A Generalization of Exceptions and Control in ML-like Languages

Carl A. Gunter
University of Pennsylvania
gunter@cis.upenn.edu

Didier Rémy
INRIA, Rocquencourt
Didier.Remy@inria.fr

Jon G. Riecke
AT&T Bell Laboratories
riecke@research.att.com

Abstract

We add functional continuations and prompts to a language with an ML-style type system. The operators significantly extend and simplify the control operators in SML/NJ, and can be themselves used to implement (simple) exceptions. We prove that well-typed terms never produce run-time type errors and give a module for implementing them in the latest version of SML/NJ.

1 Introduction

Exceptions and continuations are two common means of altering control in mostly functional call-by-value languages, even in statically typed languages. All dialects of the ML language, for instance, build in an exception mechanism, including the ML of the LCF theorem prover [7], CAML [11], and Standard ML (SML) [13]. The exception system of, *e.g.*, SML, consists of three operations: one for declaring new exceptions, one for "raising" an exception (possibly with a value), and one for "handling" raised exceptions. The second construct, the continuation mechanism, can be added in principle to any dialect of ML [3], but a full scale implementation is to our knowledge only a part of SML/NJ, the New Jersey implementation of SML. The continuation mechanism consists of two primitives: `callcc` (call-with-current-continuation) which reifies the control stack as a function and passes it to another function, and `throw` which invokes a continuation on an argument. Both mechanisms, particularly the exception mechanism, are useful: exceptions can be used to recover from errors in an efficient, elegant, and uncluttered way, and continuations can be used to implement other control features, *e.g.*, concurrency [19].

It is folklore (the authors know of no published proof) that neither 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. In this paper we provide a generalization of simple exceptions and continuations in an ML-style type system. We prove that the language is type-safe, *i.e.*, evaluation of programs cannot generate run-time type errors. There are two interesting and important aspects of the type system. First, unlike the type system of SML/NJ, our system requires no new type constructor for continuations (we introduce one for prompts). Second, the semantics of the language overcomes some of the anomalies of `callcc` in the top-level interactive loop, and allows a somewhat cleaner style of programming than `callcc`.

The assignment of types for exceptions is generally well understood, but the assignment of types for continuation-based operations is not. Sitaram and Felleisen [24] were the first to give a limited type system for continuation-based operations. They added `callcc` into PCF, a simply-typed language with basic arithmetic, and used the typing rule

$$\frac{A \vdash a : (\tau \rightarrow \texttt{nat}) \rightarrow \tau}{A \vdash (\texttt{callcc}\; a) : \tau}\ (\texttt{callcc})$$

where A specifies the types of free variables. The limitation in the typing is obvious: only a continuation whose result type is `nat` can be reified. In essence, the problem of typing `callcc` in PCF hinges on the fact that one must pick *one* return type for continuations.

The type `nat` is a canonical choice in PCF, but not in languages with more type structure. In a language with ML-style polymorphism, `callcc` ought to have a polymorphic type with an arbitrary return type. Duba, Harper, and MacQueen [3] have proposed adding a unary type constructor `cont` for typing `callcc` which hides the return type of the continuation. The type for `callcc` is[1]

```
callcc : ('1a cont -> '1a) -> '1a
```

in their proposal, which is the type given in the SML/NJ implementation. To invoke the continuation, one uses the operation

```
throw : 'a cont -> 'a -> 'b
```

Although there are only two type variables in the type of `throw`, in actuality *three* types are necessary to explain the type of `throw`. For instance, in the expression

```
==> 5 > (1 + callcc (fn k =>
             if s = "a" then throw k 2
             else size s))
```

where `==>` denotes the "prompt" of the interactive loop, there is the argument type `int` of the continuation, which must be the same as that expected by the context in which it was reified; there is the type `int` of the context in which the throw is invoked; and there is the type `bool` of the value

[1] Note that the type variable is imperative [8]; in SML/NJ this is indicated by the number ("weakness") in the variable '1a. For the moment, the reader need not worry about imperative variables.

To appear in proceedings of FPCA'95

returned to the prompt after a value is throw'n to the reified continuation. The third type is not directly represented in the types for `callcc` and `throw`, but is rather "hidden" in the abstract type constructor `cont`.

The failure to represent the prompt type can lead to difficulty with the operational behavior of `callcc`. From a theoretical standpoint, what Wright and Felleisen [28] call "strong soundness" fails to hold. A language satisfies **strong soundness** if the type obtained from evaluating an expression is the type assigned statically; the absence of run-time type errors is what Wright and Felleisen term "weak soundness". One can see why the issue of strong soundness arises in the following session of the SML/NJ interactive loop:

```
==> val c = callcc (fn k=> fn x=>
                    throw k (fn y=> x+4));
val c = fn : int -> int
==> fun g () = 5 > (c 2);
val g = fn : unit -> bool
```

The value of `g ()` should be a function that returns $2+4$ given any integer (and probably also resets the value of `g` to the previously declared value, if any), even though the type system predicts the type `bool`. The same behavior would happen any time a continuation is stored in some data structure like a closure or reference cell. SML/NJ regards this as an anomaly, and resolves the problem by placing "prompt stamps" on reified continuations and aborting with a runtime exception if a continuation is invoked under a prompt that does not match its stamp.

Our approach to typing `callcc` is simpler: we force the missing prompt type to be included in the type of the reified continuation. If we were to modify the constructor `cont`, the continuation `k` above would have a type like

```
(int -> int) cont (int -> int)
```

where the right side is the type of the prompt. This information could be used in the typing of an expression that throw's to this continuation. For instance, a top-level phrase yielding a value of boolean type in which a value is throw'n to `k` must be rejected as having a type error. A logical extension of this idea is to allow the programmer to insert explicit control points representing his own prompts. This idea is not new; Felleisen introduced first-class prompts in the untyped setting [4]. It simplifies matters to resume execution at the control point marked by the prompt, thus leaving the issue of whether to resume the computation within the reified control to the program. To make this work with types in the way outlined above, such a reification must carry the type of the enclosing prompt. Our design is intended to make it possible to check the correctness of this type statically. We achieve this by requiring that prompts be *typed* and *named*—for Felleisen's original prompts, in contrast, there is only a single, untyped prompt [4]. The reified control fragments can then be treated as functions—that is, we do not need the type constructor `cont`, only the function type constructor `->`.

Before beginning the formal treatment, let us see how one example works. To begin with, we create a new prompt by a gensym-like primitive operation `new_prompt`:

```
==> val p = new_prompt (): int prompt
val p : int prompt
```

This prompt can be set at control points expecting an integer and used to delimit a control fragment that returns an integer. Two more primitives are required: (`set` p `in` a) which *sets* prompt p in expression a, and the primitive (`cupto` p `as` k `in` b), which reifies the *control up to* p and binds this to k in the expression b. Thus,

```
==> 5 > (set p in 1 + (cupto p as k in 2 + (k 3)))
val it = false : bool
```

binds to `k` the control `1 + [_]` (the control up to the point where the prompt was set), *i.e.*, an `int`-expecting, `int`-returning continuation), and evaluates `(2 + (k 3))` in the control context `5 > [_]` (not `5 > 1 + [_]`). When `k` is invoked as a function with 3 as its argument, the expression `5 > 2 + 1 + 3` is evaluated to `false`. The reification `k` is treated as the ordinary function `fn x => 1 + x`.

Notice how similar these operations are to exceptions and `callcc`, *e.g.*, control behavior as provided by `callcc` is achieved by setting a prompt at top level. In fact, the operation that reifies the continuation is a typed version of Felleisen's "functional continuation" operator $\mathcal{F}$ [4], an operator that captures continuations as *functions* whose application does not necessarily abort the computation. In terms of macro-expressiveness, $\mathcal{F}$ can express `callcc` and other control operators. The `new_prompt` and `set` operations, though, have direct analogs in the exceptions of ML: `new_prompt` declares a new prompt just like the keyword `exception` generates a new exception value, and `set` marks a breakpoint on the call stack just as `try` in CAML or `handle` in SML marks a breakpoint (although these have a handler associated with them). After first describing the syntax and operational semantics of our language and proving that the language is type safe, we show how to express a generalization of `callcc` and simple exceptions, and show how to implement the operations in a manner as efficient as the implementation of callcc.

2 A Typed Language with Prompts

Table 1 defines the grammar of the language. The syntax is that of a restricted version of the core of ML (without base constants, references, and exceptions) with three extra constructs for manipulating the control flow of a program: `new_prompt`, `set _ in _`, and `cupto _ as _ in _`. The language is based on primitive syntax classes of variables x and prompts p. The construct `new_prompt` returns a fresh prompt; `set _ in _` establishes a new dynamic extent for the prompt to which the first subexpression evaluates and runs the second subexpression; and `cupto _ as _ in _`, where cupto is an abbreviation for "control upto", reifies a continuation. The `cupto` operation binds its second subexpression—which must be a variable—to the control up to the value of its first subexpression—which much be a prompt—in the scope of its third. Binding conventions for the λ-calculus portion of the language are the usual ones; we identify all terms up to renaming of bound variables. We use the notation $a[b/x]$ to denote the capture-free substitution of term b for variable x in term a.

The typing rules for the language are given in Table 2. Here, A stands for a type context whose syntax is given in Table 1. The operation $\texttt{close}(A,\tau)$ returns a type scheme $(\forall\,\alpha_1\ldots\alpha_n.\,\tau)$, where $\{\alpha_1,\ldots,\alpha_n\}$ is the set of type variables occurring free in τ but not in A. The syntax restricts the expression bound by `let` to be a **value**, *i.e.*, an expression that causes no immediate subcomputation. The type system becomes unsound if the syntax of `let` is left unrestricted. This phenomenon—first pointed out by Tofte [25, 26] in the context of typing references in ML—has been well-documented in the case of continuations (*cf.*

Table 1: Syntax

$a ::=$	Expression
$\quad v$	Value
$\quad \mid (a_1\ a_2)$	Application
$\quad \mid \texttt{let val}\ x = v\ \texttt{in}\ a$	Polymorphic let binding
$\quad \mid \texttt{set}\ a_1\ \texttt{in}\ a_2$	Set a prompt
$\quad \mid \texttt{cupto}\ a_1\ \texttt{as}\ x\ \texttt{in}\ a_2$	Reify control upto a prompt
$v ::=$	Value
$\quad x$	Variable
$\quad \mid ()$	Unit value
$\quad \mid \texttt{new\_prompt}$	Generate new prompt
$\quad \mid (\lambda x.\, a)$	Abstraction
$\quad \mid p$	Prompts
$\tau ::=$	Type
$\quad \alpha$	Type variable
$\quad \mid \texttt{unit}$	Unit type
$\quad \mid (\tau \rightarrow \tau)$	Function type
$\quad \mid (\tau\ \texttt{prompt})$	Type of prompts
$\sigma ::= \forall\, \alpha_1 \ldots \alpha_k.\, \tau$	Type scheme
$A ::= \emptyset \mid A[x : \sigma] \mid A[p : \tau]$	Typing context

Table 2: Typing Rules.

$$\frac{x : \forall\, \alpha_1 \ldots \alpha_n.\, \tau \in A}{A \vdash x : \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]}\ \text{(Var)} \qquad \frac{}{A \vdash () : \texttt{unit}}\ \text{(Unit)} \qquad \frac{p : \tau \in A}{A \vdash p : (\tau\ \texttt{prompt})}\ \text{(Prompt Const)}$$

$$\frac{A[x : \tau_0] \vdash a : \tau_1}{A \vdash (\lambda x.\, a) : (\tau_0 \rightarrow \tau_1)}\ \text{(Fun)} \qquad \frac{A \vdash a_1 : \tau_2 \rightarrow \tau_1 \qquad A \vdash a_2 : \tau_2}{A \vdash (a_1\ a_2) : \tau_1}\ \text{(App)}$$

$$\frac{}{A \vdash \texttt{new\_prompt} : (\texttt{unit} \rightarrow \tau\ \texttt{prompt})}\ \text{(Prompt)} \qquad \frac{A \vdash a_1 : (\tau_1\ \texttt{prompt}) \qquad A[x : (\tau_0 \rightarrow \tau_1)] \vdash a_2 : \tau_1}{A \vdash \texttt{cupto}\ a_1\ \texttt{as}\ x\ \texttt{in}\ a_2 : \tau_0}\ \text{(Cupto)}$$

$$\frac{A \vdash a_1 : (\tau\ \texttt{prompt}) \qquad A \vdash a_2 : \tau}{A \vdash \texttt{set}\ a_1\ \texttt{in}\ a_2 : \tau}\ \text{(Set)} \qquad \frac{A \vdash v : \tau_1 \qquad A[x : \texttt{close}(A, \tau_1)] \vdash a_2 : \tau_2}{A \vdash \texttt{let val}\ x = v\ \texttt{in}\ a : \tau_2}\ \text{(Let)}$$

Table 3: Operational Semantics.

$E ::=$	Evaluation context
$\quad [\_]$	Hole
$\quad \mid (E\ a) \mid (v\ E)$	Application
$\quad \mid \texttt{set}\ E\ \texttt{in}\ a \mid \texttt{set}\ p\ \texttt{in}\ E$	Set
$\quad \mid \texttt{cupto}\ E\ \texttt{as}\ x\ \texttt{in}\ a$	Cupto

Redex reductions

$$\begin{array}{rcll}
((\lambda x.\, a)\ v)/P & \longrightarrow_{red} & a[v/x]/P & \\
\texttt{let val}\ x = v\ \texttt{in}\ a/P & \longrightarrow_{red} & a[v/x]/P & \\
(\texttt{new_prompt}\ ())/P & \longrightarrow_{red} & p/\{p\} \cup P & \qquad p \notin P \\
\texttt{set}\ p\ \texttt{in}\ v/P & \longrightarrow_{red} & v/P & \\
\texttt{set}\ p\ \texttt{in}\ E_p[\texttt{cupto}\ p\ \texttt{as}\ x\ \texttt{in}\ a]/P & \longrightarrow_{red} & (\lambda x.\, a)\ (\lambda y.\, E_p[y])/P &
\end{array}$$

Context reductions

$$\frac{a_0/P_0 \longrightarrow_{red} a_1/P_1}{E[a_0]/P_0 \ \longrightarrow\ E[a_1]/P_1}$$

[8, 12, 28]). The type system adopts Wright's proposal [27] (which was well-known to Tofte) of restricting the form of `let` to "value-only polymorphism" rather than adding imperative type variables as the SML definition does. For better readability we use the syntactic sugar ($\texttt{let}\ x = a_1\ \texttt{in}\ a_2$) for $((\lambda x.\, a_2)\ a_1)$ for the monomorphic `let`. Some familiar facts follow immediately from the form of the type system. It is not hard to see that every term has a unique typing derivation, and that one may easily derive an algorithm (based on unification) that derives a principal type.

A rewriting semantics in the style of [4] (a convenient reformulation of structured operational semantics [15]) is given in Table 3. The semantics is given in two parts: the first part defines a collection of **evaluation contexts**, which specify the positions in which a redex can be reduced, and the second part specifies a collection of rules defining a binary relation $\longrightarrow_{red}$ for the reduction of redexes. Intuitively, to do a step of evaluation on a term a, one finds a context E and a redex a_0 such that $a \equiv E[a_0]$ and $a_0 \longrightarrow_{red} a_1$; then $E[a_0] \longrightarrow E[a_1]$. Our redex reductions are of the slightly more complex form $a_0/P_0 \longrightarrow_{red} a_1/P_1$, meaning "the redex a_0 with prompts P_0 reduces to expression a_1 with prompts P_1" so reductions in an evaluation context have the form $E[a_0]/P_0 \longrightarrow E[a_1]/P_1$. The set P_i—the current set of allocated prompts—is much like a "store" in an operational semantics of references, and determines the previously allocated prompts. Thus, the expression ($\texttt{new\_prompt}\ ()$) allocates a "fresh prompt" relative to the current P. Also, in the redex rules, the notation E_p denotes an evaluation context in which the hole is not in the scope of a setting of prompt p. The rules specify how to reify a continuation and pass a value up to the nearest dynamically enclosing prompt.

A few examples should make the behavior of the reduction semantics more apparent. For instance, the expression

$$\begin{array}{l}\texttt{let}\ x = \texttt{new\_prompt}\ ()\ \texttt{in}\\ \texttt{set}\ x\ \texttt{in cupto}\ x\ \texttt{as}\ k\ \texttt{in}\ (k\ (\lambda z.\, z))\end{array}$$

first allocates a fresh prompt, sets the dynamic scope to be this prompt, reifies the (empty) continuation as k, and passes to k the identity function. The final result is thus the identity function. At a high level, the formal steps are

$$\begin{array}{l}(\texttt{let}\ x = \texttt{new\_prompt}\ ()\ \texttt{in}\\ \ \texttt{set}\ x\ \texttt{in cupto}\ x\ \texttt{as}\ k\ \texttt{in}\ (k\ (\lambda z.\, z)))\ /\ \emptyset\\ \quad\longrightarrow \texttt{set}\ p\ \texttt{in cupto}\ p\ \texttt{as}\ k\ \texttt{in}\ (k\ (\lambda z.\, z))\ /\ \{p\}\\ \quad\longrightarrow (\lambda k.\, k\ (\lambda z.\, z))\ (\lambda x.\, x)\ /\ \{p\}\\ \quad\longrightarrow ((\lambda x.\, x)\ (\lambda z.\, z))\ /\ \{p\}\\ \quad\longrightarrow (\lambda z.\, z)\ /\ \{p\}\end{array}$$

This expression is also well-typed in the language: the variable x has type $((\alpha \to \alpha)\ \texttt{prompt})$ and the continuation k has type $((\alpha \to \alpha) \to (\alpha \to \alpha))$. Another example is that of an abortive computation:

$$\begin{array}{l}\texttt{let}\ x = \texttt{new\_prompt}\ ()\ \texttt{in}\\ \texttt{set}\ x\ \texttt{in cupto}\ x\ \texttt{as}\ k\ \texttt{in}\ (\lambda z.\, \lambda y.\, y)\end{array}$$

which aborts the computation and passes $(\lambda z.\, \lambda y.\, y)$ to the top-level.

There is actually more latitude in assigning operational semantics to the language than it first appears. For instance, any of the following rules preserve the strong type soundness theorem below:

$$\begin{array}{l}\texttt{set}\ p\ \texttt{in}\ E_p[\texttt{cupto}\ p\ \texttt{as}\ x\ \texttt{in}\ a]/P\\ \qquad\longrightarrow_{red}\ \texttt{set}\ p\ \texttt{in}\ ((\lambda x.\, a)\ (\lambda y.\, E_p[y]))/P\\ \texttt{set}\ p\ \texttt{in}\ E_p[\texttt{cupto}\ p\ \texttt{as}\ x\ \texttt{in}\ a]/P\\ \qquad\longrightarrow_{red}\ \texttt{set}\ p\ \texttt{in}\ ((\lambda x.\, a)\ (\lambda y.\, \texttt{set}\ p\ \texttt{in}\ E_p[y]))/P\end{array}$$

The first rule grabs the functional continuation but leaves the prompt p set in the continuation; this corresponds to the operational semantics of Felleisen's $\mathcal{F}$ operation [4]. The second rule also leaves the prompt p set, but also grabs the "set" when the functional continuation is reified; this corresponds to the operational semantics of Danvy and Filinski's `shift` operation [2]. It is easy to see how to simulate the first rule in our semantics by adding a `set` before every body of a `cupto`. Similarly, the second rule can be simulated using the first. The other direction, though, seems not to be known—that is, whether the weaker operational rules can simulate the operational semantics we have given to `cupto`. There are even further possibilities, including ones that erase all intervening `set`'s during a `cupto` [14]. We do not feel, though, that there is a clear answer to the question of which operational rule is right; suffice it to say that we have picked one, and that the other rules lead to strong type soundness as well.

3 Type Safety

We now show that reduction preserves typing and each well-typed term never gets stuck at a run-time type error.

Type safety is a subtle issue because "getting stuck at a run-time type error" is open to interpretation. Some examples of "run-time type error" require little justification. For instance, the non-well-typed term

$$((\texttt{new\_prompt}())\ (\texttt{new\_prompt}()))/\emptyset$$

cannot be reduced past a form $(p_1\ p_2)/\{p_1, p_2\}$ for some prompts p_1, p_2; the result is obviously a run-time type error because of the attempt to apply a non-function to an argument. But the issue is subtle in the presence of control operations, and for our purposes not every "stuck" term will be a run-time type error. For instance, the well-typed term

$$\texttt{let}\ x = \texttt{new\_prompt}\ ()\ \texttt{in cupto}\ x\ \texttt{as}\ k\ \texttt{in}\ k\ /\ \emptyset$$

reduces to $(\texttt{cupto}\ p\ \texttt{as}\ k\ \texttt{in}\ k)/\{p\}$ with no further reductions possible—the continuation cannot be reified since no prompt has been set. The situation for exceptions in ML is similar: well-typed terms can still result in an "uncaught exception". In general one must accept the situation, *e.g.*, the restrictions required to make it possible to determine statically that there is no division by zero would probably be unacceptable. We leave aside these concerns and adopt an analog to the ML convention, *i.e.*, the term above does not represent a run-time type error. Theorem 9 provides a precise expression of our assumptions.

We first need a few simple lemmas about the type system that are essentially independent of control operations. Proofs of the following lemmas can be found in, *e.g.*, [18].

Lemma 1 (Type Substitution) *If $A \vdash a : \tau$, then*

$$A[\tau_0/\alpha] \vdash a : \tau[\tau_0/\alpha].$$

Lemma 2 (Extension of Type Assignment) *Let B be any type assignment whose domain contains no free variables of a. Then $AB \vdash a : \tau$ iff $A \vdash a : \tau$.*

A type scheme $\forall\, \alpha_1 \ldots \alpha_n.\, \tau$ is **more general** than a type scheme $\forall\, \alpha_1 \ldots \alpha_p.\, \tau'$ if there are types $\tau_1, \ldots \tau_n$ such that $\tau[\tau_1/\alpha_1] \ldots [\tau_n/\alpha_n]$ is equal to τ'. Similarly, a type assignment A is more general than a type assignment B if they have the same domain D and, for all $x \in D$ the value $A(x)$ of A at x is more general than $B(x)$.

Lemma 3 (Generalization of Type Assignment) *If A is more general than B and $B \vdash a : \tau$, then $A \vdash a : \tau$.*

Lemma 4 (Term Substitution) *Suppose $A \vdash a_0 : \tau_0$ and $A[x : \forall \alpha_1 \ldots \alpha_n. \tau_0] \vdash a : \tau$, where $\alpha_1, \ldots, \alpha_n$ are not free in A. Then $A \vdash a[a_0/x] : \tau$.*

The proof of type safety for our particular language requires a few definitions. A type assignment A is a **prompt assignment** if $A = \emptyset[p_1 : \tau_1] \ldots [p_n : \tau_n]$, and A' is a **prompt extension** of a prompt assignment A if A' is of the form AA'' where A'' is a prompt assignment. Evaluation of expressions may create new prompts but cannot change the type of an expression; thus, we write $a_0/P_0 \subset a_1/P_1$ if P_1 contains P_0 and, for any prompt assignment A_0 with prompts P_0 and any type τ such that $A_0 \vdash a_0 : \tau$, there exists a prompt extension A_1 of A_0 such that P_1 is the domain of A_1 and $A_1 \vdash a_1 : \tau$. It is not hard to see that the relation $\subset$ is reflexive and transitive. One may also easily prove the following lemma by induction on the structure of evaluation contexts.

Lemma 5 *If $a_0/P_0 \subset a_1/P_1$, then $E[a_0]/P_0 \subset E[a_1]/P_1$.*

The important step of reduction is the capture of the current context up to a prompt. The context E used in a program $E[x]$ will be turned into a function $\lambda x. E[x]$. The following lemma will simplify the corresponding case in the proof of subject reduction.

Lemma 6 *Suppose $A \vdash E[a_1] : \tau$. Then there exists a type τ_0 such that $A \vdash a_1 : \tau_0$ and, for any term a_2 such that $A \vdash a_2 : \tau_0$, we also have $A \vdash E[a_2] : \tau$.*

Proof: The proof is by induction on the form of the evaluation context; it relies on the fact that the hole in an evaluation context is not in the scope of any binding operation. Here are three typical cases:

Case $E = [.]$: Then pick τ_0 to be τ.

Case $E = (E'\ a_0)$: From the hypothesis we know that $A \vdash E'[a_1] : \tau_1 \to \tau$ and $A \vdash a_0 : \tau_1$. Thus, by the induction hypothesis, there is a type τ_0 such that $A \vdash a_1 : \tau_0$, and, for any a_2 such that $A \vdash a_2 : \tau_0$, we have $A \vdash E'[a_2] : \tau_1 \to \tau$. The statement now follows from the typing rule App.

Case $E = (\texttt{set}\ E'\ \texttt{in}\ a_0)$: From the hypothesis, we have $A \vdash E'[a_1] : (\tau\ \texttt{prompt})$ and $A \vdash a_0 : \tau$. Thus, by the induction hypothesis, there is a type τ_0 such that $A \vdash a_1 : \tau_0$, and, for any a_2 with $A \vdash a_2 : \tau_0$, we have $A \vdash E'[a_2] : (\tau\ \texttt{prompt})$. The statement now follows from the typing rule Set. ■

Lemma 7 (Redex Contraction) *If $a_0/P_0 \longrightarrow_{red} a_1/P_1$, then $a_0/P_0 \subset a_1/P_1$.*

Proof: Each case of redex reduction can be considered independently. Assume there is a prompt assignment A_0 with prompts P_0, a type τ such that $A_0 \vdash a_0 : \tau$; we need to exhibit a prompt extension A_1 of A_0 such that P_1 is the domain of A_1 and $A_1 \vdash a_1 : \tau$.

Case $a_0 = ((\lambda x. a)\ v)$ or $(\texttt{letval}\ x = v\ \texttt{in}\ a)$: In both cases the reduction steps for these forms do not change the set of prompts. In each case, there exists a type τ_1 and a list W of type variables not in the free variables of A_0 such that $A_0 \vdash v : \tau_1$ and $A_0[x : \forall W. \tau_1] \vdash a : \tau$. Since $a_1 = a[v/x]$, it follows from Lemma 4 that $A_0 \vdash a_1 : \tau$.

Case $a_0 = (\texttt{new\_prompt}\ ())$: Note that $A_0 \vdash \texttt{new\_prompt} : (\texttt{unit} \to \tau'\ \texttt{prompt})$ and $A_0 \vdash () : \texttt{unit}$. Suppose $a_1 = p$ where $p \notin P_0$. If $P_1 = P_0 \cup \{p\}$ and $A_1 = A_0[p : \tau']$, then $A_1 \vdash a_1 : (\tau'\ \texttt{prompt})$.

Case $a_0 = (\texttt{set}\ p\ \texttt{in}\ v)$: Trivial.

Case $a_0 = (\texttt{set}\ p\ \texttt{in}\ E_p[\texttt{cupto}\ p\ \texttt{as}\ x\ \texttt{in}\ a])$: The reduction step for setting a prompt does not introduce any new prompts. Thus, $A_0 \vdash E_p[\texttt{cupto}\ p\ \texttt{as}\ x\ \texttt{in}\ a] : \tau$ and $A_0 \vdash p : (\tau\ \texttt{prompt})$. Applying Lemma 6, there exists a type τ_1 such that $A_0 \vdash (\texttt{cupto}\ p\ \texttt{as}\ x\ \texttt{in}\ a) : \tau_1$ (1) and $A_0 \vdash E_p[a_1] : \tau$ for any expression a_1 such that $A_0 \vdash a_1 : \tau_1$ (2). From (1) it follows that $A_0[x : \tau_1 \to \tau] \vdash a : \tau$ and, consequently, $A_0 \vdash \lambda x. a : (\tau_1 \to \tau) \to \tau$. Let y be a variable that appears neither in the domain of A_0 nor in E_p. By (2) and Lemma 2, $A_0[y : \tau_1] \vdash E_p[y] : \tau$, and hence $A_0 \vdash (\lambda y. E_p[y]) : (\tau_1 \to \tau)$. Thus, $A_0 \vdash (\lambda x. a)(\lambda y. E_p[y]) : \tau$ follows. ■

Theorem 8 (Subject Reduction) *If $a_0/P_0 \longrightarrow a_1/P_1$, then $a_0/P_0 \subset a_1/P_1$.*

Proof: A simple combination of Lemmas 5 and 7. ■

Note that Theorem 8 does not hold without the value-only restriction (or other restrictions on polymorphic `let`); see [8, 12, 28] for examples.

Theorem 9 (Value Halting) *Suppose A is a prompt assignment with prompts P. If $A \vdash a : \tau$ and a/P cannot be reduced, then a is either a value or a term of the form $E_p[\texttt{cupto}\ p\ \texttt{as}\ x\ \texttt{in}\ a']$.*

Proof: The proof is by induction on the size of a. The cases when $a = ()$, `new_prompt`, p, and $(\lambda x. a')$ are trivial, so consider the remaining cases:

Case $a = (a_1\ a_2)$: There must be a type τ_2 such that $A \vdash a_1 : (\tau_2 \to \tau)$. By the induction hypothesis applied to a_1/P, a_1 either is a value or has the form $E_p[\texttt{cupto}\ p\ \texttt{as}\ x\ \texttt{in}\ a']$. The latter implies a has the form $E_p[\texttt{cupto}\ p\ \texttt{as}\ x\ \texttt{in}\ a']$, so consider the case when a_1 is a value. By the induction hypothesis applied to a_2/P, a_2 either is a value or has the form $E_p[\texttt{cupto}\ p\ \texttt{as}\ x\ \texttt{in}\ a']$. Again, the latter case means that the lemma holds, so consider the case when a_2 is a value too. Note that a_1 cannot be an abstraction, since a cannot be reduced. Since a_1 has a functional type, it can only be `new_prompt`. Hence, a_2 is of type `unit`, and it must be the value $()$. However, this is not possible since a cannot be reduced. This rules out all cases but the case when a has the form $E_p[\texttt{cupto}\ p\ \texttt{as}\ x\ \texttt{in}\ a']$, so the statement holds.

Case $a = (\texttt{letval}\ x = v\ \texttt{in}\ a_1)$: Then a could be reduced, contradicting the hypothesis.

Case $a = (\texttt{set}\ a_1\ \texttt{in}\ a_2)$: Then $A \vdash a_1 : \tau\ \texttt{prompt}$ and $A \vdash a_2 : \tau$. If a_1 is not a value, then it has the form $E_p[\texttt{cupto}\ p\ \texttt{as}\ x\ \texttt{in}\ a_0]$, and therefore a is also of the form $E'_p[\texttt{cupto}\ p\ \texttt{as}\ x\ \texttt{in}\ a_0]$ where $E'_p = (\texttt{set}\ E_p\ \texttt{in}\ a_2)$. If a_1 is a value, a_1 must be a prompt q. Note that a_2 cannot be a value, for otherwise a could be reduced. Thus, a_2 must be $E_p[\texttt{cupto}\ p\ \texttt{as}\ x\ \texttt{in}\ a_0]$ where $p \neq q$ (otherwise a can be reduced). It follows that $a = E'_p[\texttt{cupto}\ p\ \texttt{as}\ x\ \texttt{in}\ a_0]$ where $E'_p = (\texttt{set}\ q\ \texttt{in}\ E_p)$.

Case $a = (\texttt{cupto}\ a_1\ \texttt{as}\ x\ \texttt{in}\ a_2)$: Then $A \vdash a_1 : \tau_1\ \texttt{prompt}$ and $A[x : \tau_0 \to \tau_1] \vdash a_2 : \tau_1$. If a_1 is not a value, it must be of the form $E_p[\texttt{cupto}\ p\ \texttt{as}\ y\ \texttt{in}\ a_0]$ and so is a. Otherwise, it must be a prompt q and E_p must not set q. Thus a is of the form $E_q[\texttt{cupto}\ q\ \texttt{as}\ x\ \texttt{in}\ a_2]$ where $E_q = E_p$. ■

The following theorem then follows immediately from the previous two theorems:

Theorem 10 (Type Safety) *Suppose $\emptyset \vdash a : \tau$. Then one of the three following must happen:*

1. *There exists a value v and a prompt assignment A with prompts P such that $a/\emptyset \longrightarrow^* v/P$ and $A \vdash v : \tau$;*
2. *$a/\emptyset \longrightarrow^* E_p[\texttt{cupto}\ p\ \texttt{as}\ x\ \texttt{in}\ a']/P$; or*
3. *The reduction sequence starting from $a/\emptyset$ is infinite.*

4 Expressiveness

4.1 Simple exceptions

Simple exceptions are a simplification of the exception mechanism found in most ML variants. We add three new syntactic forms to our language: `new_exn`, which generates a new internal name for an exception; $(\texttt{raise}\ a_1\ a_2)$, which raises an exception a_1 with value a_2; and $(\texttt{handle}\ a_1\ a_2\ a_3)$, which evaluates a_1 to exception h and a_2 to v_2, and then evaluates a_3 so that if exception h is raised with a value v, the evaluation of a_3 aborts and handler v_2 is applied to v. To describe the formal semantics, we need internal exception names h, new evaluation contexts

$$(\texttt{raise}\ E\ a) \mid (\texttt{raise}\ v\ E) \mid$$
$$(\texttt{handle}\ E\ a\ a') \mid (\texttt{handle}\ v\ E\ a) \mid (\texttt{handle}\ v\ v'\ E)$$

and new redex rules

$$\begin{aligned}(\texttt{new\_exn}\ ())/X, P &\longrightarrow_{red} h/\{h\} \cup X, P \\ & \qquad h \notin X \\ (\texttt{handle}\ h\ v\ v')/X, P &\longrightarrow_{red} v'/X, P \\ (\texttt{handle}\ h\ v\ E_h[\texttt{raise}\ h\ v'])/X, P &\longrightarrow_{red} (v\ v')/X, P\end{aligned}$$

where X is a finite set of exceptions and E_h is an evaluation context with no intervening $(\texttt{handle}\ h\ v''\ E)$ expressions. The new operations can also be typed—not surprisingly—using typings similar to those in ML. If we add a new type construction $(\tau\ \texttt{exn})$ to the syntax of types, the types of the new operations are

$$\frac{}{A \vdash \texttt{new\_exn} : (\texttt{unit} \to \tau\ \texttt{exn})}\ \text{(New Exception)}$$

$$\frac{h : \tau \in A}{A \vdash h : (\tau\ \texttt{exn})}\ \text{(Exception Const)}$$

$$\frac{A \vdash a_1 : (\tau\ \texttt{exn}) \qquad A \vdash a_2 : \tau}{A \vdash (\texttt{raise}\ a_1\ a_2) : \tau_0}\ \text{(Raise)}$$

$$\frac{A \vdash a_1 : (\tau\ \texttt{exn}) \quad A \vdash a_2 : (\tau \to \tau_0) \quad A \vdash a_3 : \tau_0}{A \vdash (\texttt{handle}\ a_1\ a_2\ a_3) : \tau_0}\ \text{(Handle)}$$

It is a simple exercise to extend the proof of Theorem 10 to the enhanced language.

Simple exceptions differ from exceptions in most ML variants in three ways. First, exceptions are generated from `new_exn` rather than declared by the keyword `exception`. This difference is inconsequential, since one may use `let` to bind an exception to a name. Second, one may not handle multiple exceptions in one handler. Again, the difference is inconsequential, since one may use multiple `handle` expressions to yield the same effect. Third, handlers must be given with respect to a specific exception. For example, in most ML variants one can write $(\texttt{handle}\ \_\ a_2\ a_3)$ that catches *any* exception raised during the evaluation of a_3—even one that is declared in a_3. This difference is substantive; wildcard patterns are a useful feature, giving the programmer the ability to recover from arbitrary errors. On the other hand, wildcard exceptions encourage some sloppiness in error handling, cause problems for reasoning about code, and prevent certain compiler optimizations (John Reppy, personal communication, August 1994).

Simple exceptions are a redundant feature in our language; we do not know about ML handlers with wildcard expressions without assuming an extensible datatype in the language.[2] That is, one may easily macro expand the three primitives for simple exceptions into our base language without exceptions but with `new_prompt`, `set`, and `cupto` (*i.e.*, simple exceptions do not change the "expressiveness", in the sense of [5], of the language). Let $[\![a]\!]$ be the notation for the translation of a term with simple exceptions to one without. The translation of `new_exn` is simply `new_prompt`, *i.e.*, $[\![\texttt{new\_exn}]\!] = \texttt{new\_prompt}$. The translation of $(\texttt{raise}\ a_1\ a_2)$ is

$$\begin{array}{l}\texttt{let}\ x_1 = [\![a_1]\!]\ \texttt{in} \\ \texttt{let}\ x_2 = [\![a_2]\!]\ \texttt{in} \\ \texttt{cupto}\ x_1\ \texttt{as}\ k\ \texttt{in}\ x_2\end{array}$$

where x_1, x_2, k are distinct fresh variables. The translation of the term $(\texttt{handle}\ a_1\ a_2\ a_3)$ is

$$\begin{array}{l}\texttt{let}\ x_1 = [\![a_1]\!]\ \texttt{in} \\ \texttt{let}\ x_2 = [\![a_2]\!]\ \texttt{in} \\ \texttt{let}\ p = \texttt{new\_prompt}\ ()\ \texttt{in} \\ \texttt{set}\ p\ \texttt{in} \\ (\lambda z.\ (\texttt{cupto}\ p\ \texttt{as}\ k\ \texttt{in}\ (x_2\ z))) \\ \quad (\texttt{set}\ x_1\ \texttt{in} \\ \quad\ \ \texttt{let}\ x_3 = [\![a_3]\!]\ \texttt{in} \\ \quad\ \ \texttt{cupto}\ p\ \texttt{as}\ k\ \texttt{in}\ x_3)\end{array}$$

where x_1, x_2, z, p, k are distinct fresh variables. The translation of an exception constant h is a prompt constant with the same name. Finally, the translation is homomorphic in all of the other operations, *e.g.*, $[\![(a_1\ a_2)]\!] = ([\![a_1]\!]\ [\![a_2]\!])$.

The translation preserves types, although this must be stated with some care. Let $[\![\tau]\!]$ be the type τ with every occurrence of $(\tau_0\ \texttt{exn})$ replaced recursively by $([\![\tau_0]\!]\ \texttt{prompt})$.

Theorem 11 *Suppose $A \vdash a : \tau$ in the extended language, and $[\![A]\!]$ is the type context with all types translated. Then $[\![A]\!] \vdash [\![a]\!] : [\![\tau]\!]$.*

The proof is a simple induction on the original typing derivation. For instance, consider $a = (\texttt{handle}\ a_1\ a_2\ a_3)$ where a_1 has type τ, a_2 has type $(\tau_0\ \texttt{exn})$, and a_3 has type $(\tau_0 \to \tau)$. To assign the right type to $[\![a]\!]$, let $p : ([\![\tau]\!]\ \texttt{prompt})$; then one may assign the type $[\![\tau]\!]$ to the `cupto` expression.

It is harder to state and prove a correctness theorem for the translation with respect to reduction: the translation of simple exceptions into the `cupto` mechanism means that one must keep careful track of which `cupto`'s and prompts belong to simple exceptions and which are in the program itself. For instance, in translating the term

$$\texttt{set}\ p\ \texttt{in}\ \texttt{handle}\ h\ (\lambda x.\,())\ (\texttt{cupto}\ p\ \texttt{as}\ k\ \texttt{in}\ ())$$

[2] An extensible datatype is an ML datatype where new constructors can be added later. Indeed, the type *exn* is such an extensible datatype in several implementations of ML, *e.g.*, CAML or SML. One can then simulate full exceptions with a unique "exception" carrying values of type *exn* and the wildcard handler becomes a regular handler.

some of the cupto's belong to the handle but one does not. To maintain the proper bookkeeping, we use a relation $a/X, P \rhd a'/P'$ where a is a term possibly involving simple exceptions and a' is a term without simple exceptions. The definition implies that $a/X, P \rhd [\![a']\!]/X \cup P$, although more terms are related. One can then prove the following

Theorem 12 *Suppose A is an exception context with exceptions X and B is a prompt context with prompts P, and $AB \vdash a : \tau$ in the exted language, and $a/X, P \rhd a'/P'$. Then*

$$a/X, P \longrightarrow^* v/X_0, P_0 \text{ iff } a'/P' \longrightarrow^* v'/P'_0$$

where $v/X_0, P_0 \rhd v'/P'_0$.

The theorem holds if the base language is extended with other ML constructs such as if-then-else and numeric constants.

4.2 Callcc

One could also add primitive operations for reifying and invoking continuations. Since the language has first-class prompts, the most natural way to add callcc to the language is to add the forms $(\texttt{callccp}\, a_1\ a_2)$, $(\texttt{throwp}\, a_1\ a_2)$, and $(\texttt{abortp}\ a_1\ a_2)$. The first arguments to callccp and abortp are the prompts delimiting the continuation to be reified or discarded. The semantics may be formalized by enhancing the grammar of evaluation contexts with

$$(\texttt{callccp}\, E\ a)$$

and the redex rules with

$$\begin{array}{l}(\texttt{set}\ p\ \texttt{in}\ E_p[\texttt{callccp}\, p\ a])/P \\ \quad\longrightarrow_{red} (\texttt{set}\ p\ \texttt{in}\ E_p[a\ (\lambda x.\, \texttt{abortp}\, p\ E_p[x])])/P \\ (\texttt{throwp}\, v_1\ v_2)/P \longrightarrow_{red} (v_1\ v_2)/P \\ (\texttt{set}\ p\ \texttt{in}\ E_p[\texttt{abortp}\, p\ a])/P \longrightarrow_{red} a/P\end{array}$$

These operations can also be given types via the rules

$$\frac{A \vdash a_1 : \tau\ \texttt{prompt} \qquad A \vdash a_2 : \tau}{A \vdash \texttt{abortp}\, a_1\ a_2 : \tau}\ (\text{Abortp})$$

$$\frac{A \vdash a_1 : \tau_0\ \texttt{prompt} \qquad A \vdash a_2 : (\tau \to \tau_0) \to \tau}{A \vdash \texttt{callccp}\, a_1\ a_2 : \tau}\ (\text{Callccp})$$

$$\frac{A \vdash a_1 : \tau \to \tau_0 \qquad A \vdash a_2 : \tau}{A \vdash \texttt{throwp}\, a_1\ a_2 : \tau_0}\ (\text{Throwp})$$

Note the difference between the typing rule for throw in [3] and SML/NJ and the typing rule for throwp here. A throw expression in SML/NJ has *any* type. Syntactically this is achieved by hiding the return type of the continuation subexpression (the first subexpression). This is (weakly) semantically sound because of the global invariant that all continuations captured with callcc start by aborting the computation and resuming somewhere else. Hence, it is clear that the type of throw a_1 a_2 may be *any* type since the computation of this expression will never return to the calling context. A throwp expression, however, explicitly mentions the return type τ_0 of the continuation. In fact, the typing rule is exactly the same as for function application, which is how the operation is implemented.

As with simple exceptions, adding these continuation operations is merely adding syntactic sugar; one can translate terms with them to terms without. For instance, the translation of a term $(\texttt{abortp}\, a_1\ a_2)$, denoted $[\![\texttt{abortp}\, a_1\ a_2]\!]$ as above, is

$$\begin{array}{l}\texttt{let}\ x_1 = [\![a_1]\!]\ \texttt{in} \\ \texttt{cupto}\ x_1\ \texttt{as}\ k\ \texttt{in}\ [\![a_2]\!]\end{array}$$

The definition of $[\![\texttt{callccp}\, a_1\ a_2]\!]$ is

$$\begin{array}{l}\texttt{let}\ x_1 = [\![a_1]\!]\ \texttt{in} \\ \texttt{cupto}\ x_1\ \texttt{as}\ k\ \texttt{in} \\ \texttt{set}\ x_1\ \texttt{in}\ (k\ ([\![a_2]\!]\ (\lambda x.\, \texttt{abortp}\, (k\ x))))\end{array}$$

Finally, $[\![\texttt{throwp}\ a_1\ a_2]\!] = ([\![a_1]\!]\ [\![a_2]\!])$. The translation is correct, *i.e.*, the analogs of Theorems 11 and 12 hold for this translation.

5 Implementation

To complete the argument that prompts and cupto are simpler and easier to use than callcc, we show that the cupto can be implemented as efficiently (in an asymptotic sense) as callcc.

Our operations—including multiple prompts—can be implemented as a module in SML/NJ with the signature in Table 4. Other implementations of functional continuation operators appear in the literature: for instance, Filinski [6] shows how to encode control operators with callcc and one reference cell under the assumption that there is one prompt. The module provides a way to translate complete programs in our language to SML programs. The module has three primitives new_prompt, set and cupto that implement the constructs of the same name. Since there is no macro facility in SML, the user of the module must adopt the conventions

$$\begin{array}{r}[\![\texttt{set}\ a_1\ \texttt{in}\ a_2]\!] = \texttt{set}\ [\![a_1]\!]\ \texttt{(fn () =>}\ [\![a_2]\!]\texttt{)} \\ [\![\texttt{cupto}\, a_1\ \texttt{as}\ x\ \texttt{in}\ a_2]\!] = \texttt{cupto}\ [\![a_1]\!]\ \texttt{(fn x =>}\ [\![a_2]\!]\texttt{)}\end{array}$$

The other constructs of our language can be translated directly into SML/NJ, *e.g.*, $[\![()]\!] = ()$ and 1

$$[\![\lambda x.\, a]\!] = (\texttt{fn x =>} [\![a]\!]).$$

Appendix A gives an implementation in SML/NJ. Our encoding is clearly of the same flavor as the untyped encoding of shift and reset [22] into Scheme with callcc, but it is not easy to relate them in a precise way, since the languages that they encode are also different.

Notice that the signature involves weak type variables (*cf.* [10, 26]). If SML were modified to have value-only polymorphism, the signature of this module would be identical but without the "weaknesses" on the type variables. The weaknesses never cause a problem in translating programs in our language to SML, since our language has "value-only" polymorphic let.

The encoding of multiple prompts is more difficult than that of a single prompt, since we have to push on the same control stack continuations to prompts of possibly different types. The elements of the stack are variant types. Since we do not know in advance the number of prompts that will be defined, we need an open variant type. Some ML variants, among them SML, provide one built-in extensible datatype of exception values. The generativity of exception declarations allows one to define a function that returns a new exception (holding a value of fixed but arbitrary type) each time it is called. Any locally extensible datatype could replace the use of exceptions; we never use exception handling or raising except for reporting errors. This is a practical justification for giving extensible datatypes full status in ML; we have met a more theoretical justification in Section 4.

Assuming that we already have an efficient implementation of callcc, *i.e.*, the cost of callcc and throw are

Table 4: Signature for Prompts.

```
signature PROMPT =
  sig
    exception Uncaught_prompt      (* to report uncaught prompt *)
    type 'a prompt
    val new_prompt : unit -> '1a prompt
    val set :  '1a prompt -> (unit -> '1a) -> '1a
    val cupto : '1a prompt -> (('2b -> '1a) -> '1a) -> '2b
  end
```

constant, the encoding yields an efficient implementation of `cupto`. The conditions are clearly met by most cps compilation strategies, such as those used in SML/NJ. Given such an efficient implementation, the cost of `new_prompt` and `set` is clearly constant. The cost of `cupto` may, however, be proportional to the size of the control stack. The cost of applying a continuation is also proportional to the small piece of control stack stored in the closure of that continuation; a similar cost, of course, exists with the more traditional `throw`. Each of the above operations takes constant time for programs that use a single prompt. In particular, this is the case for all programs with only `callcc`, whether they have been straightforwardly rewritten with `cupto`'s, or linked with a module that implements `callcc` and `throw` with `cupto`'s. Moreover, the small factor by which the cost is increased in the simulation might be compensated by the conciseness of programs using `cupto` rather than `callcc`.

For stack-based compilation strategies, `callcc` is an expensive operation; since our simulation relies on `callcc`, the simulated operations will also be expensive. We could easily extend the simulation to implement a "restoring" `cupto`, so that the common pattern

$$(\texttt{cupto}\ x\ \texttt{as}\ k\ \texttt{in set}\ x\ \texttt{in}\ k\ a)$$

is directly implemented in terms of `callcc`. Programs using only `callcc` should run about as fast with primitive `callcc` as with `callcc` simulated with restoring `cupto`. Replacing `callcc` by `cupto` thus should not decrease performance.

On the other hand, it is quite difficult to predict whether programming with functional continuations would be more efficient than programming with usual, aborting continuations. The obvious reason is that such a judgement depends on programming styles and therefore it is very subjective. Another important factor is the quality of the implementation of continuations.

Clearly, one cannot obtain better performance when simulating functional continuations with aborting continuations. When setting a prompt, the simulation reifies the current continuation (*i.e.*, copies the current stack) and continues with a quasi-empty stack. Later, when control is reified up to the prompt, the current stack represents the context up to the prompt. A primitive implementation of `cupto`'s would certainly mark the stack when setting a prompt so that copying the context from the root to the prompt is avoided.

Since continuations can express `callcc`, one may program with functional continuations using only total continuations. Capturing a functional continuation cannot be faster than capturing a total continuation of the same size. However, in many examples (see the next section), `callcc`'s come in pairs with some little protocol that actually implements functional continuations. Thus, we now consider the problem of comparing programs written with `cupto`'s with their counterpart in terms of `callcc`. That is, we compare the efficiency of our simulation to a primitive implementation of `cupto`.

For sake of simplification, we consider a restoring `cupto` and the following scenario (context and stack can be interchanged, according to which one provides better intuitions): the execution starts in an empty stack which grows to a control point E_1 where a prompt is set. The evaluation continues with a_1 and reaches a control point $E_1[\texttt{set}\ p\ \texttt{in}\ E_2[\_]]$ where the context up to p is reified as k and evaluation continues (*i.e.*, the mark and the context k are left on the stack) with a_2. The whole program is of the form $E_1[\texttt{set}\ p\ \texttt{in}\ a_1]$ where a_1 is itself

$$E_2[\texttt{cupto}\ x\ \texttt{as}\ k\ \texttt{in set}\ x\ \texttt{in}\ k\ a_2].$$

The comparison of performances naturally depends on the quality of the compilation of continuations. Let us call an implementation "naive" if it always copies the part of stack corresponding to the context that is reified. With a naive implementation of continuations, primitive functional continuations are clearly more efficient than simulated ones: both E_1 and E_2 are copied by the simulation while only E_2 is copied with a primitive implementation. There are also "smart" implementations that copy the stack lazily, *i.e.*, just before the stack is popped. Given support from the garbage collector, this may avoid copies of reified continuations that have become unreachable at the time when copying should occur. We do not know whether smart compilation would equally benefit to both the simulated and the primitive `cupto`. It might also be the case that if prompts are set frequently, cutting up the stack would become unnecessary in the case of prompts, *i.e.*, a naive primitive implementation of `cupto` might run as efficiently as a smart implementation of `callcc`.

Queinnec and Serpette have described an implementation of functional continuations [17] that never copies the stack. Roughly, their idea is to freeze some active part of the stack, and jump over that part until it becomes garbage. However, their semantics differs from ours since prompts are erased from the context during reification (see Section 7). It is not clear that their compilation schema can be applied to our semantics, and, if the schema can be applied, whether one obtains good performance. Moreover, their method requires garbage collection on the stack and it penalizes block allocation. This makes the implementation closer to a stackless implementation, and performance should be comparable to the case of CPS-implementations.

6 Programming Examples

In this section we briefly describe two small examples that give the flavor of how to use our operations and the implementation in SML/NJ: traversing a tree and coroutines. Nothing about the examples is really original or complex: indeed, neither example makes use of multiple, named prompts in the same way that the encoding of exceptions does. Nevertheless, the examples do illustrate how one programs with functional continuations in a typed setting.

6.1 Traversing a Tree

The first example, a small toy example suggested to us by Matthias Felleisen, demonstrates how functional continuations can express computations more succinctly than they can be expressed using `callcc`. Given an element of the SML datatype

```
datatype tree =
    Null | Cell of int | Pair of tree * tree
```

of binary trees with values at all internal nodes, we want to write two functions

```
get_first : tree -> answer
get_next : unit -> answer
```

that walk down the tree and output the values of the nodes, one at a time. Answers are elements of the datatype

```
datatype answer = None | Some of int
```

For instance, given the tree

```
val tree =
  Pair (Pair (Cell 1, Null), Pair (Cell 2, Cell 3))
```

the following output is required:

```
- get_next ();
val it = None : answer
- get_first tree;
val it = Some 1 : answer
- get_next ();
val it = Some 2 : answer
- get_next ();
val it = Some 3 : answer
- get_next ();
val it = None : answer
```

There is a very simple and generic solution, provided two functions `start` and `suspend` that, respectively, start the toplevel computation and suspend the evaluation of the computation, returning to toplevel, and a reference `resume`, that contains the suspended computation where to resume next:

```
fun walk Null = None
  | walk (Cell i) = suspend (Some i)
  | walk (Pair (t1,t2)) = (walk t1; walk t2)

fun get_first t = start (fn () => walk t)
fun get_next () = start (fn () => !resume None)
```

The implementation of the functions `start` and `resume` is straightforward with first-class prompts and functional continuation operations:

```
local
  val toplevel = new_prompt(): answer prompt
in
  val resume = ref (fn (x:answer) => x)
  fun start f = set toplevel f
  fun suspend v =
      cupto toplevel (fn k => ((resume := k); v))
end
```

This can be encoded directly into SML/NJ using `callcc` instead of our control operations without going through our implementation.

```
local
  val exit = ref (fn (x:answer) => x)
in
  val resume = ref (fn (x:answer) => x)
  fun start f =
    callcc (fn toplevel =>
      (exit := throw toplevel; f(); !exit None))
  fun suspend v =
    callcc (fn rest =>
      (resume := (fn x => throw rest x); !exit v))
end
```

While the encoding with prompts and functional continuations is rather natural, the encoding with aborting continuations is difficult to write (especially if written directly) and also harder to understand. The example illustrates the coupling of the two aborting continuations `exit` and `resume` that are replaced by a single functional continuation `resume` in the `cupto` version of the code.

6.2 Coroutines

The tree traversal example is not convincing: one can solve the problem by producing answers in a lazy stream without using continuations at all. The simulation of coroutines is a more interesting and practical example.

Suppose we want to implement the following signature of coroutines:

```
signature COROUTINES =
sig
  val coroutine: (unit -> unit) -> unit
  val fork: (unit -> unit) -> unit
  val yield: unit -> unit
  val exit: unit -> unit
end
```

The function `coroutine` establishes a context for running coroutine expressions; the other operations are the obvious functions. For instance, one would write

```
coroutine (fn () => (fork (exit); exit()))
```

for forking a trivial process and then exiting, which would yield control to the child process just created.

The implementation uses an internal prompt to establish the scope of the `coroutine` function, and relies on the simulation above (Section 5) and a module for queues:

```
structure Coroutines:COROUTINES =
struct
   open Prompt
   open Queue
   val p = new_prompt (): unit prompt
   val readyQ:(unit -> unit) queue = mkQueue ()
   fun coroutine f = set p f
   fun dispatch () =
       (dequeue readyQ ()) handle Dequeue => ()
   fun exit () =
       cupto p (fn k => coroutine dispatch)
   fun yield () =
       cupto p (fn k => (enqueue (readyQ,k);
                         coroutine dispatch))
   fun fork f =
       enqueue (readyQ,fn () => (f (); exit ()))
end
```

Notice that the state of a running coroutine is saved as a function in the queue when doing a yield.

The implementation of the same policy for fork using callcc is more complicated, and it seems to require a trick. The following code is slightly modified from [20]:

```
structure Coroutines:COROUTINES =
struct
   open Queue
   val readyQ:unit cont queue = mkQueue ()
   fun coroutine f = (f ()) handle Dequeue => ()
   fun dispatch () = throw (dequeue readyQ) ()
   fun exit () = dispatch ()
   fun fork f =
       let val newThread =
           callcc (fn k1 =>
              (callcc (fn k2 => (throw k1 k2));
               f (); exit ()))
       in enqueue (readyQ, newThread)
       end
   fun yield () =
       callcc (fn k => (enqueue (readyQ,k);
                        dispatch ()))
end
```

The two callcc's are necessary to get the right behavior from fork. The parent process must grab a continuation that represents the child process's continuation and put the child's continuation in the queue. The continuation k1 is the continuation that sets newThread to the child continuation, puts the continuation into the queue, and continues. The continuation k2, on the other hand, is the continuation that calls f and exit and continues with the rest of the program, which is the child's continuation. This pattern of two callcc's is quite common: the implementation of Concurrent ML, for instance, uses 4 instances of double callcc's (8 total) out of 28 callcc's. Implementing a different forking policy—where the child starts immediately—is easier

```
   fun fork f =
       callcc (fn oldThread =>
                (enqueue (readyQ,oldThread);
                 f ();
                 exit ()))
```

but is also quite easy using cupto:

```
   fun fork f =
       (enqueue (readyQ,fn () => (f (); exit ()));
        yield ())
```

Functional control operators make the implementation of coroutines simpler because one can separate the management of queues from the processes by prompts.

7 Comparison with Previous Work

We have already seen, in Section 2, how the operational semantics of our control operations compares with Felleisen's $\mathcal{F}$ and Danvy and Filinski's shift operation. Many other choices of functional continuation operations are possible, *e.g.*, Hieb and Dybvig's spawn [9] and Queinnec and Serpette's splitter [17]. See [16, 14] for a detailed comparison of the operational semantics of these operations.

With one exception, none of these papers consider type systems for functional continuations. The sole exception is Queinnec and Serpette's paper [17] on splitter, abort, and call/pc. These operations differ in some respects from our three operations of new_prompt, set, and cupto. Using a notation similar to ours, the types of the operations are

$$\begin{aligned}&\texttt{splitter} : (\tau\ \mathsf{prompt} \to \tau) \to \tau\\ &\texttt{abort} : \tau\ \mathsf{prompt} \to (\texttt{unit} \to \tau) \to \tau_0\\ &\texttt{call/pc} : \tau\ \mathsf{prompt} \to ((\tau_0 \to \tau) \to \tau_0) \to \tau_0\end{aligned}$$

The splitter operation sets a new prompt and runs the body. If abort is ever called with that prompt and an argument (a thunk), the prompt is erased and the thunk is called in the continuation before the splitter. If call/pc is ever invoked with a function, the continuation up to the prompt is reified and all its internal prompts are unset before it is passed as an argument. Using our notation and operations for clarity, the operational semantics can be expressed by the rules

$$\begin{aligned}&(\texttt{splitter}\ a)/P \longrightarrow_{red} (\texttt{set}\ p\ \texttt{in}\ (a\ p))/P \cup \{p\},\quad p \notin P\\ &(\texttt{set}\ p\ \texttt{in}\ E_p[\texttt{abort}\ p\ a]) \longrightarrow_{red} (a\ ())\\ &(\texttt{set}\ p\ \texttt{in}\ E_p[\texttt{call/pc}\ p\ a])\\ &\qquad \longrightarrow_{red} (\texttt{set}\ p\ \texttt{in}\ \langle E_p\rangle[a\ (\lambda x.\ E_p[x])])\end{aligned}$$

where $\langle E\rangle$ stands for the context E where all prompts have been unset, *i.e.*, $\langle \texttt{set}\ p\ \texttt{in}\ E\rangle$ is E and the transformation is homomorphic on other constructs (only prompts can be in the position of p, since set _ in _ expressions are all introduced by the reduction rule for splitter). We do not know if they proved a type soundness theorem as we have: the paper [17] does not state the theorem nor attempt to prove it, but using our proof technique it is easy to carry out.

Appart from this significant difference, Queinnec and Serpette's splitter also comes closest to ours in adding multiple prompts. Others, notably Sitaram and Felleisen [22] and Danvy and Filinski [2], have added multiple prompts and control operations to languages to obtain more control. The difference between these operations and our language (and Queinnec and Serpette's) is important: prompts in our proposal are hidden in an abstract type that only the compiler can manipulate, whereas in [2, 22] the representations of prompts are known to the programmer (as integers). The hidden representation of prompts is essential for implementing exceptions in a correct manner: the implementation generates fresh prompts that programmers cannot cupto. Also, having a special type of prompts makes it easy to incorporate prompts into a language like ML; we otherwise would need some cumbersome naming scheme for infinite sets of prompts at each type.

Aside from the rigorous treatment of types, the single identifiably new feature in our proposal is the decomposition of declaring a new prompt from setting a prompt, and the corresponding ability to set a prompt more than once. This is again used in our encoding of exceptions, but we know of no other natural examples which require one to set a prompt more than once.

8 Discussion

We have shown how to incorporate primitives for first-class prompts and the reification of control up to a prompt in a statically-typed language. Let us consider briefly the theoretical, programming, and compilation issues related to these primitives.

We believe that the primary theoretical benefit of using named, typed prompts arises in the simple proof of strong soundness. In fact, our choice of constructs can be used to simplify proofs of strong soundness for *other* control operations. For instance, to prove strong soundness for callcc, Wright and Felleisen [28] consider only expressions that do not contain an abort. Given their way of expressing the semantics of callcc, this is essentially equivalent to ruling out expressions containing continuations reified relative to a different top-level. The restriction works because any continuations reified in the course of the evaluation of a given expression must all be relative to the top-level for that expression. Our typing gives a way to explain the strong soundness for `callcc` more perspicuously: when the user types an expression, the interactive top-level loop simply creates a fresh prompt (with the type of the expression) and `set`'s; all `callcc`'s are then done via `cupto`'s to this fresh prompt.

To determine whether named, typed prompts are useful in programming requires some experience in writing programs. In the untyped case, prompts add significant expressive power [23, 21]; we believe the examples of [21] could be typed in our system. We also conjecture that many applications that currently uses `callcc` (such as various threads packages or CML) could benefit—for instance, the explicit prompt mechanism may simplify the implementation of threads in a interactive top-level loop. At the very least, the sense in which `callcc` can be easily encoded in our language should ensure that switching to explicit prompts will cost little.

A challenge left open by this work is still an efficient *direct* implementation of the operations, especially for stack-based compilation strategies.

Although our operations have better typing and programming properties than `callcc` in a language like ML, there is still the larger question of whether inexpensive, continuation-based operations are really necessary. Concurrency operations can be easily built using continuations, but there are not very many other good examples of programs that need continuations, and continuations are difficult to use for the non-expert programmer. It may well be that concurrency primitives are more fundamental and important than continuation operations, but until the *right* set of primitives is found it may be best to build in continuation operations.

Acknowledgements: We thank Andrew Appel for discussions about how "prompts" are encoded in the SML/NJ interactive top-level loop, Bruce Duba, Trevor Jim, and Christian Queinnec for several helpful discussions, and Matthias Felleisen and Tim Griffin for detailed comments on drafts.

A Implementation of Prompts with Callcc in SML/NJ

The following code implements the signature of Table 4.

```
structure Prompt : PROMPT =
  struct
    exception Uncaught_prompt
    exception Core_dumped

    type 'a cont = 'a General.cont
    val throw = General.throw
    val callcc = General.callcc

    type 'a control = 'a cont * exn list
    type 'a prompt =
      {In : ((bool * 'a cont)->exn),
       Out: ((bool * 'a cont)->'a cont * exn list)
            -> (exn -> 'a control)
            -> exn -> 'a control}
```

```
    fun new_prompt ()  =
      let exception Sharp of (bool * '1a cont)
          fun In (b,c) = Sharp (b,c)
          fun Out yes no x =
              case x
                of Sharp p => yes p | z => no z
      in ({In = In, Out = Out} : '1a prompt)
      end
```

The key idea of this encoding is to use a stack of control points (called pc for "**p**rompt and **c**ontinuation"):

```
    val stack = ref ([]: exn list)
    fun push pc = stack := pc :: !stack
    fun pop () =
      case !stack
           of [] => raise Uncaught_prompt
            | pc :: rest => (stack := rest; pc)
```

To set a prompt, the current continuation is captured and transformed into a control point associated with prompt p that is pushed on the control stack. The expression is run and control resumes at the control point found on top of the control stack, which must be a p prompt.

```
    fun set (p: '1a prompt) e =
      callcc (fn normal_cont =>
        let val _ =
             push (#In p (true,normal_cont))
            val v = e()
            val (effective_continuation, _) =
             #Out p
                  (fn (b,c) => (c,[]))
                  (fn sc => raise Core_dumped)
                  (pop())
        in
            throw effective_continuation v
        end)
```

When capturing control, the stack of control will be copied up to the corresponding prompt. Fake prompts (b is false) are ignored.

```
    fun pop_control (p:'a prompt) =
      let fun pop_it control =
         #Out p
              (fn (b,c) =>
               if b then (c, control)
               else pop_it (#In p (b,c):: control))
              (fn pc => pop_it (pc :: control))
              (pop())
      in pop_it [] end
```

When a control is used as a function the saved control stack will be appended to the top of the current control stack.

```
    fun push_control (pc :: control) =
        (push pc; push_control control)
      | push_control [] = ()
```

In more detail, the "cupto" first captures the control stack up to the first occurrence of the prompt p, and retrieves the continuation `abort` to jump to this point. Then it captures the current continuation `x`, aborts to the prompt and keeps running in an environment where control stands for the following function: when given a value `v` it captures the current continuation as `after` and pushes it on the control stack as a fake p prompt, since it is used as a return address, but not to stop control. Then, the saved `control` is pushed on the control stack and computation jumps to position `x`. Later, when reaching prompt p, computation will resume at position `after` instead of `abort`.

```
  fun cupto p f =
    let val (abort, control) = pop_control p
    in callcc (fn x =>
       throw abort
       (f (fn v =>
          callcc (fn after =>
            let val _ =
                  push  (#In p (false, after))
                val _ = push_control control
             in
               throw x v
            end))))
    end
end (* struct *)
```

References

[1] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] Olivier Danvy and Andrezj Filinski. Representing control: A study of the cps transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[3] Bruce F. Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 163–173. ACM, 1991.

[4] Matthias Felleisen. The theory and practice of first-class prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190. ACM, 1988.

[5] Matthias Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991.

[6] Andrzej Filinski. Representing monads. In *Conference Record of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 446–457. ACM, 1994.

[7] Michael J. C. Gordon, Robin Milner, and C.P. Wadsworth. *Edinburgh LCF: A Mechanical Logic of Computation*, volume 78 of *Lect. Notes in Computer Sci.* Springer-Verlag, 1979.

[8] Robert Harper and Mark Lillibridge. ML with `callcc` is unsound, July 1991. Message sent to the "sml" mailing list.

[9] Robert Hieb and R. Kent Dybvig. Continuations and concurrency. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 128–136, 1990.

[10] My Hoang, John C. Mitchell, and Ramesh Viswanathan. Standard ML-NJ weak polymorphism and imperative constructs. In *Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science*, pages 15–25, 1993.

[11] Xavier Leroy. *The Caml Light system, release 0.6 – Documentation and user's manual.* INRIA, 1993. Included in the Caml Light distribution.

[12] Xavier Leroy. Polymorphism by name for references and continuations. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 220–231. ACM, 1993.

[13] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[14] Luc Moreau and Christian Queinnec. Partial continuations as the difference of continuations: A duumvirate of control operators. In *Sixth International Symposium on Programming Languages, Implementations, Logics and Programs*, 1994.

[15] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus Univ., Computer Science Dept., Denmark, 1981.

[16] Christian Queinnec. A library of high level control operators. *Lisp Pointers*, 1993.

[17] Christian Queinnec and Bernard Serpette. A dynamic extent control operator for partial continuations. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 174–184. ACM, 1991.

[18] Didier Rémy. Extending ML type system with a sorted equational theory. Research Report 1766, Institut National de Recherche en Informatique, BP 105, F-78 153 Le Chesnay Cedex, 1992.

[19] John H. Reppy. *Higher-order concurrency.* PhD thesis, Computer Science Department, Cornell University, Ithaca, NY, January 1992. Available as Cornell Technical Report 92-1285.

[20] John H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1995. To appear.

[21] Dorai Sitaram. Handling control. In *Proceedings of the 1993 ACM Conference on Programming Language Design and Implementation*, pages 147–155. ACM, 1993.

[22] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, 1990.

[23] Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *LISP and Symbolic Computation*, 3:67–99, 1990.

[24] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: Full abstraction for models of control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 161–175. ACM, 1990.

[25] Mads Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, Edinburgh University, 1988.

[26] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, 1990.

[27] Andrew K. Wright. Polymorphism for imperative languages without imperative types. Technical Report COMP TR93-200, Department of Computer, Rice University, 1993.

[28] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.