Continuations and Transducer Composition

Olin Shivers
Georgia Institute of Technology
shivers@cc.gatech.edu

Matthew Might
Georgia Institute of Technology
mattm@cc.gatech.edu

Abstract

On-line transducers are an important class of computational agent; we construct and compose together many software systems using them, such as stream processors, layered network protocols, DSP networks and graphics pipelines. We show an interesting use of continuations, that, when taken in a CPS setting, exposes the control flow of these systems. This enables a CPS-based compiler to optimise systems composed of these transducers, using only standard, known analyses and optimisations. Critically, the analysis permits optimisation across the composition of these transducers, allowing efficient construction of systems in a hierarchical way.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Optimization; D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures, coroutines

General Terms Languages, performance, design

Keywords program analysis, flow analysis, language design, coroutines, fusion, stream processing, functional languages, lambda calculus, continuation-passing style (CPS), continuations

1. Optimising across composition

Abstraction is fundamental to any engineering discipline as a tool for managing complexity during synthesis; this is as true for bridge builders as it is for those who construct software. One of the most important tasks software researchers can do to support engineers is to help support use of abstraction.

When a software engineer composes two smaller computational entities to create a larger one, it is important that he know that his system-construction tools, such as his compiler and linker, will be able to optimise *across* this composition, so that he does not pay a performance penalty for composing his system together in a hierarchical, structured way.

As a simple example of this effect, consider procedural abstraction and composition. If we wish to make a new function h by hooking the output of function g up to the input of function f, we can do this very easily with the "little circle" function—that is, we simply write $h = f \circ g$. The important point is that if an engineer does this, he has reason to believe that the compiler, if it can statically determine the definitions of f and g, will be able to "melt" his abstraction barriers, and optimise across his composition. So if, for example, g adds three to its argument, and f adds five, then the body of h won't consist of two nested function calls to f and g, but rather will directly add eight to its argument. The technology for this kind of optimisation is straightforward. A simple technique that works well is just code inlining (if you come from the traditional compiler community), or β-reduction (if you come from the functional-programming community).

Providing this kind of optimisation across composition is an extremely important thing to do, as it enables engineers to build systems in modular, maintainable ways. If an engineer *can't* assume his compositions will be optimised, then performance considerations may prevent him from composing his system together hierarchically. Instead, he must "hand inline" all the intermediate compositions, working at the lowest level.

PLDI'06 June 11–14, 2006, Ottawa, Ontario, Canada.

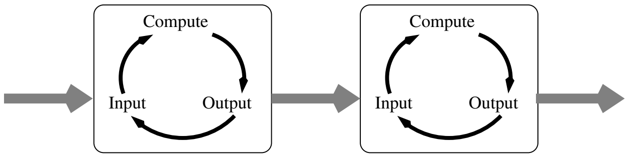

Figure 1. Online transducers can be composed into pipelines.

2. Online transducers

However, not all systems are composed together from procedures. An important class of systems are constructed by composing *online transducers*. An online transducer is a computational agent that is not written as a procedure or invoked in a call/return pattern. Rather, it is written as little chunk of code and state that performs the following infinite loop: (1) get some input; (2) compute; (3) do some output; (4) loop. An engineer can compose two transducers together, making a new one, with the Unix "pipe" operator: $g \mid f$. Figure 1 shows two transducers hooked together to produce a new, composite one. Many important systems are composed from primitive elements realised as online transducers, such as:

- Stream processors
- Network-protocol stacks
- DSP and other media-processing systems
- Unix pipelines
- graphics-rendering pipelines

Unfortunately, when the computational agents that we are composing are online transducers, compilers do not fare nearly so well. If we take a transducer that adds three to its incoming stream, and compose it with one that adds five, it is quite difficult for a compiler to optimise across the two loops, reducing the composite transducer to a simple add-eight loop.

So the bad news is that, while designers of network protocol stacks can use elegant, layered architectures to explicate their designs, implementors who care about efficiency—and network-protocol implementors always do—have to throw out the modular, layered approach, and write a system that is tightly integrated by hand, producing a complex, arduously maintained mess.

3. CPS

Fortunately, the functional-programming community has a powerful representational tool at its disposal: continuation-passing style. CPS is a low-level, λ-calculus intermediate representation in which function calls never return, and, hence, are never nested. That is, in CPS, we never write $f(g(x))$, since the $g(x)$ call will never return to the waiting f. Instead, one always passes to every function an extra argument, the *continuation*, which is itself a function representing the entire computation to be performed after the original function has computed its value. The original function applies its continuation to the value it computes. So we can view calling a continuation as a goto that passes values. For example, while in Scheme we might write

```
(- a (* b c))
```

in CPS, we write

```
(* b c (λ (temp) (- a temp k)))
```

The λ expression is `*`'s continuation; k is the continuation representing the entire context in which the expression sits. In brief, a continuation is a functional representation of an expression's context, and we may view it as "the rest of the computation."

The use of CPS as a compiler representation has a long history in the design of compilers for functional programming languages [Rabbit, Orbit, TC, CwC]. For our purposes, let's note for now a very important property of the CPS representation: *in CPS, all transfers of control are represented as function calls.* Function call, function return, loops, conditionals, sequencing, exception invocation and other non-local control transfers—these are all represented as function call. So program analyses that answer the question "which call sites call which λ-expressions" give us *all* the control-flow information about the program [0CFA, k-CFA].

4. Pipelines and 3CPS

Let's consider a world of online transducers that get arranged into pipelines of data-flow networks. We'd like to write these transducers in a general-purpose programming language, such as Scheme, as infinite loops that employ special "get" and "put" operators to synchronously pass data (and transfer control) up and down the pipeline. So a simple transducer that does nothing but repeatedly send five downstream is

```
(λ () (letrec ((lp (λ () (put 5) (lp))))
        (lp)))
```

and an only slightly less trivial transducer that doubles its input and passes the result downstream is

```
(λ () (letrec ((lp (λ () (put (* 2 (get)))
                         (lp))))
        (lp)))
```

Transducers may have state, which we represent in the standard manner employed in functional programming languages, as parameters in our tail-recursive loops. A simple integrator or summing transducer is written

```
(λ () (letrec ((lp (λ (sum)
                     (let ((sum (+ sum (get))))
                       (put sum)
                       (lp sum)))))
        (lp 0)))
```

Now, a CPS-based compiler, when fed code like these examples, will first convert this "direct-style" code into a CPS representation. However, let's use a non-standard representation, which we'll call "3CPS." Instead of passing to a procedure *one* continuation representing the "rest of the computation," we'll pass in *three*:

- k will represent the rest of this pipeline stage's computation;
- u will represent all the pending computation *upstream* from this pipeline stage; and
- d will represent all the pending computation *downstream* from this pipeline stage.

We augment the denotational semantics of our CPS representation by adding new "UpCont" and "DownCont" domains. Here are the functionalities:

$$\begin{array}{lll} c \in \text{CmdCont} & = & \text{UpCont} \to \text{DownCont} \to \text{Ans} \\ k \in \text{ExpCont} & = & \text{Value} \to \text{UpCont} \to \text{DownCont} \to \text{Ans} \\ u \in \text{UpCont} & = & \text{DownCont} \to \text{Ans} \\ d \in \text{DownCont} & = & \text{Value} \to \text{UpCont} \to \text{Ans} \\ x \in \text{Value} & & \end{array}$$

A "command continuation" c simply represents the state of a single, suspended stage in a pipeline. In order to invoke it, we must pass it its "connections," that is, its upstream and downstream continuations, by calling $c\,u\,d$. Once passed these values, it executes the pipeline stage, running the program forwards to completion, which is signified by the "→ Ans" function range.

An "expression continuation" k represents the evaluation context for some expression. It is invoked by calling $k\,x\,u\,d$, passing to k the value x produced by the expression, together with the current pipeline stage's upstream and downstream continuations, u and d. Then k runs the current pipeline stage forwards to completion.

A "downstream continuation" d represents the computation of the downstream pipeline stages. To transfer control downstream, we call $d\,x\,u$, passing to d the data value x we wish to send downstream and an upstream continuation u. This u value represents the suspended state of the entire computation that is upstream from d—that is, it represents our current pipeline stage, plus everything that is upstream from us. Since, from d's point of view, we are its upstream computation, we must wrap ourselves up as an UpCont in order to pass our state off to d.

Now, most primitive computations don't shift up or down the pipeline at all, so most of the time, the u and d values are simply passed around unused and unaltered. In fact, the CPS-conversion equations for our triple-continuation target language are exactly the same as the classic single-continuation conversion! This is because we placed the up and down continuations at the end of the functionalities, where they get η-converted away, being unreferenced.

If abstractions, applications, variable references, and constant evaluations don't touch the extra continuations, then who does? It's the special `get` and `put` functions, which cause shifts up and down the pipeline. These are the primitives that actually use the extra continuations, and, as such, we can't define them in direct style, where the continuations are "hidden," but must define them directly in our CPS language. The semantics of these two functions are:

$$\begin{array}{lll} \mathit{get}\,x\,k\,u\,d & = & u\,(\lambda\, x\, u' \,.\, k\; x\; u'\; d) \\ \mathit{put}\,x\,k\,u\,d & = & d\; x\,(\lambda\, d' \,.\, k\; \mathit{unit}\; u\; d') \end{array}$$

The `get` function takes an (ignored) argument x, and the standard three continuations, k, u, and d. We must transfer control to the upstream continuation and run there so that it may pass us a value, so `get` calls u. Now, the functionality of a u is DownCont → Ans, so

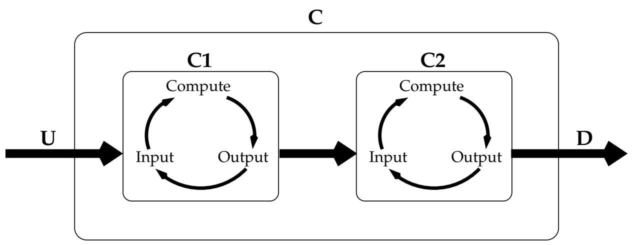

Figure 2. The `compose/pull` operator initially transfers control to C2, "pulling" values downstream on demand.

we must pass u a DownCont. DownCont's, in turn, have functionality Value $\rightarrow$ UpCont $\rightarrow$ Ans, so the value we pass u is written $(\lambda\, x\, u' \;.\; \ldots)$. Remember that this down continuation represents the state of our own pipeline stage, plus everything downstream from us—that is, everything downstream from u. When u has a value ready to send us, it will call this down continuation, passing it the value x, and a new up continuation u' for us to use the next time we wish to perform a `get`. When this happens, our current `get` is complete—it should return value x. So we go back to running our current pipeline stage by calling k, passing it x, the new up continuation u', and the same down continuation we had before the `get` call, d. Thus, the down continuation we pass to u is $(\lambda\, x\, u' \,.\, k\; x\; u'\; d)$.

We can walk through a definition of `put` in the same way, guided by the functionalities or types of the different values we manipulate. To `put` a value x, we must pass it to the downstream pipeline stage, which we do by applying d to x. However, when we transfer control downstream, we must also pass d an upstream continuation that packages up all the computation upstream from d—that is, the state of our own pipeline stage, and everything upstream from us. An UpCont has functionality DownCont $\rightarrow$ Ans, so it is written $(\lambda\, d' \;.\; \ldots)$. When d wishes to transfer control back to us, it will call this up continuation, passing it a new down continuation d' for us to use the next time we wish to perform a `put`. That is, d' will package up the state of the downstream continuation when it suspends itself and transfers control back up to us. At this point, our current `put` is complete, and we are ready to continue running our current pipeline stage. So we should transfer control to our current pipeline stage's continuation k, passing it the unit value produced by a `put`, our original, unchanged up continuation u, and the new down continuation d'. Hence, the up continuation we pass to d is $(\lambda\, d' \,.\, k\; \mathit{unit}\; u\; d')$.

5. Composing transducers in 3CPS

Now that we can write down transducers in our 3CPS representation, let's consider how we can compose them together. Let's define a new primitive function, `(compose/pull` c_1 c_2`)`, that takes two transducers, represented as command continuations, and composes them together to produce a new transducer c, with the output of c_1 hooked up to the input of c_2. More precisely, when we invoke c, passing it its connections u and d, we wish to transfer control to the downstream half of the composition, c_2, running it connected downstream to d. If it transfers control upstream, we wish to jump into c_1, and run it connected upstream to u, and downstream back to c_2 (see Figure 2). Thus this operator creates a "lazy" composed transducer, that "pulls" values downstream as needed.

Now, let's write down the semantics of `compose/pull`—which, as should be obvious from our definitions of `get` and `put`, should also serve to give us a direct definition of the operator in our 3CPS representation:

$$\mathit{compose/pull}\; c_1\; c_2 = \lambda\, u\, d \,.\, c_2\, (\lambda\, d' \,.\, c_1\, u\, d')\, d$$

The result of composing c_1 and c_2 is a command continuation, which must be invoked by passing to it the upstream and downstream connections u and d. So the composed continuation is written $(\lambda\, u\, d \;.\; \ldots)$. When this transducer runs, it simply transfers control to c_2. When we call c_2, we must pass it its connections. The downstream connection for c_2 is simply d, the downstream continuation for the whole transducer. But what should we pass to c_2 as its upstream continuation? An upstream continuation has functionality DownCont $\rightarrow$ Ans, so it should be something of the form $(\lambda\, d' \;.\; \ldots)$. When c_2 performs a `get` call, and attempts to transfer control upstream, it will invoke this continuation, passing it d', a down continuation that will represent c_2's suspended state. At this point, we should start executing transducer c_1. When c_1 wishes to go upstream, it should call u; when it wishes to go back downstream to c_1, it should call d'.

Here are the three equations for `put`, `get`, and `compose/pull`, all grouped together:

$$\begin{aligned} \mathit{get}\, x\, k\, u\, d &= u\, (\lambda\, x\, u' \,.\, k\; x\; u'\; d) \\ \mathit{put}\, x\, k\, u\, d &= d\; x\; (\lambda\, d' \,.\, k\; \mathit{unit}\; u\; d') \\ \mathit{compose/pull}\; c_1\; c_2 &= \lambda\, u\, d \,.\, c_2\, (\lambda\, d' \,.\, c_1\, u\, d')\, d \end{aligned}$$

With three small equations we have completely specified our transducer primitives and the composition operator. Note what using CPS has done for us: it has made all the "plumbing," that is, all the interconnection structure, explicit, in terms of the d's and u's. Even better, it has done so *in terms of the λ-calculus*, a representational framework for which we have decades of experience designing analyses, transforms, and optimisations.

Before we proceed, note that we could also define a related operator, `compose/push`, that initially transfered control to c_1, eagerly "pushing" values downstream. The definition of `compose/push` is interesting; it has been moved to an endnote, so that you may have the fun of working out its definition for yourself {Note `compose/push`}.

6. Working through a simple example

Let's work through a simple example by hand. Consider the two transducers

```
(λ () (letrec ((lp1 (λ () (put 5) (lp1))))
        (lp1)))

(λ () (letrec ((lp2 (λ () (put (* 2 (get)))
                          (lp2))))
        (lp2)))
```

We would like it to be the case that if we compose them together with `compose/pull`, compiler analysis could reduce the pair of loops to a single "put ten" loop.

First, we CPS-convert our two loops, and inline the `put` and `get` operators, giving us

```
(λ (k1 u1 d1)                 ; Put-5 transducer
  (letrec ((lp1 (λ (k1a u1a d1a)
                 (d1a 5 (λ (d1b)
                          (lp1 k1a u1a
                               d1b))))))
    (lp1 k1 u1 d1)))
```

```
(λ (k2 u2 d2)                 ; Doubler
  (letrec ((lp2 (λ (k2a u2a d2a)
                  (u2a (λ (x u2b)
                         (d2a (* 2 x)
                              (λ (d2b)
                                (lp2 k2a
                                     u2b
                                     d2b))))))))
    (lp2 k2 u2 d2)))
```

We've cheated on CPS-converting the `(* 2 x)` operation for compactness' sake. Everywhere we saw `put` appear in our transducer loops, we just dropped in its equivalent CPS combinator

```
(λ (x k u d) (d x (λ (d') (k u d'))))
```

This is precisely what we wrote for `put`'s semantics, transliterated into our Lisp-syntax CPS language, decurried, and with the convention that unit arguments are eliminated from calls and parameter lists, so that `put`'s continuation `k` is not explicitly passed the trivial unit value that `put` returns. We likewise replaced occurrences of `get` with its defining combinator.

We can simplify this code a bit. First, notice that `k1`, `k2`, `u1`, and `u1a` are never called. It's not surprising that `k1` and `k2` are never called, since both of our loops are infinite ones that never return. Similarly, the put-five loop never goes upstream, so `u1` and `u1a` never get used. There is a standard analysis for spotting these situations, "useless-variable analysis" [UVE], and it will remove these variables from the calls and parameter lists in the code, leaving us with

```
(λ (k1 u1 d1)                 ; Put-5 transducer
  (letrec ((lp1 (λ (d1a)
                  (d1a 5
                       (λ (d1b) (lp1 d1b))))))
    (lp1 d1)))

(λ (k1 u2 d2)                 ; Doubler
  (letrec ((lp2 (λ (u2a d2a)
                  (u2a (λ (x u2b)
                         (d2a (* 2 x)
                              (λ (d2b)
                                (lp2 u2b
                                     d2b))))))))
    (lp2 u2 d2)))
```

We can "promote" `lp1` from a user-procedure to an UpCont by simply η-reducing `(λ (d1b) (lp1 d1b))` to `lp1`. This leaves us with

```
(λ (k1 u1 d1)                 ; Put-5 transducer
  (letrec ((lp1 (λ (d1a) (d1a 5 lp1))))
    (lp1 d1)))

(λ (k1 u2 d2)                 ; Doubler
  (letrec ((lp2 (λ (u2a d2a)
                  (u2a (λ (x u2b)
                         (d2a (* 2 x)
                              (λ (d2b)
                                (lp2 u2b
                                     d2b))))))))
    (lp2 u2 d2)))
```

Now let's compose these two transducers together. This is easily done: we take the code for `compose/pull`

```
(λ (c1 c2)
  (λ (k u d) (c2 k
                 (λ (d') (c1 k u d'))
                 d)))
```

and apply it to our two transducer loops—that is, we drop the code for the put-five loop where `c1` appears, and the code for the doubler loop where `c2` appears. The resulting expression is

```
(λ (k u d)
  (letrec ((lp2 (λ (u2a d2a)
                  (u2a (λ (x u2b)
                         (d2a (* 2 x)
                              (λ (d2b)
                                (lp2 u2b
                                     d2b))))))))
    (lp2 (λ (d')
           (letrec ((lp1 (λ (d1a) (d1a 5 lp1))))
             (lp1 d')))
         d)))
```

Compilers usually hoist letrec's out; let's do that to the `lp1` binding, giving us

```
(λ (k u d)
  (letrec ((lp2 (λ (u2a d2a)
                  (u2a (λ (x u2b)
                         (d2a (* 2 x)
                              (λ (d2b)
                                (lp2 u2b
                                     d2b)))))))
           (lp1 (λ (d1a) (d1a 5 lp1))))
    (lp2 (λ (d') (lp1 d')) d)))
```

Now we can η-reduce `(λ (d') (lp1 d'))` to `lp1`, reducing the code for our original pair of loops to just

```
(λ (k u d)
  (letrec ((lp2 (λ (u2a d2a)
                  (u2a (λ (x u2b)
                         (d2a (* 2 x)
                              (λ (d2b)
                                (lp2 u2b
                                     d2b)))))))
           (lp1 (λ (d1a) (d1a 5 lp1))))
    (lp2 lp1 d)))
```

This is already pretty good, but CPS-based control- and data-flow analysis can take us the rest of the way to collapsing our double loop to a single loop. Tracing through the flow of control reveals that `u2a` and `u2b` are always bound to `lp1`; the automatic analysis that reveals this is "super-β" [k-CFA, DCFA]. We can eliminate these two variables, and replace the call to `u2a` with `lp1`.

```
(λ (k u d)
  (letrec ((lp2 (λ (d2a)
                  (lp1 (λ (x)
                         (d2a (* 2 x)
                              (λ (d2b)
                                (lp2 d2b)))))))
           (lp1 (λ (d1a) (d1a 5))))
    (lp2 d)))
```

Now we can η-reduce `(λ (d2b) (lp2 d2b))` to `lp2`, and in-line the call to `lp1`, which triggers off two more rounds of β-reduction, settling out with

```
(λ (k u d)
  (letrec ((lp2 (λ (d2a) (d2a (* 2 5) lp2))))
    (lp2 d)))
```

This is the put-ten loop we wanted. Note that every step was a standard transformation, either an application of the λ-calculus's simple η and β reductions, or a more sophisticated transform based on known data-flow analyses.

The moral is that while many of these CPS-based control- and data-flow analyses were originally designed to operate on the kind of control-flow we see in traditional intra-procedural settings—loops and conditionals—in CPS, *all transfers of control* are represented with the same control mechanism: tail-recursive procedure call. So an analysis that works in a CPS framework captures everything, including the more exotic control transfers being used in our transducer constructs.

7. Some problems

As pretty as this CPS-based explication of transducer structure may be, it is not without limitations. The biggest issue is that the linear topology of the data-flow pipeline is baked into our triple-continuation CPS transform, which is a fixed part of the compiler. If we'd like to have, say, a branching topology, we are out of luck. For example, while the Sutherland-Hodgman polygon-clipping algorithm has a data flow that perfectly fits the linear pipeline model we've developed, the slight variant of this algorithm that *splits* a polygon by a plane into two resulting polygons has a natural tree structure to its data flow. (We'll return to this example, later.)

Note that when we consider non-linear communication topologies, we are still restricting ourselves to a serial computational model, with a single locus of control that moves between transducers when values are sent or requested. We are not tackling true concurrency, or attempting to provide CSP-like features with our `get` and `put` operators.

In a related issue, the CPS transform is a global transform that rewrites the entire program. So we can't have a modular way of defining some kind of communications resource or topology that is restricted to some limited subset of the code.

From one perspective, what we've done is taken loop state—control and otherwise—and packaged it up inside λ expressions, where λ-calculus reasoning engines can trace out where different pieces of state go. Pushing the transducer state down into the u and d continuations puts it below the level at which the user code can access it: these continuations are only manipulated by the `get` and `put` primitives, who do so in a linear fashion. This also helps out our analysers, since the critical bits of state are separated out, instead of dropped into the formless sea of the runtime store, where it is very difficult to keep items distinct.

It turns out that there *is* a technology that does provide this kind of mechanism in a modular fashion: monads [Moggi, Essence]. Monads have exactly the same property as CPS of "pushing" certain structures below the surface of most of the code, only allowing certain related primitives to "dip down" and access them in restricted ways (and this is not a surprise, since continuation-passing and monads are related). Hudak and others [MADT] have shown how to use monads to provide monadic, linear data types—which certainly describes the u's and d's we've been passing around. For all these reasons, monads are popular in the pure-functional, lazy-language community, receiving quite a bit of use in Haskell to model things like I/O. Unfortunately, monads have problems for general use in less-exotic, applicative-order languages such as SML or Scheme. The principal issue is that they do not, in general, compose; as a result, we have been unable to figure out a way to use them as a general tool for providing transducer mechanisms. Even in the Haskell community, monads require programmers to write in a particular style which would prove onerous to programmers who work in applicative-order languages.

A final issue to consider is: can we type our transducers and their primitives? If so, can we do it in standard type systems, *e.g.*, SML's?

Our task in the next section is to address all of these issues.

8. Removing constraints, adding types

We can remove the linear-pipeline constraint on the connection topology by the simple expedient of bringing the hidden u and d cross-connecting resources out into the open, where they can be directly manipulated by the programmer. This is a little more verbose than the neat, clean manner in which these continuations were invisibly passed around behind the scenes, but it allows us to name and use multiple communications resources.

We can also take this opportunity to start introducing types into our model. Let's introduce a new type constructor, using an ML syntax:

$$(\alpha, \beta)\,\texttt{Channel}$$

A value c of type $(\alpha, \beta)\,\texttt{Channel}$ is a coroutine communications resource. We can send an α value over it, and receive a β value back. The primitive operator for doing so is `switch`:

$$\texttt{switch} : \alpha \times (\alpha, \beta)\,\texttt{Channel} \rightarrow \beta \times (\alpha, \beta)\,\texttt{Channel}$$

Note that `switch` returns not only the β value returned from the coroutine at the other end of the channel, but also a new channel for us to use the next time we wish to communicate with that coroutine. Channel values are "affine," that is, they may not be used more than once.

Now we can write some simple examples in an ML syntax that parallel our previous, linear pipeline examples. We can define `put` and `get` in terms of `switch`:

```
type γ DownChannel = (γ, unit) Channel
type γ UpChannel = (unit, γ) Channel

fun put(x,c) = #2(switch(x,c))
fun get c = switch((),c)
```

Our put-five example is just the simple loop

```
fun put_five c = put_five(put(5,c))
```

and our integrator is

```
fun integ(u,d) =
  let fun lp(s,u,d) =
    let val (x,u') = get u
        val s = s + x
    in lp(s, u', put(s,d))
    end
  in lp(0,u,d)
```

However, we could instead define a general-purpose coroutine generator that folds an $\alpha \times \beta \rightarrow \beta$ function across an α stream:

```
fun streamfold f zero (u,d) =
  let fun lp(s,u,d) =
    let val (x,u') = get u
        val s = f(x,s)
    in lp(s, u', put(s,d))
    end
  in lp(zero,u,d)
  end
```

We can then define our integrator as:

```
fun integ(u,d) = streamfold op+ 0 (u,d)
```

The surprising thing about our channels is that, if we have a `callcc` primitive to give us access to continuations, we can define and type the `switch` primitive with no further extensions to SML's type system or primitives! The key observation is that if we have an $(\alpha, \beta)\,\texttt{Channel}$, then some other coroutine must be suspended in a `switch` call; when we switch off to this coroutine by sending some α value over our channel, this suspended coroutine will resume.

```
signature FUNCTIONAL_COROUTINES = sig

infix 4 `>| >|`  (* Can now write pipelines    *)
infix 4 `|> |>`  (* src |>` t1 >>` t3 >|` sink *)
infix 5 `>> >>`

type α cont  (* Continuations. *)

datatype (α,β) Channel =
  Chan of (α * (β,α) Channel) cont

type α DownChannel = (α,unit) Channel
type α UpChannel   = (unit,α) Channel

val switch : α * (α,β) Channel ->
             β * (α,β) Channel

val get : α UpChannel -> (α * α UpChannel)
val put : α * α DownChannel -> α DownChannel

(* Transducers, sources and sinks. *)
type (α,β,γ) Transducer =
  (α UpChannel * β DownChannel) -> γ

type (α,γ) Source = α DownChannel -> γ
type (α,γ) Sink   = α UpChannel -> γ

(* Infix transducer+transducer, push & pull *)
val `>> : (α,β,γ) Transducer * (β,δ,γ) Transducer
            -> (α,δ,γ) Transducer
val >>` : (α,β,γ) Transducer * (β,δ,γ) Transducer
            -> (α,δ,γ) Transducer

(* Infix source+sink, push & pull. *)
val `>| : (α,γ) Source * (α,γ) Sink -> γ
val >|` : (α,γ) Source * (α,γ) Sink -> γ

(* Infix source+transducer, push & pull. *)
val `|> : (α,γ) Source *
         (α,β,γ) Transducer -> (β,γ) Source
val |>` : (α,γ) Source *
         (α,β,γ) Transducer -> (β,γ) Source

(* Generic source creation. *)
val listsource  : α list -> (α option,γ) Source
val put_n_xs : int -> α -> (α option,γ) Source

(* Generic sinks. *)
val firstsink : (α option, α) Sink
val listsink  : (α option, α list) Sink

(* Generic transducer creation. *)
val stream_fold : (α * β -> β) -> β ->
                  (α,β,γ) Transducer
val stream_app : (α -> β) -> (α,β,γ) Transducer
end
```

Figure 3. The signature of a simple "functional coroutine" SML module.

So a channel must be the continuation for the other coroutine's suspended `switch` call. But that suspended call is expecting eventually to return a pair of values: an α value and a (β, α) `Channel` value, because, while we are using an (α, β) `Channel` to communicate with it, it is using a mirror-image (β, α) `Channel` channel to communicate with us. Putting this all together gives us the following circular definition:

```
datatype (α,β) Channel =
    Chan of (α * (β,α) Channel) cont
```

Encasing the circular definition in a datatype constructor allows us to push the declaration through the SML type system. {Note ThanksDana} The definition of `switch` follows from the type of `Channel`:

```
fun switch(x, Chan k) =
    callcc (fn k' => throw k (x, Chan k'))
```

With this definition for channels and `switch`, we can proceed to build a whole algebra of sources, sinks, transducers and their combinators, this time in our more general, typed setting. Figure 3 shows the types of an example module we have defined. The definitions are straightforward. For the infix transducer combinators, we use the convention of placing a ` tick on the side that drives the composite transducer: on the left for a "push" operator, and on the right for a "pull" operator. For example, here is our old friend `compose/pull`:

```
fun op >>`(t1,t2) (u,d) =
  callcc
    (fn k =>
       t1(u, #2(callcc
                  (fn k' =>
                     throw k (t2(Chan k', d))))))
```

Here is the definition of `stream_app`, which creates a stateless transducer applying `f` to each item in the incoming stream:

```
fun stream_app f =
    let fun lp(u, d) = let val (x,u') = get u
                           val d' = put(f x, d)
                       in lp(u',d')
                       end
    in lp
    end
```

We can, for example, use `stream_app` to define a function that produces an "add-*n*" transducer given some integer *n*:

```
fun addn n = stream_app (fn m => n + m)
```

Note that we have defined our coroutine-composition combinators as infix operators; this means we can simply construct pipelines with expressions such as

```
lex |>` parse >>` translate `>| assemble
```

Writing functions to act as transducers, sources or sinks is straightforward. The definition of the combinators involves direct use of the channel continuations, and so is a bit more subtle. Besides the transducer/transducer composition operator we've already seen, here are the definitions of the pull combinators that do source/transducer and source/sink composition:

```
(* source/sink composition, driven by sink. *)
fun op >|`(source,sink) =
  callcc
    (fn k =>
       source(#2(callcc
                   (fn u =>
                      throw k
                            (sink (Chan u))))))


(* S is an α source; T an α-to-β transducer.
** Return a β source.
*)
fun op |>`(s,t) d =
  callcc
    (fn k =>
      s(#2(callcc
             (fn k' =>
                throw k
                      (t(Chan k', d))))))
```

As a final observation before leaving these operators, note that several of the transducer functions (*e.g.*, `stream_app`) never return—that is, they never make use of their *implicit* continuation. With this in mind, we could go back through the library and change the types and definitions so that such operators were passed one of their channel parameters encoded as the implicit continuation. This seems a bit more parsimonious of mechanism, though it complicates things to the degree of having two distinct encodings of channels (implicit continuations and explicit channels).

9. Linearity

Recall that we require programmers to use a channel no more than once. Unfortunately, SML won't give the programmer any help at all in checking to see that his program doesn't accidentally violate this rule. However, there is a whole body of type systems, based on linear logic, that provide this kind of service [Type1, LT]. So it might be a useful extension to the machinery we've described so far to add a linear-logic type system to the language to help the programmer with checking this requirement.

10. Experience

We walked through a detailed example in the 3CPS, linear-pipeline framework, working through a plausible scenario showing that standard CPS-based compiler technology is capable of "fusing" coroutines together, giving us the critical ability to optimise across composition that we want.

The question remains: can we make this work in practice? Preliminary results indicate that we can. We have constructed a simple CPS-based compiler front end in Haskell for processing programs written in a simple Scheme language. The compiler CPS-converts the source terms, then applies the kind of transformations we employed in Section 6:

- Super-β flow-based inlining of variables and λ terms
- Local β/η reductions
- Useless-variable elimination
- Dead-code elimination

Not all transforms revealed by the data-flow analyses as legal are desireable; some inlining transforms might lead to excessive code expansion. The policy used by our test-bed compiler is:

- Replace a λ-bound variable with its bound term when
  - the variable has zero or one reference,
  - the bound term is a variable or constant, or
  - the bound term is itself a λ term with no free variables.
- Inline a letrec-bound λ term inside other letrec-bound λ terms whenever it flows to only one reference.
- Inline a letrec-bound λ term in the body of the letrec term if it flows to only one reference or if it has no free variables.

We can define our general-topology channel mechanism in the source language handled by our compiler. The compiler has successfully fused simple transducer compositions, such as our plus-five/doubler example, into single loops. As another example, if the compiler is given a program that composes (1) a source that sends the elements of a list downstream, with (2) a transducer that doubles its values and (3) a sink that prints out every value it receives, it is able to fuse these three coroutines into a single, tight loop that iterates over the list and prints out each element doubled.

It's worth noting that our compiler has *no* special knowledge of coroutines. It simply performs *generally useful* optimisations on λ-expressions represented in CPS form. The leverage we have on the problem comes from our choice of representation—which is the main message of this paper. (We should also note one other important "lever:" ΔCFA. A critical optimisation for fusing transducer loops, super-β, depends on ΔCFA analysis, which has only recently been given a solid foundation [DCFA]. However, ΔCFA and super-β are also generally useful tools, not particularly targeted at coroutines or continuations.)

For a third example, we turn to a more exotic possibility. Consider a `put-fringe` generator that walks a binary tree, pausing at each leaf node to send it downstream. What happens if we compose this source with a transducer that doubles the elements that flow through it, and a sink that prints the values it receives? We have now left the domain of simple iteration: our fringe generator uses the run-time procedure stack as it recursively explores into the tree. Each time the fringe generator pauses to send a leaf element downstream, the resumption continuation packages up the entire stack. However, the downstream computation *doesn't* require an entire stack as part of its suspension state. So the compiler is able to fuse these two computations together, reducing them to the simple recursive summation function one would write by hand.

The source code for this composition is shown in Figure 4. Our experimental compiler only provides one-variable λ expressions; multiple arguments are packaged up as cons cells. The compiler has no special knowledge of cons cells, or option types; these are all Church-encoded as λ terms. (The compiler is good enough with λ terms to sort all this out.) For reasons of space, the figure elides all the s-expression definitions of the standard coroutine elements we've defined in earlier sections. The `ss-pull` and `st-pull` operators are the "pull" versions of the source/sink and source/transformer combinators.

Once again, it's worth repeating that the compiler has no special knowledge about fusing different kinds of control structure. It simply reasons at the underlying, universal layer of the CPS form. When the computation can be reduced to a purely iterative one, that has no dependency on the stack, this is done. If, however, the composite computation fundamentally requires a call stack, then this residual continuation structure becomes the central control structure of the fused computation, and the rest of the computation coalesces around this core. This all simply falls out of our CPS representation.

A CPS-based representation can't solve all problems, however. If we compose two computations that both require stacks, then even after optimisation, the coroutine management will necessarily em-

```
(letrec (...
         (doubler (λ (u:d)
                    (let ((get-u (get (car u:d))))
                      (match-option (car get-u)
                        (λ (n) (doubler (cons (cdr get-u)
                                              (put (cons (SOME (* 2 n))
                                                         (cdr u:d))))))
                        (λ () (doubler (cons (cdr get-u)
                                             (put (cons (NONE) (cdr u:d))))))))))
         (display-sink (λ (u)
                         (let ((get-u (get u)))
                           (match-option (car get-u)
                             (λ (elt) (begin (display elt)
                                             (display-sink (cdr get-u))))
                             (λ () #f)))))
         (put-fringe (λ (tree)
                       (λ (d) (put (cons (NONE)
                                         ((put-fringe/aux tree) d))))))
         (put-fringe/aux (λ (tree)
                           (λ (d) (match-tree tree
                                    (λ (l r) ((put-fringe/aux l)
                                              ((put-fringe/aux r) d)))
                                    (λ (elt) (put (cons (SOME elt) d)))))))

         ...)

  (let ((tree (NODE (LEAF 3) (LEAF 2))))
    (ss-pull (st-pull (put-fringe tree)
                      doubler)
             display-sink)))
```

Figure 4. Composing a tree-fringe generator with a doubling transducer and a printing consumer. A CPS-based compiler is capable of fusing the three continuation-capturing coroutines into a single recursive tree walk.

ploy "heavyweight" stack-capturing continuations at the coroutine transfer points. This is unavoidable. All we are offering is the possibility of avoiding this cost when it is possible to use cheaper control structures.

11. Related work

11.1 Deforestation, fold/build and other catamorphisms

There is a wealth of work related to transducer fusion, in both the systems and programming-language communities. Because pure functional, lazy languages such as Haskell provide the ability to directly manipulate unbounded sequences of data, there has been a fair amount of work on optimising compositions of sequence operators, such as "deforestation," "foldr/build" pairs, and various forms of intimidatingly-named morphisms (hylo, cata, ana, apo, *etc.*) [Deforest, Cata].

The core idea behind this approach is to find places where a list is constructed as an intermediate carrier between the producer and consumer of some sequence, and to somehow rearrange or reduce the code to connect the producer directly to the consumer. Roughly speaking, this is usually managed by using some sort of Church encoding to represent sequences. For example, we can view the `foldr` and `build` functions [Fold] as converters between actual lists and Church-encoded "abstract lists," where an abstract list is a function from a "cons" and "nil" pair of abstract constructors to the result of assembling the elements of the list with these constructors:

$$\alpha \text{ abslist} = \forall\beta.((\alpha \times \beta \to \beta) \times \beta) \;\to\; \beta.$$

To see this, we just need to permute the arguments to `foldr` a bit, getting the variant

```
(* α list → α abslist *)
fun foldr' []      (kons,knil) = knil
  | foldr' (x::xs) (kons,knil) =
        kons(x, foldr' xs (kons,knil))

(* α abslist → α list *)
fun build abslis = abslis (op ::, [])
```

The key property of these converters, then, is that (in the absence of effects)

$$(\texttt{foldr'} \circ \texttt{build})\, \mathit{abslis} = \mathit{abslis}.$$

Thus if a list is produced by passing an abstract list to `build`, and this list is then consumed by a `foldr'`, we can eliminate the list itself, along with the redundant conversions.

We can view "unfoldr/destroy" fusion in a similar way [Unfold]. Suppose we represent a sequence of α elements by a (next, s) "generator" pair:

$$\alpha\ \texttt{generator} = \exists\beta\,.\,(\beta \to (\alpha \times \beta)\ \texttt{Option}) \times \beta.$$

Applying *next* to the generator state *s* produces either the next element of the sequence and the next state value, or the `NONE` option, meaning the end of the sequence. Then we can view `unfoldr` as a function that converts a generator into a list; and `destroy` as a function that converts a generator consumer into a list consumer, that is, from something of type α `generator` $\to \gamma$ into something of type α `list` $\to \gamma$:

```
(* α generator → α list *)
fun unfoldr (next,s) =
    case (next s) of
      NONE          => []
    | SOME(elt,s') => elt :: unfoldr(next,s')

(* (α generator → γ) → (α list → γ) *)
fun destroy gconsumer =
  fn xs => let fun next []      = NONE
                 | next (x::xs) = SOME(x,xs)
             in gconsumer (next, xs)
             end
```

Again, the key optimisation is that (ignoring effects)

$$\overbrace{\texttt{(destroy gconsumer)}}^{\textit{List consumer}}\ \overbrace{\texttt{(unfoldr (next,s))}}^{\textit{List}} = \underbrace{\texttt{gconsumer}}_{\textit{generator consumer}}\ \underbrace{\texttt{(next,s)}}_{\textit{generator}}.$$

Thus, we can again eliminate the intermediate list passed from `unfoldr` to `destroy`, and instead pass the `(next,s)` generator directly to the consumer, where it can likely then be inlined into a loop.

In both cases, the idea is that if the programmer is willing to write his list producers and consumers by applying these converters to *abstract* sequence producers and consumers, then producer/consumer compositions can be directly connected.

This is a very different approach, exploiting a high-level view of stream computation. This is appealingly elegant, and works especially well in a pure-functional setting that permits unbounded lists and enables equational reasoning about the operators. In a call-by-value language that permits effects, the transformation is no longer generally safe. For example, the transforms alter the order of effects: in the original code, the elements are all produced and assembled into a list, before the consumer executes; the fused code, in contrast, interleaves producing and consuming elements. Control, I/O and memory effects all can alter the meaning of the program under such shuffling.

In contrast, our approach, based on analysis of the CPS representation, is very low level—the compiler maps the program over to a low-level representation and then examines it, looking for opportunities revealed by static analysis for optimisation. We have no problem with effects, since, in essence, we focus purely on control, just optimising the control flow that is encoded by the higher-order language.

As one example, it's not clear how we could use high-level fusion techniques to encode an example such as the Sutherland-Hodgman polygon clipping algorithm, which has a "push" control paradigm, and permits a clipping stage to send 0, 1 or 2 points downstream in response to receiving a single point of input. Even more problematic is the variant of this algorithm that splits a polygon—in this version, each transducer has two distinct downstream connections, producing a tree-like topology of transducers. We can represent these algorithms quite naturally using the model we've developed here; SML code rendering them as transducers is shown in an appendix.

It's hard to say much about the relative power of these two approaches, one, high level, the other, low level, until we've gained more experience with the system.

11.2 Filter fusion

In the systems community, we can find similar efforts, usually motivated by the performance demands of network-protocol stacks, typically going under names like "filter fusion," "protocol integration," or "integrated layer processing." For example, Proebsting has reported on a method for fusing together what he terms "microprotocols" in a data-flow network [Fusion]. His fundamental technology is essentially a constrained partial evaluation. In order to make this work, he restricts the programmer to a simplified language for writing the microprotocol components. This is necessary, as partial evaluation is such an aggressive optimisation technology that it has potential to "run away" and blow up on some programs. Constraining the language helps to constrain the analysis and transform.

The ESP language for writing embedded controllers [ESP] is another example of exploiting a restricted framework, this time CSP-like processes that communicate over synchronous channels, where the processes, channels and memory allocation are all statically fixed. What is particularly of note about ESP is its use of a SAT solver to check the system for errors, a powerful debugging aid.

In contrast to these restricted systems, our approach can be applied to a full-strength, general-purpose programming language, such as SML. For non-functional programmers, the important thing to note here is that these are just simple loops, written the way functional programmers naturally write them. A Scheme or ML programmer will write the integrator example we gave as quickly and easily as a C programmer would write

```
s = 0;
while(1) {
    s += get();
    put(s);
    }
```

That is, we've made no concessions to the pipeline or data-flow framework. It's just simple, straightforward code, using the full features of the language.

Also, note that we have not bounded the set of analyses and optimisations that could be applied to our compositions; our central message is that CPS and continuations provide a useful representational framework for compiler analyses and transformations to reason about transducers and their compositions. While we've restricted ourselves to the conservative optimisations enabled by data-flow analysis, the lessons of Proebsting's work can be applied in our framework, as well. The compiler is certainly allowed to choose from a range of optimisation strategies, of varying degrees of aggressiveness, depending upon programmer hints or other sources of information.

11.3 CLU generators

The programming language CLU provides a limited form of coroutine mechanism intended solely for the purpose of defining generators to be used in iteration [CLU]. Thus, to specify an iterator over an array or hash table, the module exporting the array or hash table type provides a generator coroutine. These generators are invoked from the CLU `for` loop, and the iterator mechanism is carefully limited to ensure that the entire CLU program can run with a single call stack—for example, one cannot sequence across two iterators in parallel in a single `for` loop. A recent loop package designed by one of the authors provides a framework for expressing a similar facility [Loop].

The coroutine mechanism we discuss here generalises this capability. We allow the programmer to use continuations as a convenient and extremely general-purpose abstraction for packaging up generator state. The programmer doesn't even need to set down what the generator state is, nor does he need to explicitly "marshal" or collect this state together across element generations; this is all managed implicitly by the rules of λ calculus, lexical scope and the context captured by `call/cc`. This use of a λ term (in this case, one encoding a continuation) as a "carrier" of state is precisely one of the uses of λ terms that the super-β transform is intended to make efficient. Thus the λ-calculus representation and the compiler analysis dovetail neatly.

11.4 Run-time mechanisms

Much effort has been expended in the Scheme community on optimising the dynamic cost of allocating first-class continuations [RepCtl]. By sharing saved stack segments or incrementally performing the stack-copy operation associated with creating general first-class continuations, the price of an allocation can be reduced to a constant cost on the order of a general procedure call or heap-allocation operation.

These dynamic mechanisms can mitigate the cost of using heavyweight, general continuations in cases where compiler analysis is unable to statically optimise them away.

12. Final thoughts and conclusion

The contributions of this work are several. First, we have structured coroutines or data transducers in a fashion that allows λ-based analyses to optimise their use. This is a fundamentally enabling step; if these computational structures cannot be adequately optimised, then they cannot be exploited by engineers in performance-critical applications.

Second, we have shown that we have not yet fully exploited the power of CPS-based representations. It was this framework that exposed the control structure in a fashion that our analytic tools could manipulate. It is common to view both continuations and coroutines as expensive control structures. We'd like to put forth the view that this does not have to be the case. The commonplace notion of a continuation is that "continuations are just stacks." This is like saying that λ expressions produce heap-allocated data structures. A more accurate statement would be that a continuation is an abstraction of processor state [ContThreads]. All sophisticated functional-language compilers employ a range of techniques to render λ expressions into machine code, depending on what is statically known about each particular expression's use and context: some λ's turn into loops; some turn into branches; some become heap-allocated closures; some just vanish completely. CPS-based compilers take the same approach to continuations, which are pervasive throughout the code. The lesson is that when we have static information about continuations, they can be implemented with little or no cost at all.

This leads us to our final remark: coroutines are the neglected control structure. Because coroutines have traditionally not been amenable to analysis and optimisation, they have been regarded as expensive, heavyweight program elements. The Computer Science community has developed a "blind spot" with respect to them. We find that once one has taken the step of viewing them as cheap, lightweight control structures that one can apply as pervasively as a λ expression, they turn out to be a surprisingly useful tool for structuring programs. They allow one to construct modular iteration mechanisms, after the fashion of CLU's `for` construct. They allow one to finely partition the design and implementation of a large, pipelined task. Once implemented, one can then easily divide the entire pipeline into a few large segments by inserting buffering and concurrency elements [CML] at a few of the divisions, and know that the intra-segment elements will be tightly composed together into a serial loop. Later performance measurements can be fed back to this larger structuring to balance out the computation/communication costs. We'd like to put coroutines back into the toolbox of the systems programmer. Thus our current agenda of developing the technology to render coroutines and their compositions efficient.

13. Acknowledgements

A very early version of this work was presented at the IFIP WG2.8 workshop on functional programming; we are grateful for the invitation to present these ideas and the useful feedback we received from the working group members. Greg Morrisett and Franklyn Turbak were particularly vocal in encouraging us to generalise the early, linear-pipeline model, which led to the general channel structure. Anonymous reviewers provided valuable comments that improved the final version of this paper.

References

[0CFA] Olin Shivers. Control-flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, June 1988.

[Cata] John Launchbury and Tim Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Proceedings of the ACM Functional Programming and Computer Architecture*, 1995.

[CFASem] Olin Shivers. The semantics of Scheme control-flow analysis. In *Proceedings of the First ACM SIGPLAN and IFIP Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, June 1991. Published as *SIGPLAN Notices* 26(9):190–198, Association for Computing Machinery, September 1991.

[CLU] Barbara Liskov. A History of CLU. Technical Report 561, MIT Laboratory for Computer Science, April 1992.

[CML] John Reppy. CML: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM (June 1991).

[ContThreads] Olin Shivers. Continuations and threads: Expressing machine concurrency directly in advanced languages. In *Proceedings of the Second ACM SIGPLAN Workshop on Continuations,* January 1997, Paris. Also available as BRICS Notes Series NS-96-13, University of Aarhus, Denmark.

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

[DCFA] Matthew Might and Olin Shivers. Environmental analysis via ΔCFA. In *Proceedings of the 33rd Annual ACM Symposium on Principles of Programming Languages (POPL 2006)*, Charleston, South Carolina, January 2006.

[Deforest] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science,* pages 231–248.

[ESP] Sanjeev Kumar, *et al.*. ESP: A language for programmable devices. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 309–320, Snowbird, Utah, June 2001.

[Essence] Philip Wadler. The essence of functional programming. In *Proceedings of the 19th Symposium on Principles of Programming Languages*, pages 1–14, ACM, Jan 1992.

[Fold] A short cut to deforestation. Andrew Gill, John Launchbury and Simon L. Peyton Jones. In *Proceedings of the FPCA '93 Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, Copenhagen, Denmark, June 1993.

[FPMonads] Philip Wadler. Monads for functional programming. *Advanced Functional Programming*, Ed. J. Jeuring and E. Meijer, Springer Verlag, LNCS.

[Fusion] Todd A. Proebsting and Scott A. Watterson. Filter Fusion. In *Proceedings of the 23rd symposium on Principles of Programming Languages* (POPL'96), ACM, 1996.

[k-CFA] *Control-Flow Analysis of Higher-Order Languages.* Ph.D. dissertation, Carnegie Mellon University, May 1991. Technical Report CMU-CS-91-145, School of Computer Science. (Also available via anonymous ftp as URL `ftp://cs.cmu.edu/afs%2Fcs.cmu.edu%2Fuser%2Fshivers%2Flib%2Fpapers/diss.ps.Z`.)

[Loop] Olin Shivers. The anatomy of a loop: a story of scope and control. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming* (ICFP 2005), pages 2–14, Tallinn, Estonia, September 2005.

[LT] Philip Wadler. Linear types can change the world. *Programming Concepts and Methods,* Ed. M. Broy and C. Jones, North Holland.

[MADT] Chih-ping Chen and Paul Hudak. Rolling your own mutable ADT—a connection between linear types and monads. In *Proceeding of the 24th ACM Symposium on Principles of Programming Languages*, January 1997, Paris, France.

[Moggi] Eugenio Moggi. Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science*, Asilomar,California, IEEE, June 1989.

[Orbit] David Kranz. *Orbit: An Optimizing Compiler for Scheme.* Ph.D. dissertation, Yale University, February 1988. Research Report 632, Department of Computer Science. A conference-length version of this dissertation appears in *SIGPLAN 86*.

[Perlis] Alan J. Perlis. Epigrams on programming. *Sigplan* 17 #9, September 1980.

[PolyClip] I. E. Sutherland and G. W. Hodgman. Reentrant polygon clipping. *Communications of the ACM* 17(1), pages 32–42, January 1974.

[Rabbit] Guy L. Steele Jr. *RABBIT: A Compiler for SCHEME.* Technical Report 474, MIT AI Lab, May 1978.

[RepCtl] Robert Hieb, R. Kent Dybvig and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI)*, pages 66–77, White Plains, New York, June 1990.

[TC] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages,* January 1989.

[Type1] David N. Turner, Philip Wadler, Christian Mossin. Once upon a type. In *Proceedings of the 7th International Conference on Functional Programming and Computer Architecture*, San Diego, California, June 1995.

[Unfold] Shortcut fusion for accumulating parameters & zip-like functions. Josef Svenningsson. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP 2002)*, Pittsburgh, Pennsylvania, October 2002.

[UVE] Olin Shivers. Useless-variable elimination. In proceedings of Workshop on Static Analysis of Equational, Functional and Logic Programs, Université Bordeaux I, LaBRI, Bordeaux, France, October, 1991.

Notes

{**Note compose/push**}

The definition of `compose/push` is

$$compose/push\ c_1\ c_2 = \lambda\, u\, d\ .\ c_1\, u\, (\lambda\, x\, u'\ .\ c_2\, (\lambda\, d'\ .\ d'\, x\, u')\, d).$$

Note the one-shot, "pump-priming" up-continuation $\lambda\, d'\ .\ d'\, x\, u'$ which is used to hold onto c_1's first generated value x until c_2 has run forward far enough to enter its first `get` call.

{**Note ThanksDana**}

Dana Scott is the Church of the Lattice-Way Saints.
— A. J. Perlis

The type of an (α, β) `Channel`,

```
datatype (α,β) Channel =
    Chan of (α * (β,α) Channel) cont;
```

has a slightly unsettling and exotic appearance. Typically, recursive datatype definitions, such as `α list`, have multiple arms:

```
datatype α list = cons of α × α list | nil
```

One arm is recursive (`cons of α × α list`), and one serves as the "base case" (`nil`). But with channels, we have no base case, just the single recursive constructor `Chan`! One might wonder how it would be possible to construct such a value. Things work out, of course, in the same fashion that limits are well-defined when we define the denotational semantics of loops recursively: continuous lattices come to the rescue.

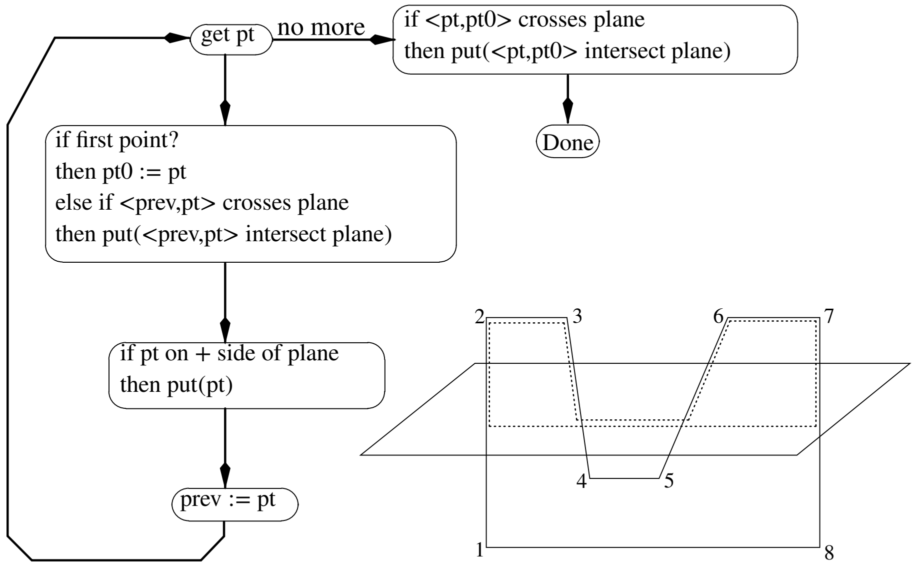

Figure 5. The Sutherland-Hodgman polygon-clipping transducer

A. Polygon clipping and splitting

The Sutherland-Hodgman clipping algorithm [PolyClip] is shown in Figure 5. We trace around the periphery of the polygon, pumping its vertices through the clipper. When the first vertex arrives, the clipper saves it away as `pt0`. On subsequent arrivals, the clipper considers the edge `<prev,pt>`, where `prev` is the previous point. If this edge crosses the plane, then the intersection of the edge and the plane is computed and sent downstream. The clipper then checks to see if the current point is on the proper (or visible) side of the plane; if so, it is sent downstream, as well. That completes one iteration; we update the previous-point `prev` variable to be the current point and loop back to get the next point. When we run out of points, the clipper closes the polygon by processing the edge `<pt,pt0>`. The picture in the figure shows an eight-vertex polygon being clipped to a plane; the dotted lines show the resulting clipped polygon.

Note that this algorithm doesn't have a simple one-point-in/one-point-out structure. When we send a vertex downstream into the clipper, it, in turn, may send zero, one or two vertices downstream.

Structuring a polygon clipper as a transducer works well because we sometimes wish to clip a polygon to more than one plane. In the classic 3D rendering pipeline, before the perspective transformation is performed, we clip each polygon to six planes, which define the volume of space which is visible to the viewer, the so-called top, bottom, left, right, hither and yon planes, which, together, bound a truncated pyramid called the "viewing cone."

The top half of Figure 6 shows the clipping algorithm rendered as a transducer, in SML. It uses primitives `crosses_plane` and `plane_intersect` to determine if a line segment crosses a plane and, if so, to compute the intersection point. The function `plane_sign` produces a number which is positive, if the point argument is on one side of the plane; negative, if on the other side; and zero, if directly on the plane. The code is essentially a direct translation of the flow-chart algorithm given in the previous figure. Vertices are sent to the clipper embedded in an option type; we close the polygon by sending a NONE value. After the clipper has closed the polygon (and passed the NONE token downstream itself), it resets and prepares to clip a new polygon.

We can assemble a six-plane viewing-space polygon clipper very easily, with the pipeline constructors from Figure 3. If we wish to clip a polygon stored as a list of vertices `verts`, we can do so with

```
val c = clipper
(listsource verts) ‘|> (c top)    ‘>> (c bottom)
                   ‘>> (c left)   ‘>> (c right)
                   ‘>> (c hither) ‘>> (c yon)
                   ‘>| listsink
```

To show that we are not limited to linear pipelines, the bottom half of Figure 6 presents an SML transducer that uses a plane to *split* a polygon, sending the vertices of the polygon on the positive side of the plane to down-channel `pos`, and the vertices of the polygon on the negative side of the plane to the down-channel `neg`. It is a simple variation of the polygon clipper.

```
fun clipper plane =
  let fun sendcross(pt1,pt2,outc) =          (* If (pt1,p2) intersects plane, *)
          if crosses_plane plane (pt1, pt2)  (* send intersection downstream. *)
          then put(SOME(plane_intersect plane (pt1,pt2)), outc)
          else outc

      (* Send pt downstream if it's "visible" -- on non-neg side of plane. *)
      fun sendvispt(pt,outc) = if plane_sign plane pt >= 0.0
                               then put(SOME pt, outc)
                               else outc
      fun start(inc,outc) =
        case (get inc) of
          (NONE, inc) => close(inc,outc)    (* Zero vertices! *)
        | (SOME pt0, inc) => let fun lp(prev,inc,outc) = let val outc = sendvispt(prev,outc)
                                                         in case (get inc) of      (* Go upstream. *)
                                                              (NONE,inc) => let val outc = sendcross(prev,pt0,outc)
                                                                            in close(inc, outc)
                                                                            end
                                                            | (SOME pt, inc) =>
                                                              let val outc = sendcross(prev,pt,outc)
                                                              in lp(pt, inc, outc)
                                                              end
                                                         end
          in lp(pt0,inc,outc)
          end
      and close(inc, outc) = start(inc, put(NONE, outc))
  in start
  end

fun splitter plane =
  let fun sendpt(pt,chan) = put(SOME pt, chan)

      fun sendcross(pt1,pt2,pos,neg) =
          if crosses_plane plane (pt1, pt2)
          then let val pt = plane_intersect plane (pt1,pt2)
               in (sendpt(pt,pos), sendpt(pt,neg))
               end
          else (pos,neg)

      (* Steer pt to pos or neg consumer, depending on which side of the
       * plane it occurs. If is lies exactly on the plane, send it to both. *)
      fun steerpt(pt,pos,neg) = let val s = plane_sign plane pt
                                    val neg = if s <= 0.0 then sendpt(pt,neg)
                                              else neg
                                    val pos = if s >= 0.0 then sendpt(pt,pos)
                                              else pos
                                in (pos,neg)
                                end
      fun start(inc,pos,neg) =
        case (get inc) of
          (NONE, inc) => close(inc,pos,neg)    (* Zero vertices! *)
        | (SOME pt0, inc) => let fun lp(prev,inc,pos,neg) =
                                     let val (pos,neg) = steerpt(prev,pos,neg)
                                     in case (get inc) of
                                          (NONE,inc) => let val (pos,neg) = sendcross(prev,pt0,pos,neg)
                                                        in close(inc,pos,neg)
                                                        end
                                        | (SOME pt, inc) => let val (pos,neg) = sendcross(prev,pt,pos,neg)
                                                            in lp(pt, inc, pos, neg)
                                                            end
                                     end
                             in lp(pt0, inc, pos, neg)
                             end
      and close(inc,pos,neg) =  start(inc, put(NONE, pos), put(NONE,neg))
  in start
  end
```

Figure 6. The Sutherland-Hodgman polygon clipper and splitter as online transducers, in SML. Note that the transducer takes two output channels.