Types for Delimited Control Operators

Asumu Takikawa

May 2, 2012

1 Introduction

There are a great variety of control constructs in programming languages: conditionals, goto, coroutines, threads, processes, break, return, and, of course, continuations. The advantage of continuations over the others is that many of the other control constructs are expressible in terms of them. Unfortunately, the global continuations operators (e.g., `call/cc`) that are implemented in languages like Scheme and SML can be difficult to use. Delimited continuations are an alternative approach that provide better abstraction. However, designing a type system to accommodate them is non-trivial.

Before looking at delimited continuations, let's quickly review global continuations as exemplified by `call/cc`. The `call/cc` operator in Scheme and ML reifies the context of a program. That is, we take our evaluation contexts `E` and turn them into special values such as `(cont E)`. A continuation value just installs the given context in place of the current one:

$$E_0[((\texttt{cont}\ E_1)\ v)] \longrightarrow E_1[v]\ \text{[cont]}$$

Meanwhile, `call/cc` lets us capture a context as a continuation and passes it to a function that we define:

$$E[(\texttt{call/cc}\ v)] \longrightarrow (v\ (\texttt{cont}\ E))\ \text{[call/cc]}$$

However, continuations are complicated to reason about and, worse, are misleadingly similar to functions while not acting like them. For example, function composition does not make sense for continuations:

```
(call/cc (λ (k) ((compose add1 k) 0)))
```

Without looking closely, this might look like it will evaluate to 1. The real result is 0. While languages with continuations will allow the composition of `add1` and `k` as above, the resulting function is not useful. In other words, `call/cc` continuations do not act like real functions. You can see this in the reduction rule for continuation application, which is very different from function application.

2 Delimited control

To deal with the deficiencies of global continuations, we can use *delimited continuations*. As the name implies, delimited continuations come with a term—often called a prompt—that delimits the extent of the captured continuation. Surprisingly, even though delimited continuations seem to be more limited than global continuations, they are in fact *more expressive*.

Unlike global continuations, exemplified by `call/cc`, there are many competing formulations of delimited continuations. The original formulation by Felleisen et al. is sometimes called *dynamic* delimited continuations (or `prompt` and `control`) (Felleisen et al. 1987). We will first take a look at a variation due to Danvy and Filinski called `shift` and `reset` (Danvy and Filinski 1989). The semantics for these two sets of control operators are remarkably similar, but their reduction rules

differ in one subtle way. The most important resulting difference, for the purposes of this talk, is that Danvy and Filinski's *static* delimited continuations are easier to reason about in a typed setting.

Our first set of delimited control operators are `(shift k e)` and `(reset e)`. The former is a binding construct that binds its parameter `k` to a continuation function in the body `e`. In this regard, it is similar to `call/cc` except that `call/cc` uses a lambda expression to handle the binding. The latter term `(reset e)` represents a delimiter for any `shift`s that occur inside `e`.

The reduction rules for `shift` and `reset` are shown below. We will assume that our language has some standard set of evaluation contexts `E` and also contexts `F`. The `F` contexts only differ from `E` in that they do *not* contain `reset` terms.

$$E[(\texttt{reset}\ v)] \longrightarrow E[v] \qquad \text{[reset]}$$

$$E[(\texttt{reset}\ F[(\texttt{shift}\ x\ e)])] \longrightarrow E[(\texttt{reset}\ \text{subst}[\![x, (\lambda\ (x_0)\ (\texttt{reset}\ F[x_0])), e]\!])] \qquad \text{[shift]}$$

The rule for `reset` just produces whatever value is contained in the delimiter. In other words, the delimiter is just a marker for how far to capture the continuation.

When a `shift` is contained inside a `reset`, the evaluation context that is contained *within* the delimiter—here `F`—is captured and provided to the body of `shift`. This is the key difference between delimited and global continuations. The continuations captured by `call/cc` represent the *entire* rest of the program. However, in practice captured continuations do have a limit: e.g., the command prompt at the REPL. In a sense, delimited continuations are formalizing what REPLs already do in an ad-hoc way.

There is another key difference that is shown in the rules above. When a continuation is captured with `call/cc`, it is reified into a `(cont k)` that will eventually completely replace the current evaluation context. Meanwhile, delimited continuations are real functions and retain the evaluation context when they are applied. This difference is made clear in the rule for `shift`, where the captured context is copied into the body of the lambda. The captured context isn't reified, but actually integrated as an expression. This effectively means the continuation is composed into the context where the continuation is invoked.

Note that the use of the `F` context is important in the rule above. Since `F` contains no `reset` terms, the rule is specifically only capturing up to the *innermost* delimiter. In this system, it is not possible to skip delimiters when capturing continuations.

Let's look at some examples. This first example shows the reduction sequence for a simple expression using shift and reset. The shift inside just contains a constant number in the body. Since the continuation bound to `k` is not used at all, this discards the continuation and just produces `0`.

```
'(reset (+ 1 (shift k 0)))  →
'(reset 0)  →
0
```

In this next example, the continuation is just immediately invoked. When a continuation is immediately invoked, it simply acts as the identity:

```
'(reset (+ 1 (shift k (k 0))))  →
'(reset ((λ (x_0) (reset (+ 1 x_0))) 0))  →
'(reset (reset (+ 1 0)))  →
'(reset (reset 1))  →
'(reset 1)  →
1
```

In a sense, this isn't surprising because the use of `k` is just re-adding the continuation it already had up to the closest reset. We can also just return the continuation bound to `k`, which will just be a plain function value:

```
'(reset (+ 1 (shift k k)))  →
'(reset (λ (x_0) (reset (+ 1 x_0))))  →
'(λ (x_0) (reset (+ 1 x_0)))
```

The lambda term that is returned just contains a body that will execute the addition that was in the original continuation. The function can be passed around and used in other contexts without completely replacing them.

We mentioned previously that the original formulation of delimited continuations is very similar to Danvy and Filinski's. For comparison, here are the reduction rules for `prompt` and `control`:

$$E[(\texttt{prompt}\ v)] \longrightarrow E[v] \quad \text{[prompt]}$$

$$E[(\texttt{prompt}\ F[(\texttt{control}\ x\ e)])] \longrightarrow E[(\texttt{prompt}\ \text{subst}[\![x, (\lambda\ (x_0)\ F[x_0]), e]\!])] \quad \text{[control]}$$

The usage of the terms is exactly the same as `shift` and `reset`. Where they differ is in a tiny modification to the reduction rule for `shift`/`control`. The rule for `shift` installs a delimiter `reset` just around the invocation of the captured continuation whereas the rule for `control` just applies the captured continuation as is.

This difference is significant. The extra delimiter for `shift` constrains any inner `shift`s to capture only up to the current `shift`. In particular, this means that we can reason about the captured continuations entirely locally. This is advantageous for typing these delimited continuations.

Now let's look at a more interesting example to see how delimited continuations are more flexible than `call/cc`. Suppose that instead of building currying into our language, we want to be able to express that using delimited continuations. Here is a definition of a curried list append function in Racket:

```
(define (append1 lst)
  (if (null? lst)
      (shift k k)
      (cons (car lst) (append1 (cdr lst)))))
```

Note the use of `shift` in the null case. Now we can apply the function as follows: `(reset (append1 '(1 2 3)))`. Effectively, this causes the use of `shift` in the function to return a continuation that performs `(cons 1 (cons 2 (cons 3 [])))`, where `[]` is a hole awaiting a value. This continuation is return all the way to the enclosing `reset`.

The power of these delimited operators is that the return behavior is truly non-local. The pair of delimiter and control operator do not communicate except by the structure of the context. Compare this to `call/cc`, which is unable to express this program as concisely. The issue is that a single `call/cc` is only able to return up to the point of the continuation capture. In order to return to some other point in the program, *another* continuation is needed. Since this continuation must be threaded to the original `call/cc`, either extra function arguments or mutable state is necessary.

There is one issue with the code for `append1` above. We needed to wrap `reset` around the function call, which is extra noise that we would like to remove. We can employ what Danvy and Filinski call a *control abstraction* and wrap the function: `(define append (λ (v) (reset (append1 v))))`. Now the fact that `append` uses delimited continuations is completely hidden from the observer. We say that this version of `append` is *pure* because it is free of control effects.

3 Cupto and exceptions

There is one other delimited control operator that we will discuss called the `cupto` operator introduced by Gunter et al (Gunter et al. 1995). These are closer to `prompt` and `control`, but are amenable to typing because the prompts are all named. To create named prompts, there is a `new-prompt` term. The `set` term is the delimiter and `cupto` is the control operator that will bind and possibly use the continuation. Here are the reduction rules:

```
E[(new-prompt)]  ⟶                          [new-prompt]
E[(% x_p)]
                                  where x_p = (gensym)

E[(set p v)]  ⟶                             [set]
E[v]

E_0[(set (% x_p) E_1[(cupto (% x_p) x e)])]  ⟶  [cupto]
E_0[subst[[x, (λ (x_0) E_1[x_0]), e]]]
```

The `new-prompt` term just reduces to a value `(% x_p)` where `x_p` is some freshly generated name. Both the `cupto` and `set` terms take one of these prompt values as an argument. What is interesting about this operator is that it is motivated as a generalization of both exceptions and continuations. You can see this in how the prompt mechanism works.

Instead of having prompts and shifts be paired via the dynamic context, they are now paired explicitly via the generative prompt values created by `new-prompt`. A `cupto` can jump to any delimiter that it has access to through the prompt value. This variation is similar to the `prompt`/`control` semantics—notice the lack of the extra delimiter around the `cupto` point—except that the active delimiter is dropped after reduction. Since `cupto` can jump to contexts that are arbitrary high up on the stack anyway, there is no reason to keep the active context around.

Notice also that this set of control operators completely discards the current delimiter `set` after a `cupto`. This is different from both the Felleisen et al and Danvy and Filinski semantics. For its target application of exceptions, this is not really a problem. If you need to retain the delimiter, you can always install it manually in the use of `cupto`. Also, like `prompt` and `control`, the `cupto` operator does not install an additional delimiter inside the continuation function.

By now you probably have a basic idea of how delimited control operators work. Also, it is clear that there are many different kinds of control operators defined by fine-tuning of certain parameters for the basic operators. In other words, if you want to adopt delimited control operators, you need to ask several questions like the following:

- Should the operator install an additional prompt?
- Should the active delimiter be dropped?
- Should prompts be tagged or determined dynamically?

The answers to these questions will determine the kind of delimited control operator to use. Most of them can macro-express the others, so the decision becomes more important either in a typed context or if you are looking for a pragmatic choice that matches your application.

We have mentioned types only in passing so far. We might now ask: how do we add types to these operators? That really depends on what operators you are considering. Let us start with the `cupto` system because it has a very straightforward type system.

The reason that the `cupto` type system is straightforward is because named prompts provide a hook that we can hang types onto. We will add a prompt type that simply carries around another type. Thus, the rule for the `new-prompt` term is simple:

$$\frac{}{\Gamma \vdash (\texttt{new-prompt}) : (\textbf{prompt}\,\tau)}\ \text{T-NewP}$$

The τ occurs free in the prompt type because this type system is intended to be used in an ML-like language with type inference.

The rule for prompts is also simple and can be derived from the operational semantics. A delimiter with a value inside will just step to that value, so the type of the whole delimiter must be the

type of the value. In addition, we'll require the prompt value to have type $(\mathbf{prompt}\,\tau)$ where τ is the type of the value inside the delimiter.

$$\text{T-PROMPT}\quad \frac{\Gamma \vdash e_0 : (\mathbf{prompt}\,\tau) \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash (\mathtt{set}\,e_0\,e_1) : \tau}$$

The most interesting part of the type rules is the rule for `cupto`. The rule also showcases the primary limitation of this type system for delimited control.

$$\text{T-CUPTO}\quad \frac{\Gamma \vdash e_0 : (\mathbf{prompt}\,\tau_1) \qquad \Gamma, x : \tau_0 \to \tau_1 \vdash e_1 : \tau_1}{\Gamma \vdash (\mathtt{cupto}\,e_0\,x\,e_1) : \tau_0}$$

In this rule, the main body of the `cupto` is type-checked in an environment with a binding for the type of the continuation. The continuation takes an argument of type τ_0, which is the type of the hole in the context around the `cupto` up to the delimiter. The result of both the continuation and the body of the `cupto` need to have type τ_1 because that is the type that the entire delimited expression should have. This makes sense, but there is a limitation: the type of the value returned by the `cupto` is fixed. More specifically, a `cupto` that just returns the continuation itself will be ill-typed.

This type system is very simple, but unfortunately the limitation in the cupto rule is problematic for certain use cases of continuations. For example, recall the append example that was presented earlier. In the implementaiton of append, the `shift` simply returns the bound continuation `k` to a context that is clearly expecting a list. The type system for cupto explicitly disallows this use. In other words, the append example is not expressible in the cupto language with types.

Another way of phrasing this problem is that the cupto type system does not allow *answer type modification*. The cupto operator can only provide values that the context is known to already accept.

While these limitations may sound severe, the cupto system is still sufficient to encode many uses of control. In particular, the cupto system was designed as a generalization of exception operators. Gunter et al actually introduce an encoding of exceptions in their cupto paper, which we will describe now.

The exception language that we will consider consists of the following operators: `new-exn`, `(raise e_0 e_1)`, and `(handle e_0 e_1 e_2)`. The `new-exn` term creates a new exception value while the other two terms respectively raise an exception and handle an exception using the provided function.

Let's start with the translation of `new-exn`:

```
(new-exn)
=>
(new-prompt)
```

Exception values are just represented as prompt values. Next we will look at the `raise` term. This term takes the exception value and a value to raise to the handler.

```
(raise e1 e2)
=>
(let ([x1 e1]
      [x2 e2])
  (cupto x1 k x2))
```

A `raise` translates simply to a `cupto`. The continuation parameter `k` is bound but never used. Since all that the exception needs to do is abort to the handler (i.e., communication is one-way), there is no need to use the continuation. The `let` is there just to force the expressions to evaluate in order.

Next, the translation of `handler`, which takes an exception value, a handler function, and the code for the body:

```
(handler e1 e2 e2)
=>
(let ([x1 e1]
      [x2 e2]
      [p (new-prompt)])
  (set p ((λ (z) (cupto p k (x2 z)))
          (set x1
               (let ([x3 e3])
                 (cupto p k x3))))))
```

This translation is somewhat more complicated. The handler translates to code that first creates its own local prompt and sets up a delimiter for it. Within the delimiter, a lambda expression is immediately applied to an expression with another delimiter. This delimiter is named by the exception value, so that if `x3` contains a `raise` to that exception, it will be caught by the delimiter. After it is caught by the delimiter, the value will flow into the lambda expression and be given as an argument to `x2`, which is the evaluated handler function. If there is no `raise`, the `cupto` in the let form will jump to the local delimiter and that will be the end.

This shows that the `cupto` system is expressive enough to support exceptions despite its type system limitations. In practice, there are probably many cases where answer type modification is not necessary to support useful and interesting control idioms with delimited continuations.

4 Types for shift and reset

Since Gunter et al's system cannot deal with answer type modification, one might ask if that is possible in general. Danvy and Filinski's original type system for shift and reset actually does accommodate answer type modification. The main tradeoff for this expressiveness is that the type system is quite complex.

To accommodate answer type modification, Danvy and Filinski make the type-checking relation a five-place relation. The relation will now track the *original* result type and the *new* result type. The result type is the type of the value that will be provided to the nearest delimiter by a jump.

To start, let's look at the rules for constants. Since constants have no control effects, they will never affect the result type at all (i.e., they are *pure*). In other words, they are going to be polymorphic in the result type:

$$\textsc{T-Int}\quad \Gamma;\alpha \vdash n : \texttt{int};\alpha$$

In this rule, the α is both the old and new result type. The computation of a constant integer will not matter for the result type at all.

Similarly, a `reset` is also pure. This makes sense given our earlier discussion of control abstractions using `reset`. The type rule is shown below:

$$\textsc{T-Reset}\quad \frac{\Gamma;\sigma \vdash e : \sigma;\tau}{\Gamma;\alpha \vdash (\texttt{reset}\, e) : \tau;\alpha}$$

The expression e inside of the delimiter can have any control effect because the delimiter will shield the context from it. Hence, the whole delimiter is pure and just has the same α for old and new result types. However, the result types for the inner expression look strange. The old result type

and the type of the expression are both σ. This makes sense because if there is a control effect, it will potentially change the type of the value in the `reset` from σ to some new type.

In particular, consider what would happen if the expression inside is a constant.

$$\frac{}{\Gamma; \texttt{int} \vdash n : \texttt{int}; \texttt{int}}$$
$$\Gamma; \alpha \vdash (\texttt{reset}\, e) : \texttt{int}; \alpha$$

Unification will cause the α and σ to be the same type due to both the T-INT and T-RESET rules. In turn, both of these are instantiated with `int` because of the constraints in T-RESET. This makes everything work out as the type of the whole delimiter is now `int` as desired.

Next, we will look at the rule for `shift`:

T-SHIFT
$$\frac{\Gamma, f : \tau/\delta \to \alpha/\delta; \sigma \vdash e : \sigma; \beta}{\Gamma; \alpha \vdash (\texttt{shift}\, x\, e) : \tau; \beta}$$

The rule binds the type for the continuation, which has a function type. Notably, this rule shows that we need to change the types of functions because they may have control effects themselves. The extra types in the function types are the old and new result types for functions.

Type-checking the body of the `shift` again just assumes that the type of the old answer (σ) is the type of the body expression itself. If anything in the body changes that type, then β will reflect that. Now notice that the type of the whole `shift` is *not* σ because the `shift` will never return to its immediate context unless the continuation is actually called (otherwise it will abort to the delimiter). Instead, the type of the whole `shift` (i.e., the type that its context expects) is going to be the input type of the continuation because the continuation's argument will be plugged into this context. The result type of the continuation has to be the old result type of `shift` (α) because the context would also produce that type if the `shift` were not there.

Finally, if the `shift` just aborts to the delimiter, we can return whatever type is consistent with the new result type of the body expression. If the body is a constant, this will just be the type of the constant.

Note that the continuation itself in `shift` is polymorphic in the answer type because of the extra `reset` that is always wrapped around the context in a continuation function.

Application and lambdas are fairly straightforward except for the need to figure out the ordering of the result types in the application. Here are the rules:

T-LAM
$$\frac{\Gamma, x : \sigma; \alpha \vdash e : \tau, \beta}{\Gamma; \delta \vdash \lambda x.e : \sigma/\alpha \to \tau/\beta, \delta}$$

T-APP
$$\frac{\Gamma; \delta \vdash e_0 : \sigma/\alpha \to \tau/\epsilon, \beta \qquad \Gamma; \epsilon \vdash e_1 : \sigma; \delta}{\Gamma; \alpha \vdash e_0\, e_1 : \tau, \beta}$$

The lambda rule just checks what the effects the body of a lambda would have and appropriately instantiates the function type with the corresponding result types. The lambda expression itself is polymorphic in its result type because lambdas are values and thus pure.

The application rule is slightly trickier. We have seen that all of the pure expressions will have the same old and new result types. Thus, in order to determine the effect of the entire application, we need to determine if the control effects of the function or argument position will be the effect of the entire expression. Note that the final effect cannot be *both* effects because an effect will first abort and change the context so that the other effect cannot be run at all.

The ordering in the rule shows that the function position is evaluated first. If the function has an effect, then its result type β is used since β is the effect of the whole expression. On the other hand, if the function position is pure, then it will simply propagate the old result type δ. This old result type is the new result type of the argument expression. Thus, if the function does not cause an effect, then the argument will get to set the new result type. Otherwise, there is no way that the argument could cause an effect because the context has changed.

Similarly, because the function in the argument itself is invoked *after* both the function and argument are evaluted to values, its final result type ϵ is propagated to the argument's old result type. That is, if the argument had no effect, the new result type of the whole expression will be the function's latent effect.

Answer type modification is clearly necessary even for relatively simple examples, but it greatly increases the complexity of the type system as we have just seen.

5 Continuations in the real world

So far, we have talked about a variety of delimited control operators and their use in small examples. While delimited control operators are actually implemented for some languages (e.g., SML, Scheme, Racket, etc.), they have not seen mainstream adoption in the form that we have seen so far. This might raise the question: are delimited continuations useful in practice?

It turns out that mainstream languages such as C#, JavaScript, Python, and Ruby all use a form of delimited control called `yield`. Unlike the operators we have seen so far, the implmentations of control in these languages are more limited and hidden so that most programmers need not be aware of how they are delimited. However, James and Sabry have shown that the yield operator can be modeled using delimited control operators and that a sufficiently powerful generalization of yield can also model delimited control (James and Sabry 2011).

Here is one example from their paper that demonstrates how Ruby uses a form of delimited control:

```
def inject(state)
  self.each { |v| state = yield(state, v) }
  state
end

def useInject()
  total = [1..1000].inject(0) { |sum, v|
    if prime?(v)
       sum + v
    else
       sum
    end
  }
end
```

This example contains two methods. The `inject` method uses the `yield` operator to suspend computation and return two values to the context. The other loops over the integers from 1 to 1000 and sums the prime numbers using `inject`. At the end of a loop iteration, the `sum` that is returned is implicitly given to the suspended computation of `inject` (i.e., the continuation) and sets the next state within the body of `inject`.

Viewing this as a use of delimited continuations, you can consider the `yield` as something akin to a `shift` or `control`. The delimiter is implicitly integrated into the looping construct. This is a common pattern in imperative languages, where `yield` is often used to create iterators or looping. In other words, these languages bake a special case of delimited control into certain aspects of the language.

6 Conclusion

Delimited control operators are powerful and can subsume uses of global continuations. When truly global behavior is desired, a prompt can be applied at the top-level, which simulates the behavior of `call/cc`. Certain patterns of delimited control can even be found in mainstread languages, though usually not with the full expressiveness of the traditional operators such as `shift` and `reset`. Delimited control is a useful addition to a language's toolbox, but care must be made to ensure that the design fits the use cases and that a type system for it balances usefulness and complexity.

Bibliography

Olivier Danvy and Andrzej Filinski. A Functional Abstraction of Typed Contexts. University of Copenhagen, , 1989.

Matthias Felleisen, Daniel Friedman, Bruce Duba, and John Merrill. Beyond Continuations. Indiana University, 216, 1987.

Carl A Gunter, Didier Remy, and Jon G Riecke. A Generalization of Exceptions and Control in ML-like Languages. In *Proc. International Conference on Functional Programming Languages and Computer Architecture*, 1995.

Roshan P James and Amr Sabry. Yield : Mainstream Delimited Continuations. In *Proc. Theory and Practice of Delimited Continuations*, 2011.