Implementing Escape Continuations in C

Jean-Michel Hufflen

University of Franche-Comté, CNRS,
FEMTO-ST institute, F-25000 Besançon, France

14 December 2023

Motivation

Continuation semantics

Examples

Bridge Scheme/C

Future work

Functions `setjmp` and `longjmp`

Included into the C language:

```
#include <setjmp.h>

int setjmp(jmp_buf env) ;
void longjmp(jmp_buf env, int val) ;
```

Functions `setjmp` and `longjmp`

Included into the C language:

```
#include <setjmp.h>

int setjmp(jmp_buf env) ;
void longjmp(jmp_buf env, int val) ;
```

`setjmp(env)` *memoizes* the current environment into the `env` variable and returns 0.

Functions `setjmp` and `longjmp`

Included into the C language:

```
#include <setjmp.h>

int setjmp(jmp_buf env) ;
void longjmp(jmp_buf env, int val) ;
```

`setjmp(env)` *memoizes* the current environment into the `env` variable and returns `0`.
`longjmp(env,val)` starts over from the return point where `env` was saved, but the returned value at this point is now `val`.

Using `setjmp`/`longjmp`

If you write something like:

$$\texttt{if (setjmp(env))}\ \underbrace{\texttt{...}}_{\text{❷}}\ \texttt{; else}\ \underbrace{\texttt{...}}_{\text{❶}}\ \texttt{;}$$

that allows you to start a computation at ❶; if you want to *escape* it, use:

```
longjmp(env,1)
```

and the computation will terminate by running ❷, getting rid of ❶.

Using `setjmp/longjmp`

If you write something like:

$$\texttt{if (setjmp(env))}\ \underbrace{\texttt{...}}_{❷}\ \texttt{; else}\ \underbrace{\texttt{...}}_{❶}\ \texttt{;}$$

that allows you to start a computation at ❶; if you want to *escape* it, use:

```
longjmp(env,1)
```

and the computation will terminate by running ❷, getting rid of ❶.
A kind of *goto*, but to a place already reached,

Using `setjmp/longjmp`

If you write something like:

$$\texttt{if (setjmp(env))}\ \underbrace{\ldots}_{❷}\ ;\ \texttt{else}\ \underbrace{\ldots}_{❶}\ ;$$

that allows you to start a computation at ❶; if you want to *escape* it, use:

```
longjmp(env,1)
```

and the computation will terminate by running ❷, getting rid of ❶.
A kind of *goto*, but to a place already reached, or a kind of *backtrack*.

Using `setjmp/longjmp`

If you write something like:

$$\texttt{if (setjmp(env))}\ \underbrace{\texttt{...}}_{❷}\ \texttt{;}\ \texttt{else}\ \underbrace{\texttt{...}}_{❶}\ \texttt{;}$$

that allows you to start a computation at ❶; if you want to *escape* it, use:

```
longjmp(env,1)
```

and the computation will terminate by running ❷, getting rid of ❶.

A kind of *goto*, but to a place already reached, or a kind of *backtrack*.

These functions have been used to implement a rough mechanism of *exceptions* but, in reality, they are closed to *continuations*.

Continuations in Scheme

$$(F_1\ (\texttt{call/cc}\ G_1)) \equiv (F_1\ (G_1\ \nleftarrow F_1))$$

If F_1 is invoked within G_1's body, it is applied and yields a *direct* result, without returning to a caller.

Continuations in Scheme

$$(F_1\ (\texttt{call/cc}\ G_1)) \equiv (F_1\ (G_1 \quad \nleftarrow F_1))$$

If F_1 is invoked within G_1's body, it is applied and yields a *direct* result, without returning to a caller. ‘$(\texttt{F}_1\ \ \texttt{...})$’ is for ‘normal’ run.

Continuations in Scheme

$$(F_1\ \ (\texttt{call/cc}\ \ G_1)) \equiv (F_1\ \ (G_1\ \ \not\leftarrow F_1))$$

If F_1 is invoked within G_1's body, it is applied and yields a *direct* result, without returning to a caller. '$\texttt{(F}_1\ \texttt{...)}$' is for 'normal' run.

Examples:

```
(+ 1 (call/cc (lambda (f1) (* 2 (f1 2023))))) ⟹
      2024
(+ 1 (call/cc (lambda (f1) (* 2 2023)))) ⟹ 4047
```

Continuations in Scheme

$$(F_1\ \texttt{(call/cc}\ G_1\texttt{))} \equiv \texttt{(}F_1\ \texttt{(}G_1\ \nleftarrow F_1\texttt{))}$$

If F_1 is invoked within G_1's body, it is applied and yields a *direct* result, without returning to a caller. ‘$\texttt{(F}_1\ \texttt{...)}$’ is for ‘normal’ run.
Examples:

```
(+ 1 (call/cc (lambda (f1) (* 2 (f1 2023))))) ⟹
      2024
(+ 1 (call/cc (lambda (f1) (* 2 2023)))) ⟹ 4047
```

Continuations captured by means of the `call/cc` function may be saved and applied *overwhelmingly*, but we are not interested in this feature.

Continuations in Scheme

$$\texttt{(}F_1\ \texttt{(call/cc}\ G_1\texttt{))} \equiv \texttt{(}F_1\ \texttt{(}G_1\ \nleftarrow F_1\texttt{))}$$

If F_1 is invoked within G_1's body, it is applied and yields a *direct* result, without returning to a caller. ‘$\texttt{(F}_1\ \texttt{...)}$’ is for ‘normal’ run.
Examples:

```
(+ 1 (call/cc (lambda (f1) (* 2 (f1 2023))))) ⟹
        2024
(+ 1 (call/cc (lambda (f1) (* 2 2023)))) ⟹ 4047
```

Continuations captured by means of the `call/cc` function may be saved and applied *overwhelmingly*, but we are not interested in this feature.
`call/ec` ⟸ *escape* continuations, dynamic extent only.

Programming languages with continuations

`call/cc` present in some languages, e.g., Ruby.

Programming languages with continuations

`call/cc` present in some languages, e.g., Ruby.
Standard ML of New Jersey provides *typed continuations*, and *escape continuations* as *typed control continuations* (being `'a control_cont` type).

Programming languages with continuations

`call/cc` present in some languages, e.g., Ruby.

Standard ML of New Jersey provides *typed continuations*, and *escape continuations* as *typed control continuations* (being `'a control_cont` type).

Haskell ⟸ *continuation monad*.

Programming languages with continuations

`call/cc` present in some languages, e.g., Ruby.

Standard ML of New Jersey provides *typed continuations*, and *escape continuations* as *typed control continuations* (being `'a control_cont` type).

Haskell ⟸ *continuation monad*.

Compiling such languages by generating C code—e.g., bigloo—a C compiler being in charge of low-level details.

C compiler certified

Some research projects aim to ensure that C codes are correctly put into action.

- Frama-C, developed using OCaml, but not fully proved yet,

C compiler certified

Some research projects aim to ensure that C codes are correctly put into action.

- Frama-C, developed using OCaml, but not fully proved yet,
- an attempt to use Coq (CompCert).

A continuation semantics for C

Functions:

$$mExp : \mathrm{Exp} \to \mathrm{Env}_f \to \mathrm{Env} \to \mathrm{State} \to \mathrm{C}_e \to (\mathcal{D} \times \mathrm{State})$$

where:

- **State** is the program's state,
- **Env** and **Env_f** are environments,
- **C_e** $= (\mathcal{D} \times \mathrm{State}) \to (\mathcal{D} \times \mathrm{State})$,
- $\mathcal{D}$ is the set of all *possible values*.

Example: assignment

$$mExp : \mathrm{Exp} \to \mathrm{Env}_f \to \mathrm{Env} \to \mathrm{State} \to \mathrm{C}_e \to (\mathcal{D} \times \mathrm{State})$$

$$\begin{array}{l} mExp\ (assignment(var, e))\ \rho_f\ \rho\ s\ k = \\ \quad \mathbf{let}\ k_0 = (v_0, s_0) \mapsto k(v_0, s_0 \oplus [\rho(var) \mapsto v_0]) \\ \quad \mathbf{in}\ mExp\ e\ \rho_f\ \rho\ s\ k_0 \\ \quad \mathbf{end} \end{array}$$

Example: assignment

$$mExp : \mathsf{Exp} \to \mathsf{Env}_f \to \mathsf{Env} \to \mathsf{State} \to \mathsf{C}_e \to (\mathcal{D} \times \mathsf{State})$$

$$\begin{array}{l} mExp\ (assignment(var, e))\ \rho_f\ \rho\ s\ k = \\ \quad \mathbf{let}\ k_0 = (v_0, s_0) \mapsto k(v_0, s_0 \oplus [\rho(var) \mapsto v_0]) \\ \quad \mathbf{in}\ mExp\ e\ \rho_f\ \rho\ s\ k_0 \\ \quad \mathbf{end} \end{array}$$

Within such a framework, the *future* of any computation is always explicit.

Functions `setjmp/longjmp`

$$mExp : \mathsf{Exp} \to \mathsf{Env}_f \to \mathsf{Env} \to \mathsf{State} \to \mathsf{C}_e \to (\mathcal{D} \times \mathsf{State})$$

$$\begin{aligned} &mExp\ (\mathtt{setjmp}(env))\ \rho_f\ \rho\ s\ k = \\ &\quad (0, s \oplus [\rho(env) \mapsto (k, s)]) \end{aligned}$$

Functions `setjmp/longjmp`

$$mExp : \mathsf{Exp} \to \mathsf{Env}_f \to \mathsf{Env} \to \mathsf{State} \to \mathsf{C}_e \to (\mathcal{D} \times \mathsf{State})$$

$$\begin{aligned}&mExp\,(\texttt{setjmp}(env))\ \rho_f\ \rho\ s\ k = \\ &\quad (0, s \oplus [\rho(env) \mapsto (k, s)])\end{aligned}$$

$$\begin{aligned}&mExp\,(\texttt{longjmp}(env), e)\ \rho_f\ \rho\ s\ k = \\ &\quad \textbf{let } (k_0, s_0) = s(\rho(env)) \\ &\quad \textbf{in } mExp\ e\ \rho_f\ \rho\ s_0\ k_0 \\ &\quad \textbf{end}\end{aligned}$$

Our framework

Showing examples: tree-sum-ep.scm—or tree-sum-ep.sml if a strongly typed language is preferred—and tree-sum.c.

Our framework (continued)

A function launches a computation by means of recursive auxiliary functions.

Our framework (continued)

A function launches a computation by means of recursive auxiliary functions.
These auxiliary functions are not internal, so all the arguments are explicit.

Our framework (continued)

A function launches a computation by means of recursive auxiliary functions.

These auxiliary functions are not internal, so all the arguments are explicit.

In particular, one argument of them expresses the way to *escape* a recursive computation leading to a dead end. For a C program, this is a variable being type `jmp_buf`.

Our framework (continued)

A function launches a computation by means of recursive auxiliary functions.

These auxiliary functions are not internal, so all the arguments are explicit.

In particular, one argument of them expresses the way to *escape* a recursive computation leading to a dead end. For a C program, this is a variable being type `jmp_buf`.

One way to express escaping a computation.

More formally

$$\texttt{(call/ec (lambda (exit-0) }\underbrace{\texttt{...}}_{\text{❶}}\texttt{))}$$

vs

$$\texttt{if (setjmp(env)) }\underbrace{\texttt{...}}_{\substack{\texttt{escape}\\ \texttt{function's}\\ \texttt{call}}}\texttt{ ; else }\underbrace{\texttt{...}}_{\substack{\text{❶}\\ \texttt{in}\\ \text{C}}}\texttt{ ;}$$

Basic equivalences

Between Scheme types and C ones, e.g., integers, trees. . .

Our functions

Let f be a Scheme (or Standard ML) function, let x (resp. y) be an element of f's domain (resp. codomain); if we denote the respective implementations in C by $C(f)$, $C(x)$, $C(y)$, then:

$$f(x) = y \Longrightarrow C(f)(C(x)) = C(y)$$

Our functions

Let f be a Scheme (or Standard ML) function, let x (resp. y) be an element of f's domain (resp. codomain); if we denote the respective implementations in C by $C(f)$, $C(x)$, $C(y)$, then:

$$f(x) = y \Longrightarrow C(f)(C(x)) = C(y)$$

How?

Our functions

Let f be a Scheme (or Standard ML) function, let x (resp. y) be an element of f's domain (resp. codomain); if we denote the respective implementations in C by $C(f)$, $C(x)$, $C(y)$, then:

$$f(x) = y \Longrightarrow C(f)(C(x)) = C(y)$$

How? Operational semantics of Scheme and continuation semantics of C.

Our functions

Let f be a Scheme (or Standard ML) function, let x (resp. y) be an element of f's domain (resp. codomain); if we denote the respective implementations in C by $C(f)$, $C(x)$, $C(y)$, then:

$$f(x) = y \Longrightarrow C(f)(C(x)) = C(y)$$

How? Operational semantics of Scheme and continuation semantics of C.
By induction on the level of recursive calls.

Our functions

Let f be a Scheme (or Standard ML) function, let x (resp. y) be an element of f's domain (resp. codomain); if we denote the respective implementations in C by $C(f)$, $C(x)$, $C(y)$, then:

$$f(x) = y \Longrightarrow C(f)(C(x)) = C(y)$$

How? Operational semantics of Scheme and continuation semantics of C.

By induction on the level of recursive calls.

Strong induction.

Future work

Several ways to escape dead-end computations.

Future work

Several ways to escape dead-end computations.
Make explicit rules as *patterns*.

Future work

Several ways to escape dead-end computations.
Make explicit rules as *patterns*.
Implementation of C's continuation semantics using Coq.

Future work

Several ways to escape dead-end computations.
Make explicit rules as *patterns*.
Implementation of C's continuation semantics using Coq.
Writing the complete article ⟸ looking for proof-readers next spring.