Harvard School of Engineering and Applied Sciences — CS 152: Programming Languages

Delimited control

Lecture 21 Thursday, April 16, 2015

1 Error handling and control

Consider the following calculus for a small untyped language with errors and error handlers.

$$\begin{aligned} e &::= x \mid \lambda x.\, e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \mid \mathsf{Error}\ v \mid \mathsf{raise}\ e \mid \mathsf{try}\ e_1\ \mathsf{catch}\ x.e_2 \\ v &::= n \mid \lambda x.\, e \mid \mathsf{Error}\ v \end{aligned}$$

$$\begin{aligned} E &::= [\cdot] \mid E\ e \mid v\ E \mid E + e \mid v + E \mid \mathsf{raise}\ E \mid \mathsf{try}\ E\ \mathsf{catch}\ x.e \\ F &::= [\cdot] \mid E\ e \mid v\ E \mid E + e \mid v + E \mid \mathsf{raise}\ E \end{aligned}$$

$$\frac{}{\mathsf{raise}\ v \longrightarrow \mathsf{Error}\ v} \qquad \frac{}{\mathsf{try}\ v\ \mathsf{catch}\ x.e \longrightarrow v}\ v \neq \mathsf{Error}\ v' \qquad \frac{}{\mathsf{try}\ \mathsf{Error}\ v\ \mathsf{catch}\ x.e \longrightarrow e\{v/x\}}$$

$$\frac{}{E[F[\mathsf{Error}\ v]] \longrightarrow E[\mathsf{Error}\ v]}$$

The semantics of the language are the same as the one we saw in lecture 19 (we show here only the most interesting rules — we have discussed the rest extensively in lecture 19). The formulation of the semantics is slightly different to that of lecture 19. In lecture 19, we introduced a set of rules dedicated to propagate errors all the way to the next enclosing error handler (or the top of the program if such a handler didn't exist). In the above rules, we take advantage of the flexibility of evaluation contexts to eliminate the need for propagating rules. In particular, we define a new kind of evaluation context, F, that are a subset of the evaluation contexts, E. The difference between f and E is that F contexts do not contain error handling frames. Thus in if an expression is of the form $E[F[e]]$, we can conclude that if there is an exception handler that is supposed to catch errors the e raises, then the corresponding frame is in the E part of the evaluation context and not in the F part. This reasoning explains our last inference rule. Since we must propagate an error to the closest error handler, we can safely drop the part of the evaluation context that does not contain any error handlers. Notice, that if the evaluation context has no error handling frames, the rule is going to return the error as the result of the whole program, as we expect.

The F evaluation contexts are not just an elegant way to avoid the multiple error propagation rules. They also help us expose an interesting fact about our error handling semantics. As we discussed above, our semantics forget the part of the evaluation context between an error and its enclosing exception handler ,i.e., when an error occurs, the semantics resume the evaluation not from the point in the evaluation trace where the error occurred but the control jumps directly in the error handling code. All this code can access from the point in the evaluation tract where the error occurred is the error message. The SCC semantics below make this fact even more obvious;

$$\begin{array}{rcl} \langle e, E\rangle & \longrightarrow_{SCC} & \langle e', E'\rangle \\ \hline \langle \mathsf{raise}\ e, E\rangle & \longrightarrow_{SCC} & \langle e, E[\mathsf{raise}\ [\cdot]]\rangle \\ \langle \mathsf{try}\ e_1\ \mathsf{catch}\ x.e_2, E\rangle & \longrightarrow_{SCC} & \langle e_1, E[\mathsf{try}\ [\cdot]\ \mathsf{catch}\ x.e_2]\rangle \\ \langle v, E[\mathsf{try}\ [\cdot]\ \mathsf{catch}\ x.e]\rangle & \longrightarrow_{SCC} & \langle v, E\rangle \\ \langle \mathsf{Error}\ v, F\rangle & \longrightarrow_{SCC} & \langle \mathsf{Error}\ v, [\cdot]\rangle \\ \langle \mathsf{Error}\ v, E[\mathsf{try}\ F[\cdot]\ \mathsf{catch}\ x.e]\rangle & \longrightarrow_{SCC} & \langle e\{v/x\}, E\rangle \end{array}$$

The interesting rule is the last one. When the control register holds an error, we inspect the contents of the context register, find the error handler closest to the hole, drop all frames between that handler and the hole, move the error handling expression of the handler to the control register and set the context register to the evaluation context that surrounds the error handler.

The following example shows our error handling CC machine in action:

$$\begin{array}{rl}
& \langle (\lambda x.\, \mathsf{try}\ 42 + x\ \mathsf{catch}\ y.y)\ \lambda x.\, x, [\cdot]\rangle \\
\xrightarrow{2}_{SCC} & \langle \mathsf{try}\ 42 + \lambda x.\, x\ \mathsf{catch}\ y.y, [\cdot]\rangle \\
\xrightarrow{2}_{SCC} & \langle 42, \mathsf{try}\ [\cdot] + \lambda x.\, x\ \mathsf{catch}\ y.y\rangle \\
\longrightarrow_{SCC} & \langle \lambda x.\, x, \mathsf{try}\ 42 + [\cdot]\ \mathsf{catch}\ y.y\rangle \\
\longrightarrow_{SCC} & \langle \mathsf{Error}\ 0, \mathsf{try}\ 42 + [\cdot]\ \mathsf{catch}\ y.y\rangle \\
\longrightarrow_{SCC} & \langle 0, [\cdot]\rangle
\end{array}$$

Observe, how in the last rule our machine discards the $42 + [\cdot]$ part of the evaluation context and resumes computation from the $y\{0/y\}$ error handling exception that has access only to the error message 0.

In many setting, though, it is useful to be able to resume evaluation after an error exactly from the point of the evaluation trace where an error occurs. For instance, consider a program that controls an airplane's landing gear. The author of this program anticipates the possibility of a division by zero and installs error handlers around any piece of the program that may perform a division (remember that in the presence of higher-order functions these are not just the pieces of the text of the program where a division occurs). Now due to the critical nature of the program, it is important for the correct evaluation of the program, if a division by zero takes place to resume the evaluation exactly from the point in the trace where the division occurs after replacing zero with a default safe number.

Error handlers that allow us to resume evaluation from the exact point that raises an error are called *resumable*. To describe their semantics precisely, we extend our calculus with an $\mathsf{try}\ e_1\ \mathsf{resume}\ f,x.e_2$ construct and specify rules in our SCC machine for the new construct:

$$e ::= \ldots \mid \mathsf{try}\ e_1\ \mathsf{resume}\ f,x.e_2$$

$$E ::= \ldots \mid \mathsf{try}\ E\ \mathsf{resume}\ f,x.e$$

$$\begin{array}{rcl}
\langle e, E\rangle & \longrightarrow_{SCC} & \langle e', E'\rangle \\
\hline
\langle \mathsf{try}\ e_1\ \mathsf{resume}\ f,x.e_2, E\rangle & \longrightarrow_{SCC} & \langle e_1, E[\mathsf{try}\ [\cdot]\ \mathsf{resume}\ f,x.e_2]\rangle \\
\langle v, E[\mathsf{try}\ [\cdot]\ \mathsf{resume}\ f,x.e]\rangle & \longrightarrow_{SCC} & \langle v, E\rangle \\
\langle \mathsf{Error}\ v, E[\mathsf{try}\ F[\cdot]\ \mathsf{resume}\ f,x.e]\rangle & \longrightarrow_{SCC} & \langle e\{\lambda z.\, F[z]/f\}\{v/x\}, E\rangle
\end{array}$$

The last rule captures the essence of resumable error handlers. When the control operator is an error, similar to the standard error handling rule, we inspect the context for the closest error handler. But now instead of passing to the error handling code only the error message, we also pass the part of the evaluation context F between the error and the error handler. To do so, we reify F as a function $\lambda x.\, F[x$ that the error handling code can call with an appropriate argument.

The following variant of our previous example shows resumable error handlers in action:

$$\begin{aligned}
& \langle(\lambda x.\, \mathsf{try}\ 42 + x\ \mathsf{resume}\ f{,}y.f\ y)\ \lambda x.\, x, [\cdot]\rangle \\
\xrightarrow{3}_{SCC}\ & \langle \mathsf{try}\ 42 + \lambda x.\, x\ \mathsf{resume}\ f{,}y.f\ y, [\cdot]\rangle \\
\xrightarrow{2}_{SCC}\ & \langle 42, \mathsf{try}\ [\cdot] + \lambda x.\, x\ \mathsf{resume}\ f{,}y.f\ y\rangle \\
\longrightarrow_{SCC}\ & \langle \lambda x.\, x, \mathsf{try}\ 42 + [\cdot]\ \mathsf{resume}\ f{,}y.f\ y\rangle \\
\longrightarrow_{SCC}\ & \langle \mathsf{Error}\ 0, \mathsf{try}\ 42 + [\cdot]\ \mathsf{resume}\ f{,}y.f\ y\rangle \\
\longrightarrow_{SCC}\ & \langle(\lambda z.\, 42 + z)\ 0, [\cdot]\rangle \\
\xrightarrow{5}_{SCC}\ & \langle 42, [\cdot]\rangle
\end{aligned}$$

In the previous to the last step of the evaluation trace, we pass to the error handling code, not only the error message 0 but also the evaluation context $42 + [\cdot]$ reified as the function $\lambda z.\, 42 + z$. This enables the error handling code to resume the addition by applying this latter function to the error message.

2 Delimited control operators

The discussion of resumable error handlers raises a question: why is a resumable error handler the only special construct that can capture part of the context and use it? Wouldn't it be useful for programmers to be able to capture some part of the evaluation context at some point in the evaluation trace and use it when they see fit?

The answer is yes. In fact, a piece of a context reified as a function is called in the programming languages literature a *functional continuation*. Operators that create functional continuations have been studied extensively over the last fifty and there have been numerous proposals of how to add them to production programming languages. Nowadays, in languages such as Scheme, Racket, Ruby, Scala, Haskell, programmers can find sets of primitives for creating and managing functional continuations. Different primitives have different semantics and expressiveness. The major distinction is to whether they allow programmers to specify what part of the context they want to capture (*delimited* continuations) or if the primitives capture the entire context by default. Another difference is whether a call to a functional continuation extends the current context with the captured continuation (*composable* continuations) or the call discards some part of the context before installing the captured continuation.

The various versions of continuation primitives can significantly improve the design of programming languages and software. First, they are the underlying mechanism to understand and implement a series of seemingly unrelated language features: exceptions and exception handlers (like the ones we discuss above), coroutines, lazy data structure generation and traversal, goal-driven backtracking. Second, continuation primitives simplify the design and implementation of a diverse variety of programs such as interactive interpreters and web servers. In general, continuation primitives give immense expressive power to programmers; they give to programmers total control over the execution of a program. Programmers can use them to jump from one part of an evaluation to another and back per will. Thus they are often compared to giving the ability to programs to time-travel.

In the remainder of this section, we discuss the semantics of delimited continuations. We start by adding to a simple untyped language three new constructs. The first, $\%\ e$, is called a prompt and installs around e a delimiter, i.e., a marker that tells to the runtime of our language that if during the execution of e we try to capture a continuation then the runtime should capture the context up to the prompt. The second, $\mathsf{abort}\ e$, gives to programmers a way to discard the part of the context between a call to abort and the closest prompt. The third, $\mathsf{call/c}\ \lambda f.\, e$, is the most important. It captures the continuation of the call to call/c up to the closest prompt, reifies it as a function and passes it to e. Here is the syntax of our language:

$$\begin{aligned}
e &::= x \mid \lambda x.\, e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \mid \%\ e \mid \mathsf{abort}\ e \mid \mathsf{call/c}\ v \\
v &::= n \mid \lambda x.\, e
\end{aligned}$$

Now that we have some intuition about our continuation primitives, we can describe their semantics formally as rules of a SCC machine:

$$\begin{aligned} E &::= [\cdot] \mid E\ e \mid v\ E \mid E + e \mid v + E \mid \%\ E \mid \mathsf{abort}\ E \\ F &::= [\cdot] \mid F\ e \mid v\ F \mid F + e \mid v + F \mid \mathsf{abort}\ F \end{aligned}$$

$$\begin{array}{rcl} \langle e, E\rangle & \longrightarrow_{SCC} & \langle e', E'\rangle \\ \hline \\ \langle \%\ e, E\rangle & \longrightarrow_{SCC} & \langle e, E[\%\ [\cdot]]\rangle \\ \langle v, E[\%\ [\cdot]]\rangle & \longrightarrow_{SCC} & \langle v, E\rangle \\ \langle \mathsf{abort}\ e, E\rangle & \longrightarrow_{SCC} & \langle e, E[\mathsf{abort}\ [\cdot]]\rangle \\ \langle v, E[\%\ F[\mathsf{abort}\ [\cdot]]]\rangle & \longrightarrow_{SCC} & \langle v, E\rangle \\ \langle v, F[\mathsf{abort}\ [\cdot]]\rangle & \longrightarrow_{SCC} & \langle v, [\cdot]\rangle \\ \langle \mathsf{call/c}\ \lambda f.\, e, E[\%\ F[[\cdot]]]\rangle & \longrightarrow_{SCC} & \langle e\{\lambda x.\, F[x]/f\}, E\rangle \\ \langle \mathsf{call/c}\ \lambda f.\, e, F[[\cdot]]\rangle & \longrightarrow_{SCC} & \langle e\{\lambda x.\, F[x]/f\}, [\cdot]\rangle \end{array}$$

The first two rules deal with prompts. The first rule installs a prompt, i.e., it marks the current context and switches the control of the machine to the expression under the prompt. The second discards a prompt marker from the current context when the expression after the machine fully evaluates to a value the expression under the prompt. The next two rules deal with abort. The first rule kicks off the evaluation inside an abort while the second pops an abort frame the current context when the machine fully evaluated the expression inside an abort. The last two rules describe the meaning of call/c. The first one captures the part of the context between the call to call/c and the closest prompt, reifies it as a function and substitutes the function for the free variable of the body of the function inside call/c. Then the machine pops F from the context and switches control to the body of the latter function. Notice, that the capture of the continuation results in the discard of the relevant prompt. The second rule fires in the absence of a prompt in which case the machine captures the whole current context.

Let's examine the semantics of prompts and call/c through the evaluation trace of a small example:

$$\begin{array}{l} \quad \langle (1 + (1 + (\lambda x.\, \%\ (1 + (1 + \mathsf{call/c}\ \lambda f.\, f\ x)))\ 0)), [\cdot]\rangle \\ \xrightarrow{7}_{SCC} \langle \%\ (1 + (1 + \mathsf{call/c}\ \lambda f.\, f\ 0)), 1 + (1 + [\cdot])\rangle \\ \longrightarrow_{SCC} \langle 1 + (1 + \mathsf{call/c}\ \lambda f.\, f\ 0), 1 + (1 + \%\ [\cdot])\rangle \\ \xrightarrow{4}_{SCC} \langle \mathsf{call/c}\ \lambda f.\, f\ 0, 1 + (1 + \%\ (1 + (1 + [\cdot])))\rangle \\ \longrightarrow_{SCC} \langle (\lambda x.\, 1 + (1 + x))\ 0, 1 + (1 + [\cdot])\rangle \\ \xrightarrow{9}_{SCC} \langle 4, [\cdot]\rangle \end{array}$$

The first interesting step is the evaluation of the call to the prompt. At that point the machine transforms the current context $1 + (1 + [\cdot])$ to $1 + (1 + \%\ [\cdot])$ in order to keep track of the prompt. Thus when the evaluation trace reaches the application call/c $\lambda f.\, f\ 0$, the machine captures the part of the context under the prompt, $1 + (1 + [\cdot])$, turns into the function $\lambda x.\, 1 + (1 + x)$ and substitutes the function for f in the body of $\lambda f.\, f\ 0$. Then the machines switches control to the result of the substitution $\lambda x.\, 1 + (1 + x))\ 0$.

Here is another similar example that demonstrates the interaction between prompts and abort:

$$\begin{array}{l} \quad \langle (1 + (1 + (\lambda x.\, \%\ (1 + (1 + \mathsf{abort}\ x)))\ 0)), [\cdot]\rangle \\ \xrightarrow{7}_{SCC} \langle \%\ (1 + (1 + \mathsf{abort}\ 0)), 1 + (1 + [\cdot])\rangle \\ \longrightarrow_{SCC} \langle 1 + (1 + \mathsf{abort}\ 0), 1 + (1 + \%\ [\cdot])\rangle \\ \xrightarrow{4}_{SCC} \langle \mathsf{abort}\ 0, 1 + (1 + \%\ (1 + (1 + [\cdot])))\rangle \\ \longrightarrow_{SCC} \langle 1 + (1 + 0), [\cdot]\rangle \\ \xrightarrow{6}_{SCC} \langle 2, [\cdot]\rangle \end{array}$$

As the evaluation trace shows, when the machine evaluates the application to abort, it drops from the current context $1 + (1 + \%\, 1 + (1 + [\cdot])$ the part up to the prompt, $1 + (1 + [\cdot])$, together with the prompt and resumes evaluation with the remaining context $1 + (1 + [\cdot])$.

As we mentioned above continuation primitives give programmers total control on the evaluation of a program and programmers can use them to implement jumps from one point of the evaluation trace to another. Programming languages that do not have continuation primitives usually have built-in features that, in a limited manner, can switch control of the evaluation from one point in the evaluation trace to another. Such examples are loops. The following code snippet shows how we can implement with continuation primitives a loop that calls a given function an infinite number of times:

$$\begin{array}{l}
\mathsf{let}\ \mathit{now} = \lambda x.\, \mathsf{call/c}\ \lambda k.\, k\ \mathsf{in} \\
\mathsf{let}\ \mathit{goto} = \lambda \mathit{when}.\, \mathit{when}\ \mathsf{in} \\
\mathsf{let}\ \mathit{beginning} = (\lambda f.\, \%\ ((\lambda \mathit{beginning}.\, f\ 0; \mathit{goto}\ \mathit{beginning})\ (\mathit{now}\ 0)))\ \lambda x.\, 42\ \mathsf{in} \\
\mathit{goto}\ \mathit{beginning}
\end{array}$$

The first line of the code defines a function now that when called returns the context of the call up to the closest prompt. We can use this function to obtain the context at any given point in a program evaluation. The second line defines a function goto that consumes a continuation and applies it to itself. Thus this function switches control to its argument continuation and evaluates the body of that argument. Observe now the third line of the program. The call $\mathit{now}\ 0$ captures the continuation at the beginning of the evaluation of the third line of the program up to the prompt. Thus the third line returns the context $(\lambda \mathit{beginning}.\, f\ 0; \mathit{goto}\ \mathit{beginning})\ [\cdot]$ reified as a continuation. After we obtain the continuation, in the main body of the program, we call $\mathit{goto}\ \mathit{beginning}$ which in essence applies $\mathit{beginning}$ to $\lambda \mathit{beginning}.\, f\ 0; \mathit{goto}\ \mathit{beginning}$. Thus we perform the call to f once and then we call goto again with $\mathit{beginning}$ as its argument. As a result, our program jumps back to the point before the evaluation of the body of $(\lambda \mathit{beginning}.\, f\ 0; \mathit{goto}\ \mathit{beginning})$, evaluates the body of that function with $\mathit{beginning}$ as the argument of the function, and repeats this process again and again. Thus the program is an infinite loop that evaluates $f\ 0$ at each iteration.

Before, we conclude this section, we discuss the names of prompt and call/c. The first, prompt, is inspired by the prompt of interactive interpreters, i.e., the cue to the user to write the expression that the interpreter evaluates next. This expression is an arbitrary piece of code in the interpreted language that can raise errors or try to jump somewhere in the code of the interpreter itself. Thus it can destroy the interpreter. However, when we interact with interpreters this never happens no matter how hard we try. This is because the prompt of the interpreter operates as a delimiter that limits how the user's expression affects the interpreter. Thus it is very similar to the way we use continuation prompts. The second, call/c, is short for *call with continuation*. The name is reminiscent of the call/cc operator of Scheme (also know as *call with current continuation*). However call/cc's semantics is different than that of our call/c. Scheme's call/cc is not delimited, i.e., it captures the whole context instead of the context up to the closest prompt. Racket comes with a version of call/cc that is delimited by prompts. However, when we call a continuation captured with call/cc, the runtime of Racket aborts the context of the call to the closest prompt or to the nearest continuation frame (if any) shared by the current and the captured continuation. (Scheme's call/cc discards the whole context of the call). Notice that this is not the case for our call/c. We discard the context up to and including the prompt not when we call the continuation but when we capture it. This deviation from the standard semantics of call/cc affects call/c's expressiveness (in fact call/c is less expressive than call/cc) but leads to a simpler presentation of the material in these lecture notes.

3 Building exception handlers with control operators

In this section, we bring our discussion of delimited continuations full-circle. We examine how we can use the primitives of the previous section to approximate the error handlers we discuss in the first section of the lecture notes. To do so, we use a definitional translation to compile try e_1 catch $x.e_2$, try e_1 resume $f,x.e_2$ and raise e to expressions in the language of the previous section.

We begin with try e_1 catch $x.e_2$ and raise e:

$$\begin{aligned}\mathcal{T}[\![\text{try } e_1 \text{ catch } x.e_2]\!] &= \lambda x.\, \mathcal{T}[\![e_2]\!]\ (\%\ \mathcal{T}[\![e_1]\!])\\ \mathcal{T}[\![\text{raise } e]\!] &= \text{abort } \mathcal{T}[\![e]\!]\end{aligned}$$

Our starting point is that abort is quite similar to raise. So we translate raise e to abort $\mathcal{T}[\![e]\!]$. To allow try e_1 catch $x.e_2$ to obtain the result of $\mathcal{T}[\![e]\!]$ from the call to abort and pass it to e_2, in the translation of try e_1 catch $x.e_2$, we install a prompt around $\mathcal{T}[\![e_1]\!]$ that stops the propagation of the result of $\mathcal{T}[\![e]\!]$ upwards in the current context. Now that we confined this value, we can apply it to the function $\lambda x.\, \mathcal{T}[\![e_2]\!]$.

This translation works well if e_1 uses raise. What happens when this is not the case, i.e., e_1 evaluates to a value? Then again we apply v to the exception handling code $\lambda x.\, \mathcal{T}[\![e_2]\!]$. So our translation is not correct...

To fix the problem, we need some way to skip the evaluation of the exception handling code when there is no call to raise. We achieve this by (i) installing a prompt around the application of the result of $\mathcal{T}[\![e_1]\!]$ to $\lambda x.\, \mathcal{T}[\![e_2]\!]$ and (ii) turning the result of $\mathcal{T}[\![e_1]\!]$ into a function that when called aborts this prompt:

$$\begin{aligned}\mathcal{T}[\![\text{try } e_1 \text{ catch } x.e_2]\!] &= \%\ ((\lambda x.\, \mathcal{T}[\![e_2]\!])\ ((\%\ (\text{let } res = \mathcal{T}[\![e_1]\!] \text{ in } \lambda x.\, \text{abort } res))\ 0))\\ \mathcal{T}[\![\text{raise } e]\!] &= \text{let } res = \mathcal{T}[\![e]\!] \text{ in abort } \lambda x.\, res\end{aligned}$$

Notice that we apply first 0 to $\lambda x.$ abort res before we pass the result of the application to $\lambda x.\, \mathcal{T}[\![e_2]\!]$. Thus if $\mathcal{T}[\![e_1]\!]$ evaluates to a value without raising an exception, the application results into discarding the prompt around the call to $\lambda x.\, \mathcal{T}[\![e_2]\!]$. As a result, we avoid evaluating the error handling code.

Our translation so far does not handle the resumable exception handler try e_1 resume f,$x.e_2$. To accommodate the resumable exception handler, we have to capture the context around a call to raise in e_1. We accomplish it by (i) installing yet another prompt around e_1 in the translation of try e_1 resume f,$x.e_2$ and (ii) having the translation of raise make a call to call/c before aborting:

$$\begin{aligned}\mathcal{T}[\![\text{try } e_1 \text{ resume } f,x.e_2]\!] &= \%\ (((\%\ (\text{let } res = \%\ \mathcal{T}[\![e_1]\!] \text{ in } \lambda x.\, \text{abort } res))\ 0)\ \lambda f.\, \lambda x.\, \mathcal{T}[\![e_2]\!])\\ \mathcal{T}[\![\text{raise } e]\!] &= \text{abort } (\text{let } res = \mathcal{T}[\![e]\!] \text{ in call/c } \lambda k.\, \lambda x.\, \lambda h.\, (h\ k)\ res)\\ \mathcal{T}[\![\text{try } e_1 \text{ catch } x.e_2]\!] &= \%\ (((\%\ (\text{let } res = \%\ \mathcal{T}[\![e_1]\!] \text{ in } \lambda x.\, \text{abort } res))\ 0)\ \lambda f.\, \lambda x.\, \mathcal{T}[\![e_2]\!])\end{aligned}$$

There are a few more subtle points about our final translation. In try e_1 resume f,$x.e_2$, in case of an exception, e_2 must have access to both the continuation k we captured with call/c in the translation of raise and the result of raise's argument. For that reason, we translate the e_1 part of try e_1 resume f,$x.e_2$ into the application $(\lambda x.$ abort $res)$ 0 where res is the result of the evaluation of $\mathcal{T}[\![e_1]\!]$. If $\mathcal{T}[\![e_1]\!]$ evaluates without raising an exception, the application results into a call to abort that skips the application of $\lambda f.\, \lambda x.\, \mathcal{T}[\![e_2]\!]$ to the result of $(\lambda x.$ abort $res)$ 0. Thus we avoid running the exception handling code and simply return the result of $\mathcal{T}[\![e_1]\!]$. If $\mathcal{T}[\![e_1]\!]$ raises an exception, the result of the translation of raise aborts to the prompt of the let in the translation of try e_1 resume f,$x.e_2$ and we end up with $\%\ ((\lambda x.\, (\lambda h.\, (h\ k)\ res)\ 0)\ \lambda f.\, \lambda x.\, \mathcal{T}[\![e_2]\!])$. The latter expression reduces to $\%\ \mathcal{T}[\![e_2]\!]\{f/k\}\{x/res\}$, i.e., the desired result. The translation of try e_1 catch $x.e_2$ is the same as that of try e_1 resume f,$x.e_2$. Since in try e_1 catch $x.e_2$, f is not free in e_2 , the substitution of k for f does not affect e_2 and only res is accessible in e_2.

As a final remark, notice that our translation is correct only if (i) any uses of continuation operators introduced in a translated program are due to our translation and (ii) programs use raise exclusively inside the try part of exception handlers. If these severe restrictions do not hold, programmer' s code can easily mess with the exception handling protocol we have implemented. This is the best we can do, though, with our limited continuation operators. Also, our translation does not interoperate with errors raised by primitive operators, only with exceptions from our raise. In a production language, such as Racket, that does compile exceptions and exception handlers to code that uses continuation operators (i) a more expressive set of continuation primitives guarantees the correctness of the compilation and (ii) primitives raise exceptions using the Racket equivalent of our raise.