Higher-Order and Symbolic Computation, 5, 275–307 (1999)

Unchecked Exceptions can be Strictly More Powerful than Call/CC *

MARK LILLIBRIDGE ** mdl@pa.dec.com
Systems Research Center
Compaq Computer Corporation
130 Lytton Avenue
Palo Alto, CA 94301

Received November 25, 1995; Revised March 30, 1996

Editor: Olivier Danvy

Abstract. We demonstrate that in the context of statically-typed purely-functional lambda calculi without recursion, unchecked exceptions (e.g., SML exceptions) can be strictly more powerful than call/cc. More precisely, we prove that a natural extension of the simply-typed lambda calculus with unchecked exceptions is strictly more powerful than all known sound extensions of Girard's F_ω (a superset of the simply-typed lambda calculus) with call/cc.

This result is established by showing that the first language is Turing complete while the later languages permit only a subset of the recursive functions to be written. We show that our natural extension of the simply-typed lambda calculus with unchecked exceptions is Turing complete by reducing the untyped lambda calculus to it by means of a novel method for simulating recursive types using unchecked-exception–returning functions. The result concerning extensions of F_ω with call/cc stems from previous work of the author and Robert Harper.

Keywords: Studies of programming constructs, control primitives, exceptions, recursion, λ-calculus, type theory, functional programming

1. Introduction

The relationship between the programming features of *exceptions* and *call/cc* (call with current continuation) in statically-typed purely-functional programming languages has been an open question for some time. Carl Gunter and colleagues write in a recent paper [11]:

> It is folklore (the authors know of no published proof) that neither [unchecked] exceptions nor continuations can be expressed as a macro in terms of the other (at least if no references are present), even though they are closely related.

Exceptions in statically-typed programming languages come in two kinds, *checked* and *unchecked*. The difference lies in the fact that the type of a function must

* This research was sponsored by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168.
** Supported by a National Science Foundation Graduate Fellowship while this work was done.

declare only the checked exceptions that it may throw, not the unchecked ones. Most programming languages provide only one kind of exception. For example, all user-declared exceptions in CLU [16] are checked exceptions[1] while SML [17] provides only unchecked exceptions. Java [2], from which this terminology comes, is unusual in providing both kinds.

In this paper we demonstrate that under reasonable assumptions unchecked exceptions cannot be expressed as a macro using only call/cc in a statically-typed purely-functional lambda calculi without recursion, thus partially answering half of Gunter *et al.*'s open question. Left open is the question of whether or not unchecked exceptions can be defined using a macro in terms of call/cc and either a fixpoint operator, recursive types, or some similar feature.

We do this by showing that when call/cc is added in a sound manner to even as powerful a statically-typed purely-functional lambda calculus as Girard's F_ω [10, 22], the set of functions expressible in the resulting language is still a subset of the recursive functions, and that a natural extension with unchecked exceptions of even so limited a language as the simply-typed lambda calculus ($\lambda^{\rightarrow}$) is Turing complete; that is, it permits all computable functions to be expressed. (In particular, unlike in the first case, potentially non-terminating functions can be written.) This demonstrates that unchecked exceptions are strictly more powerful than call/cc for statically-typed purely-functional lambda calculi without recursion — even allowing arbitrary transformations on programs instead of macros cannot reduce unchecked exceptions to call/cc.

The first of these results is a previous result of the author with Robert Harper [13]. The second is new to this paper and involves a novel method for simulating recursive types using unchecked-exception–raising functions.

Future work on answering Gunter *et al.*'s question will require studying explicitly what can be defined using a macro because the approach taken in this paper of comparing language power (what functions a language can denote) cannot produce negative results when the base language is already Turing complete, as is the case when recursion is present. Fellesisen's work [7] provides an excellent foundation for this task. Although it was not Gunter *et al.*'s primary concern, Andrzej Filinski has shown recently that it is possible to define unchecked exceptions in terms of call/cc and references [9].

2. The Power of Call/CC

Like assignment, both call/cc and exceptions can have non-local effects. Adding non-local effects to a lambda calculus requires specifying an evaluation order in order to retain determinism. Consider, for example, the following untyped code fragment:

```
call/cc (λk. f(k 3, k 4))
```

If we evaluate arguments left-to-right, then this expression evaluates to `3`; if we instead evaluate right-to-left, then it evaluates to `4`. Each different evaluation order yields a different extension of F_ω with call/cc.

The author and Robert Harper considered a number of these extensions in a recent paper [13] obtaining a number of results, which we summarize briefly here: Four evaluation strategies were considered, differing in whether they use call-by-name or call-by-value parameter passing and in whether or not they evaluate beneath type abstractions ($\Lambda\alpha{::}K.M$). We know of no other proposed evaluation strategies for F_ω extended with non-local effects.

Not evaluating beneath type abstractions treats type instantiation ($M[A]$) as a significant computation step, possibility including effects. Strategies of this type are used in Quest [3] and LEAP [21], and are directly compatible with extensions that make significant uses of types at run time [14] (e.g., "dynamic" types [1, 3]). Since polymorphic expressions are kept distinct from their instances, the anomalies that arise in implicitly polymorphic languages in the presence of references [25] and control operators [12] do not arise.

Strategies that evaluate beneath type abstractions are inspired by the operational semantics of ML [5, 17]. Evaluation proceeds beneath type abstractions, leading to a once-for-all-instances evaluation of polymorphic terms. Type instantiation is retained as a computation step, but its force is significantly attenuated by the fact that type expressions may contain free type variables, precluding primitives that inductively analyze their type arguments. The superficial efficiency improvement gained by evaluating beneath type abstractions comes at considerable cost since it is incompatible with extensions such as mutable data structures and control operators [25, 12].

The call/cc operator takes one normal argument, which it then calls with the current continuation; if the argument returns, then its value is the result of the call/cc expression [4]. For example (ignoring types), **call/cc** `(λf.3)` yields 3 and **call/cc** `(λf.f 4; 3)` yields 4. The obvious type for call/cc in F_ω was used: [2]

$$\textbf{call/cc} \;:\; \forall\alpha{:}\Omega.\ ((\forall u{:}\Omega.\alpha{\to}u){\to}\alpha){\to}\alpha$$

In English, this means that **call/cc**`[`A`]` M has type A when M is a function that takes an A-accepting continuation and returns a value of type A, where A-accepting continuations have type $\forall u{:}\Omega.A{\to}u$, reflecting the fact that they do not return.

This typing does not distinguish continuations from normal functions. If desired, an abstract type constructor (e.g., α **cont**) can be used to keep these separate. The only effect of this change is to rule out some programs, so it can only decrease the power of F_ω plus call/cc.

F_ω extended with call/cc was shown to be sound under all the strategies except for the strategy most like ML (the call-by-value, evaluating beneath type abstractions strategy) which was shown to be unsound for full F_ω. Restricting so that polymorphism can only be used on values, not general expressions,[3] restores soundness for this strategy. This restriction has since been adopted for SML where it is called the value restriction [18].

Typed CPS (continuation-passing style) transforms were then given for each strategy from the appropriate sound subset into F_ω and proven correct. The proofs of correctness showed that if M terminates with value V then the transformed version of M terminates with a (possibly further reduced) transformed version of V. This

was done by showing that each reduction step on the original term corresponds to at least one reduction step of the transformed term. Since F_ω is known to be strongly normalizing (see [10]), this means that all programs in the sound subsets must terminate. Hence, adding call/cc to F_ω in a sound manner permits only recursive functions to be written; F_ω plus call/cc is therefore not Turing complete.

It should be noted that because the simply-typed lambda calculus ($\lambda^{\rightarrow}$), the polymorphic lambda calculus (F_2), and the core of ML, *Core*-ML [19, 5], are proper subsets of F_ω that this result applies to adding call/cc to them as well.

3. A Language with Unchecked Exceptions

Unlike the case with call/cc, we will need to consider only one language with unchecked exceptions. To maximize the strength of our results, we choose to add unchecked exceptions to the weakest typed lambda calculus, the simply-typed lambda calculus ($\lambda^{\rightarrow}$). Because exceptions can have non-local effects, we must fix an evaluation order before we can extend $\lambda^{\rightarrow}$. We choose to use call-by-value (CBV) evaluation as it seems to be the most natural choice in this setting. (The question of whether or not to evaluate under type abstractions does not arise here because there are no type abstractions in $\lambda^{\rightarrow}$.)

3.1. The CBV Simply-Typed Lambda Calculus ($\boldsymbol{\lambda_v^{\rightarrow}}$)

We define the syntax of $\lambda_v^{\rightarrow}$ as follows:

$$
\begin{array}{lrcl}
\textit{Types} & A, B & ::= & \mathbf{unit} \mid A_1 \rightarrow A_2 \mid A_1 + A_2 \\
\textit{Terms} & M & ::= & x \mid () \mid \lambda x{:}A.M \mid M_1\, M_2 \mid \mathbf{left}_A(M) \mid \mathbf{right}_A(M) \mid \\
 & & & M \ \mathbf{of}\ \mathbf{left}(x_1{:}A_1) => M_1;\ \ \mathbf{right}(x_2{:}A_2) => M_2 \\
 & & & \\
\textit{Assignments} & \Gamma & ::= & \emptyset \mid \Gamma, x{:}A
\end{array}
$$

The meta-variables x and y range over *term variables*. As usual, we identify α-convertible terms and types. We use $\mathrm{FV}(M)$ to denote the free term variables of M and $[M/x]M'$ for the usual capture-avoiding substitution of M for x in M'.

In addition to the usual first-class functions ($\lambda x{:}A.M$), we have included a singleton type **unit** with sole value () and disjoint sums (with type $A+B$). The disjoint sums are not necessary for the primary results of this paper; we have included them for use in Section 4.1, which provides motivation for the encoding of recursive types using unchecked exceptions. The **unit** type is included so that we have at least one base type; any other base type with at least one value would work as well.

The type system of $\lambda_v^{\rightarrow}$ consists of a set of rules for deriving judgements of the following forms:

$$
\begin{array}{ll}
\triangleright\, \Gamma & \textit{well-formed assignment} \\
\Gamma \triangleright M : A & \textit{well-formed term}
\end{array}
$$

The rules for deriving these judgements are standard and can be found in Figure 1.

$$\triangleright\, \emptyset \qquad \text{(A-EMPTY)}$$

$$\frac{\triangleright\, \Gamma \quad x \notin \mathrm{dom}(\Gamma)}{\triangleright\, \Gamma, x{:}A} \qquad \text{(A-EXTEND)}$$

$$\frac{\triangleright\, \Gamma}{\Gamma \triangleright x : \Gamma(x)} \qquad \text{(T-VAR)}$$

$$\frac{\triangleright\, \Gamma}{\Gamma \triangleright () : \mathbf{unit}} \qquad \text{(T-UNIT)}$$

$$\frac{\Gamma, x{:}A_1 \triangleright M : A_2}{\Gamma \triangleright \lambda x{:}A_1.M : A_1 \to A_2} \qquad \text{(T-ABS)}$$

$$\frac{\Gamma \triangleright M_1 : A_2 \to A \quad \Gamma \triangleright M_2 : A_2}{\Gamma \triangleright M_1\, M_2 : A} \qquad \text{(T-APP)}$$

$$\frac{\Gamma \triangleright M : A}{\Gamma \triangleright \mathbf{left}_{A+B}(M) : A{+}B} \qquad \text{(T-LEFT)}$$

$$\frac{\Gamma \triangleright M : B}{\Gamma \triangleright \mathbf{right}_{A+B}(M) : A{+}B} \qquad \text{(T-RIGHT)}$$

$$\frac{\Gamma \triangleright M : A_1{+}A_2 \quad \Gamma, x_1{:}A_1 \triangleright M_1 : B \quad \Gamma, x_2{:}A_2 \triangleright M_2 : B}{\Gamma \triangleright M\ \mathbf{of\ left}(x_1{:}A_1) => M_1;\ \ \mathbf{right}(x_2{:}A_2) => M_2 : B} \qquad \text{(T-CASE)}$$

Figure 1. Typing rules for $\lambda_v^{\rightarrow}$

A *context* C is an expression of $\lambda_v^{\rightarrow}$ with a single *hole*, written $[]$:

$$\begin{array}{rcl} \textit{Contexts}\ \ C & ::= & [] \mid \lambda x{:}A.\, C \mid C\, M \mid M\, C \mid \mathbf{left}_A(C) \mid \mathbf{right}_A(C) \mid \\ & & C\ \mathbf{of}\ \mathbf{left}(x_1{:}A_1) \Rightarrow M_1;\ \ \mathbf{right}(x_2{:}A_2) \Rightarrow M_2 \mid \\ & & M\ \mathbf{of}\ \mathbf{left}(x_1{:}A_1) \Rightarrow C;\ \ \mathbf{right}(x_2{:}A_2) \Rightarrow M_2 \mid \\ & & M\ \mathbf{of}\ \mathbf{left}(x_1{:}A_1) \Rightarrow M_1;\ \ \mathbf{right}(x_2{:}A_2) \Rightarrow C \end{array}$$

The hole in C may be *filled* by an expression M, written $C[M]$, by replacing the hole with M, incurring capture of free variables in M that are bound at the occurrence of the hole. For example, if $C = \lambda x{:}\mathbf{unit}.[]$, and $M = f\ x$, then $C[M] = \lambda x{:}\mathbf{unit}.f\ x$. The variables that are bound within the hole of a context are said to be *exposed (to capture)* by that context. We write $EV(C)$ for the exposed variables of a context.

Type checking in $\lambda_v^{\rightarrow}$ is compositional in the sense that if an expression is well-typed, then so are all its constituent expressions:

LEMMA 1 (DECOMPOSITION) *Suppose that* $\lambda_v^{\rightarrow} \vdash \Gamma \triangleright C[M] : A$ *such that* $EV(C) \cap dom(\Gamma) = \emptyset$.[4] *Then there exists* Γ', *and* B *such that:*

- $dom(\Gamma') = EV(C)$
- $\lambda_v^{\rightarrow} \vdash \Gamma, \Gamma' \triangleright M : B$

Proof: Routine induction on the structure of contexts. ■

Furthermore, only the type of a constituent of a well-formed expression is relevant to typing. Consequently any constituent may be replaced by a term of the same type without affecting typability of the whole expression:

LEMMA 2 (REPLACEMENT) *Suppose that* $\lambda_v^{\rightarrow} \vdash \Gamma \triangleright C[M] : A$, *with* $\lambda_v^{\rightarrow} \vdash \Gamma, \Gamma' \triangleright M : B$ *in accordance with the decomposition lemma. If* $\lambda_v^{\rightarrow} \vdash \Gamma, \Gamma'', \Gamma' \triangleright M' : B$ *then* $\lambda_v^{\rightarrow} \vdash \Gamma, \Gamma'' \triangleright C[M'] : A$.

Proof: Routine induction on typing derivations. ■

Following Plotkin [23] and Felleisen [8], we specify an operational semantics by defining the set of *values* (V), the set of *evaluation contexts* (E), and the *one step evaluation relation* for that semantics. One-step evaluation is a binary relation on programs that is defined by a set of rules of the form $E[R] \hookrightarrow M$, where E is an evaluation context, R is an expression, the *redex*, and M is determined as a function of E and R. In this case $E[R]$ is said to *evaluate to* M *in one step*. We define $\hookrightarrow^+$ to be the irreflexive, transitive closure and $\hookrightarrow^*$ to be the reflexive, transitive closure of the $\hookrightarrow$ relation.

A *program* is a well-typed closed term. We will arrange things so that a program P is either a value or can be represented in exactly one way as $E[R]$ where E is an evaluation context and R is a redex. Because our evaluation relations will be functions of E and R, this means that our languages will be deterministic. Using this method, the call-by-value semantics of $\lambda_v^{\rightarrow}$ are defined as follows:

$$
\begin{array}{rcl}
V & ::= & x \mid () \mid \lambda x{:}A.M \mid \mathbf{left}_A(V) \mid \mathbf{right}_A(V) \\
E & ::= & [] \mid E\,M \mid V\,E \mid \mathbf{left}_A(E) \mid \mathbf{right}_A(E) \mid \\
 & & E\ \mathbf{of}\ \mathbf{left}(x_1{:}A_1) \Rightarrow M_1;\ \ \mathbf{right}(x_2{:}A_2) \Rightarrow M_2 \\
R & ::= & (\lambda x{:}A.M)\,V \mid \\
 & & \mathbf{left}_B(V)\ \mathbf{of}\ \mathbf{left}(x_1{:}A_1) \Rightarrow M_1;\ \ \mathbf{right}(x_2{:}A_2) \Rightarrow M_2 \mid \\
 & & \mathbf{right}_B(V)\ \mathbf{of}\ \mathbf{left}(x_1{:}A_1) \Rightarrow M_1;\ \ \mathbf{right}(x_2{:}A_2) \Rightarrow M_2
\end{array}
$$

$$
\begin{array}{rcl}
E[(\lambda x{:}A.M)\,V] & \hookrightarrow & E[[V/x]M] \\
E[\mathbf{left}_B(V)\ \mathbf{of}\ \mathbf{left}(x_1{:}A_1) \Rightarrow M_1;\ \ \mathbf{right}(x_2{:}A_2) \Rightarrow M_2] & \hookrightarrow & E[V/x_1]M_1] \\
E[\mathbf{right}_B(V)\ \mathbf{of}\ \mathbf{left}(x_1{:}A_1) \Rightarrow M_1;\ \ \mathbf{right}(x_2{:}A_2) \Rightarrow M_2] & \hookrightarrow & E[V/x_2]M_2]
\end{array}
$$

In the absence of case expressions, these rules specify that the leftmost, outermost term of the form $(\lambda x{:}A.M)\,V$ may be reduced in place by replacing it with $[V/x]M$. Note that the case expression first evaluates its first argument then evaluates exactly one of its branches. Sequential execution $(M_1;\ M_2)$ can be obtained by writing $(\lambda x{:}A_1.M_2)M_1$, where A_1 is the type of M_1 and x is not free in M_2. The definition of term variables (x) as values is for the benefit of the transformation rules later in this paper.

THEOREM 1 (PROGRESS) *If M is a closed, well-typed expression of type A, then either M is a value, or else there exist a unique evaluation context E and unique redex R such that $M = E[R]$.*

Proof: The proof proceeds by induction on the structure of typing derivations using Lemma 1. ■

LEMMA 3 (SUBSTITUTION) *If $\lambda_v^{\rightarrow} \vdash \Gamma, x{:}A, \Gamma' \triangleright M' : B$ and $\lambda_v^{\rightarrow} \vdash \Gamma \triangleright M : A$ then $\lambda_v^{\rightarrow} \vdash \Gamma, \Gamma' \triangleright [M/x]M' : B$.*

Proof: Routine induction on typing derivations. ■

THEOREM 2 (SUBJECT REDUCTION) *If $\lambda_v^{\rightarrow} \vdash \Gamma \triangleright M : A$ and $M \hookrightarrow N$, then $\lambda_v^{\rightarrow} \vdash \Gamma \triangleright N : A$.*

Proof: By Theorem 1, $M = E[R]$ for some unique evaluation context E and redex R. Hence, by Lemma 1 there exists a type B and assignment Γ' such that $\lambda_v^{\rightarrow} \vdash \Gamma, \Gamma' \triangleright R : B$. By inspecting the definition of the one-step evaluation rules, we see that $N = E[N']$ and $R \hookrightarrow N'$ for some N'. By Lemma 2, we need to show only that $\lambda_v^{\rightarrow} \vdash \Gamma, \Gamma' \triangleright N' : B$.

We proceed by cases on the form of R. If $R = (\lambda x{:}A'.M')\,V$, then $\lambda_v^{\rightarrow} \vdash \Gamma, \Gamma', x{:}A' \triangleright M' : B$, $\lambda_v^{\rightarrow} \vdash \Gamma, \Gamma' \triangleright V : A'$, and $N' = [V/x]M'$ by inspection; hence, $\lambda_v^{\rightarrow} \vdash \Gamma, \Gamma' \triangleright N' : B$ by Lemma 3 as required. The remaining cases are handled similarly. ■

We will be proving versions of these theorems for many different systems. We will omit their proofs when they are essentially the same as the ones for $\lambda_v^{\rightarrow}$.

3.2. $\boldsymbol{\lambda_v^{\rightarrow}}$ extended with unchecked exceptions ($\boldsymbol{\lambda_v^{\rightarrow,E}}$)

We chose to add exceptions to $\lambda_v^{\rightarrow}$ by introducing a family of unchecked exceptions parameterized by the type of the values they carry. For each type A, there is an associated unchecked exception flavor with an operation to raise an exception of that flavor ($\mathbf{raise}_A(M)$) and an operation to catch and handle just exceptions of that flavor (M **handle** $x{:}A \Rightarrow M'$).

Unlike in SML, it is not possible to have different flavors of A-carrying exceptions in our system. A more powerful exception system than the one we describe here would have the ability to dynamically generate new exception flavors at runtime. We do not need this additional power for any of the results in this paper. Indeed, we shall see that with the exception of the simulation of recursive types using unchecked exceptions result in Section 4.2, we will need only one exception flavor to establish our results. This fact means that our primary result about exceptions ($\lambda_v^{\rightarrow,E}$ is Turing Complete) can be extended to systems like SML that require exception flavors to be declared before they can be used.

Note that because functions are first-class in $\lambda^{\rightarrow}$, exceptions may carry functional values. Our results depend on this ability. While this ability is seldom used in applicative programming, its analog is used often in object-oriented programming (objects can be thought of as a form of multiple-entry closure).

The syntax of $\lambda_v^{\rightarrow,E}$ is an extension of that of $\lambda_v^{\rightarrow}$:

$$\mathit{Terms}\ \ M\ ::=\ \ldots \mid \mathbf{raise}_A(M) \mid M\ \mathbf{handle}\ x{:}A \Rightarrow M'$$

No new types are required, but a new type rule is required for each of the new operators:

$$\frac{\Gamma \triangleright M : A}{\Gamma \triangleright \mathbf{raise}_A(M) : B} \qquad \text{(T-RAISE)}$$

$$\frac{\Gamma \triangleright M : B \quad \Gamma, x{:}A \triangleright M' : B}{\Gamma \triangleright M\ \mathbf{handle}\ x{:}A \Rightarrow M' : B} \qquad \text{(T-HANDLE)}$$

Note that $\mathbf{raise}_A(M)$ may be assigned any type since it never returns normally. The properties of decomposition and replacement remain true:

$$\begin{aligned}\mathit{Contexts}\ \ C\ ::=\ \ & \ldots \mid \mathbf{raise}_A(C) \mid C\ \mathbf{handle}\ x{:}A \Rightarrow M \mid \\ & M\ \mathbf{handle}\ x{:}A \Rightarrow C\end{aligned}$$

LEMMA 4 (DECOMPOSITION) *Suppose that $\lambda_v^{\rightarrow,E} \vdash \Gamma \triangleright C[M] : A$ such that $EV(C) \cap dom(\Gamma) = \emptyset$. Then there exists Γ' and B such that:*

- $dom(\Gamma') = EV(C)$
- $\lambda_v^{\rightarrow,E} \vdash \Gamma, \Gamma' \triangleright M : B$

LEMMA 5 (REPLACEMENT) *Suppose that* $\lambda_v^{\rightarrow,E} \vdash \Gamma \triangleright C[M] : A$, *with* $\lambda_v^{\rightarrow,E} \vdash \Gamma, \Gamma' \triangleright M : B$ *in accordance with the decomposition lemma. If* $\lambda_v^{\rightarrow,E} \vdash \Gamma, \Gamma'', \Gamma' \triangleright M' : B$ *then* $\lambda_v^{\rightarrow,E} \vdash \Gamma, \Gamma'' \triangleright C[M'] : A$.

As expected, the semantics of $\lambda_v^{\rightarrow,E}$ is an extension of that of $\lambda_v^{\rightarrow}$. Because execution of an $\lambda_v^{\rightarrow,E}$ program can produce an uncaught exception instead of a value, it is necessary to generalize our semantic framework so that programs evaluate to outputs (O), a superset of the values (V). The definitions for $\lambda_v^{\rightarrow,E}$'s semantics follows. F here represents a single frame of an evaluation context that is not a handle expression. We use it to combine all the cases where we propagate a exception up through a non-handle expression into one rule.

$$
\begin{array}{rcl}
O & ::= & V \mid \mathbf{raise}_A(V) \\
V & ::= & \ldots \\
E & ::= & \ldots \mid \mathbf{raise}_A(E) \mid E \ \mathbf{handle}\ x{:}A \Rightarrow M
\end{array}
$$

$$
\begin{array}{rcl}
R & ::= & \ldots \mid F[\mathbf{raise}_A(V)] \mid O \ \mathbf{handle}\ x{:}A \Rightarrow M \\
F & ::= & [\,]\, M \mid V\, [\,] \mid \mathbf{left}_A([\,]) \mid \mathbf{right}_A([\,]) \mid \mathbf{raise}_A([\,]) \mid \\
 & & [\,]\ \mathbf{of}\ \mathbf{left}(x_1{:}A_1) \Rightarrow M_1;\ \ \mathbf{right}(x_2{:}A_2) \Rightarrow M_2
\end{array}
$$

$$
\begin{array}{rcll}
\ldots & \hookrightarrow & \ldots & \\
E[F[\mathbf{raise}_A(V)]] & \hookrightarrow & E[\mathbf{raise}_A(V)] & \\
E[V \ \mathbf{handle}\ x{:}A \Rightarrow M] & \hookrightarrow & E[V] & \\
E[(\mathbf{raise}_A(V)) \ \mathbf{handle}\ x{:}B \Rightarrow M] & \hookrightarrow & E[[V/x]M] & (A = B) \\
E[(\mathbf{raise}_A(V)) \ \mathbf{handle}\ x{:}B \Rightarrow M] & \hookrightarrow & E[\mathbf{raise}_A(V)] & (A \neq B)
\end{array}
$$

THEOREM 3 (PROGRESS) *If M is a closed, well-typed expression of type A, then either M is an output, or else there exist a unique evaluation context E and unique redex R such that $M = E[R]$.*

THEOREM 4 (SUBJECT REDUCTION) *If* $\lambda_v^{\rightarrow,E} \vdash \Gamma \triangleright M : A$ *and* $M \hookrightarrow N$, *then* $\lambda_v^{\rightarrow,E} \vdash \Gamma \triangleright N : A$.

4. The Power of Unchecked Exceptions

In this section we prove that $\lambda_v^{\rightarrow,E}$ is Turing complete. We start by motivating the reduction we will use.

4.1. Motivation

It is standard practice when giving the semantics of untyped programming languages such as Scheme [4], to explain unchecked exceptions by use of a transform similar in spirit to a CPS transform wherein expressions returning a value of "type" A are transformed to expressions that return a value of "type" $A + \sigma$, where σ is

the "type" of all the values that may be carried by exceptions. If the original expression evaluates normally to a value V of type A then the transformed expression evaluates to the value $\mathbf{left}_{\langle A\rangle+\langle\sigma\rangle}(V')$, where V', $\langle A\rangle$, and $\langle\sigma\rangle$ are suitably transformed versions of V, A, and σ. If, on the other hand, the original expression when evaluated raises an uncaught exception carrying the value V of type σ then the transformed expression evaluates to $\mathbf{right}_{\langle A\rangle+\langle\sigma\rangle}(V')$.

The control flow of the original program is then simulated explicitly using case expressions. For example, the translation of $\mathbf{raise}(M)$ would first run the transformed code for M then case on the result. If it was a left value, then we would return it after first switching its tag to right. This corresponds to M returning a value normally, which is then turned into an exception carrying that value. If the result was a right value, we would return it untouched. This corresponds to M raising an exception, which then passes through the raise expression uncaught.

4.1.1. Exceptions carrying base types Such a transform is easily written in the statically-typed case where there is only one flavor of exception and where σ, the type of values it carries, is a base type (b). For simplicity, let us consider transforming the subset of $\lambda_v^{\rightarrow,E}$ that has no disjoint sums and only has exceptions carrying values of type σ. Terms from this language, which we call σ-only, may not include $\mathbf{left}_A(M)$, $\mathbf{right}_A(M)$, or M **of left**$(x{:}A) => M_1$; **right**$(y{:}B) => M_2$ as sub-terms and may include raise or handle sub-terms of only the form $\mathbf{raise}_\sigma(M)$ or M **handle** $x{:}\sigma => M'$. σ-only is closed under $\lambda_v^{\rightarrow,E}$ evaluation:

LEMMA 6 (CLOSURE) *If M is a σ-only term and $M \hookrightarrow N$ under $\lambda_v^{\rightarrow,E}$ then N is a σ-only term.*

Proof: Inspection of the one-step evaluation relation for $\lambda_v^{\rightarrow,E}$ shows that it maps σ-only terms to σ-only terms. ■

A transform of this kind from σ-only to $\lambda_v^{\rightarrow}$ for the case when σ is a base type can be found in Figure 2. Unless we say otherwise, σ will be assumed to be a base type throughout the rest of this section (Section 4.1.1).

The main transform is computed by the judgement $\Gamma \triangleright M : A \Rightarrow M'$, which says that the σ-only computation M of type A transforms to the $\lambda_v^{\rightarrow}$-term M', which will have type $[A]$ under $\langle -\rangle$ applied pointwise to Γ ($\langle\Gamma\rangle$). A sub-judgement, $\Gamma \triangleright V : A \Rightarrow_v V'$, is used to transform σ-only values into $\lambda_v^{\rightarrow}$ values with type $\langle A\rangle$ under $\langle\Gamma\rangle$. A number of macros used by the transform can be found in Figure 3.

Note that the transformation of a value differs from the transformation of the computation that immediately returns that value because only the later will be wrapped in a left tag. The B-APP-V and B-RAISE-V rules are specialized versions of the B-APP and B-RAISE rules that produce more optimized code for the case when M_1 or M is a value. This optimization is needed to ensure that one $\lambda_v^{\rightarrow,E}$ evaluation step on a σ-only term is simulated by one or more $\lambda_v^{\rightarrow}$ evaluation steps on the transformed version of the term (see Theorem 5 below). Without this simulation result, it is much harder to prove the correctness of the transforms.

$$\begin{array}{rcl} \langle b \rangle & = & b \\ \langle A_1 + A_2 \rangle & = & \langle A_1 \rangle + \langle A_2 \rangle \\ \langle A_1 \to A_2 \rangle & = & \langle A_1 \rangle \to [A_2] \\ \\ [A] & = & \langle A \rangle + \langle \sigma \rangle \end{array}$$

$$\frac{\triangleright\, \Gamma}{\Gamma \triangleright x : \Gamma(x) \Rightarrow_v x} \quad \text{(BV-VAR)}$$

$$\frac{\triangleright\, \Gamma}{\Gamma \triangleright () : \mathbf{unit} \Rightarrow_v ()} \quad \text{(BV-UNIT)}$$

$$\frac{\Gamma, x{:}A_1 \triangleright M : A_2 \Rightarrow M'}{\Gamma \triangleright \lambda x{:}A_1.M : A_1 \to A_2 \Rightarrow_v \lambda x{:}\langle A_1 \rangle.M'} \quad \text{(BV-ABS)}$$

$$\frac{\Gamma \triangleright V : A \Rightarrow_v V'}{\Gamma \triangleright V : A \Rightarrow \mathbf{left}_{[A]}(V')} \quad \text{(B-VALUE)}$$

$$\frac{\Gamma \triangleright V : A_2 \to A \Rightarrow_v V' \quad \Gamma \triangleright M : A_2 \Rightarrow M'}{\Gamma \triangleright V\ M : A \Rightarrow \mathit{applyv}(V', M', A_2, A)} \quad \text{(B-APP-V)}$$

$$\frac{\begin{array}{c} M_1 \text{ not a } \lambda_v^{\to,E}\text{-value} \\ \Gamma \triangleright M_1 : A_2 \to A \Rightarrow M_1' \quad \Gamma \triangleright M_2 : A_2 \Rightarrow M_2' \end{array}}{\Gamma \triangleright M_1\ M_2 : A \Rightarrow \mathit{apply}(M_1', M_2', A_2, A)} \quad \text{(B-APP)}$$

$$\frac{\Gamma \triangleright V : \sigma \Rightarrow_v V'}{\Gamma \triangleright \mathbf{raise}_\sigma(V) : A \Rightarrow \mathit{raisev}(A, V')} \quad \text{(B-RAISE-V)}$$

$$\frac{M \text{ not a } \lambda_v^{\to,E}\text{-value} \quad \Gamma \triangleright M : \sigma \Rightarrow M'}{\Gamma \triangleright \mathbf{raise}_\sigma(M) : A \Rightarrow \mathit{raise}(A, M')} \quad \text{(B-RAISE)}$$

$$\frac{\Gamma \triangleright M : A \Rightarrow M' \quad \Gamma, x{:}\sigma \triangleright N : A \Rightarrow N'}{\Gamma \triangleright M\ \mathbf{handle}\ x{:}\sigma => N : A \Rightarrow \mathit{handle}(x, A, M', N')} \quad \text{(B-HANDLE)}$$

Figure 2. Exception transform for σ a base type

$$
\begin{array}{rcll}
apply(M_1, M_2, A_2, A) & = & M_1 \textbf{ of} & (x \notin FV(M_2)) \\
 & & \quad \textbf{left}(x{:}\langle A_2 \rightarrow A \rangle) => & \\
 & & \quad\quad applyv(x, M_2, A_2, A); & \\
 & & \quad \textbf{right}(x{:}\langle \sigma \rangle) => \textbf{right}_{[A]}(x) & \\
 & & & \\
applyv(V, M, A_2, A) & = & M \textbf{ of} & (x \notin FV(V)) \\
 & & \quad \textbf{left}(x{:}\langle A_2 \rangle) => V\ x & \\
 & & \quad \textbf{right}(x{:}\langle \sigma \rangle) => \textbf{right}_{[A]}(x) & \\
 & & & \\
raise(A, M) & = & M \textbf{ of} & \\
 & & \quad \textbf{left}(x{:}\langle \sigma \rangle) => raisev(A, x) & \\
 & & \quad \textbf{right}(x{:}\langle \sigma \rangle) => \textbf{right}_{[A]}(x) & \\
 & & & \\
raisev(A, V) & = & \textbf{right}_{[A]}(V) & \\
 & & & \\
handle(x, A, M, N) & = & M \textbf{ of} & \\
 & & \quad \textbf{left}(x{:}\langle A \rangle) => \textbf{left}_{[A]}(x); & \\
 & & \quad \textbf{right}(x{:}\langle \sigma \rangle) => N &
\end{array}
$$

Figure 3. Macros used by the transform

The transforms are defined on all well-typed σ-only values and terms when σ is a base type:

LEMMA 7 (COMPLETENESS) *Let V and M be σ-only values and terms respectively.*

1. *If $\lambda_v^{\rightarrow,E} \vdash \Gamma \triangleright V : A$ then $\exists$ a $\lambda_v^{\rightarrow}$-value V' such that $\vdash \Gamma \triangleright V : A \Rightarrow_v V'$.*
2. *If $\lambda_v^{\rightarrow,E} \vdash \Gamma \triangleright M : A$ then $\exists$ a $\lambda_v^{\rightarrow}$-term M' such that $\vdash \Gamma \triangleright M : A \Rightarrow M'$.*

Proof: Routine simultaneous induction on the derivations of $\lambda_v^{\rightarrow,E} \vdash \Gamma \triangleright V : A$ and $\lambda_v^{\rightarrow,E} \vdash \Gamma \triangleright M : A$. ■

(For certain Γs, Ms, and As, the transform can yield several different M's that differ only in the types they contain because the variable A in the B-RAISE rule is unconstrained due to the polymorphism of the $\mathbf{raise}_\sigma(-)$ operator.)

Moreover, they result in well-typed $\lambda_v^{\rightarrow}$ terms and values with the appropriate types:

LEMMA 8 (PRESERVATION) *Let V and M be drawn from σ-only while V' and M' are drawn from $\lambda_v^{\rightarrow}$.*

1. *If $\vdash \Gamma \triangleright V : A \Rightarrow_v V'$ then $\lambda_v^{\rightarrow,E} \vdash \Gamma \triangleright V : A$ and $\lambda_v^{\rightarrow} \vdash \langle\Gamma\rangle \triangleright V' : \langle A\rangle$.*
2. *If $\vdash \Gamma \triangleright M : A \Rightarrow M'$ then $\lambda_v^{\rightarrow,E} \vdash \Gamma \triangleright M : A$ and $\lambda_v^{\rightarrow} \vdash \langle\Gamma\rangle \triangleright M' : [A]$.*

Proof: Proved by simultaneous induction on the derivations of $\Gamma \triangleright V : A \Rightarrow_v V'$ and $\Gamma \triangleright M : A \Rightarrow M'$. ■

They also distribute over substitution:

LEMMA 9 (SUBSTITUTION) *Suppose $\vdash \Gamma \triangleright V : A \Rightarrow_v V'$. Then:*

1. *If $\vdash \Gamma, x{:}A, \Gamma' \triangleright V_2 : B \Rightarrow_v V_2'$ then $\vdash \Gamma, \Gamma' \triangleright [V/x]V_2 : B \Rightarrow_v [V'/x]V_2'$.*
2. *If $\vdash \Gamma, x{:}A, \Gamma' \triangleright M : B \Rightarrow M'$ then $\vdash \Gamma, \Gamma' \triangleright [V/x]M : B \Rightarrow [V'/x]M'$.*

Proof: Proved by routine simultaneous induction on the derivations of $\Gamma, x{:}A, \Gamma \triangleright V_2 : B \Rightarrow_v V_2'$ and $\Gamma, x{:}A, \Gamma' \triangleright M : B \Rightarrow M'$, using the fact that N is a value iff $[V/x]N$ is a value. ■

Proving the transforms correct is done through showing that each evaluation step in $\lambda_v^{\rightarrow,E}$ on an original σ-only term is simulated in $\lambda_v^{\rightarrow}$ on the transformed term by at least one step:

THEOREM 5 (SIMULATION) *Suppose M is a σ-only term, $\vdash \Gamma \triangleright M : A \Rightarrow M'$ and $M \hookrightarrow N$ under $\lambda_v^{\rightarrow,E}$. Then there exists N' such that $\vdash \Gamma \triangleright N : A \Rightarrow N'$ and $M' \hookrightarrow^+ N'$ under $\lambda_v^{\rightarrow}$.*

Proof: Proved by induction on M. Sample cases:

LAM: Here $M = \lambda x{:}A_1.M_1$. This case is impossible because inspection of $\lambda_v^{\rightarrow,E}$'s one-step evaluation relation shows that it never applies to lambdas.

APP1M: Here $M = M_1\, M_2$, where M_1 is not a $\lambda_v^{\rightarrow,E}$ output. By B-APP, $\exists A_2, M_1', M_2'$ such that $\Gamma \triangleright M_1 : A_2 \rightarrow A \Rightarrow M_1'$, $\Gamma \triangleright M_2 : A_2 \Rightarrow M_2'$, and $M' = apply(M_1', M_2', A_2, A)$.

Inspection of the semantic definitions for $\lambda_v^{\rightarrow,E}$ shows that it must be the case that there exists N_1 such that $N = N_1\, M_2$ and $M_1 \hookrightarrow N_1$ under $\lambda_v^{\rightarrow,E}$. By the induction hypothesis, $\exists N_1'$ such that $\Gamma \triangleright N_1 : A_2 \rightarrow A \Rightarrow N_1'$ and $M_1' \hookrightarrow^+ N_1'$ under $\lambda_v^{\rightarrow} \Rightarrow M' = apply(M_1', M_2', A_2, A) \hookrightarrow^+ apply(N_1', M_2', A_2, A)$ under $\lambda_v^{\rightarrow}$.

Two cases need to be considered:

- N_1 is a $\lambda_v^{\rightarrow,E}$ value: By B-VALUE, $\exists V_1$ such that $\Gamma \triangleright V_1 : A_2 \rightarrow A \Rightarrow_v V_1'$ and $N_1' = \mathbf{left}_{[A_2 \rightarrow A]}(V_1')$. It is easy to verify that $apply(N_1', M_2', A_2, A)$ $= apply(\mathbf{left}_{[A_2 \rightarrow A]}(V_1'), M_2', A_2, A) \hookrightarrow applyv(V_1', M_2', A_2, A) \Rightarrow M' \hookrightarrow^+ applyv(V_1', M_2', A_2, A)$ (all under $\lambda_v^{\rightarrow}$). By B-APP-V, $\Gamma \triangleright N : A \Rightarrow applyv(V_1', M_2', A_2, A)$.
- otherwise: By B-APP, $\vdash \Gamma \triangleright N : A \Rightarrow apply(N_1', M_2', A_2, A)$.

HANR: Here $M = (\mathbf{raise}_\sigma(V_1))\ \mathbf{handle}\ x{:}\sigma => M_2 \hookrightarrow [V_1/x]M_2$ under $\lambda_v^{\rightarrow,E}$. By B-HANDLE and B-RAISE-V, $\exists V_1', M_2'$ such that $\Gamma \triangleright V_1 : \sigma \Rightarrow_v V_1'$, $\Gamma, x{:}\sigma \triangleright M_2 : A \Rightarrow M_2'$, and $M' = handle(x, A, \mathbf{right}_{[A]}(V_1'), M_2') \hookrightarrow [V_1'/x]M_2'$ under $\lambda_v^{\rightarrow}$. Hence by Lemma 9, $\vdash \Gamma \triangleright [V_1/x]M_2 : A \Rightarrow [V_1'/x]M_2'$.

■

To see where this result goes wrong if we do not use the specialized versions of *apply* and *raise*, consider the case where $M = V\, M_1$, $N = V\, M_2$, and $M_1 \hookrightarrow M_2$ under $\lambda_v^{\rightarrow,E}$. To get our desired simulation result, we will need to show that $M' = apply(\mathbf{left}_{[A_2 \rightarrow A]}(V'), M_1', A_2, A) \hookrightarrow^+ apply(\mathbf{left}_{[A_2 \rightarrow A]}(V'), M_2', A_2, A) = N'$ under $\lambda_v^{\rightarrow}$ for appropriate A_2 and A. It is easy to verify that $apply(\mathbf{left}_{[A_2 \rightarrow A]}(V'), M_1', A_2, A) \hookrightarrow^+ applyv(V', M_1', A_2, A)$ $\hookrightarrow^+ applyv(V', M_2', A_2, A)$ under $\lambda_v^{\rightarrow}$ (the second part by induction). However, it is not the case that $applyv(V', M_2', A_2, A) \hookrightarrow apply(\mathbf{left}_{[A_2 \rightarrow A]}(V'), M_2', A_2, A)$ under $\lambda_v^{\rightarrow}$ so the result fails. (The later is an expansion of the former.)

LEMMA 10 (STOPPING) *If $\vdash \Gamma \triangleright M : A \Rightarrow M'$ then M is a $\lambda_v^{\rightarrow,E}$ output iff M' is a $\lambda_v^{\rightarrow}$ value.*

Proof: Proved by inspection of the definitions of the transform and the semantics of $\lambda_v^{\rightarrow}$ and $\lambda_v^{\rightarrow,E}$. ■

THEOREM 6 (CORRECTNESS) *Suppose P is a σ-only program of type A. Then there exists P' such that $\vdash \triangleright P : A \Rightarrow P'$ and*

1. *The evaluation of P under $\lambda_v^{\rightarrow,E}$ terminates iff the evaluation of P' under $\lambda_v^{\rightarrow}$ terminates.*

2. *If $P \hookrightarrow^* O$ under $\lambda_v^{\rightarrow,E}$ then $\exists O'$ such that $\vdash \triangleright O : A \Rightarrow O'$ and $P' \hookrightarrow^* O'$ under $\lambda_v^{\rightarrow}$.*

3. *If $P' \hookrightarrow^* V'$ under $\lambda_v^{\rightarrow}$, then exists a σ-only output O such that $\vdash \triangleright O : A \Rightarrow V'$ and $P \hookrightarrow^* O$ under $\lambda_v^{\rightarrow,E}$.*

Proof: P' exists by Lemma 7. Consider the (possibly infinite) chain of terms resulting from applying the one-step evaluation function of $\lambda_v^{\rightarrow,E}$ repeatedly to P ($=P_0$): P_1, P_2, ...

By repeatedly applying Theorem 5, we can generate a corresponding chain of $\lambda_v^{\rightarrow}$ terms $P'_0 (= P')$, P'_1, P'_2, ... such that $\vdash \triangleright P_i : A \Rightarrow P'_i$ and $\forall i.\ P'_i \hookrightarrow^+ P'_{i+1}$ under $\lambda_v^{\rightarrow}$. By Lemma 10, the chains must either both be infinite in length, or both be of the same length (say n) with P_n a $\lambda_v^{\rightarrow,E}$ output and P'_n a $\lambda_v^{\rightarrow}$ value. By Lemma 6, $\forall i.\ P_i$ is a σ-only term. The desired results follow. ■

Problems arise if we try and use this transform when σ is a non-base type. In particular, if σ is an arrow type, infinite recursion results at the type level, preventing the transform from working because infinite types are not permitted in $\lambda_v^{\rightarrow}$. (For example, if $\sigma = \mathbf{int} \rightarrow \mathbf{int}$ then $\langle\sigma\rangle = \langle\mathbf{int}\rangle\rightarrow[\mathbf{int}] = \mathbf{int}\rightarrow(\langle\mathbf{int}\rangle+\langle\sigma\rangle) = \mathbf{int}\rightarrow(\mathbf{int}+(\mathbf{int}\rightarrow(\langle\mathbf{int}\rangle +\langle\sigma\rangle))) = \ldots$.)

This suggests that if we add recursive types to the destination calculus, we can make the transform work on arrow types. We will consider such a transform in Section 4.1.3 after we introduce a suitable target calculus.

4.1.2. $\lambda_v^{\rightarrow}$ extended with recursive types ($\lambda_v^{\rightarrow,\mu}$) We can obtain such a calculus by adding recursive types ($\mu\alpha.A$) to $\lambda_v^{\rightarrow}$. We use a formulation of recursive types where there is an isomorphism $\mu\alpha.A \cong [\mu\alpha.A/\alpha]A$ mediated by two built in primitives, $\mathbf{unroll}_{\mu\alpha.A}(-)$ in the forward direction and $\mathbf{roll}_{\mu\alpha.A}(-)$ in the backward direction, such that $\mathbf{unroll}_{\mu\alpha.A}(\mathbf{roll}_{\mu\alpha.A}(V)) \hookrightarrow V$. The syntax of $\lambda_v^{\rightarrow,\mu}$ is an extension of that of $\lambda_v^{\rightarrow}$:

$$
\begin{array}{lrcl}
\textit{Types} & A, B & ::= & \ldots \mid \alpha \mid \mu\alpha.A \\
\textit{Terms} & M & ::= & \ldots \mid \mathbf{roll}_A(M) \mid \mathbf{unroll}_A(M) \\
\\
\textit{Assignments} & \Gamma & ::= & \ldots \mid \Gamma, \alpha
\end{array}
$$

The meta-variable α ranges over *type variables*. We use $FTV(A)$ to denote the free type variables of A and $[A/\alpha]A'$ for the usual capture-avoiding substitution of A for α in A'.

The type system of $\lambda_v^{\rightarrow,\mu}$ extends that of $\lambda_v^{\rightarrow}$. Because types may now contain type variables, an additional judgement is needed to check that types do not contain type variables unbound by the current assignment:

$$\frac{\triangleright\,\Gamma \quad \alpha \notin \mathrm{dom}(\Gamma)}{\triangleright\,\Gamma, \alpha} \qquad (\textsc{a-type})$$

$$\frac{\Gamma \triangleright A \quad x \notin \mathrm{dom}(\Gamma)}{\triangleright\,\Gamma, x{:}A} \qquad (\textsc{a-extend2})$$

$$\frac{\triangleright\,\Gamma \quad \alpha \in \mathrm{dom}(\Gamma)}{\Gamma \triangleright \alpha} \qquad (\textsc{k-var})$$

$$\frac{\triangleright\,\Gamma}{\Gamma \triangleright \mathbf{unit}} \qquad (\textsc{k-unit})$$

$$\frac{\Gamma \triangleright A \quad \Gamma \triangleright B}{\Gamma \triangleright A{\to}B} \qquad (\textsc{k-arrow})$$

$$\frac{\Gamma \triangleright A \quad \Gamma \triangleright B}{\Gamma \triangleright A{+}B} \qquad (\textsc{k-dsum})$$

$$\frac{\Gamma, \alpha \triangleright A}{\Gamma \triangleright \mu\alpha.A} \qquad (\textsc{k-rec})$$

$$\frac{\Gamma \triangleright M : A \quad \Gamma \triangleright B}{\Gamma \triangleright \mathbf{left}_{A+B}(M) : A{+}B} \qquad (\textsc{t-left2})$$

$$\frac{\Gamma \triangleright M : B \quad \Gamma \triangleright A}{\Gamma \triangleright \mathbf{right}_{A+B}(M) : A{+}B} \qquad (\textsc{t-right2})$$

$$\frac{\Gamma \triangleright M : [\mu\alpha.A/\alpha]A \quad \Gamma \triangleright \mu\alpha.A}{\Gamma \triangleright \mathbf{roll}_{\mu\alpha.A}(M) : \mu\alpha.A} \qquad (\textsc{t-roll})$$

$$\frac{\Gamma \triangleright M : \mu\alpha.A}{\Gamma \triangleright \mathbf{unroll}_{\mu\alpha.A}(M) : [\mu\alpha.A/\alpha]A} \qquad (\textsc{t-unroll})$$

Figure 4. New type rules for $\lambda_v^{\rightarrow,\mu}$

$\Gamma \triangleright A$ *well-formed type*

The rules for implementing this new judgement as well as for typing $\mathbf{roll}_A(M)$ and $\mathbf{unroll}_A(M)$ appear in Figure 4. Rules A-EXTEND2, T-LEFT2, and T-RIGHT2 replace the corresponding rules of $\lambda_v^{\rightarrow}$; they are needed to ensure that if $\vdash \Gamma \triangleright M : A$ then $\vdash \Gamma \triangleright A$. The properties of decomposition and replacement remain true:

$$\mathit{Contexts}\ \ C\ ::=\ \ldots \mid \mathbf{roll}_A(C) \mid \mathbf{unroll}_A(C)$$

LEMMA 11 (DECOMPOSITION) *Suppose that* $\lambda_v^{\rightarrow,\mu} \vdash \Gamma \triangleright C[M] : A$ *such that* $EV(C) \cap dom(\Gamma) = \emptyset$. *Then there exists* Γ' *and* B *such that:*

- $dom(\Gamma') = EV(C)$
- $\lambda_v^{\rightarrow,\mu} \vdash \Gamma, \Gamma' \triangleright M : B$

LEMMA 12 (REPLACEMENT) *Suppose that* $\lambda_v^{\rightarrow,\mu} \vdash \Gamma \triangleright C[M] : A$, *with* $\lambda_v^{\rightarrow,\mu} \vdash \Gamma, \Gamma' \triangleright M : B$ *in accordance with the decomposition lemma. If* $\lambda_v^{\rightarrow,\mu} \vdash \Gamma, \Gamma'', \Gamma' \triangleright M' : B$ *then* $\lambda_v^{\rightarrow,\mu} \vdash \Gamma, \Gamma'' \triangleright C[M'] : A$.

The semantics of $\lambda_v^{\rightarrow,\mu}$ is an extension of that of $\lambda_v^{\rightarrow}$:

$$\begin{array}{rcl} V & ::= & \ldots \mid \mathbf{roll}_A(V) \\ E & ::= & \ldots \mid \mathbf{roll}_A(E) \mid \mathbf{unroll}_A(E) \\ R & ::= & \ldots \mid \mathbf{unroll}_A(\mathbf{roll}_A(V)) \end{array}$$

$$\begin{array}{rcl} \ldots & \hookrightarrow & \ldots \\ E[\mathbf{unroll}_A(\mathbf{roll}_A(V))] & \hookrightarrow & E[V] \end{array}$$

THEOREM 7 (PROGRESS) *If* M *is a closed, well-typed expression of type* A, *then either* M *is a value, or else there exist a unique evaluation context* E *and unique redex* R *such that* $M = E[R]$.

THEOREM 8 (SUBJECT REDUCTION) *If* $\lambda_v^{\rightarrow,\mu} \vdash \Gamma \triangleright M : A$ *and* $M \hookrightarrow N$, *then* $\lambda_v^{\rightarrow,\mu} \vdash \Gamma \triangleright N : A$.

4.1.3. Exceptions carrying functional types By retargeting our transform from $\lambda_v^{\rightarrow}$ to $\lambda_v^{\rightarrow,\mu}$ and altering its definition slightly, we can make it work even when σ is an arrow type. Figure 5 contains the necessary modifications to our transform: the new transform is composed of the definitions of this figure plus those of Figures 2 and 3, except that where there are two definitions for the same name, only the new definition is used.

The most important change here is to the definitions of what types transformed values ($\langle - \rangle$) and computations ($[-]$) have. Instead of transformed exceptional values (the arguments of $\mathbf{right}_A(-)$) being of type $\langle \sigma \rangle$, they now are of type

$$
\begin{array}{rcl}
\langle b \rangle_\alpha & = & b \\
\langle A_1 + A_2 \rangle_\alpha & = & \langle A_1 \rangle_\alpha + \langle A_2 \rangle_\alpha \\
\langle A_1 \to A_2 \rangle_\alpha & = & \langle A_1 \rangle_\alpha \to [A_2]_\alpha \\
[A]_\alpha & = & \langle A \rangle_\alpha + \alpha \\
\gamma & = & \mu\alpha.\langle \sigma \rangle_\alpha
\end{array}
$$

$$
\begin{array}{rcl}
\langle A \rangle & = & \langle A \rangle_\gamma \\
[A] & = & [A]_\gamma \quad (= \langle A \rangle + \gamma)
\end{array}
$$

$$
\begin{array}{rcll}
apply(M_1, M_2, A_2, A) & = & M_1 \textbf{ of} & (x \notin FV(M_2)) \\
 & & \quad \textbf{left}(x{:}\langle A_2 \to A \rangle) \Rightarrow & \\
 & & \quad\quad applyv(x, M_2, A_2, A); & \\
 & & \quad \textbf{right}(x{:}\gamma) \Rightarrow \textbf{right}_{[A]}(x) &
\end{array}
$$

$$
\begin{array}{rcll}
applyv(V, M, A_2, A) & = & M \textbf{ of} & (x \notin FV(V)) \\
 & & \quad \textbf{left}(x{:}\langle A_2 \rangle) \Rightarrow V\ x & \\
 & & \quad \textbf{right}(x{:}\gamma) \Rightarrow \textbf{right}_{[A]}(x) &
\end{array}
$$

$$
\begin{array}{rcl}
raise(A, M) & = & M \textbf{ of} \\
 & & \quad \textbf{left}(x{:}\langle \sigma \rangle) \Rightarrow raisev(A, x) \\
 & & \quad \textbf{right}(x{:}\gamma) \Rightarrow \textbf{right}_{[A]}(x)
\end{array}
$$

$$
raisev(A, V) = \textbf{right}_{[A]}(\textbf{roll}_\gamma(V))
$$

$$
\begin{array}{rcl}
handle(x, A, M, N) & = & M \textbf{ of} \\
 & & \quad \textbf{left}(x{:}\langle A \rangle) \Rightarrow \textbf{left}_{[A]}(x); \\
 & & \quad \textbf{right}(x{:}\gamma) \Rightarrow (\lambda x{:}\langle \sigma \rangle.\, N)\ \textbf{unroll}_\gamma(x)
\end{array}
$$

Figure 5. Exception transform using recursive types

$\gamma = \mu\alpha.\langle\sigma\rangle_\alpha$, where $\langle - \rangle_\alpha$ is the old definition of $\langle - \rangle$ modified to use α as the type of transformed exception values. These definitions ensure that $\langle A \rangle$ and $[A]$ are always well defined.

Unlike before, though, explicit roll and unroll operations need to be inserted to convert between γ and $\langle\sigma\rangle = \langle\sigma\rangle_\gamma$; these show up in the *raisev* and *handle* functions. The only other change required is to the type argument of the **right**$(x : -)$ pattern matcher in the case statements (from $\langle\sigma\rangle$ to γ).

Although σ is now free to be any $\lambda_v^{\rightarrow,E}$ type, the transforms are still defined on only well-typed σ-only values and terms:

LEMMA 13 (COMPLETENESS) *Let V and M be σ-only values and terms respectively.*

1. *If $\lambda_v^{\rightarrow,E} \vdash \Gamma \triangleright V : A$ then $\exists$ a $\lambda_v^{\rightarrow,\mu}$-value V' such that $\vdash \Gamma \triangleright V : A \Rightarrow_v V'$.*
2. *If $\lambda_v^{\rightarrow,E} \vdash \Gamma \triangleright M : A$ then $\exists$ a $\lambda_v^{\rightarrow,\mu}$-term M' such that $\vdash \Gamma \triangleright M : A \Rightarrow M'$.*

Proof: Routine simultaneous induction on the derivations of $\lambda_v^{\rightarrow,E} \vdash \Gamma \triangleright V : A$ and $\lambda_v^{\rightarrow,E} \vdash \Gamma \triangleright M : A$. ■

All of the results previously proved for the original transform carry over to the new transform; we omit proofs that are essentially the same as the previous versions:

LEMMA 14 (PRESERVATION) *Let V and M be drawn from σ-only while V' and M' are drawn from $\lambda_v^{\rightarrow,\mu}$.*

1. *If $\vdash \Gamma \triangleright V : A \Rightarrow_v V'$ then $\lambda_v^{\rightarrow,E} \vdash \Gamma \triangleright V : A$ and $\lambda_v^{\rightarrow,\mu} \vdash \langle\Gamma\rangle \triangleright V' : \langle A \rangle$.*
2. *If $\vdash \Gamma \triangleright M : A \Rightarrow M'$ then $\lambda_v^{\rightarrow,E} \vdash \Gamma \triangleright M : A$ and $\lambda_v^{\rightarrow,\mu} \vdash \langle\Gamma\rangle \triangleright M' : [A]$.*

LEMMA 15 (SUBSTITUTION) *Suppose $\vdash \Gamma \triangleright V : A \Rightarrow_v V'$. Then:*

1. *If $\vdash \Gamma, x{:}A, \Gamma' \triangleright V_2 : B \Rightarrow_v V_2'$ then $\vdash \Gamma, \Gamma' \triangleright [V/x]V_2 : B \Rightarrow_v [V'/x]V_2'$.*
2. *If $\vdash \Gamma, x{:}A, \Gamma' \triangleright M : B \Rightarrow M'$ then $\vdash \Gamma, \Gamma' \triangleright [V/x]M : B \Rightarrow [V'/x]M'$.*

THEOREM 9 (SIMULATION) *Suppose M is a σ-only term, $\vdash \Gamma \triangleright M : A \Rightarrow M'$ and $M \hookrightarrow N$ under $\lambda_v^{\rightarrow,E}$. Then there exists N' such that $\vdash \Gamma \triangleright N : A \Rightarrow N'$ and $M' \hookrightarrow^+ N'$ under $\lambda_v^{\rightarrow,\mu}$.*

Proof: Proved by induction on M. Most cases are very similar to the previous theorem, with one exception:

HANR: Here $M = (\mathbf{raise}_\sigma(V_1))$ **handle** $x{:}\sigma \Rightarrow M_2 \hookrightarrow [V_1/x]M_2$ under $\lambda_v^{\rightarrow,E}$ with $x \notin \mathrm{FV}(V_1')$. By B-HANDLE and B-RAISE-V, $\exists V_1', M_2'$ such that
$\Gamma \triangleright V_1 : \sigma \Rightarrow_v V_1'$, $\Gamma, x{:}\sigma \triangleright M_2 : A \Rightarrow M_2'$, and
$M' = \mathit{handle}(x, A, \mathbf{right}_{[A]}(\mathbf{roll}_\gamma(V_1')), M_2')$
$\hookrightarrow [\mathbf{roll}_\gamma(V_1')/x]((\lambda x{:}\langle\sigma\rangle.\, M_2')\ \mathbf{unroll}_\gamma(x))$
$= (\lambda x{:}\langle\sigma\rangle.\, M_2')\ \mathbf{unroll}_\gamma(\mathbf{roll}_\gamma(V_1'))$
$\hookrightarrow (\lambda x{:}\langle\sigma\rangle.\, M_2')\ V_1' \hookrightarrow [V_1'/x]M_2'$ under $\lambda_v^{\rightarrow,\mu}$. Hence by Lemma 15,
$\vdash \Gamma \triangleright [V_1/x]M_2 : A \Rightarrow [V_1'/x]M_2'$.

■

LEMMA 16 (STOPPING) *If* $\vdash \Gamma \triangleright M : A \Rightarrow M'$ *then* M *is a* $\lambda_v^{\rightarrow,E}$ *output iff* M' *is a* $\lambda_v^{\rightarrow,\mu}$ *value.*

THEOREM 10 (CORRECTNESS) *Suppose* P *is a* σ*-only program of type* A*. Then there exists* P' *such that* $\vdash \triangleright P : A \Rightarrow P'$ *and*

1. *The evaluation of* P *under* $\lambda_v^{\rightarrow,E}$ *terminates iff the evaluation of* P' *under* $\lambda_v^{\rightarrow,\mu}$ *terminates.*
2. *If* $P \hookrightarrow^* O$ *under* $\lambda_v^{\rightarrow,E}$ *then* $\exists O'$ *such that* $\vdash \triangleright O : A \Rightarrow O'$ *and* $P' \hookrightarrow^* O'$ *under* $\lambda_v^{\rightarrow,\mu}$*.*
3. *If* $P' \hookrightarrow^* V'$ *under* $\lambda_v^{\rightarrow,\mu}$*, then exists a* σ*-only output* O *such that* $\vdash \triangleright O : A \Rightarrow V'$ *and* $P \hookrightarrow^* O$ *under* $\lambda_v^{\rightarrow,E}$*.*

A question naturally arises: since, as we have seen, the transform suggests that unchecked exceptions carrying arrow types have an inherently recursive character, can we simulate recursive types using unchecked exceptions capable of carrying arrow types? The following section answers this question in the affirmative.

4.2. Simulating recursive types with exceptions

Suppose we wish to encode the $\lambda_v^{\rightarrow,\mu}$ values of a recursive type $\mu\alpha.A$ using the $\lambda_v^{\rightarrow,E}$ values of some type $\overline{\mu\alpha.A}$. What properties do we need of $\overline{\mu\alpha.A}$? Most importantly, we need the following isomorphism so we can implement roll and unroll on our encoded terms:[5]

$$\overline{\mu\alpha.A} \;\cong\; \overline{[\mu\alpha.A/\alpha]A}$$

Ideally, we would like our transform to be compositional so that $\overline{[A/\alpha]B} = [\overline{A}/\alpha]\overline{B}$; this means that the desired isomorphism is really between $\overline{\mu\alpha.A}$ and $[\overline{\mu\alpha.A}/\alpha]\overline{A}$.

Because recursive types have no non-local effects, we should be able to make only local changes to recursive values and types. This means that $\overline{\ }$ should distribute over type constructors other than μ (e.g., $\overline{A+B} = \overline{A}+\overline{B}$) and that $\overline{b}$ should equal b. Accordingly, we should have that $\overline{A} = A$ when A contains no recursive types.

Thus, if we consider the simple case where A contains no recursive types, the isomorphism means that we have to be able to pack a value of type $[\overline{\mu\alpha.A}/\alpha]A$ into a value of type $\overline{\mu\alpha.A}$. We cannot do this in $\lambda_v^{\rightarrow}$ because, in general, values of the former type will contain much more information than values of the later type.

If we are to succeed, we need a way to "smuggle" more information out of some values than their type suggests they can provide. Functions that can raise unchecked exceptions provide this ability. Consider a function of the simplest functional type **unit→unit** that can raise an exception carrying values of type σ: if we apply the transform of the last section, we see that it can be modeled by a $\lambda_v^{\rightarrow,\mu}$ value of

$$
\begin{array}{rcll}
\overline{b} & = & b & \\
\overline{A_1{+}A_2} & = & \overline{A_1} + \overline{A_2} & \\
\overline{A_1{\to}A_2} & = & \overline{A_1} \to \overline{A_2} & \\
\overline{\mu\alpha.A} & = & \mathbf{unit} \to \mathbf{unit} & \\
 & & & \\
\overline{\emptyset} & = & \emptyset & \\
\overline{\Gamma, x{:}A} & = & \overline{\Gamma}, x{:}\overline{A} & \\
 & & & \\
 & & & \\
\overline{x} & = & x & \\
\overline{()} & = & () & \\
\overline{\lambda x{:}A.M} & = & \lambda x{:}\overline{A}.\,\overline{M} & \\
\overline{M_1\,M_2} & = & \overline{M_1}\ \ \overline{M_2} & \\
\overline{\mathbf{roll}_{\mu\alpha.A}(V)} & = & \mathit{packv}(\overline{[\mu\alpha.A/\alpha]A},\, \overline{V}) & \\
\overline{\mathbf{roll}_{\mu\alpha.A}(M)} & = & \mathit{pack}(\overline{[\mu\alpha.A/\alpha]A},\, \overline{M}) & (M \text{ not a } \lambda_v^{\to,\mu}\text{-value}) \\
\overline{\mathbf{unroll}_{\mu\alpha.A}(M)} & = & \mathit{unpack}(\overline{[\mu\alpha.A/\alpha]A},\, \overline{M}) & \\
\overline{\mathbf{left}_A(M)} & = & \mathbf{left}_{\overline{A}}(\overline{M}) & \\
\overline{\mathbf{right}_A(M)} & = & \mathbf{right}_{\overline{A}}(\overline{M}) &
\end{array}
$$

$$
\begin{array}{c}
\overline{M\ \mathbf{of}\ \mathbf{left}(x{:}A) => M_1;\ \ \mathbf{right}(y{:}B) => M_2} \\
= \\
\overline{M}\ \mathbf{of}\ \mathbf{left}(x{:}\overline{A}) => \overline{M_1};\ \ \mathbf{right}(y{:}\overline{B}) => \overline{M_2}
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{pack}(A, M) & = & (\lambda x{:}A.\ \mathit{packv}(A, x))\ M & \\
\mathit{packv}(A, M) & = & \lambda x{:}\mathbf{unit}.\ \mathbf{raise}_A(M) & (x \notin FV(M)) \\
\mathit{unpack}(A, M) & = & (M\,();\mathit{fail}(A))\ \mathbf{handle}\ x{:}A => x & \\
\mathit{fail}(A) & = & \mathbf{raise}_{\mathbf{unit}}(()) &
\end{array}
$$

Figure 6. Encoding recursive types with exceptions

type $\langle\mathbf{unit}{\to}\mathbf{unit}\rangle = \langle\mathbf{unit}\rangle \to [\mathbf{unit}] = \mathbf{unit} \to (\langle\mathbf{unit}\rangle{+}\gamma) = \mathbf{unit} \to (\mathbf{unit}{+}\gamma)$, where γ is a rolled version of σ. Here σ can be any type so that our $\mathbf{unit}{\to}\mathbf{unit}$ $\lambda_v^{\to,E}$ function can "smuggle" out a value of type σ by raising a unchecked exception carrying values of that type. Note that checked exceptions, by contrast, do not allow smuggling because functions that raise checked exceptions must include the types that those exceptions carry in their functional type (e.g., $\mathbf{unit}{\to}\mathbf{unit}\ \mathbf{throws}\ \sigma$).

These facts suggest that we set $\overline{\mu\alpha.A} = \mathbf{unit}{\to}\mathbf{unit}$ and pack values of type $\overline{[\mu\alpha.A/\alpha]A}$ into $\mathbf{unit}{\to}\mathbf{unit}$ functions by wrapping a $\mathbf{unit}$-accepting lambda ($\lambda x{:}\mathbf{unit}.[]$) around code that raises the packed value. Unpacking can then be done by calling

one of these functions and catching the resulting exception, which will be carrying the packed value. Such a transform from $\lambda_v^{\rightarrow,\mu}$ to $\lambda_v^{\rightarrow,E}$ can be found in Figure 6.

The transform acts on types as expected, with the exception that it acts on only closed $\lambda_v^{\rightarrow,\mu}$ types and *simple assignments* (assignments that declare no type variables) because there is nothing in $\lambda_v^{\rightarrow,E}$ that a type variable could be mapped to. (Remember that $\lambda_v^{\rightarrow,E}$ does not have type variables.) Because of this restriction, the transform is only guaranteed to be defined on $\lambda_v^{\rightarrow,\mu}$ types and terms that are valid under a valid $\lambda_v^{\rightarrow,\mu}$ simple assignment:

LEMMA 17 (COMPLETENESS AND PRESERVATION) *Let M, A, and Γ be $\lambda_v^{\rightarrow,\mu}$ terms, types, and simple assignments respectively. Then:*

1. *If $\lambda_v^{\rightarrow,\mu} \vdash \triangleright\, \Gamma$ then $\overline{\Gamma}$ exists and $\lambda_v^{\rightarrow,E} \vdash \triangleright\, \overline{\Gamma}$.*
2. *If $\lambda_v^{\rightarrow,\mu} \vdash \Gamma \triangleright A$ then $\overline{\Gamma}$ and $\overline{A}$ exist and $\overline{A}$ is a $\lambda_v^{\rightarrow,E}$ type.*
3. *If $\lambda_v^{\rightarrow,\mu} \vdash \Gamma \triangleright M : A$ then $\overline{\Gamma}$, $\overline{A}$, and $\overline{M}$ exist and $\lambda_v^{\rightarrow,E} \vdash \overline{\Gamma} \triangleright \overline{M} : \overline{A}$.*

Proof: Proved by straightforward simultaneous induction on the derivations. ■

The transform treats terms in a straightforward way, rewriting instances of roll and unroll into instances of the macros *pack*[v] and *unpack* respectively and replacing all instances of recursive types with **unit**→**unit**. Otherwise, terms are left unchanged. This transform is thus far more local than the previous ones, which required global rewriting to insert the necessary plumbing. The specialized *packv* macro is used to produce the more optimized code needed to make a simulation result go through (see Theorem 11 below).

When a transformed program is evaluated, $unpack(A, -)$ will never be called on any **unit**→**unit** function not generated by $packv(A, -)$,[6] ensuring that *unpack*'s argument will raise an exception when called. However, because nothing prevents the programmer from applying *unpack* directly to $\lambda_v^{\rightarrow,E}$ terms that are not the result of transforming a $\lambda_v^{\rightarrow,\mu}$ term, $\lambda_v^{\rightarrow,E}$'s type system insists that we give well-typed code for the case where *unpack*'s argument returns. The term $fail(A)$ of type A in *unpack* serves this purpose.

Because raise is polymorphic in its return type, we have found it easiest to define $fail(A)$ as $\mathbf{raise}_{\mathbf{unit}}(())$. However, any function that produces a term of type A will do here. The proof of the transform's correctness proceeds similarly to that of the previous transforms:

LEMMA 18 (SUBSTITUTION) *If $\overline{V}$ and $\overline{M}$ exist then $\overline{[V/x]M} = [\overline{V}/x]\overline{M}$.*

Proof: Proved by induction on M. ■

THEOREM 11 (SIMULATION) *If M is a $\lambda_v^{\rightarrow,\mu}$ term, $\lambda_v^{\rightarrow,\mu} \vdash \Gamma \triangleright M : A$, Γ is simple, and $M \hookrightarrow N$ under $\lambda_v^{\rightarrow,\mu}$ then $\overline{M} \hookrightarrow^+ \overline{N}$ under $\lambda_v^{\rightarrow,E}$.*

Proof: Proved by induction on M. $\overline{M}$ exists by Lemma 17. Sample cases:

BETA: Here $M = (\lambda x{:}A.M)\,V \hookrightarrow [V/x]M = N$ under $\lambda_v^{\rightarrow,\mu}$. Hence, $\overline{M} = (\lambda x{:}\overline{A}.\overline{M})\,\overline{V} \hookrightarrow [\overline{V}/x]\overline{M} = \overline{N}$ under $\lambda_v^{\rightarrow,E}$ by Lemma 18.

CAN: Here $M = \mathbf{unroll}_B(\mathbf{roll}_B(V)) \hookrightarrow V = N$ under $\lambda_v^{\rightarrow,\mu}$. By T-ROLL, B must have the form $\mu\alpha.A$. Let $A' = [\mu\alpha.A/\alpha]A$. Then
$\overline{M} = unpack(\overline{A'},\ packv(\overline{A'}, \overline{V}))$
$= ((\lambda x{:}\mathbf{unit}.\ \mathbf{raise}_{\overline{A'}}(\overline{V}))\,();\, fail(A))\ \mathbf{handle}\ x{:}\overline{A'} => x$
$\hookrightarrow (\mathbf{raise}_{\overline{A'}}(\overline{V}); fail(A))\ \mathbf{handle}\ x{:}\overline{A'} => x$
$\hookrightarrow \mathbf{raise}_{\overline{A'}}(\overline{V})\ \mathbf{handle}\ x{:}\overline{A'} => x$
$\hookrightarrow [\overline{V}/x]x = \overline{V} = \overline{N}$ under $\lambda_v^{\rightarrow,E}$ $(x \notin FV(\overline{V}))$.

■

LEMMA 19 (STOPPING) *If $\overline{M}$ exists then it is a $\lambda_v^{\rightarrow,E}$ value iff M is a $\lambda_v^{\rightarrow,\mu}$ value.*

Proof: Proved by inspection of the transform. ■

THEOREM 12 (CORRECTNESS) *Suppose P is a $\lambda_v^{\rightarrow,\mu}$ program such that $\lambda_v^{\rightarrow,\mu} \vdash \Gamma \triangleright P : A$ where Γ is simple. Then $\overline{P}$ exists and*

1. *The evaluation of P under $\lambda_v^{\rightarrow,\mu}$ terminates iff the evaluation of $\overline{P}$ under $\lambda_v^{\rightarrow,E}$ terminates.*
2. *If $P \hookrightarrow^* V$ under $\lambda_v^{\rightarrow,\mu}$ then $\overline{P} \hookrightarrow^* \overline{V}$ under $\lambda_v^{\rightarrow,E}$.*
3. *If $\overline{P} \hookrightarrow^* V'$ under $\lambda_v^{\rightarrow,E}$, then exists a $\lambda_v^{\rightarrow,\mu}$ value V such that $\overline{V} = V'$ and $P \hookrightarrow^* V$ under $\lambda_v^{\rightarrow,\mu}$.*

Proof: $\lambda_v^{\rightarrow,E} \vdash \overline{\Gamma} \triangleright \overline{P} : \overline{A}$ by Lemma 17. Consider the (possibly infinite) chain of terms resulting from applying the one-step evaluation function of $\lambda_v^{\rightarrow,\mu}$ repeatedly to P $(=P_0)$: P_1, P_2, ...
By repeatedly applying Theorems 8 and 11, we get that $\overline{P}$ $(=\overline{P_0}) \hookrightarrow^+ \overline{P_1} \hookrightarrow^+ \overline{P_2} \ldots \hookrightarrow^+$ under $\lambda_v^{\rightarrow,E}$. If evaluation of P does terminate, say at P_n, then by Theorem 7 and Lemma 19 P_n is a $\lambda_v^{\rightarrow,\mu}$ value and $\overline{P_n}$ is a $\lambda_v^{\rightarrow,E}$ value; moreover, by Theorems 3 and 4, evaluation of $\overline{P}$ under $\lambda_v^{\rightarrow,E}$ terminates at $\overline{P_n}$, with all previous evaluation steps being on non-$\lambda_v^{\rightarrow,E}$ values. The desired results follow.
■

Because we have a correct transform from full $\lambda_v^{\rightarrow,\mu}$ to $\lambda_v^{\rightarrow,E}$, we have reduced our goal of proving $\lambda_v^{\rightarrow,E}$ Turing complete to proving $\lambda_v^{\rightarrow,\mu}$ Turing complete:

COROLLARY 1 (REDUCTION) *If $\lambda_v^{\rightarrow,\mu}$ is Turing complete then so is $\lambda_v^{\rightarrow,E}$.*

Proof: Follows from Theorem 12 and the fact that the transform defines an algorithm. (Recall that a language L is Turing complete iff it can express all computable functions. This requirement can be expressed more formally as there

exist recursive functions f_i and f_o such that for all Turing-machine descriptions M and Turing-machine inputs I, $f_i(M, I)$ is a valid program of L that terminates with value $f_o(O)$ iff Turing machine M on input I halts with output O.) ■

It is a simple matter to restrict the transform to the subset of $\lambda_v^{\rightarrow,\mu}$ that contains no disjoint sums, sumless $\lambda_v^{\rightarrow,\mu}$. Terms of sumless $\lambda_v^{\rightarrow,\mu}$ may not contain $\mathbf{left}_A(M)$, $\mathbf{right}_A(M)$, or M **of** **left**$(x{:}A) => M_1$; **right**$(y{:}B) => M_2$ as sub-terms. The resulting transform yields terms in $\lambda_v^{\rightarrow,E}$ minus disjoint sums (sumless $\lambda_v^{\rightarrow,E}$, defined similarly). The proofs of correctness are almost identical, with the addition of the following Lemma; we omit them to save space.

LEMMA 20 (CLOSURE)

1. *If M is a sumless $\lambda_v^{\rightarrow,\mu}$ term and $M \hookrightarrow N$ under $\lambda_v^{\rightarrow,\mu}$ then N is a sumless $\lambda_v^{\rightarrow,\mu}$ term.*
2. *If M is a sumless $\lambda_v^{\rightarrow,E}$ term and $M \hookrightarrow N$ under $\lambda_v^{\rightarrow,E}$ then N is a sumless $\lambda_v^{\rightarrow,E}$ term.*

Proof: Proved by inspection of the one-step evaluation relations for $\lambda_v^{\rightarrow,\mu}$ and $\lambda_v^{\rightarrow,E}$. ■

The correctness of the more restrictive transform will allow us to show that just sumless $\lambda_v^{\rightarrow,E}$ is Turing complete:

COROLLARY 2 (REDUCTION) *If sumless $\lambda_v^{\rightarrow,\mu}$ is Turing complete then so is sumless $\lambda_v^{\rightarrow,E}$ (and hence $\lambda_v^{\rightarrow,E}$).*

Our next step is to establish that sumless $\lambda_v^{\rightarrow,\mu}$ is Turing complete; we will do this by showing that it can be used to simulate the call-by-value untyped lambda calculus.

4.3. Simulating the CBV untyped lambda calculus

The syntax of the call-by-value untyped lambda calculus (λ_v) is very simple:

$$Terms \;\; M \;::=\; x \mid \lambda x.M \mid M_1\, M_2$$

λ_v has no type system; programs are simply closed terms. The semantics of λ_v is defined as follows:

$$\begin{array}{l} V \;::=\; x \mid \lambda x.M \\ E \;::=\; [] \mid E\, M \mid V\, E \\ R \;::=\; (\lambda x.M)\, V \end{array}$$

$$E[(\lambda x.M)\, V] \;\hookrightarrow\; E[[V/x]M]$$

LEMMA 21 (PROGRESS) *If M is a closed term, then either M is a value, or else there exist a unique evaluation context E and unique redex R such that $M = E[R]$.*

Proof: Proved by induction on M. ■

It is well known that adding recursive types to the simply-typed lambda calculus allows the full untyped lambda calculus to be simulated; see, for example, exercise 2.6.7 of Mitchell [20]. The following encoding of λ_v in sumless $\lambda_v^{\rightarrow,\mu}$ suffices for our purposes:

$$\begin{array}{rcl} \diamond & = & \mu\alpha.\,\alpha \rightarrow \alpha \\ \\ \{x\} & = & x \\ \{\lambda x.\,M\} & = & \mathbf{roll}_\diamond(\lambda x{:}\diamond\,.\,\{M\}) \\ \{M\,N\} & = & \mathbf{unroll}_\diamond(\{M\})\,\{N\} \end{array}$$

Here roll and unroll convert between $\diamond \rightarrow \diamond$ and $\diamond$. Under appropriate assumptions, the transform results in a well-formed sumless $\lambda_v^{\rightarrow,\mu}$ term:

THEOREM 13 (PRESERVATION) *Suppose Γ is a $\lambda_v^{\rightarrow,\mu}$ assignment of the form $x_1{:}\diamond$, $x_2{:}\diamond, \ldots, x_n{:}\diamond$, $\lambda_v^{\rightarrow,\mu} \vdash \Gamma$, and M is a λ_v term with $FV(M) \subseteq dom(\Gamma)$ then:*

1. *$\{M\}$ is a sumless $\lambda_v^{\rightarrow,\mu}$ term*
2. *$\lambda_v^{\rightarrow,\mu} \vdash \Gamma \triangleright \{M\} : \diamond$*

Proof: Proved by induction on M. ■

The transform is proved correct by the usual methods:

LEMMA 22 (SUBSTITUTION) $\{[V/x]M\} = [\{V\}/x]\{M\}$

LEMMA 23 (STOPPING) *M is a λ_v value iff $\{M\}$ is a sumless $\lambda_v^{\rightarrow,\mu}$ value.*

THEOREM 14 (SIMULATION) *Suppose M is a closed λ_v term and $M \hookrightarrow N$ under λ_v. Then $\{M\} \hookrightarrow^+ \{N\}$ under $\lambda_v^{\rightarrow,\mu}$.*

Proof: Proved by induction on M using Lemma 23 as needed. Sample case:

BETA: Here $M = (\lambda x.M')\,V \hookrightarrow [V/x]M' = N$ under λ_v. Hence, $\{M\} = \mathbf{unroll}_\diamond(\{\lambda x.M'\})\,\{V\} = \mathbf{unroll}_\diamond(\mathbf{roll}_\diamond(\lambda x{:}\diamond\,.\,\{M'\}))\,\{V\} \hookrightarrow (\lambda x{:}\diamond\,.\,\{M'\})\,\{V\} \hookrightarrow [\{V\}/x]\{M'\} = \{[V/x]M'\} = \{N\}$ under $\lambda_v^{\rightarrow,\mu}$ by Lemmas 23 and 22.

■

THEOREM 15 (CORRECTNESS) *Suppose P is a λ_v program. Then*

1. *The evaluation of P under λ_v terminates iff the evaluation of $\{P\}$ under $\lambda_v^{\rightarrow,\mu}$ terminates.*

2. *If $P \hookrightarrow^* V$ under λ_v then $\{P\} \hookrightarrow^* \{V\}$ under $\lambda_v^{\rightarrow,\mu}$.*

3. *If $\{P\} \hookrightarrow^* V'$ under $\lambda_v^{\rightarrow,\mu}$, then exists a λ_v value V such that $\{V\} = V'$ and $P \hookrightarrow^* V$ under λ_v.*

Proof: Consider the (possibly infinite) chain of closed terms resulting from applying the one-step evaluation function of λ_v repeatedly to P ($=P_0$): P_1, P_2, ...

By repeatedly applying Theorem 14, we get that $\{P\}$ ($=\{P_0\}$) $\hookrightarrow^+$ $\{P_1\}$ $\hookrightarrow^+$ $\{P_2\}$ $\hookrightarrow^+$ ... under $\lambda_v^{\rightarrow,\mu}$. If evaluation of P does terminate, say at P_n, then by Lemmas 21 and 23 P_n is a λ_v value and $\{P_n\}$ is a $\lambda_v^{\rightarrow,\mu}$ value; moreover, by Theorems 7, 13, and 8, evaluation of $\{P\}$ under $\lambda_v^{\rightarrow,\mu}$ terminates at $\{P_n\}$, with all previous evaluation steps being on non-$\lambda_v^{\rightarrow,\mu}$ values. The desired results follow. ■

Thus, we have that sumless $\lambda_v^{\rightarrow,\mu}$ (and hence sumless $\lambda_v^{\rightarrow,E}$) is Turing complete if λ_v is Turing complete, which we will show in the next section:

COROLLARY 3 (REDUCTION) *If λ_v is Turing complete then so is sumless $\lambda_v^{\rightarrow,\mu}$.*

Proof: Follows from Theorem 15 and the fact that the transform defines an algorithm. ■

By careful inspection of our transforms, we can sharpen this result to apply to an even smaller subset of $\lambda_v^{\rightarrow,E}$, (**unit**→**unit**)→(**unit**→**unit**)-only:

THEOREM 16 (REDUCTION)
If λ_v is Turing complete then so is (**unit**→**unit**)→(**unit**→**unit**)*-only.*

Proof: Observe that the transform from λ_v to sumless $\lambda_v^{\rightarrow,\mu}$ uses only one recursive type, namely $\diamond = \mu\alpha.\, \alpha \rightarrow \alpha$, and that we can construct a value of type $\overline{[\diamond/\alpha]\diamond} = \overline{\diamond} \rightarrow \overline{\diamond} = (\textbf{unit}\rightarrow\textbf{unit}) \rightarrow (\textbf{unit}\rightarrow\textbf{unit})$ by using an identity function.

If we set $fail(\diamond\rightarrow\diamond) = \lambda x{:}\textbf{unit}\rightarrow\textbf{unit}.\, x$, then the combined transform on a λ_v term M ($\overline{\{M\}}$) yields a term in (**unit**→**unit**)→(**unit**→**unit**)-only. The combination of Lemma 6, Theorem 15, and a slightly modified Theorem 12 (the proof is essentially unchanged because only $fail$'s type matters) prove the combined transform correct. ■

As a demonstration of the combined transform, we have provided working SML code in Figure 7 that uses the transform to encode a non-terminating λ_v term. The code is entirely monomorphic and uses only those features present in (**unit**→**unit**)→(**unit**→**unit**)-only plus a small amount of syntactic sugar.

```
(*
 * Prepare to simulate values of the recursive type
 * \mu a. a -> a using a ML exception of type
 * (unit->unit)->(unit->unit):
 *)
type star  = unit -> unit;
exception E of star->star;

fun roll(x:star->star):star =
         fn y:unit => raise E(x);
fun unroll(x:star):star->star =
         (x(); (fn y:star => y)) handle E(z) => z;


(*
 * Define an encoding of the untyped lambda calculus in
 * ML using the previous simulation:
 *
 * The rules for encoding using the below functions are as
 * follows:
 *
 *    encode(x)    = x
 *    encode(\x.M) = lam(fn x => encode(M))
 *    encode(M N)  = app(encode(M),encode(N))
 *)
fun app(x:star,y:star):star  = (unroll x) y;
fun lam(x:star->star):star   = roll(x);


(*
 * As an example, we use the encoding of omega = w w
 * where w = \x.x x to write a hanging function:
 *
 * (Omega reduces to itself in one beta-reduction step,
 *  resulting in an infinite reduction sequence.)
 *)
fun hang() = let val w = lam (fn x => app(x,x))
             in app(w,w) end;
```

Figure 7. SML code to encode the untyped lambda calculus

4.4. The power of the CBV untyped lambda calculus ($\boldsymbol{\lambda_v}$)

It is well known that the call-by-value lambda calculus is Turing complete; unfortunately, we were unable to find any directly applicable results in the literature to this effect. Accordingly, we will very briefly sketch in this section how such a proof might go.

The basic idea is to simulate a Turing machine T by encoding its *instantaneous descriptions* (its state, head location(s), and tape contents) as λ_v terms and by writing a λ_v function that implements the move function for T. In order to perform this encoding, we will need the ability to encode booleans, natural numbers, pairs, and one-ofs (a one-of is either none, or some value). We use the standard encodings (Church numerals, etc.) here; each value is represented by an equivalence class:

$$\begin{array}{rcl}
bool(true) & = & \{M|\ \forall V_1, V_2.\ M\,V_1\,V_2 \hookrightarrow^* V_1\} \\
bool(false) & = & \{M|\ \forall V_1, V_2.\ M\,V_1\,V_2 \hookrightarrow^* V_2\} \\
 & & \\
num(n) & = & \{M|\ \forall V_1, V_2.\ M\,V_1\,V_2 \hookrightarrow^* V_2^n\,V_1\} \\
 & & \\
pair(V_1, V_2) & = & \{M|\ M\,fst \hookrightarrow^* V_1 \text{ and } M\,snd \hookrightarrow^* V_2\} \\
fst & = & \lambda x.\lambda y.\,x \\
snd & = & \lambda x.\lambda y.\,y \\
 & & \\
none & = & \{M|\ \forall V_1, V_2.\ M\,V_1\,V_2 \hookrightarrow^* V_1\} \\
some(V) = & = & \{M|\ \forall V_1, V_2.\ M\,V_1\,V_2 \hookrightarrow^* V_2\,V\}
\end{array}$$

All terms, values, and reduction steps in this section are from λ_v. The meta-variable n ranges over natural numbers while the meta-variable t ranges over booleans. The notation $M^n N$ denotes $M(M(\ldots M(N)\ldots))$ where M is repeated n times. It can be shown that these set definitions are closed under λ_v evaluation and that all of their members must evaluate to a functional value.

These equivalence classes are non-empty:

$$\begin{array}{rcl}
\overline{true} & = & \lambda x.\lambda y.\,x \\
\overline{false} & = & \lambda x.\lambda y.\,y \\
 & & \\
\overline{n} & = & \lambda z.\lambda s.\,s^n\,z \\
 & & \\
\mathbf{pair}(M_1, M_2) & = & (\lambda v_1.\lambda v_2.\ \lambda c.\ c\,v_1\,v_2)\ M_1\ M_2 \\
 & & \\
\mathbf{none} & = & \lambda x.\lambda y.\,x \\
\mathbf{some}(M) & = & (\lambda v.\ \lambda x.\lambda y.\,y\,v)\ M
\end{array}$$

It is easy to verify that if M_1, M_2, and M evaluate to values then each of these definitions results in a value that belongs to the appropriate equivalence class. For example, $\overline{n} \in num(n)$. Given these definitions, we can define various operations on these data types:

$$
\begin{array}{rcl}
not(M) & = & M\ \overline{false}\ \overline{true} \\
and(M_1, M_2) & = & \lambda x.\lambda y.\, M_1\ (M_2\, x\, y)\ y \\
or(M_1, M_2) & = & \lambda x.\lambda y.\, M_1\ x\ (M_2\, x\, y)
\end{array}
$$

$$
\begin{array}{rcl}
inc(M) & = & \lambda z.\, \lambda s.\, s\, (M\, z\, s) \\
dec(M) & = & \lambda z.\, \lambda s.\, M\ \mathbf{pair}(z, z)\ (\lambda p.\, \mathbf{pair}(p\, snd, s(p\, snd)))\ fst \\
equal_0(M) & = & M\ \overline{true}\ (\lambda x.\, \overline{false}) \\
equal_{n+1}(M) & = & and(equal_n(dec(M)), not(equal_0(M)))
\end{array}
$$

$$
is{-}none(M) \ = \ M\ \overline{true}\ (\lambda x.\, \overline{false})
$$

THEOREM 17 (CORRECTNESS) *Suppose* $M_1 \in bool(t_1)$, $M_2 \in bool(t_2)$, *and* $N \in num(n)$. *Then*

1. $not(M_1) \in bool(\neg t_1)$
2. $and(M_1, M_2) \in bool(t_1 \wedge t_2)$
3. $or(M_1, M_2) \in bool(t_1 \vee t_2)$
4. $inc(N) \in num(n+1)$.
5. *If* $n = 0$ *then* $dec(N) \in num(0)$ *else* $dec(N) \in num(n-1)$.
6. $equal_{n'}(N) \in bool(n = n')$.
7. *If* $M \in none$ *then* $is{-}none(M) \in bool(true)$.
8. *If* $M \in some(V)$ *then* $is{-}none(M) \in bool(false)$.

From these basic data types, it is easy to derive n-ary tuples and lists — a list is defined as either none or some pair of a value (the head) and a list (the tail) — as well as conditional expressions and switching based on a natural number. With this machinery, it is easy to encode an instantaneous description of a Turing machine T as well as the appropriate move function on encoded descriptions.

For example, if we use the definition of a Turing machine given in Section 7.2 of Hopcroft et al. [15], then one of T's instantaneous descriptions can be encoded as the tuple $(q, \sigma_1, s, \sigma_2)$, where q is T's state (encoded as a natural number); σ_1 is a list of the tape symbols (represented by natural numbers) to the left of the head, rightmost first; s is the tape symbol under the head; and σ_2 is a list of the tape symbols to the right of the head, leftmost first.

The move function for T, which maps instantaneous descriptions to the next instantaneous description, is then implemented by branching on T's state followed by the symbol under the head. Each branch simply returns the appropriate next description, which is easily derived from the old one. If the original description was d, then an action to move left after writing 2 then enter state 13 would look something like $(13, cons(2, \#2\, d), hd(\#4\, d), tl(\#4\, d))$, where hd returns 0 — the

blank symbol — and *tl* returns nil when applied to nil, and $\#n$ selects the nth component of a tuple. It is also easy to write a function $halted(d)$ that returns $\overline{true}$ if d describes a instantaneous description of a halted Turing machine (just check to see if d's state is a final one).

Finally, to complete our simulation, we need to write a function *run* that takes a starting instantaneous description and repeatedly applies the move function until T is in a halting state. We will use the following fixpoint operator to do this:

$$\mathbf{fix}(V) = (\lambda f.\, V\ \lambda a.\, f\ f\ a)\ (\lambda f.\, V\ \lambda a.\, f\ f\ a)$$

It is easy to check that $\mathbf{fix}(-)$ defines a fixpoint:

$$\mathbf{fix}(V) \hookrightarrow V\ \lambda x.\, \mathbf{fix}(V)\ x$$

The definition of *run* using **fix** might then look like the following:

$$run = \mathbf{fix}(\lambda f.\lambda d.\ \ \mathbf{if}\ halted\, d\ \mathbf{then}\ d\ \mathbf{else}\ f(move\, d))$$

THEOREM 18 (TURING COMPLETENESS) *λ_v is Turing complete.*

Proof: The proof sketch of this section can be fleshed out and turned into a complete proof. ■

COROLLARY 4 (TURING COMPLETENESS)
The following languages are Turing complete: $\lambda_v^{\rightarrow,\mu}$, sumless $\lambda_v^{\rightarrow,\mu}$, $\lambda_v^{\rightarrow,E}$, sumless $\lambda_v^{\rightarrow,E}$, and (**unit**→**unit**)→(**unit**→**unit**)*-only.*

Proof: Follows from Theorems 18 and 16 plus Corollaries 3 and 2. ■

Throughout this paper, we have made the usual assumption when dealing with lambda calculi that different result values such as $\overline{true}$ and $\overline{false}$ can be distinguished. If we instead only allow non-functional values to be distinguished from each other (e.g., all functions are indistinguishable from each other), we need to add at least two non-functional values (e.g., true and false) to λ_v to make it Turing complete.

No additions, however, are required to prove sumless $\lambda_v^{\rightarrow,E}$ Turing complete under this alternative assumption because we can distinguish between our simulation of λ_v yielding a value in $bool(true)$ and a value in $bool(false)$ using the following sumless $\lambda_v^{\rightarrow,E}$ function:

$$\begin{aligned} test(x) &= \overline{\{(x\ signal\ \lambda y.y)\ (\lambda z.z)\}};\ () \\ \\ signal &= packv(\star\rightarrow\star,\ \lambda x{:}\star\,.\,\mathbf{raise}_{\mathbf{unit}}(())) \\ \star &= \mathbf{unit}\rightarrow\mathbf{unit} \end{aligned}$$

Here, if $M \in bool(true)$ then $test(M)$ evaluates $\overline{\{signal\, \lambda z.z\}}$, which raises a unit-carrying exception, and if $M \in bool(false)$ then $test(x)$ evaluates to $()$. It is easy

to use *test* to extract whatever information is required from a Turing machine's output.

A similar observation applies to (**unit**→**unit**)→(**unit**→**unit**)-only; we need only change *signal* to raise a (**unit**→**unit**)→(**unit**→**unit**) exception instead.

4.5. Related work

Two previous papers by de Groote [6] and Rehof et al. [24], consider the termination (and hence complexity) of the simply-typed lambda calculus extended with a non-standard notion of unchecked exceptions. Unlike the results of this paper, however, they show that their systems are strongly normalizing and hence not Turing complete.

Their notion of exceptions differs greatly from ours because their motivation is to create a language whose type system corresponds to classical logic through the Curry-Howard isomorphism while we are interested in exceptions as used in real programming languages. Unlike the exceptions we have been describing, their exception flavors are syntactically required to each have exactly one handler.

This requirement makes it impossible for a function to have an exceptional return that different callers can choose to catch and process as they choose; their style of exceptions is therefore not very useful for programming. Moreover, our encoding of recursive types will not work in their system because we need this ability to "smuggle" out information from exception-raising functions. Arguably, call/cc has a similar limitation (each continuation can only be "handled" at one place), which prevents using it to "smuggle" information out of functions.

5. Conclusion

We have shown by a novel method that unchecked exceptions can be used to simulate recursive types. From this result and the well known fact that the untyped lambda calculus can be encoded in the simply-typed lambda calculus ($\lambda^{\rightarrow}$) plus recursive types, it follows that the untyped lambda calculus can be encoded in $\lambda^{\rightarrow}$ suitably extended with unchecked exceptions. Because the untyped lambda calculus is Turing complete, this implies that all computable functions can be written in an extension of $\lambda^{\rightarrow}$ with unchecked exceptions. The ability to have exceptions of distinguishable flavors, possibly carrying values of different types, is not required.

From previous work of the author's with Robert Harper [13], it is known that all proposed sound methods of adding call/cc to F_ω (a superset of $\lambda^{\rightarrow}$) preserve the fact that all programs terminate. It follows from this that only a subset of the recursive functions can be written in these extensions of F_ω. Since the set of all computable functions is proper superset of the recursive functions, $\lambda^{\rightarrow}$ when suitably extended with unchecked exceptions is strictly more powerful than all reasonable extensions of F_ω with call/cc. Hence, under reasonable assumptions, not even a full global transformation on programs can rewrite away unchecked exceptions in F_ω extended with call/cc.

Acknowledgments

We are grateful to Robert Harper, Mark Leone, Cormac Flanagan, and the anonymous referees for their comments on earlier drafts of this work.

Notes

1. CLU does provide one built-in unchecked exception called *failure*, which carries a string.
2. For technical reasons, the paper actually uses a fully-applied version of the primitive described here. However, all of the paper's results apply immediately to this version as well.
3. More precisely, terms of the form $\Lambda\alpha{:}K.\, M$ are allowed only when M is a call-by-value value. (Because this strategy evaluates under lambda abstractions, $\Lambda\alpha{:}K.\, M$ is considered a value here only when M is itself a value.)
4. The condition on the exposed variables can always be satisfied by alpha-renaming $C[M]$ appropriately.
5. Technically, we only need an isomorphism between the set of transformed terms with original type $\mu\alpha.A$ and the set of transformed terms with original type $[\mu\alpha.A/\alpha]A$.
6. $\lambda_v^{\rightarrow,\mu}$'s type system ensures that closed values of recursive type A can only be constructed using $\mathbf{roll}_A(-)$.

References

1. Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin*. ACM, January 1989.
2. Ken Arnold and James Gosling. *The Java Programming Language*. Addison Wesley, 1996.
3. Luca Cardelli. Typeful programming. Technical Report 45, DEC Systems Research Center, 1989.
4. William Clinger and Jonathan Rees. Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-Sep. 1991.
5. Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.
6. Philippe de Groote. A simple calculus of exception handling. In M. Dezani and G. Plotkin, editors, *Second International Conference on Typed Lambda Calculi and Applications*, pages 201–215. Springer-Verlag LNCS 902, 1995.
7. Matthias Felleisen. On the expressive power of programming languages. In *Science of Computer Programming*, volume 17, pages 35–75, 1991.
8. Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 10(2):235–271, 1992.
9. Andrzej Filinski. *Controlling Effects*. PhD thesis, Carnegie Mellon University, 1996. Available as Carnegie Mellon University technical report CMU-CS-96-119.
10. Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieure*. PhD thesis, Université Paris VII, 1972.
11. Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A generalization of exceptions and control in ML-like languages. In *1995 Conference on Functional Programming and Computer Architecture*, pages 12–23, La Jolla, CA, June 1995.
12. Robert Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. *LISP and Symbolic Computation*, 6(4):361–380, November 1993.
13. Robert Harper and Mark Lillibridge. Operational interpretations of an extension of F-omega with control operators. *Journal of Functional Programming*, 6(3):393–418, May 1996.
14. Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, CA, January 1995.

15. John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
16. Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.
17. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.
18. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (revised edition)*. MIT Press, 1997.
19. John Mitchell and Robert Harper. The essence of ML. In *Fifteenth ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988.
20. John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.
21. Frank Pfenning and Peter Lee. LEAP: A language with eval and polymorphism. In *TAPSOFT '89, Proceedings of the International Joint Conference on Theory and Practice in Software Development, Barcelona, Spain*, pages 345–359. Springer-Verlag LNCS 352, March 1989.
22. Benjamin Pierce, Scott Dietzen, and Spiro Michaylov. Programming in higher-order typed lambda-calculi. Technical Report CMU-CS-89-111, Carnegie Mellon University, Pittsburgh, Pennsylvania, March 1989.
23. Gordon Plotkin. Call-by-name, call-by-value, and the lambda calculus. *Theoretical Computer Science*, 1:125–159, 1975.
24. J. Rehof and M. H. Soerensen. The λ_Δ calculus. In M. Hagiya and J. Mitchell, editors, *Theoretical Aspects of Computer Software*, pages 516–542. Springer-Verlag LNCS 789, 1994.
25. Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, November 1990.