K, H\rangle & = & \\
\multicolumn{3}{l}{\quad \breve{e}_1 : \langle \lambda v.\ \mathtt{case}\ v\ \kappa\ \breve{T}(\dot{e}_2)\ \langle K, H\rangle\ \dot{T}(\dot{e}_3), H\rangle} \\
\mathtt{case}\ \breve{e}_1\ \kappa\ \breve{e}_2\ \breve{e}_3 : \langle K, H\rangle & = & \\
\multicolumn{3}{l}{\quad \breve{e}_1 : \langle \lambda v.\ \mathtt{case}\ v\ \kappa\ \breve{T}(\dot{e}_2)\ \langle K, H\rangle\ \breve{T}(\dot{e}_3)\ \langle K, H\rangle, H\rangle} \\
\mathtt{case}\ \kappa{\cdot}v_1\ \kappa\ \dot{e}_2\ \dot{e}_3 : \mathtt{Return} & = & \dot{e}_2 : \mathtt{Return} \\
\mathtt{case}\ \kappa{\cdot}v_1\ \kappa\ \dot{e}_2\ \dot{e}_3 : \langle K, H\rangle & = & \dot{e}_2 : \langle K, H\rangle \\
\mathtt{case}\ \kappa{\cdot}v_1\ \kappa\ \breve{e}_2\ \dot{e}_3 : \langle K, H\rangle & = & \dot{e}_2 : \langle K, H\rangle \\
\mathtt{case}\ \kappa{\cdot}v_1\ \kappa\ \dot{e}_2\ \breve{e}_3 : \langle K, H\rangle & = & K(\dot{e}_2 : \mathtt{Return}) \\
\mathtt{case}\ \kappa{\cdot}v_1\ \kappa\ \breve{e}_2\ \breve{e}_3 : \langle K, H\rangle & = & \dot{e}_2 : \langle K, H\rangle \\
\mathtt{case}\ \kappa'{\cdot}v_1\ \kappa\ \dot{e}_2\ \dot{e}_3 : \mathtt{Return} & = & \dot{e}_3 : \mathtt{Return} \\
\mathtt{case}\ \kappa'{\cdot}v_1\ \kappa\ \dot{e}_2\ \dot{e}_3 : \langle K, H\rangle & = & \dot{e}_3 : \langle K, H\rangle \\
\mathtt{case}\ \kappa'{\cdot}v_1\ \kappa\ \breve{e}_2\ \dot{e}_3 : \langle K, H\rangle & = & K(\dot{e}_3 : \mathtt{Return}) \\
\mathtt{case}\ \kappa'{\cdot}v_1\ \kappa\ \dot{e}_2\ \breve{e}_3 : \langle K, H\rangle & = & \breve{e}_3 : \langle K, H\rangle \\
\mathtt{case}\ \kappa'{\cdot}v_1\ \kappa\ \breve{e}_2\ \breve{e}_3 : \langle K, H\rangle & = & \breve{e}_3 : \langle K, H\rangle \\
\mathtt{case}\ \underline{\kappa'{\cdot}v_1}\ \kappa\ \dot{e}_2\ \dot{e}_3 : \mathtt{Return} & = & \underline{\kappa'{\cdot}v_1} \\
\mathtt{case}\ \underline{\kappa'{\cdot}v_1}\ \kappa\ e_2\ e_3 : \langle K, H\rangle & = & K(\underline{\kappa'{\cdot}v_1})
\end{array}
$$

Figure 12: Definition of the infix operator ‘:’

Notices, Vol. 32, No 5, pages 261–272, Las Vegas, Nevada, June 1997. ACM Press.

[JW96] Suresh Jagannathan and Andrew Wright. Flow-directed inlining. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 31, No 5, pages 192–205. ACM Press, May 1996.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, September 1981.

[Ses89] Peter Sestoft. Replacing function parameters by global variables. In Joseph E. Stoy, editor, *Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture*, pages 39–53, London, England, September 1989. ACM Press.

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

[SW94] Paul Steckler and Mitchell Wand. Selective thunkification. In Baudouin Le Charlier, editor, *Static Analysis*, number 864 in Lecture Notes in Computer Science, pages 162–178, Namur, Belgium, September 1994. Springer-Verlag.

[YR97] Kwangkeun Yi and Sukyoung Ryu. Towards a cost-effective estimation of uncaught exceptions in SML programs. In Pascal Van Hentenryck, editor, *Static Analysis*, number 1302 in Lecture Notes in Computer Science, pages 98–113, Paris, France, September 1997. Springer-Verlag.