—Last revision: September 1998—

Return Types for Functional Continuations*

Carl A. Gunter
University of Pennsylvania

Didier Rémy
INRIA, Rocquencourt

Jon G. Riecke
Bell Laboratories

Abstract

We study the typing of control operators in a language with an ML-style type system. We introduce a new set of control operators that subsume most control operators that have been proposed for languages such as Scheme or ML. We prove subject reduction for the extended language. We also show how the new operators can easily encode a variety of already known constructs: callcc, shift and reset, exceptions, references as well as new variants of these constructs. We also describe an implementation of functional continuations in terms of the more primitive callcc operator.

1 Introduction

Control operations are a heavily used feature in mostly functional languages. For instance, all dialects of the ML language—including the ML of the LCF theorem prover [7], CAML [10], and Standard ML (SML) [13]—build in an exception mechanism. Exceptions give the programmer the ability to recover from errors in a modular and efficient fashion. Continuations are another, though less common, control facility that can be added to any dialect of ML [3]. The continuation mechanism consists of two primitives: `callcc` (call-with-current-continuation) which reifies the entire control context as a function and passes it to another function, and `throw` which invokes a continuation on an argument, aborting the current computation. Continuations can be used to implement other control features, *e.g.*, concurrency [18]. Both exceptions and continuations preserve the inherent type soundness properties of ML.

In this paper we give a generalization of the continuation mechanism of [3] for a language with 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 generalization. First, unlike the type system of SML/NJ, our system requires no new type constructor for continuations; continuations have functional type. These "continuations" are really "functional continuations". Functional continuations—a programming language feature in which portions of the control context can be reified as an ordinary, non-abortive function—have been studied in the context of untyped languages [2, 4], but not in the context of ML-like languages. Second, functional continuations overcome some of the anomalies of `callcc` in the top-level interactive loop, allow a cleaner style of programming than `callcc`, and increase the expressiveness of the language .

Type systems for continuation-based operations are not well understood. Sitaram and Felleisen [22] were the first to give a limited type system for continuation-based operations. They added `callcc`

*A preliminary version entitled *A generalization of exceptions and control in ML-like languages* appeared in *Proceedings of the 1995 Conference on Functional Programming and Computer Architecture*, ACM Press, 1995.

into PCF, a simply-typed language with basic arithmetic, and used the typing rule

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

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 complex 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

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

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, consider typing the following expression

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

at the top-level, 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 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 [26] 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". This difference between strong and weak 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` applied to `()` should be a function that, given any value, returns 6, and so the type should be a function type. Nevertheless, 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 the initial continuations and aborting with a runtime exception if an initial continuation is invoked under a prompt that does not match its stamp.

Our approach to typing `callcc` is simpler: we force the missing type of the *prompt* 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 [4] and Danvy and Filinski [2] study first-class prompts in the untyped setting. 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, such a reification must carry the type of the enclosing prompt. Our design makes it possible to check the correctness of this type statically. We achieve this by requiring that prompts be *typed* and *named*—for Felleisen's and Danvy and Filinski's original prompts, in contrast, there is only a single, untyped prompt [2, 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] or Danvy and Filinski's `shift` operator, operators that capture continuations as *functions* whose application does not necessarily abort the computation. In terms of macro-expressiveness, $\mathcal{F}$ and `shift` 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 in the control context 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 as efficiently as `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 up to", reifies a function from the control context. The `cupto` operation binds its second subexpression—which must be a variable—to the control up to the value of its first subexpression—which must evaluate to 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 [13, 25]. The type system becomes unsound if the syntax of `let` is left unrestricted, (this phenomenon—first pointed out by Tofte [23, 24] in the context of typing references in ML—has been well-documented in the case of continuations [8, 11, 26]). 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, *e.g.*, one may easily construct an algorithm (based on unification) that derives a principal type, as in ML.

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. 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 $E[a_0] \longrightarrow E[a_1]$. The 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 (`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. To simplify the examples, the term ($\texttt{let}\ x = a\ \texttt{in}\ a'$)—where the binding of x is not a value—stands for the term $((\lambda x.\, a')\ a)$. For instance, the expression

```
let x =new_prompt () in
set x in cupto x as k in (k (λz. z))
```

Table 1: Syntax

$a ::=$		Expression
	v	Value
	$\mid (a_1\ a_2)$	Application
	$\mid \texttt{let}\ x = v\ \texttt{in}\ a$	Polymorphic let binding
	$\mid \texttt{set}\ a_1\ \texttt{in}\ a_2$	Set a prompt
	$\mid \texttt{cupto}\ a_1\ \texttt{as}\ x\ \texttt{in}\ a_2$	Reify control up to a prompt
$v ::=$		Value
	x	Variable
	$\mid ()$	Unit value
	$\mid \texttt{new\_prompt}$	Generate new prompt
	$\mid (\lambda x.\, a)$	Abstraction
	$\mid p$	Prompts
$\tau ::=$		Type
	α	Type variable
	$\mid \texttt{unit}$	Unit type
	$\mid (\tau \to \tau)$	Function type
	$\mid (\tau\ \texttt{prompt})$	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.

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

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

$$\frac{}{A \vdash \texttt{new\_prompt} : (\texttt{unit} \to \tau\ \texttt{prompt})}\ \text{(Prompt)}$$

$$\frac{A \vdash a_1 : (\tau_1\ \texttt{prompt}) \qquad A[x : (\tau_0 \to \tau_1)] \vdash a_2 : \tau_1}{A \vdash \texttt{cupto}\ a_1\ \texttt{as}\ x\ \texttt{in}\ a_2 : \tau_0}\ \text{(Cupto)}$$

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

Table 3: Operational Semantics.

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

Redex reductions

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

Context reductions

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

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

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

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:

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

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:

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

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]. All of these choices (as well as other, less interesting choices) of operational rules lead to strong type soundness. We have merely focused on one of the more powerful forms.

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 is a run-time type error. For instance, the well-typed term

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

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". 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.

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 τ_1, $\ldots$ τ_n such that $\tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n] = \tau'$, where $\tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ is the result of simultaneously substituting the τ_i's for the α_i's. 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 $E[x]$ is 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: By induction on the form of the evaluation context; the proof 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 $(\texttt{let}\ x = v\ \texttt{in}\ a)$: 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 `cupto` 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, 11, 26] 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 $a = (\texttt{let}\ x = v\ \texttt{in}\ a_1)$**:** 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 either*

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.*

Table 4: Signature for Prompts.

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

4 Expressiveness

In this section, we consider several extensions to the base language: regular (aborting) continuations, simple exceptions, and global references.

In this section, we assume that cupto is provided in the language as a module with the following interface given in table 4. That is, we will write `cupto` a_1 $(\lambda x.\, a_2)$ and `set` a_1 $(\lambda().\,)a_2$ instead of `cupto` a_1 `as` x `in` a_2 and `set` a_1 `in` a_2.

4.1 Callcc

Using prompts, we can provide a safe implementation of `callcc`. However, as opposed to callcc in SML/NJ, a different prompt is set (here explicitly) at the begenning of each phrase. The type system will then prohibit interaction of control between different toplevel phrases, and avoid the runtime anomaly of callcc in SML/NJ.

This extension is given as a module with the following interface:

```
type ('a,'b) fcont
type 'a toplevel
val shift : 'a toplevel -> (('b,'a) cont -> 'a) -> 'b
val reset : 'a toplevel -> (unit -> 'a) -> 'a
val callcc : 'a toplevel -> (('a,'b) cont -> 'b) -> 'b
val throw : 'a toplevel -> ('a,'b) cont -> 'b -> 'c
val eval : ('a toplevel -> 'a) -> 'a
```

Each phrase will be evaluated after setting a new prompt.

```
let eval a =
  let toplevel = new_prompt () in
    set toplevel (fun () -> a toplevel);;
```

The type of the prompt passed to a is also the type of the toplevel phrase. (Here, the user is responsible for setting the prompt at the beginning of each phrase, by calling the eval function.) The abstract type `'a toplevel` of this prompt (see the interface) forces the user to call eval once, but it does not prevent from several calls to eval, which would be erroneous.

```
type 'a toplevel = 'a prompt
```

Continuations are all returning to the toplevel. Thus, their type `('a,'b) fcont` uses an extra parameter `'b` to remember the type of the toplevel phrase.

```
type ('a,'b) fcont = 'a -> 'b;;
```

The encoding uses an auxiliary operator `shift` that captures the current evaluation up to the toplevel prompt, then resets the toplevel prompt, and starts a new evaluation in which the aborted computation is bound.

```
let shift toplevel a = cupto toplevel (fun k -> set toplevel (fun () -> a k))
```

The typing constraint imposes that the return of the new expression is also the type of the toplevel phrase. Note that the traditional `abort` is in fact a special case of `shift` when the new evaluation does not uses the aborted continuation). The operator `reset` is just the same as set:

```
let reset = set
```

The most interesting operation is `callcc`, which can easily be expressed with shift :

```
let callcc toplevel a =
  shift toplevel
    (fun k -> k (fun () -> (a (fun r -> k (fun () -> r)))))
  ();;
```

The current continuation k is shifted but it is immediately reinstalled before the evaluation proceeds with a. The only difficulty is to get the evaluation order right.

Here, k is captured as a non-aborting functional continuation. Thus, when k is resumed it must first abort the current continuation. This is realized the following function.

```
let throw toplevel v1 v2 = shift toplevel (fun k -> v1 v2);;
```

Since continuations captured with callcc have the abstract type "('a,'b) cont", they can only be reified by calling `throw`. This ensures that the current evaluation will always be aborted right before a continuation is reinstalled. This use of throw instead of a functional continuation allows a better typing of reification: since reification abort the current evaluation, the return type a throw expression is unconstrained. .

These continuations gives a better account of continuations in an interactive language. However, they do not provide more safety: a new distinct prompt is set at the begenning of each phrase so that different toplevel phrases may return values of different types. Thus, if a control operation is transmitted from one phrase to another one, e.g. through a closure or a reference cell, the prompt of the old control will not match the current prompt and an uncaught-prompt error will occur dynamically. This will be detected statically whenever the return type of the two continuations do not match. However, this is not detected when the two phases have the same return type. We could strengthen security by giving each toplevel prompt a different abstract type.

The uncaught-prompt error will occur even if the continuation does not actually attempt to return to an old toplevel. In the follow section, we explain how to recover the less permissive behavior of Sml/Nj.

4.2 Stamped toplevel prompts

We now provide a module with the following interface:

```
type 'a cont
val callcc: ('a cont -> 'a) -> 'a
val throw: 'a cont -> 'a -> 'b
val eval: (unit -> 'a) -> 'a
```

To allow control to cross toplevel boundaries, one solution is to make all toplevel prompts identical. Thus, one could make the following definitions:

```
let toplevel = new_prompt();;
let callcc f = callcc toplevel f;;
let throw v1 v2 = throw toplevel v1 v2;;
```

This will actually work, but all phrases will be forced to have the same type. Indeed, if several phrases exchange control, they must have the same prompt and therefore they must return values of the same type.

The implementation of callcc in Sml/Nj allows several phrases to communicate as long as a phrase never returns to the toplevel prompt of another one. This is actually sufficient to ensure safety, but the proof of this fact relies on a global invariant and cannot be derived from types.

We can easily provide an implementation of the above behavior, but we must bypass the type system at one point. In the implementation we will assume that the toplevel prompt is unit. (Any arbitrary contant type would be fine).

```
type 'a cont = ('a, unit) fcont;;
```

Then, we replace the eval function by

```
let phrase = ref (ref ());;
exception Toplevel;;

let eval a =
  let p = ref() in
  phrase := p;
  set (Obj.magic toplevel)
    (fun () -> let r = a() in if !phrase == p then r else raise Toplevel);;
```

The reference `phrase` contains a marker of the current phrase. The exception `Toplevel` is used to report a prompt mismatch as in Sml/nj.

The function `eval` differs from the previous definition in two ways. First, it disables the typing constraint between the type of the phrase and the one of the prompt. This is corrected by marking each phrase with a new stamp, and checking before the result is returned that the static marker of the phrase is also the marker of the phrase that is currently being evaluated.

There is another slightly less natural encoding that does not require any magic. Instead, once can use the exceptional mechanism of cupto (see next section) to "jump other the type constraint". Only the eval function need to be rewritten:

```
let eval a =
  let p = ref() in
```

```
phrase := p;
let q =  new_prompt() in
set q
  (fun () ->
      set toplevel
        (fun () ->
           let r = a() in
           if !phrase == p then cupto q (fun _-> r) else raise Toplevel);
      raise Toplevel);;
```

The potential type error has been replace by a potential uncaught prompt. Of course none will ever occur if our implementation is correct, but the is a meta property.

Note that one could also have used a local exception instead of a local prompt to produce the same effect. As is well known, local exceptions provide are an extensible datatype (several indenpendent extensible datatypes can then be obtained using the generativity of the module system). In turn, extensible datatypes provides some weak form of dynamics, which is one what is needed in the above example. As we shall see in the next section exception as implementable in terms of cupto's...

4.3 Simple exceptions

Simple exceptions are a simplification of the exception mechanism found in most ML variants. Simple exceptions require three new forms: `new_exn`, which generates a new internal name for an exception; (`raise` a_1 a_2), which raises an exception a_1 with value a_2; and (`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. The semantics of exceptions uses internal exception names h, new evaluation contexts

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

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 (`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 (τ `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.

The ability to encode simple exceptions represents a distinct increase in expressive power over the language with `callcc`. The argument, due to Mark Lillibridge, is remarkably simple. In the polymorphic λ-calculus without recursion, adding `callcc` does not change the terminating nature of computations, while adding simple exceptions suddenly permits the programmer to write nonterminating computations. The encoding, which actually permits the encoding of the entire untyped λ-calculus, appears in [12].

Simple exceptions differ from the form of exceptions found in SML and CAML 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 (`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.

Simple exceptions are a redundant feature in our language.[1] That is, one may easily 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 (`raise` a_1 a_2) is

```
let x1 = [[a1]] in
let x2 = [[a2]] in
cupto x1 as k in x2
```

where x_1, x_2, k are distinct fresh variables. The translation of the term (`handle` a_1 a_2 a_3) is

```
let x1 = [[a1]] in
let x2 = [[a2]] in
let p =new_prompt () in
set p in
(λz. (cupto p as k in (x2 z)))
  (set x1 in
   let x3 = [[a3]] in
   cupto p as k in x3)
```

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.*, $[\![(a_1\ a_2)]\!] = ([\![a_1]\!]\ [\![a_2]\!])$.

Exceptions can be provided as a module with the following signature:

```
type 'a exn
```

[1] We do not know how to encode ML handlers with wildcard expressions without an extensible datatype in the language. 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.

```
val new_exn : unit -> 'a exn
val raise : 'a exn -> 'a -> 'b
val handle : 'a exn -> ('a -> 'b) -> (unit -> 'b) -> 'b
```

and implementation

```
Open cupto;;
type 'a exn = 'a prompt;;
let new_exn = new_prompt;;

let raise p v = cupto p (fun _ -> v);;

let handle p h a =
  let q = new_prompt() in
  set q (fun () ->
           h (set p (fun () ->
                       let v = a() in cupto q (fun _ -> v))));;
```

4.4 References

Functional continuations can also be used to encode reference cells. This fact is not surprising: references can be encoded with Danvy and Filinski's `shift` and `reset`, and `shift` and `reset` can be encoded using our operations. Nevertheless, the ability to encode reference cells demonstrates a large increase in expressive power over a language with `callcc`: reference cells that store functions give one the ability to encode nonterminating computations, just as exceptions do. For instance, the SML program

```
let val r = ref (fn () => ())
    val f = fn x => (!r) x
in  r:= f;
    f ()
end
```

diverges, even though the code involves no recursion or recursive types.

Precisely, we will provide a module with the following interface:

```
type 'a ref
val ref : 'a prompt -> 'b -> ('b ref -> 'c) ->  'c
val (!) : 'a ref -> 'a
val (:=) : 'a ref -> 'a -> unit
val eval :  ('a  prompt -> 'a) -> 'a
```

The type `'a memory` is the type of the store for a program returning values of type `'a`.

The principle of our implementation is to closely follow the store small-step reduction semantics for references, which can be defined as follows. Expressions are extended with a finite collection of locations writte with letter l, and three primitives `ref` , ! and :=. Programs are now run in a toplevel store composed of a list of bindings. We actually prefer to treat bindings as local constructs, i.e. we extend the language of expressions, and evaluation contexts with bindings

$$a :: [] \mid \texttt{loc}\ l = a\ \texttt{in}\ aE ::= [] \mid \texttt{loc}\ l = v\ \texttt{in}\ E$$

We write E_l for a context that does not define l on the its path. We also write † for the toplevel mark. Evaluations rules are:

$$\begin{array}{rcll} \dagger E[\texttt{ref}\ v] & \longrightarrow & \dagger\texttt{loc}\ l = v\ \texttt{in}\ E[l] & l \notin dom\,(S) \\ \dagger E[\texttt{loc}\ l = v\ \texttt{in}\ E_l[!l]] & \longrightarrow & \dagger E[\texttt{loc}\ l = v\ \texttt{in}\ E_l[v]] & \\ \dagger E[\texttt{loc}\ l = v\ \texttt{in}\ E_l[l := v']] & \longrightarrow & \dagger E[\texttt{loc}\ l = v'\ \texttt{in}\ E_l[()]] & \\ \texttt{loc}\ l = v\ \texttt{in}\ v' & \longrightarrow & v' & l \notin dom\,(v') \end{array}$$

Functional expressions do interact a with the store since:

$$\frac{[a_1] \longrightarrow [a_1]}{\dagger E[a_1] \longrightarrow \dagger E[a_2]}\ \text{(Functional)}$$

Since new locations are always added in front, the evaluation of a program with originally no location stops with a value preceded by a sequence of store bindings.

In the implementation, we use one prompt to get direct access to the top of the store where new cells can be dynamically added. As usual, we evaluate a phrase after having set a new toplevel prompt

```
let eval a =
  let toplevel = new_prompt() in
    set toplevel (fun () -> a toplevel);;
```

New reference cells can be inserted in front of the current evaluation.

```
let ref toplevel (x : 'a) (a : 'a ref -> 'b)  =
  let return x = cupto toplevel (fun _ -> x) in
  let a p () = a p in
    (cupto toplevel
       (fun z ->
          (set toplevel
             (fun () ->
                let p = new_prompt() in
                  store p x (set p (fun () -> return (z (a p)))))))
    ) ();;
```

Here, `x` is the initial value of the reference cell to be created and `a` is the part of the program that should have the reference in its static scope. Thus it is an expression abstracted over the prompt `p` that will serve as a handle to reach the new reference. The function `return` is used to abort the evaluation when it terminates and jump over all the reference remaining of the store. This also prevents the reference cells to be constrained to being of the same type. The next line is just to freeze the evaluation of "`a p`" and thus drive the evaluation in a correct order. Finally, the evaluation context is captured up to the toplevel, the toplevel prompt is reset, the cell is installed, and the evaluation continues with value `p` as a handle to the new reference cell.

The cell itself is implementated as a continuation `store x p []` that is defined as follows. It waits for an action of the following form:

```
type 'a action =
    Write of ('a * (unit -> 'a action))
  | Read of ('a -> 'a action);;
```

and processes the action accordingly.

```
let rec store p (x:'a) (action : 'a action) =
 match action with
    Write (y,f) -> store p y (set p f)
  | Read f -> store p x (set p (fun () -> f x));;
```

A `Read f` request is answered by reinstalling the same store and the previous evaluation context `f` with value `x`. A `Write (y,f)` request is answered by reinstalling a the store with value `y` and the previous evaluation context `f` with value `()`. Thus, the two remaining primitives are straightforward:

```
let (!) r = cupto r (fun f -> Read f);;
let (:=) r x = cupto r (fun f -> Write (x,f));;
```

The type of reference cell is the type of its prompt handle. That is,

```
type 'a ref = 'a action prompt;;
```

Another possible encoding would be to have a single reference cell and represent the store as an association list (using extensible-data types). The encoding we gave here is more direct and more interesting. It would be possible to extend the encoding, for instance to enable explicit deallocation of reference cells during evaluation, as can be done in *C* for instance. This operation would of course be unsafe and result in an uncaught prompt.

4.5 Mixing the encodings

The three previous encodings can be merged together. However, they should be hierarchical, the higher ones preserving the invariants of the lower ones. This is actually the case if the encoding of the store is the outer one (its prompts are outer all other prompts), then the encoding of callcc, and the last the encoding of exceptions.

5 Multiple prompts

Cupto is a generalization of known control operators in two ways. On the one hand, it slightly differs from Danvy and Fellinsky's shift/reset and Felleisen's C control operators in the way it captures and resets the prompt itself. These differences are minor from a typing point of view. Our choice increases expressiveness, since other operators can be derived, but it simultaneaously weakens our invariants, which may make reasoning on programs harder. On the other hand, Cupto allows for multiple prompts. Although multiple prompts have been proposed earlier in the litterature as a way of hierachizing control, multiple prompts are more essential in our proposal since they directly allow for continuations returning values of different types.

In this section, we show that multiple prompts can in fact be derived from cuptos with a single prompt using an extensible data-type. Such an extensible data-type could be primitive, as for instance the type of exceptions, or provided as a library. To be independent, we will use the following library:

```
module type Extensible = sig
  type t
  type 'a constructor
```

```
  val create : unit -> 'a constructor
  val inject : 'a constructor -> 'a -> t
  val matches : t -> 'a constructor -> ('a -> 'b) -> (t -> 'b) -> 'b
end;;
```

which we can implement as follows

```
module Make (X : sig end) : Extensible = struct
  type t = Obj.t
  type 'a constructor = unit ref
  let create () = ref ()
  let inject c v = Obj.repr (c,v)
  let matches x c f g =
    let xc, xv = Obj.magic x in if xc == c then f xv else g x
end;;
```

The operation `create` builds a new data constructor of the extensible datatype; `inject` takes a constructor and a value and injects the value into the extensible datatype; and `matches` branches on the value of the constructor, calling the third argument or the fourth depending on whether or not the first and second arguments match. Note the use of casts, and the fact that `t`—an abstract type outside the module—is given a quite meaningless, but concrete, implementation type in the module. (Another way of implementing the extensible datatype idea without casts is to use the type of exceptions.) Thus, the stack is a stack of elements of the extensible datatype `t`.

Then we can implement a module of signature that reimplements cupto in terms of cupto, but using only a single prompt from structure `C`. Its interface is [2]

```
module Mcupto (C : PROMPT) : PROMPT =   struct
```

First, we define a new extensible datatype:

```
  module E = Extensible.Make  (struct end)

  type  action = Return of E.t | Cupto of (E.t * ((unit->action) -> action))
  type 'a rc = R of 'a | C of (((unit->action) -> action) -> action)
  type 'a prompt = 'a rc E.constructor
  let new_prompt() = E.create()

  let inject_C p x = E.inject p (C x)
  and matches_C q p f g = E.matches q p (fun (C y) -> f y) g

  exception Bug
  let inject_R p x = Return (E.inject p (R x))
  and is_R p (Return x) = E.matches x p (fun (R y) -> y) (fun z -> raise Bug)
```

The heart of the implementation is the following code

[2] To be exact, `PROMPT` should here be a restriction of the signature `PROMPT` without the `Uncaught_prompt` exception, but this is unsignificant.

```
  let control = C.new_prompt()
  let compose set k' k x = k' (fun () -> set (fun () -> k x))
  let rec set_control p a =
    let comp = compose (set_control p) in
    match C.set control a with
      Cupto (qa, k) ->
        matches_C qa
          p (fun a -> a k)
          (fun qa-> C.cupto cut (fun k'-> Cupto (qa, comp k' k)) ())
    |  x -> x

  let cupto_control p a =
    let a' k' k = a (compose (set_control p) k k') in
    C.cupto control (fun k' -> Cupto (inject_C p (a' k'), fun x -> x()))
```

Then, `set` and `cupto` are simply wrapping coercions around their -control versions

```
  let set p a =  is_R p (set_control p (fun () -> inject_R p (a())))
  let cupto p a =
    cupto_control p (fun k -> inject_R p (a (fun x -> is_R p (k x))))
end;;
```

The encoding is interesting for several reasons. It is definitely more complicated that the previous encodings, which suggests why multiple prompts should be provided as a primitive construction, rather than encoded. As previous encoding, this one is well-type, but somehow bypass the type system using extensible data-types and partial parttern matching: this is of course, at the price of possible dynamic uncaught prompts. The encoding is correct, but the proof is external as opposed to a (partial) internal proof given by the type system.

6 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.

The Appendix gives an implementation in SML/NJ. It has 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.

The time analysis of our implementation is important to consider. The analysis depends crucially upon the implementation of `callcc`. Assuming that the cost of `callcc` and `throw` are constant—as they are in cps compilation strategies used by, *e.g.*, SML/NJ [1]—the encoding yields an efficient implementation of the operations. Examining the code in the Appendix, the `new_prompt` and `set` are clearly constant time operations. The cost of `cupto` may, however, be proportional to the number of `set`'s that have been done before. If the program contains just *one* prompt—as is

the case with programs written using only `callcc`—all operations take constant time. 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, and therefore the simulated operations will also be expensive. A direct implementation of our operations might be more efficient. For example, a primitive implementation of `cupto`'s could mark the stack when setting a prompt, so that one could avoid copying the entire context. Of course, since functional continuations can express `callcc`, reifying a functional continuation cannot be faster than capturing an abortive continuation of the same size. However, in many examples (see the next section), `callcc`'s come in pairs and, in effect, implement some portion of functional continuations. In other words, the use of functional continuations in *applications* might actually increase the performance.

How can we compare the efficiency of our implementation via `callcc` to a primitive implementation of `cupto`? For sake of simplicity, consider a restoring `cupto` and the following scenario. First, the execution starts in an empty context which grows to a control point E_1 where a prompt is set. Then 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 with a_2. The whole program is of the form $E_1[\texttt{set}\ p\ \texttt{in}\ a_1]$ where a_1 is itself

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

The comparison of performance naturally depends on the quality of the compilation of continuations. Let us call an implementation "naive" if it always copies the part of context 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 context lazily, *i.e.*, just before the context 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 both the simulated and the primitive `cupto`. It might also be the case that if prompts are set frequently, cutting up the context 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 8). 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 penalizes block allocation. This makes the implementation closer to a stack-less implementation, and performance should be comparable to the case of CPS-implementations.

7 Programming Examples

7.1 Meta-programming

Manipulating control is by nature more difficult than sending values to fonctions. For that reason, it has often been argued that callcc is and should remain a meta-programming construct. That is, it should only be used by experts to implement libraries which would then be simpler and safer

to use. A typical example are subroutines. Some encoding with of subroutines with callcc can be found in . The encoding of subroutines with cupto is similar.

The section on encodings has shown that with respect to meta-programming, cupto is more expressive than callcc. Some of the encodings are not at all possible with callcc. Others would be much more difficult. Here, we show a few useful constructs than are both immediate applications of cupto and small variations on know constructs.

Patchers are a very restrictive forms of cupto that cannot capture the continuation, but only insert a patch when the execution will return at some current prompt and resume immediately.

```
let patch p f = cupto p (fun k -> f (set p k));;
```

For instance, an exception may sometimes be replaced by a warning, which should leave the execution but later report a possible mistake.

```
type 'a warning = 'a prompt;;
let new_warning = new_prompt;;
let warning p s = patch p (fun (v, l) -> (v, s::l));;
let handle p h a = let (v,l) = set p (fun () -> a(), []) in h v l;;

let arith = new_warning();;

let arith_handler v = function
      [] -> v | [s] -> print_string (s^"\n"); v
    | _ ->   print_string "Warning: many arithmetic errors\n"; v;;

let handle_arith a = handle arith arith_handler a;;

let (/) x y = if y = 0 then warning arith "division"; max_int) else x / y;;
let (mod) x y = if y = 0 then warning arith "modulo"; max_int) else x mod y;;
```

One could think of using a global reference. Indeed, the patching itself changes the state of a reference on the stack. However, the scoping rules to reach the reference are dynamics, as those or prompt or, equivalently those of exceptions. That is a warning could be dynamically trapped, as an exception can. This suggests a notion of reference following similar scoping rules.

References with backups Here, we extend the encoding of references with cupto to provide a backuping operation. Backuping follows the stack discipline, that is, the reference is implicitlty reset to its old value saved on the stack when the stack is popped. More precisely, the reduction semantics is

$$\texttt{loc}\ l = v\ \texttt{in}\ E_l[\texttt{backup}\ l\ a] \longrightarrow \texttt{loc}\ l = v\ \texttt{in}\ E_l[\texttt{loc}\ l = v\ \texttt{in}\ a]$$

Actually one also need to add a cleaning up rule

$$\texttt{loc}\ l = v\ \texttt{in}\ E_l[\texttt{loc}\ l = v\ \texttt{in}\ v'] \longrightarrow \texttt{loc}\ l = v\ \texttt{in}\ E_l[v']$$

which allows to normally restore the saved location. Note that since a localtion can only be non local if is already is global, local always locations can be clean up.

We provide backup's as a new primitive of the following type:

```
value backup : 'a ref -> (unit -> 'b) -> 'b
```

In a single-thread language, this can be trivially implemented by the following code:

```
let backup r a = let x = !r in let v = a() in r := x; v;;
```

Actually, in ML one must carefully protects against exceptions:

```
let backup r a =
  let x = !r in try let v = a() in r := x; v with z -> r := x; raise z
```

However, in languages with threads, callc, or cupto, this will not work anymore. We can easily add the following function to our implementation of references:

```
let backup r a =
  let q = new_prompt() in
  let return x = cupto q (fun _ -> x) in
  set q (fun () -> store r (!r) (set r (fun () -> return (a()))));;
```

Here, since the new value is stored on the stack, the code is multi-thread compliant. Remark that warnings can be trivially be easily encoded with backup.

Pushy exceptions Pushy exceptions are a variant of exceptions used in some lisp dialects where the handler is executed on the top of the stack. It may then decide either to continue of to stop the current evaluation. We won't detailed the implementation here, but they can easily be encoded by mixing exceptions with either warnings or backups.

Shift/reset with multiple promts In section 4, we have shown an encoding of shift/reset as a side effect of the encoding of callcc. Hierarchical shift/reset has been proposed. In fact our encoding naturally allows shift/reset with multiple promps, as a module with the following interface:

```
type 'a prompt
val shift : 'a prompt -> (('b -> 'a) -> 'a) -> 'b
val reset : 'a prompt -> (unit -> 'a) -> 'a
val new_prompt : unit -> 'a prompt
```

and the following trivial implementation, which shows that shift/reset is a simple restriction of cupto.

```
type 'a prompt = 'a Cupto.prompt;;
let shift p a = Cupto.cupto p (fun k -> Cupto.set p (fun () -> a k));;
let reset = Cupto.set;;
let new_prompt = Cupto.new_prompt;;
```

Here, the toplevel prompt has not been set, the user should ensure that "shift" should will only occur in a dynamic context that contains a "reset" occurs.

8 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 [14, 16] 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

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

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

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

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.

Apart 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.

9 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 [26] 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 [20, 21]; we believe the examples of [20] 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, Andrzej Filinski, Dan Friedman, Trevor Jim, and Christian Queinnec for several helpful discussions, Chris Okasaki for revealing problems with exceptions in our original SML/NJ implementation, and Matthias Felleisen and Tim Griffin for detailed comments on drafts.

References

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

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

[3] B. F. Duba, R. Harper, and D. 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] M. 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] M. Felleisen. On the expressive power of programming languages. *Science of Computer Programming*, 17:35–75, 1991.

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

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

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

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

[10] X. Leroy. The Objective Caml system. Software and documentation available via the URL `http://pauillac.inria.fr/ocaml/`, 1996.

[11] X. 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.

[12] M. Lillibridge. Exceptions are strictly more powerful than call/cc. Technical Report CMS-CS-95-178, School of Computer Science, Carnegie Mellon University, 1995.

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

[14] L. Moreau and C. 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] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus Univ., Computer Science Dept., Denmark, 1981.

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

[17] C. Queinnec and B. 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] J. H. Reppy. *Higher-order concurrency.* PhD thesis, Computer Science Department, Cornell University, Ithaca, NY, January 1992. Available as Cornell Technical Report 92-1285.

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

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

[21] D. Sitaram and M. Felleisen. Control delimiters and their hierarchies. *LISP and Symbolic Computation*, 3:67–99, 1990.

[22] D. Sitaram and M. 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.

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

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

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

[26] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 1994. To appear.

A Encoding cupto with callcc

We first describe the implementation in SML/NJ using `callcc`, and then give a sketch of the proof that this encoding is correct. The code has also been ported to the bytecode version of Objective CAML with `callcc` implemented in C.

A.1 Implementation of prompts with callcc in SML/NJ

There are two key ideas in the implementation. The first is to use a stack of continuations to model the prompts that have been `set`, each continuation representing the rest of the computation after the `set` expression terminates normally. A difficulty with types arises immediately, however: since prompts can have different types, continuations corresponding to different `set`'s may expect values of different types. For instance, in the code

```
let p = new_prompt ()
    q = new_prompt ()
in  1 + set p (fun () => if set q (fun () => true) then 2 else 3)
end
```

the continuation executed after the first `set` expects an `int`, whereas the continuation executed after the second `set` expects a `bool`. In other words, the continuations have types `(int cont)` and `(bool cont)` respectively. To solve the typing problem, we need a way to coerce these differently typed continuations into a single type.

A second key idea is needed in order to implement the peculiarities of our operations. When performing a `cupto`, the stack must be unwound to the point of the corresponding `set`. All but the last unwound continuation must be restored to the stack if the functional continuation is ever called, because they can be `cupto`'ed. The last unwound continuation, corresponding to the continuation after the `set`, can never be `cupto`'ed because of our semantics that erases the `set`. Nevertheless,

we must still be able to *jump* to that continuation after the `set` expression terminates. Thus, continuations on the stack come in two flavors: those that represent `set`'s that have not been `cupto`'ed and those that do.

First, we define some basic exceptions, abbreviations, and the stack operations themselves.

```
exception Prompt;;
exception Bug;;  (* should never be raised *)
type 'a control = 'a cont * exn list;;
module E = Extensible.Make (struct end);;
let stack = ref ([]: E.t list);;
let push pc = stack := pc :: !stack;;
let pop () =  match !stack with [] -> raise Prompt
                  | pc :: rest -> (stack := rest; pc);;
```

Second, we define two primitive types and two operations.

```
type 'a result = Value of 'a | Exception of exn;;
let freeze f x =  try Value (f x) with z -> Exception  z;;
let unfreeze = function Value x -> x | Exception z -> raise z;;
type 'a prompt = (bool * 'a result cont) E.constructor;;
```

The `value` type constructor wraps two possible outcomes for a computation, either a value or an exception, into a single type; `freeze` is a way of running a computation, and either catching the exception or returning the value; and `unfreeze` unlocks a frozen computation, re-raising the exception if one was raised. These operations help in controlling when exceptions get raised. The `prompt` type constructor abbreviates pairs of a continuation and a boolean, where the boolean is true if the corresponding `set` has not been `cupto`'ed. The operation

```
let new_prompt = E.create;;
```

simply creates a new constructor of the extensible datatype, one that will actually be of type `'a prompt`.

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

```
let set p e =
  unfreeze
    (callcc (fun normal_continuation ->
      let z = push (E.inject p (true, normal_continuation)) in
      let v = freeze e() in
      let (effective_continuation,_) =
           E.matches (pop())
             p (fun (b,c) -> (c,[]))
             (fun sc ->
                match v with
                    Value _ -> raise Bug
                  | Exception z -> raise z)
      in
          (throw effective_continuation v)
      ))
```

There are two cases when the body `e` is run: either `e` does one or more additional `set`'s and then raises an exception, or `e` does no more `set`'s or terminates normally (or both). In the first case, the exception gets re-raised by the fourth argument to the `E.cases` call. In the second case, the top element on the stack should have a constructor matching `p`. In that case, the value is thrown to the normal continuation, which then unfreezes the result.

When capturing control, we need to copy the stack of control into a list—the top of the stack being at the end of the list—up to the corresponding prompt. The following operation does this, copying `set`'s that have already been `cupto`'ed (`b` is false) along the way.

```
let pop_control (p:'a prompt) =
  let rec pop_more control =
     E.matches (pop())
        p (fun (b, c) ->
              if b then (c, control)
              else pop_more (E.inject p (b,c):: control))
         (fun pc -> pop_more (pc :: control))
  in pop_more [];;
let rec push_control =  function
      (pc :: control) -> (push pc; push_control control)
    | [] -> ();;
```

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

```
let cupto p f =
  let (abort, control) = pop_control p in
  let reified x v =
     unfreeze (callcc (fun after ->
       let z = push  (E.inject p (false, after)) in
       let z = push_control control in
       throw x v)) in
    callcc (fun x -> throw abort (freeze f (reified x)))
```

In words, `cupto` first unwinds the top of the control stack up to the first occurrence of the prompt `p`, calling that portion of the stack `control`, and retrieves the continuation `abort`. Note that when the continuation `abort` is captured—during a `set` operation—the continuation refers to the rest of the computation *after* the `set` operation returns. The `cupto` operation next captures the current continuation `x`, and then calls `f` with a function `(reified x)` that represents the reified portion of the stack. If `f` never calls this function `reified` and never does a `cupto`, the value returned by `f` is returned to the continuation `abort`, the part of the computation *after* the corresponding `set`.

The function `reified` is the trickiest part of the code. If `(reified x)` is called with a value `v`, it captures the current continuation `after` and pushes it on the control stack as an already `cupto`'ed `set` of the prompt `p`. This prompt cannot be `cupto`'ed, but can be used as a return address. 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`.

Other functional continuation operations can be implemented using modifications of the code. For instance, the `call/pc` operation of Queinnec and Serpette does not require the copying of the stack done here in `pop_control`, since `set`'s are erased during reification in the execution of `call/pc`. Similarly, in implementing Felleisen's $\mathcal{F}$ operation, there need be no distinction between

`set`'s that have been `cupto`'ed and those that have not, since the $\mathcal{F}$ operation does not erase the `set` during reification. We leave the modifications to the reader.