ACM SIGPLAN Continuation Workshop

Saturday, September 24, 2011
Tokyo, Japan (co-located with ICFP)

Session 1 (chair: Yukiyoshi Kameyama)

9:00–10:00 **Continuations and classical logic: using continuations as a tool for logic**
(invited talk)
Koji NAKAZAWA (Kyoto University)

It is well known as the Curry-Howard isomorphism that there is a neat correspondence between logical systems and typed calculi, in particular, the intuitionistic natural deduction and the simply typed lambda calculus. In his paper in 1989, Griffin showed that the correspondence is extended to classical logic and calculi with control operators, and then some typed calculi based on classical logic have been proposed and studied from viewpoints of both logic and programming languages.

In this talk, I show how continuations relate to classical logic, and that we can use ideas from continuations to prove a fundamental property of logic, that is, normalization theorem for some proof systems of classical logic.

10:00–10:30 Tea break

Session 2 (chair: Tiark Rompf)

10:30–10:45 **Visualizing continuations**
Naoki TAKASHIMA, Yukiyoshi KAMEYAMA

Direct manipulation of delimited continuations allows one to write elegant and modular programs. However, it is often hard for beginners to understand their behavior due to their semantical difficulty. To ease such a burden, we have designed a new language Redex for *visualizing* delimited continuations. It has nested (multi-prompt) delimited-control operators and a serialization mechanism. The latter gives the source-term representation, rather than binary representation, of any represented values in the language so that one can see the delimited continuations at any time of execution of a program. We believe that such a feature is very useful for learning delimited continuations.

10:45–11:00 **Demonstration of Continuation based C on GCC**
Shinji KONO

We have implemented a C-like Continuation based programming language. Continuation based C, CbC was implemented using micro-C on various architectures, and we have tried several CbC programming experiments. Here we report a new implementation of CbC compiler based on GCC 4.5. Since it contains full C capability, we can use both CbC and C.

11:10–11:35 **Using delimited continuations for distributed computing with the CIEL engine**
Derek G. MURRAY, Malte SCHWARZKOPF, Christopher SMOWTON, Steven SMITH, Anil MADHAVAPEDDY, Steven HAND

CIEL is a universal execution engine for distributed computation, designed to achieve high scalability and reliability when run on a commodity cluster. CIEL supports the full range of MapReduce-style computations, and additionally Turing-powerful data-dependent control-flow that permits efficient, fault-tolerant evaluation of iterative and dynamic programming problems that are difficult to express in a pure MapReduce framework. CIEL also has a clear separation between the execution engine and the programming language interfaces, and so in this talk I will describe the integration of delimited continuations (Scala, OCaml), monadic workflow (Haskell), pure continuations (Stackless Python), and manual callbacks (Java).

11:35–12:00 **Swarm: transparent scalability through portable continuations**
James DOUGLAS

Transparent scalability is an elusive characteristic sought for successful software projects which inevitably outgrow themselves. A common way to approach the design of such applications is with the MapReduce pattern, which requires considerable foresight into how the application can be broken down into the functional map and reduce operations. A problem with this and similar approaches is the investment required at the beginning of development; the problem domain must be carefully analyzed and a solution crafted to support the predicted scalability needs. It would be preferable if applications could be developed simply and cheaply, then later, when necessary, made scalable without reworking the existing source code. We present an approach to building transparently scalable applications using Swarm, a framework which enables code execution to "follow the data" within Scala's serializable delimited continuations. Swarm abstracts the location of data across a distributed system from the developer, eliminating costly architectural and modeling requirements of popular distributed computing patterns and frameworks. We explain the design of an example implementation of a Twitter-like Web application which uses Swarm's continuation-passing style collections, and show how the developer is unburdened by the complexity of scalability. We demonstrate how this Swarm-based application can be transparently scaled without requiring changes to the code or accommodation by the architecture.

12:00–13:30 Lunch break

Session 3 (chair: Chung-chieh Shan)

13:30–14:30 **Continuation semantics in linguistics** (invited talk)
Mats ROOTH (Cornell University)

For decades, typed lambda calculus has been an essential part of the toolkit for work on semantics in theoretical linguistics. While practitoners in linguistics have been aware that the same logical and type-theoretic methods are used in theoretical computer science, the advantages of this in importing ideas into linguistics are only starting to be cashed out. The chief success so far is the "continuation semantics" for linguistic phenomena including scope and coreference. In this talk, I will explain the linguistic intuition about continuation semantics for scope, and look at some of my own research on intonational focus and ellipsis. I will also talk about using lambda calculators in teaching, and anticipated benifits of using CS-derived ideas in linguistic semantics.

14:35–15:00 **'Focus movement' by delimited continuations**
Daisuke BEKKI, Kenichi ASAI

In the past 10 years, the application of the notion of continuations to natural language semantics has been pursued in order to capture non-local aspects of semantic composition. The phenomenon of "focus" is an example of such aspects, in which the "focused element" induces a universal quantification which refers to the meaning of the whole sentence. In this talk, we will first introduce a theory of the meta-lambda calculus, a kind of two-level typed lambda calculus, to define a monadic translation by which shift/reset operators are defined via continuation monad. We then introduce Bekki and Asai's analysis of focus in which focus contains a shift operator and the adverbial "only" denotes a reset operator. Such a compositional encoding of focus becomes possible through the clear semantics of the meta-lambda calculus based on category theory.

15:00–15:30 Tea break

Session 4 first half (chair: Oleg Kiselyov)

15:30–15:55 **Modular rollback through free monads**
Conor MCBRIDE, Olin SHIVERS, Aaron TURON

Control operators prove to be an excellent tool for decoupling concerns, and in particular for separating error repair or user interaction from the processing of correct input. In a paper to appear in this year's ICFP, Shivers and Turon give one such use case, a programming pattern for "modular rollback through control logging." Using this pattern, an input processor can be written in a direct way, without any knowledge of a user's ability to back up and alter the input, or a repair module's ability to fix errors. In this talk, we will explore the theoretical underpinnings of the programming pattern, using free monads and Filinski's reify/reflect to factor the implementation. We will show, in particular, that the pattern can be seen as a particular mode of use of monadic representation.

15:55–16:20 **Yield, the control operator: applications and a conjecture**
Roshan P. JAMES, Amr SABRY

In previous work, "Yield: Mainstream delimited continuations" (TPDC 2011), we presented a generalized version of the yield control operator that was distilled from studying yield operators of various programming languages. In this brief abstract,

1. we extend that presentation to establish the connection of yield with dynamic binding, dynamic scope and generalized stack inspection in the spirit of Kiselyov et al (SIGPLAN Not. 2006),
2. we outline a lightweight workflow infrastructure in the spirit of Lu and Gannon (eScience 2008) and
3. we provide a yield monad transformer that allows yield to be composed with other effects.

Finally, we pose a question of considerable theoretical interest: do delimited continuations expressed using yield in combination with session types shed light on answer-type polymorphism?

16:20–16:45 **Correctness of functions with shift and reset**
Noriko HIROTA, Kenichi ASAI

Although shift and reset have become used to write various interesting functions, the understanding of those functions is not always simple. As an attempt to better understand their behavior, we formalize and prove correct some functions written with shift and reset in Coq. Building on Sozeau and Kiselyov's formalization of shift and reset using Generalized Continuation Monad, we first formalize Kameyama and Hasegawa's axioms for shift and reset in Coq. We then write a few functions in monadic style and prove them correct using Kameyama and Hasegawa's axioms and the standard monad laws. By carefully not unfolding the definition of monadic operators, we can effectively prove correctness of functions in direct style. We report on two case studies of this approach: reverse and times, and mention that non-trivial generalization of hypothesis is required to properly characterize the behavior of continuations.

16:45–16:55 Short break without tea

Session 4 second half (chair: Amr Sabry)

16:55–17:20 **The limit of the CPS hierarchy**
Josef SVENNINGSSON

We present a language which we refer to as *the limit of the CPS hierarchy*. It allows for an unbounded number of levels of continuations. We present a semantics in the form of an abstract machine.

17:20–17:45 **Non-deterministic search library**
Kenichi ASAI, Chihiro KANEKO

Non-deterministic programming has been used as a non-trivial application of (delimited) continuations. We report on our experience of using OchaCaml, an extension of Caml Light with (polymorphically typed) shift and reset, to write a search problem using non-deterministic operators. We provide a library for non-deterministic operations implemented using shift and reset and show how it enables us to write a search problem in direct style, using party puzzles as a concrete example.

Thanks to the Continuation Workshop program committee:

- Kenichi Asai (Ochanomizu University, Japan)
- Małgorzata Biernacka (University of Wroclaw, Poland)
- Hugo Herbelin (PPS-πr^2, INRIA, France)
- Oleg Kiselyov
- Julia Lawall (University of Copenhagen, Denmark)
- Tiark Rompf (EPFL, Switzerland)
- Hayo Thielecke (University of Birmingham, UK)

Thanks to the Continuation Workshop organizers: Yukiyoshi Kameyama (University of Tsukuba) and Oleg Kiselyov. Thanks to the ICFP organizers, especially

- Manuel Chakravarty (University of New South Wales)
- Zhenjiang Hu (National Institute of Informatics)
- Soichiro Hidaka (National Institute of Informatics)
- Gabriele Keller (University of New South Wales)
- Derek Dreyer (Max Planck Institute for Software Systems)

Welcome and enjoy!
Chung-chieh Shan (Cornell University), program chair

Visualizing Continuations

Naoki Takashima
taka@logic.cs.tsukuba.ac.jp

Yukiyoshi Kameyama
kameyama@acm.org

Department of Computer Science, University of Tsukuba

1 Overview

Control operators for delimited continuations [3, 2] are powerful and promising mechanisms for functional programs. Nevertheless, their semantics makes it hard for beginners to understand the behavior, and it is often a time-consuming task for ordinary programmers to get used to them. Even experts on control operators sometimes get confused when they use more than one set of control operators. We need better languages and better tools to make them more practical.

We believe that "visualizing" continuations is the key to solving this problem. One can easily understand literals such as integers and strings, since we can "see" them, by, for instance, printing them. We can understand the behavior of functions, since they are written as source codes, or their values (closures) can be seen on the fly by debuggers etc. On the contrary, continuations captured by control operators do not exist in source codes, and one cannot print their values during the execution of programs. Although delimited continuations are sometimes regarded as functional values in theory, we cannot easily see them just like functions in practice.

Having this idea in mind, we have designed a new programming language `Redex` [1] as an experimental tool for our ideas. `Redex` implements the following three distinguished features: delimited-control operators, serialization, and interoperability. The first feature enables one to access delimited continuations in several different styles. The second is the key to visualizing `Redex` values including functions and (delimited) continuations. The last is to reinforce the expressive power of `Redex` so as to make more practical visualization methods possible.

In the talk, we will demonstrate how these features provide an interesting testbed for *playing with delimited continuations*.

2 Programming Language `Redex`

Delimited-control operators

In the literature, many control operators for delimited continuations have been proposed including Felleisen's control/prompt and Danvy and Filinski's shift/reset, which differ whether captured continuations contain a reset at its top or not. In general, we may consider more variants:

```
reset #p { E[shift #p : k -> e] }
==>   reset #p { let k = fun y -> reset #p { E[y] } in e }
```

This is the semantics of shift/reset (if we ignore `#p`), but we get three other variants by omitting either or both of `reset` in the reduced term.

As `Redex` aims at providing a thinking tool, it provides all four control operators above. Moreover they can be *nested* by attaching names `#p` to each use of control operators, and one can simultaneously use arbitrary combination of control operators in `Redex`.

Serialization

The most notable feature of `Redex` is serialization; one can convert any[1] `Redex` value into a string, and convert it back to an equivalent value. Here is a sample `Redex` program for append

```
func append = function
  | [] -> shift #p : k -> k
  | hd :: tl -> hd :: append tl
let f = reset #p { append [1, 2, 3] }
do printn ! serialize f
```

[1] There are some .NET values which cannot be serialized.

which, when executed will print the following string as a result:

```
(\ $Internal_0003 -> (reset #p { (($OpCons 1) (($OpCons 2) (($OpCons 3)
$Internal_0003))) }));;
```

Serialization in `Redex` does not produce an unreadable bit sequence; rather, the produced string represents an original source term as a tree (modulo the generated bound variables). Let us take another example:

```
let f =
  let x = 10 in let y = 20 in
  reset #p { (shift #p : k -> k) + y }
do printn (serialize f)
```

in which the delimited continuation captured by `shift` contains a free variable `y`. Serializing such a continuation will accompany a kind of closure conversion:

```
(\ $Internal_0002 -> (reset #p { ((\ y -> (($OpAdd $Internal_0002) y)) 20) }));;
\end{quote}
```

Interoperability with F#

Another notable feature of `Redex` is interoperability with Microsoft's .NET framework. It enables one to use various libraries from `Redex` programs. We can call .NET functions, and even create a .NET object as a `Redex` value, and send a message to this object from a `Redex` function. See Fig. 1 for a simple example.

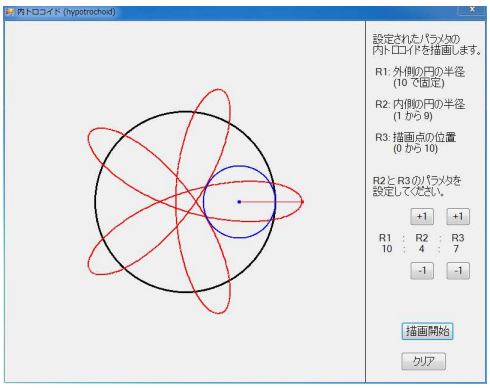


Figure 1:

As a more substantial example, a version of Fibonacci function implemented in `Redex` is shown in the appendix. It combines various features in `Redex`: (1) functional programming, (2) delimited-control operators "shift" and "reset", (3) serialization, and (4) the .NET class "StreamWriter". It is now very easy to implement a GUI for manipulating (serialized) delimited continuations, and so on.

3 Concluding Remark

`Redex` is implemented in F#, and is freely available through the URL [1]. (One needs .NET framework in advance.) Its syntax resembles OCaml's, while it is dynamically typed unlike most other functional languages. Although it remains a prototype implementation, we can play with delimited continuations and serialization, and moreover, the interoperability with .NET framework enables one to enjoy them more.

References

[1] The redex programming language. http://logic.cs.tsukuba.ac.jp/~taka/proj/redex/.

[2] Olivier Danvy and Andrzej Filinski. Abstracting control. In *LISP and Functional Programming*, pages 151–160, 1990.

[3] Matthias Felleisen. The theory and practice of first-class prompts. In *Principles of Programming Languages*, pages 180–190, 1988.

A Fibonacci in `Redex`

We show an implementation of Fibonacci function, which uses various feature of `Redex`.

The function computes each value of Fibonacci sequence one by one. Namely, when it is called, it returns a pair of the next value and the continuation (to compute the rest of the sequence). Then it serializes the captured continuation and saves it to a file using .NET class "StreamWriter" so that one can continue the rest of the computation afterwords (when the function is called next time).

```
func gen_fib v0 v1 =
  reset #p {
    let next = v0 + v1 in
    do shift #p : k -> (k, next) in
    gen_fib v1 next
  }

open System.IO
let cont_file = "cont.txt"

let write_cont k =
  let str = serialize k in
  let w_stream = new StreamWriter (cont_file) in
  w_stream.WriteLine str;
  w_stream.Dispose ()

let main () =
  let (k, v) =
    if call File.Exists(cont_file) then
      (let r_stream = new StreamReader(cont_file) in
      let str = r_stream.ReadToEnd () in
      r_stream.Dispose ();
      eval str ())
    else gen_fib 1 1 in
  printn v;
  write_cont k

do main ()
```

If we execute the program, it prints "2", "3", "5", "8", and so on, for each invocation of the program.

Demonstration of Continuation based C on GCC

Shinji KONO

e-mail:kono@ie.u-ryukyu.ac.jp
Information Engineering, University of the Ryukyus
Nishihara-cyo 1, Okinawa, 903-01, Japan

June 26, 2011

We have implemented C like Continuation based programming language. Continuation based C, CbC [1, 2] was implemented using micro-C on various architecture, and we have tried several CbC programming experiments. Here we report new implementation of CbC compiler based on GCC 4.5. Since it contains full C capability, we can use both CbC and C.

CbC's basic programming unit is a code segment. It is not a subroutine, but it looks like a function, because it has input and output. We can use C struct as input and output interfaces.

```
struct interface1  { int i; };
struct interface2  { int o; };

__code f(struct interface1 a) {
    struct interface2 b; b.o=a.i;
    goto g(b);
}
```

In this example, a code segment `f` has `input a` and sends `output b` to a code segment `g`. There is no return from code segment `b`, `b` should call

another continuation using `goto`. Any control structure in C is allowed in CwC language, but in case of CbC, we restrict ourselves to use `if` statement only, because it is sufficient to implement C to CbC translation. In this case, code segment has one input interface and several output interfaces (fig.).

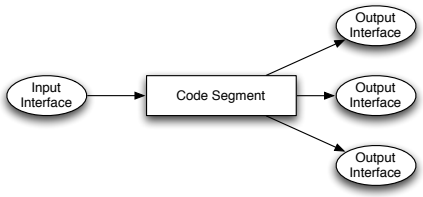


Figure 1: code

We have found interfaces should be a dual of code segment. It should be data segment. Concept of data segment is important in parallel computation and distributed computation because location of data is important. Data segment is not only a data structure in computation, but it has computation also. If data segment are moved it requires copying time and code segment have to wait for the data segment.

We will show an implementation of data segment in our Cerium Task Manager for Cell architecture and multi processor.

We built basic application such as grep and its demonstration will be shown.

References

[1] Shinji Kono , "Implementing Continuation based language in GCC ," in *Continuation Festa 2008*, April 2008.

[2] Shinji KONO, "CbC," March 2008. [Online]. Available: https://sourceforge.jp/projects/cbc/

Using Delimited continuations for Distributed Computing with the CIEL Engine

Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy and Steven Hand

Presenter: Anil Madhavapeddy

Tools for programming distributed environments generally fall into one of two camps. Some, such as MapReduce or Dryad, insulate the programmer from many of the difficulties of distributed programming, at the expense of restricting the programming model. Others, such as MPI, provide a much more generic framework but force developers to consider all of the low-level details. We believe this is a false choice: the framework should provide a simple interface to naive programmers, but allow more demanding users to express more complicated designs when necessary.

Our CIEL distributed execution engine is able to expose this trade-off and return control to its users. CIEL enables programs written in different languages and programming models to run as tasks in a distributed parallel computation. The key feature of CIEL is its ability to support data-dependent control flow during a distributed job, which allows it to run unbounded iterative and recursive algorithms. CIEL tasks have a mechanism for spawning additional tasks and can perform iteration using a tail-recursive style, with explicit continuation-passing.

While this is sufficiently expressive, it is not the most intuitive programming model for all applications. We have extended CIEL to support executors which present the illusion of a single thread of execution that is transparently broken down into parallel tasks. As a result, the programmer can write straight-line code in several languages, such as Python, Scala and OCaml, which will be executed as a reliable distributed job. These so-called 'threaded executors' use continuations to serialise the state of the program when a remote data reference is unavailable, and the CIEL engine takes care of resuming execution (potentially on a different host) when that reference subsequently becomes available.

In this talk, we will briefly outline how the CIEL engine works, and then demonstrate the OCaml and Scala executors use serialisable delimited continuations to interface with CIEL. A more detailed overview of the OCaml interface is also available separately [3].

[1] CIEL: a universal engine for distributed data-flow programming; USENIX NSDI 2011; online at http://anil.recoil.org/papers/2011-nsdi-ciel.pdf

[2] A Polyglot Approach to Cloud Programming; draft paper online at

http://www.cl.cam.ac.uk/~ms705/pub/papers/2011-ciel-socc-draft.pdf

[3] DataCaml - a first look at distributed dataflow programming using OCaml
http://anil.recoil.org/2011/06/18/datacaml-with-ciel.html

CW2001

ACM SIGPLAN Continuation Workshop 2011

Swarm

Transparent Scalability Through Portable Continuations

`https://github.com/sanity/Swarm`

Submitted on:

Jul 2, 2011

Submitted by:

James Douglas

`james@earldouglas.com`

Abstract

Transparent scalability is an elusive characteristic sought for successful software projects which inevitably outgrow themselves. A common way to approach the design of such applications is with the MapReduce pattern, which requires considerable foresight into how the application can be broken down into the functional map and reduce operations. A problem with this and similar approaches is the investment required at the beginning of development; the problem domain must be carefully analyzed and a solution crafted to support the predicted scalability needs. It would be preferable if applications could be developed simply and cheaply, then later, when necessary, made scalable without reworking the existing source code. We present an approach to building transparently scalable applications using Swarm, a framework which enables code execution to "follow the data" within Scala's serializable delimited continuations. Swarm abstracts the location of data across a distributed system from the developer, eliminating costly architectural and modeling requirements of popular distributed computing patterns and frameworks. We explain the design of an example implementation of a Twitter-like Web application which uses Swarm's continuation-passing style collections, and show how the developer is unburdened by the complexity of scalability. We demonstrate how this Swarm-based application can be transparently scaled without requiring changes to the code or accommodation by the architecture.

Supplementary material

Code selections

The following code represents a selection from the Swarm source code hosted on GitHub. It includes a loop which listens for incoming serialized continuations from remote nodes, and sends them off to be dereferenced, executed, and/or relocated as appropriate.

Swarm's `listen` loop:

```
  val server = new java.net.ServerSocket(port);

  var runnable = new Runnable() {
    override def run() = {
      while (true) {
        val socket = server.accept()
        val ois = new java.io.ObjectInputStream(socket.getInputStream())
        val bee = ois.readObject().asInstanceOf[(Unit => Bee)]
        debug("resuming execution from " + local)
        Swarm.continue(bee)
      }
    }
  }
  Swarm.executor.execute(runnable)
```

Swarm's `continue` method:

```
  def continue(f: Unit => Bee)(implicit tx: Transporter) {
    execute(reset(f()))
  }
```

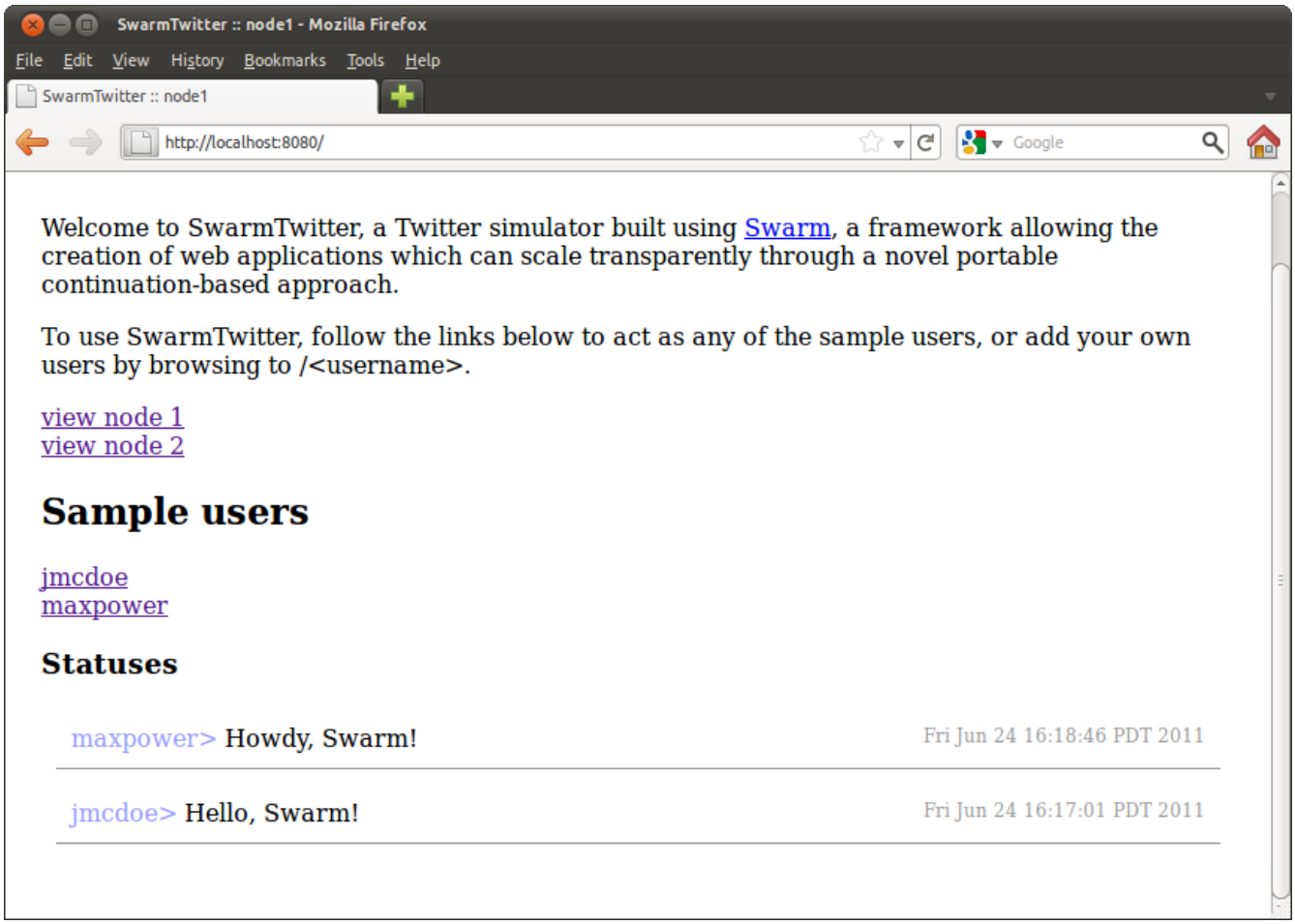
Swarm's `execute` method:

```
  def execute(bee: Bee)(implicit tx: Transporter) {
    bee match {
      case RefBee(f, ref) if (tx.isLocal(ref.location)) =>
        if (!Store.exists(ref.uid)) {
          val newRef = Store.relocated(ref.uid)
          ref.relocate(newRef.uid, newRef.location)
          tx.transport(f, ref.location)
        } else {
          Swarm.continue(f)
        }
      case RefBee(f, ref) => tx.transport(f, ref.location)
      case IsBee(f, destination) if (tx.isLocal(destination)) =>
        Swarm.continue(f)
      case IsBee(f, destination) => tx.transport(f, destination)
      case NoBee() =>
    }
  }
```

Demo screenshot

The following screenshot shows the main view of the Swarm Twitter demo after statuses have been submitted by different users. These statuses are made available on multiple server nodes via Swarm.

Swarm Twitter main view:



"Focus Movement" by Delimited Contintuations

Daisuke Bekki Kenichi Asai

Ochanomizu University
Faculty of Science, Department of Information Science *

September 24, 2011

1 Linguistic Background: Focus in Formal Semantics

Rooth (1992) discussed the truth conditions of the two sentences (1a) and (1b) which structually differ only in the location of focus (indicated by $[\,]_F$): in a situation where Mary introduced Bill and Tom to Sue, with no other introductions, (1a) is false while (1b) is true.

(1) a. Mary only introduced $[\text{Bill}]_F$ to Sue.
b. Mary only introduced Bill to $[\text{Sue}]_F$.

In order to account for such a contrast, Rooth (1992) claimed that the semantic compositions for sentences such as (1) involves the notion of *alternative sets*: for example, $\{x \mid \mathit{introduce}(m, x, s)\}$ for (1a) and $\{x \mid \mathit{introduce}(m, b, x)\}$ for (1b), with which the truth conditions for (1a) and (1b) can be represented as (2a) and (2b), respectively.

(2) a. $\forall x(x \in \{x \mid \mathit{introduce}(m, x, s)\} \leftrightarrow x = b)$
b. $\forall x(x \in \{x \mid \mathit{introduce}(m, b, x)\} \leftrightarrow x = s)$

In line with the generative tradition, Wagner (2006), among others, adopts an operation called "focus movement" which is an instance of covert movements, in order to obtain the structures for alternative sets.

On the other hand, Barker (2004) pointed out that the alternative sets exactly correspond to the *continuations* (Stratchey and Wadsworth (1974)) of the semantic representations of the focused elements, and proposed a possible implementation by means of the `fcontrol/run` operators (Sitaram and Felleisen (1990)). However, whether those operators can be defined by Barker's version of continuations was not clear enough. Shan (2007) further developed and extended this idea of a continuation-based analysis to various lingistic side-effects, including quantification, binding, wh-questions, and superiority effects.

This abstract aims at elaborating the analysis of focus in Bekki and Asai (2010), which attempted to show that delimited continuations may replace covert movements with purely compositional calculations. We first introduce the meta-lambda calculus, a kind of two-level calculus, and define (a variant of) `shift` and `reset` (Danvy and Filinski (1990)) in terms of the meta-lambda calculus. We then show how thus defined `shift` and `reset` can be used to explain focus movements in a compositional way. It appears that the use of a two-level calculus for semantic representations naturally separate covert movements from semantic representations of base sentences.

*2-1-1 Otsuka, Bunkyo-ku, Tokyo 112-8610, Japan.

2 Meta-Lambda Calculus

The syntax of the meta-lambda calculus M consists of the center column of the following table:

	meta-lambda calculus	two-staged lambda calculus
base-level variable	x	$\underline{x}$
base-level abstraction	$\lambda x.\, M$	$\underline{\lambda}\, \underline{x}.\, M$
base-level application	$M_1\, M_2$	$M_1 \underline{@} M_2$
meta-level variable	X	$\overline{x}$
meta-level variable with substitution	$X[M/x]$	(binding-time error)
meta-level abstraction	$\zeta X.\, M$	$\overline{\lambda} \overline{x}.\, M$
meta-level application	$M_1 \lightning M_2$	$M_1 \overline{@} M_2$

In addition to the standard base-level variables, abstractions, and applications, the meta-lambda calculus has corresponding meta-level terms: meta variables (possibly with substitution), meta abstractions $\zeta x.\, M$, and meta applications $M_1 \lightning M_2$.

The intended meaning of the meta-lambda terms are close to the standard two-staged lambda calculus, whose syntax is shown in the right column of the above table. In the two-staged lambda calculus, meta-level (or static) terms are reduced first to generate base-level (or dynamic) terms. Likewise, we can regard meta-level reduction in the meta-lambda calculus as generation of base-level terms.

However, there are two important differences. First, in the meta-lambda calculus, base-level reduction can be done before meta-level reduction. While a term like $\overline{\lambda} \overline{x}.\, (\underline{\lambda}\, \underline{y}.\, \overline{x}) \underline{@} \underline{a}$ results in a binding-time error because the static lambda cannot be reduced away, corresponding meta-lambda term $\zeta X.\, (\lambda y.\, X)\, a$ is well-typed and is reduced to $\zeta X.\, X[a/y]$ where $X[a/y]$ represents that when X is instantiated to a base-level term, the variable y occurring in it is replaced with a. A similar notion of meta variables with substitution appears in Nanevski et al. (2008).

Secondly, α-equivalence of both the meta- and base-level terms are maintained in the meta-lambda calculus. In the two-staged lambda calculus, it is typically required to generate fresh variables when residualizing dynamic lambdas. In the meta-lambda calculus, such hygiene is maintained automatically in a way similar to the multi-staged language by Kim et al. (2006). However, the detailed comparison is still left as a future work. Currently, the syntax, type system, and categorical semantics of the meta-lambda calculus are fixed, and the soundness of α-, β-, and η-equivalence has been shown by Masuko and Bekki (2011).

3 Handling Focus with Continuations

Our analysis achieves purely compositional analysis of focus by translating a semantic representation into CPS using the following set of rules.

Definition 1 (Translation to continuation monad)**.** The *translation rules* of lambda terms into meta-lambda calculus are defined as follows.

$$
\begin{aligned}
[\![x]\!]_c &= \zeta\kappa.(\kappa \lightning x) \\
[\![c]\!]_c &= \zeta\kappa.(\kappa \lightning c) \\
[\![\lambda x.M]\!]_c &= \zeta\kappa.([\![M]\!]_c \lightning (\zeta v.\kappa \lightning (\lambda x.v))) \\
[\![\forall x(M)]\!]_c &= \zeta\kappa.([\![M]\!]_c \lightning (\zeta v.\kappa \lightning (\forall x(v)))) \\
[\![MN]\!]_c &= \zeta\kappa.([\![M]\!]_c \lightning (\zeta m.[\![N]\!]_c \lightning (\zeta n.\kappa \lightning (mn))))
\end{aligned}
$$

This translation is an identity translation if we pass an identity function as the initial continuation and reduce meta-level terms at translation time. It traverses all the subterms including under the binders, and reconstructs the original term.

The `shift`/`reset` operators are then defined in the following way.

Definition 2 (Control operators)**.**

$$\begin{aligned}[\![\texttt{shift}\ \kappa.M]\!]_c &= \zeta\kappa.[\![M]\!]_c \natural (\zeta x.x)\\ [\![\texttt{reset}(M)]\!]_c &= \zeta\kappa.\kappa\natural([\![M]\!]_c \natural (\zeta x.x))\end{aligned}$$

Although the definition of `shift` above looks unfamiliar, it is almost the same as the original definition of `shift` except that the continuation κ is *not* reified as a function but is used as is as the meta-level function. In this setting, `shift` captures the meta-level continuation or translation-time continuation, so to speak.

The above definition of `shift` and `reset` enables us to define the *focus operators* as follows, where focus is interpreted referring to its alternative set that is captured by the `shift` operator, and the adverbial "only" is the `reset` operator that determines the scope of the corresponding alternative set.

Definition 3 (Focus operator)**.** For any meta-lambda term $M : e$ and $N : e \to t$,

$$\begin{aligned}[M]_F &\stackrel{def}{\equiv} \texttt{shift}\ \kappa.\lambda x.\forall z(\kappa z x \leftrightarrow z = M)\\ only(N) &\stackrel{def}{\equiv} \texttt{reset}(N)\end{aligned}$$

Then the semantic representations for (1a) and (1b) are respectively calculated as follows.

$$\begin{aligned}[\![[b]_F]\!]_c &= [\![\texttt{shift}\ \kappa.\lambda x.\forall y(\kappa y x \leftrightarrow y = b)]\!]_c\\ &= \zeta\kappa.([\![\lambda x.\forall y(\kappa y x \leftrightarrow y = b)]\!]_c \natural (\zeta x.x))\\ &= \zeta\kappa.((\zeta k.\lambda x.\forall y(\kappa y x \leftrightarrow y = b)) \natural (\zeta x.x))\\ &= \zeta\kappa.\lambda x.\forall y(\kappa y x \leftrightarrow y = b)\end{aligned}$$

$$\begin{aligned}&[\![(only\ ((introduce\ [b]_F)\ s))\ m]\!]_c\\ =\ &\zeta\kappa.([\![only\ ((introduce\ [b]_F)\ s)]\!]_c \natural (\zeta f.[\![m]\!]_c \natural (\zeta x.\kappa\natural(fx))))\\ =\ &\zeta\kappa.([\![\texttt{reset}((introduce\ [b]_F)\ s)]\!]_c \natural (\zeta f.(\zeta k.k\natural m)\natural(\zeta x.\kappa\natural(fx))))\\ =\ &\zeta\kappa.\kappa\natural((([\![[b]_F]\!]_c \natural (\zeta z.(introduce\ z\ s)))m)\\ =\ &\zeta\kappa.\kappa\natural((\zeta\kappa.\lambda x.\forall z(\kappa z x \leftrightarrow z = b)\natural(\zeta z.(introduce\ z\ s)))m)\\ =\ &\zeta\kappa.\kappa\natural((\lambda x.\forall z(introduce\ z\ s\ x \leftrightarrow z = b)m))\\ =\ &\zeta\kappa.\kappa\natural\forall z(introduce\ z\ s\ m \leftrightarrow z = b)\end{aligned}$$

$$\begin{aligned}&[\![(only\ ((introduce\ b)\ [s]_F)\ m]\!]_c\\ =\ &\zeta\kappa.([\![only\ ((introduce\ b)\ [s]_F)]\!]_c \natural (\zeta f.[\![m]\!]_c \natural (\zeta x.\kappa\natural(fx))))\\ =\ &\zeta\kappa.([\![\texttt{reset}((introduce\ b)\ [s]_F)]\!]_c \natural (\zeta f.(\zeta k.k\natural m)\natural(\zeta x.\kappa\natural(fx))))\\ =\ &\zeta\kappa.\kappa\natural((([\![[s]_F]\!]_c \natural (\zeta z.(introduce\ b\ z)))m)\\ =\ &\zeta\kappa.\kappa\natural((\zeta\kappa.\lambda x.\forall z(\kappa z x \leftrightarrow z = s)\natural(\zeta z.(introduce\ b\ z)))m)\\ =\ &\zeta\kappa.\kappa\natural((\lambda x.\forall z(introduce\ b\ z\ x \leftrightarrow z = s)m))\\ =\ &\zeta\kappa.\kappa\natural\forall z(introduce\ b\ z\ m \leftrightarrow z = s)\end{aligned}$$

4 Summary and Future Work

We have demonstrated how the meta-lambda calculus can be used to explain covert movements in a compositional way. The use of two-level calculus appears to open a new area in the semantic representations of movements. However, the detailed investigation of both the meta-level lambda calculus and its use in the semantics of natural languages is still on-going.

We will investigate the analysis with further theoretical implications and empirical consequences, especially incorpolating the revisions on the formulation of meta-lambda calculus recently made by Masuko and Bekki (2011).

References

Barker, C. (2004) "Continuations in Natural Language", In the Proceedings of H. Thielecke (ed.): *the Fourth ACM SIGPLAN Continuations Workshop (CW'04). Technical Report CSR-04-1, School of Computer Science, University of Birmingham, Birmingham B15 2TT.* United Kingdom, pp.1–11.

Bekki, D. (2009) "Monads and Meta-Lambda Calculus", In: H. Hattori, T. Kawamura, T. Ide', M. Yokoo, and Y. Murakami (eds.): *New Frontiers in Artificial Intelligence (JSAI 2008 Conference and Workshops, Asahikawa, Japan, June 2008, Revised Selected Papers from LENLS5)*, Vol. LNAI 5447. Springer, pp.193–208.

Bekki, D. and K. Asai. (2010) "Representing Covert Movements by Delimited Continuations", In: K. Nakakoji, Y. Murakami, and E. McCready (eds.): *New Frontiers in Artificial Intelligence (JSAI-isAI 2009 Workshops, Tokyo, Japan, November 2009, Selected Papers from LENLS7)*, Vol. LNAI 6284. Heidelberg, Springer, pp.161–180.

Danvy, O. and A. Filinski. (1990) "Abstracting Control", In the Proceedings of *LFP90, the 1990 ACM Conference on Lisp and Functional Programming.* pp.151–160.

Kim, I.-S., K. Yi, and C. Calcagno. (2006) "A Polymorphic Modal type System for Lisp-Like Multi-Staged Languages", In the Proceedings of *the 33rd ACM Symposium on Principles of Programming Languages (January 2006).* pp.257–268.

Masuko, M. and D. Bekki. (2011) "Categorical Semantics of Meta-Lambda Calculus", In the Proceedings of *the 13th JSSST Workshop on Programming and Programming Languages (PPL2011) (in Japanese).* Joozankei, Japan, pp.60–74.

Nanevski, A., F. Pfenning, and B. Pientka. (2008) "Contextual Modal Type Theory", *ACM Transactions on Computational Logic (June 2008)* **9**(3). Article 23.

Rooth, M. (1992) "A Theory of Focus Interpretation", *Natural Language Semantics* **1**, pp.75–116.

Shan, C.-c. (2007) "Linguistic side effects", In: C. Barker and P. Jacobson (eds.): *Direct compositionality.* Oxford University Press, pp.132–163.

Sitaram, D. and M. Felleisen. (1990) "Control delimiters and their hierarchies", *LISP and Symbolic Computation* **3**(1), pp.67–99.

Stratchey, C. and C. Wadsworth. (1974) "Continuations: a mathematical semantics for handling full jumps", Technical report, Oxford University, Computing Laboratory.

Wagner, M. (2006) "NPI-Licensing and Focus Movement", In the Proceedings of E. Georgala and J. Howell (eds.): *SALT XV.* Ithaca, NY: CLC Publications.

Modular rollback through free monads

Conor McBride* Olin Shivers† Aaron Turon‡

September 14, 2011

There are countless situations in which a simple input-consuming program is obscured by code for error robustness or interactive features. For example, lexers, parsers, typecheckers and servers must all potentially deal with erroneous input. Interactive development environments allow input to change over time, without incurring the cost of a complete reprocessing. And so on. Ideally, the code for basic input processing would be separate from the code to handle errors or make interactive changes. In a paper to appear in ICFP 2011, Shivers and Turon propose a programming pattern, *modular rollback through control logging*, for accomplishing this task.

The basic idea is to write the input-processing program against an abstract interface (API) for retrieving its input: as simple as a "next character" function for a lexer, or a tree destructurer for a typechecker. The code is written without any knowledge of or assumptions about error handling or user interaction.

A separate input-provider module implements the abstract interface. It uses first-class control (`call/cc`) to *discover* the computation performed by the input processor. It does this by *control logging*: at every invocation of "next character", for example, a continuation is captured. These captured continuations allow the input provider to return to previous control states of the input processor, attempting error repairs or simply allowing alternative input to be provided. The *only* knowledge shared between the provider and processor modules is the input API; it is therefore possible to plug together varying combinations of these modules. Finally, if side-effects are performed only via the API, it is possible to record code to roll back those side-effects as well—as in the case for removing typed characters when a user pressed the delete key. The API then guarantees that it is impossible to break the welding of *performing* a side-effect from *logging* the appropriate rollback.

In this talk, we will explore the theoretical underpinnings of the programming pattern using *free monads* and Filinski's *monadic reflection*.

A free monad syntactically records requests to perform the a given set of side-effecting operations; it does not actually perform them. The operations themselves are given as a *signature* from which the monad is freely generated—and this signature can include Hoare-style specifications via appropriate type

*University of Strathclyde
†Northeastern University
‡Northeastern University

structure. For example, we can think of the "get next character" interface for a lexer in terms of a monadic operation, `next`, with precondition `Unit` (easy to satisfy) and postcondition `Char` (the input).

Given a computation $c : M(\alpha)$ of type α within a free monad M, we can *evaluate* it, going from syntax to semantics:

$$eval : \forall \alpha . M(\alpha) \to F(\alpha)$$

where F is an appropriate functor (often, in HASKELL, the `IO` monad).

Supposing we have written our input processor against a free-monadic API, choices of evaluators correspond to choices of input providers. A straightforward one goes straight to `IO`, translating the syntactic requests for input into real interactions with an underlying input stream. But we can also implement backtracking via rollback, using an evaluator that maintains an appropriate stack of rollback instructions. Rather than using powerful control operators, we take advantage of the fact that, by dint of using a free monad, the input processor has *already* exposed its computational structure. Indeed, that structure is directly available as syntax, which can easily be squirreled away into the rollback stack.

This technique clarifies the rollback logging process: it's just a particular way of mapping from one monad (freely generated from the API) to another (real `IO`), whereby the rollback stack becomes a simple accumulator. On the other hand, it requires that we write the input-processing code in a monadic style, explicitly using the free monad in anticipation of possibly-different evaluators.

Using Filinski's reify/reflect operations (from "Representing Monads"), we can get the best of both worlds, allowing the input processing to be written in direct style, and using delimited control to shift that code into the free monad. Thus, the modular rollback pattern is just a particular mode of use of Filinski's technique—but unusual, in that we use the technique to shift existing code into a new monad, in an entirely modular way.

Yield, the Control Operator

Applications and a Conjecture

Roshan P. James
Indiana University
rpjames@cs.indiana.edu

Amr Sabry
Indiana University
sabry@cs.indiana.edu

1. Introduction

In previous work [JS11], we presented a generalized version of the *yield* control operator, distilled from examining *yield* in various programming languages, with the following monadic semantics:

$$\begin{array}{lcl} \mathit{run}\ e & \rightarrow & \mathit{Result}\ e' \\ & & \mathit{if}\ \langle e, \Box\rangle \mapsto^* \langle \mathit{return}\ e', \Box\rangle \\ \mathit{run}\ e & \rightarrow & \mathit{Susp}\ e'\ (\lambda x.\mathit{run}\ E[\mathit{return}\ x]) \\ & & \mathit{if}\ \langle e, \Box\rangle \mapsto^* \langle \mathit{yield}\ e', E\rangle \end{array}$$

In this brief abstract, (i) we extend that presentation to establish the connection of *yield* with dynamic binding, dynamic scope and generalized stack inspection in the spirit of Kiselyov et al [KcSS06], (ii) we outline a lightweight workflow infrastructure in the spirit of Lu and Gannon [LG08], and (iii) we provide a *yield* monad transformer that allows *yield* to be composed with other effects. Finally, we pose a question of considerable theoretical interest: do delimited continuations expressed using *yield* in combination with session types [Hon93a] shed light on answer-type polymorphism?

2. Dynamic Binding, Mutable Variables and Stack Inspection

Dynamic binding, mutable state and stack inspection all have established connections with delimited continuations [KcSS06, Fee03, Mor98]. The reductions below [KcSS06] describe the operational semantics for *dlet*, *get*, *set*, and *inspect* with the side condition that contexts E' do not have *dlet* bindings of the variable n.

$$\begin{array}{rcl} E[\mathit{dlet}\ n = v\ \mathit{in}\ v'] & \mapsto & E[v'] \\ E[\mathit{dlet}\ n = v\ \mathit{in}\ E'[\mathit{get}\ n]] & \mapsto & E[\mathit{dlet}\ n = v\ \mathit{in}\ E'[v]] \\ E[\mathit{dlet}\ n = v\ \mathit{in}\ E'[\mathit{set}\ n\ v']] & \mapsto & E[\mathit{dlet}\ n = v'\ \mathit{in}\ E'[v']] \\ E[\mathit{dlet}\ n = v\ \mathit{in}\ E'[\mathit{inspect}\ n\ f]] & \mapsto & E[(\lambda z.\mathit{dlet}\ n = v\ \mathit{in}\ E'[z])(f\ v)] \end{array}$$

To encode dynamic operations, we define a monad Dyn in terms of the *yield* monad. Dynamically bound variables are instances of the opaque type Name, over which equality is defined (satisfies Eq).

```
type Dyn t r = Yield t (Cmd t) r
data Cmd t = Lookup Name
           | Assign Name t
           | Inspect Name (t -> Dyn t t)
foldYield :: Monad m => Yield i o r
   -> s -> ((o, s) -> m (i, s)) -> m (r, s)

binder :: Cmd t -> (Name, t) -> Dyn t (t, (Name, t))
binder o s@(x, v) = ev o
  where
  ev (Lookup x')     | x == x' = return (v, s)
  ev (Assign x' v')  | x == x' = return (v', (x, v'))
  ev (Inspect x' f) | x == x' = tag (f v)
  ev a = tag (yield a)
  tag e = liftM (\x->(x,s)) e

dlet :: Name -> t -> Dyn t r -> Dyn t r
dlet x v e = liftM fst $ foldYield e (x, v) binder
```

The implementation can be summarized as follows — the `Cmd` represents the abstract commands for each effect and the combinator `binder` acts as an interpreter for the `Cmd`. The combinator `dlet` installs `binder` on the stack using the `foldYield` combinator (whose straightforward definition is skipped). Operators `get`, `set` and `inspect` simply dispatch the appropriate `Cmd` up the stack.

```
get :: Name -> Dyn t t
get x = yield (Lookup x)

set :: Name -> t -> Dyn t t
set x v = yield (Assign x v)

inspect :: Name -> (t -> Dyn t t) -> Dyn t t
inspect a f = yield (Inspect a f)
```

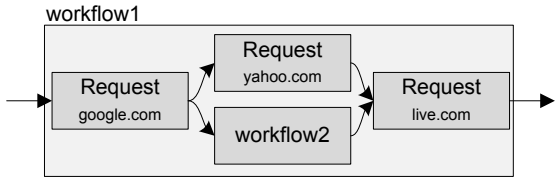
An example illustrating all three effects is given below. Here `x` and `y` are predefined `Name` instances.

```
dynamicGet _ = get x

exampleAll =
    dlet x 1 $
    dlet y 1 $
    do v1 <- dynamicGet ()      -- v1 is 1
       dlet x 2 $
        do v2 <- dynamicGet ()  -- v2 is 2
           set x 3
           v3 <- get x          -- v3 is 3, v4 is 1
           v4 <- inspect y dynamicGet
           return ()
```

3. Asynchronous Workflows

Workflows are units of computation that may do I/O, communicate with other workflows, and possibly invoke or spawn workflows. Our example here is an adaption of a library for web service orchestration by Wu and Gannon [LG08]. In the original example, basic WS-BPEL constructs are implemented using *yield* in C#. Our implementation eliminates several globals needed by the original C# version that were used to send and receive values from *iterators*. Here is a simplified "meta" search-engine workflow:




```
workflow1 :: AsyncProc ()
workflow1 =
 do v1 <- webRequest "www.google.com"
    writeDB (v1 ++ " from google")
    (v2, v3) <- parallel $
       (webRequest "www.yahoo.com", workflow2)
    writeDB (v2 ++ " from yahoo")
    writeDB (v3 ++ " from workflow2")
    v4 <- webRequest "www.live.com"
    writeDB (v4 ++ " from live")
    return ()
```

Despite having asynchronous I/O and the ability to spawn parallel workflows, the code retains a simple sequential structure. We model `AsyncProc` computations as *iterators* which *yield* when they execute a blocking call, thus delegating the blocking semantics of the operations to some top-level scheduler. The scheduler that receives I/O requests and suspended *iterators* can evaluate them in any desired order.

```
type AsyncProc a = Yield OperationResult Operation a
data Operation =
  WebRequest String | WriteDb ...
data OperationResult =
  WebResult String | DbResult ...

writeDB  :: String -> AsyncProc Bool
parallel :: (AsyncProc a,AsyncProc b)->AsyncProc (a,b)
webRequest :: String -> AsyncProc String
webRequest url =
  do WebResult r <- yield (WebRequest url)
     return r

roundRobinEngine :: [AsyncProc ()] -> IO ()
```

In essence, this implementation abstracts an effect using a delimited control operation and delegating its interpretation to a top-level.

4. Monad Transformer for *yield*

We provide a monad transformer for *yield* that allows *yield* to be composed with other effects. The implementation is a straightforward extension of our previous continuation-passing style implementation [JS11].

```
data IteratorT i o m r
  = Done r
  | Susp o (i -> m (IteratorT i o m r))
data YieldT i o m r = YieldT { unYT :: (forall b .
  (r -> m (IteratorT i o m b)) -> m (IteratorT i o m b)) }

instance Monad m => Monad (YieldT i o m)  where
  return a = YieldT (\k -> k a)
  (>>=) (YieldT ma) f=YieldT (\k->ma (\v->(unYT (f v)) k))
instance MonadTrans (YieldT i o) where
  lift m = YieldT (\k -> m >>= k)
```

We can now define *yield* and the `YieldT` version of *run* as:

```
yield :: (Monad m) => o -> YieldT i o m i
yield o = YieldT (\k -> return (Susp o k))

runYieldT :: Monad m=>YieldT i o m r-> m (IteratorT i o m r)
runYieldT (YieldT it) = it (return . Done)
```

5. Answer Type Polymorphism

Communicating sub-processes called *iterators* are the basic abstraction provided by *yield*. Session types have been proposed as means of describing the input-output interaction of concurrent processes [GVR02, Hon93b]. When viewed as session types, *iterators* with the simple type system for *yield* which have fixed input and output types, are trivial process descriptions.

From previous work [JS11], we know that *yield-run* and *shift-reset* can macro express each other. Correspondingly, when encoded using *yield*, the types for *shift-reset* and also restrictive – they result in a fixed answer type and a fixed continuation argument type:

```
type SR ans r = Yield In (Out ans) r
data In a = In { unIn :: a }
data Out ans a = Out (a -> ans) -> SR ans ans

shift :: ((a -> ans) -> SR ans ans)->SR ans a
reset :: SR ans ans -> ans
```

However, if we grant *yield* a more expressive type system as in the parametric types discussed in Section 3.1 of our previous paper [JS11], it results in a more interesting session type for the corresponding *iterator* and in a more expressive monadic type for *shift-reset*.

```
type SR i ans r = Yield i ((i -> ans) -> C i ans ans) r
data C i ans r = C { unC :: SR i ans r }

shift::((a -> ans) -> SR a ans ans) -> SR a ans a
reset :: SR a ans ans -> ans
```

This suggests that typing for *yield iterators* is intimately connected to the seemingly disparate areas of session types and answer type polymorphism and begs the question, what is the full relationship between these? If we adopt a richer type system for *iterators*, does that help relate these areas of research?

References

[Fee03] Marc Feeley. Srfi 39, 2003.

[GVR02] S. Gay, V. Vasconcelos, and A. Ravara. Session types for inter-process communication. *Tech Report, Dept. of Computer Science, Univ. of Glasgow*, 2002.

[Hon93a] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer Berlin / Heidelberg, 1993.

[Hon93b] Kohei Honda. Types for dyadic interaction. In *CONCUR'93*, volume 715 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1993.

[JS11] Roshan P. James and Amr Sabry. Yield : Mainstream delimited continuations. In *TPDC*, 2011.

[KcSS06] Oleg Kiselyov, Chung chieh Shan, and Amr Sabry. Delimited dynamic binding. *SIGPLAN Not.*, 2006.

[LG08] Wei Lu and D. Gannon. A Library for Asynchronous Concurrent Service Orchestration. In *eScience, 2008. eScience'08. IEEE Fourth International Conference on*, pages 230–237, 2008.

[Mor98] L. Moreau. A Syntactic Theory of Dynamic Binding. *Higher-Order and Symbolic Computation*, 11(3):233–279, 1998.

Correctness of Functions with Shift and Reset

Noriko Hirota Kenichi Asai
Department of Information Science, Ochanomizu University
{hirota.noriko, asai}@is.ocha.ac.jp

1 Introduction

Although shift and reset have become used to write various interesting functions, the understanding of those functions is not always simple. As an attempt to better understand their behavior, we formalize and prove correct some functions written with shift and reset in Coq.

The traditional way to prove correctness of those functions is to first remove shift and reset via CPS transformation. However, this method does not give us insight into how to reason about functions written with shift and reset. Instead, we formalize Kameyama and Hasegawa's axioms for shift and reset [1] in Coq and use them to prove correctness of functions directly. Slightly refining on Sozeau and Kiselyov's formalization of shift and reset using Generalized Continuation Monad (GCM for short) [2], we write functions in monadic style. They are proven correct using Kameyama and Hasegawa's axiom and the standard monad laws. By carefully *not* unfolding the definition of monadic operators, we can effectively prove correctness of functions in direct style. In this abstract, we report on two case studies of this approach: `reverse` and `times`, and mention that non-trivial generalization of hypothesis is required to properly characterize the behavior of continuations.

2 Formalization of Axioms

Following Sozeau and Kiselyov's formalization, we write functions with shift and reset in (generalized) monadic style. The monadic type `Cont I O A` represents a type of an expression `e` where the type of `e` is `A` but evaluation of `e` changes the answer type from `I` to `O`. Internally, it is defined as `(A -> I) -> O`, but we will not expand the definition of `Cont` except for proving Kameyama and Hasegawa's axioms. The monadic operations *bind* $e\ \lambda x.M$ and *return* e are expressed `x <-- e;; M` and `ret e`, respectively.

In order to represent the problem of answer type polymorphism, we have slightly changed the formalization of shift following Kiselyov's suggestion[1] as follows:

```
Definition shift {tau a s b}
  (f : (tau -> a) -> Cont s b s) :
  Cont a b tau :=
  fun k => f k id_fun.
```

The difference is in the type of the argument of `f`. To make the captured continuation `k` answer-type polymorphic, the type of `k` is pure rather than monadic. It means that when `k` is applied, we have to turn the result into the monadic context by inserting *return*. For example, we have to write `ret (k 1)` rather than just `k 1`.

Using these definitions, we can formalize Kameyama and Hasegawa's axioms as lemma in Coq and prove them correct with respect to CPS semantics. For example, the axiom $\langle F[SM]\rangle = \langle M(\lambda x.\langle F[x]\rangle)\rangle$ where x does not occur free in F **(reset-S)** is formalized as follows in Coq:

```
Lemma reset_S : forall {A B C I O}
  (M: (A -> O) -> Cont B C B)
  (F: A -> Cont I O I),
  reset (x <-- shift M;; F x) =
```

[1] Personal communication.

```
  (reset (k <-- ret (fun x =>
                     run (reset (F x)));;
          M k) : Cont a a C).
```

Because the axiom is written in monadic style, the pure context can be simply represented as `F` and thus the captured continuation as `fun a => run (reset (F a))` where the function `run` transforms a monadic type into a pure type.

We provide two useful tactics, `apply-axioms` and `apply-axioms-in`, that automatically simplify a goal expression and a hypothesis, respectively, by applying simple axioms as much as possible. More complex axioms whose eager application would lead to non-termination have to be applied manually.

3 Case Studies

In this section, we prove the correctness of two recursive functions, `reverse` and `times`. Our study here may suggest some general direction for the correctness proof of other recursive functions.

3.1 Reverse

The definition of `reverse` using shift and reset is as follows:

```
Fixpoint rev {A} (lst: list A) :=
  match lst with
| nil => ret (nil:list A)
| (e :: rest) => shift (fun k =>
    y <-- rev rest;;
    a <-- ret (k y);;
    ret (e :: a))
  end.

Definition reverse {A} (lst: list A) :=
  reset (a := list A) (rev lst).
```

We want to prove that `reverse lst` is always equal to the reverse of `lst`.

As usual, we cannot directly prove it by induction on `lst`, because the induction hypothesis is too weak. Instead, we generalize the context of `rev` suitably so that the equation holds for any pure context `k`:

```
Lemma lemma1 : forall {A} (lst: list A)
  (k: list A ->
      Cont (list A) (list A) (list A)),
  reset (k nil) = k nil ->
  reset (a <-- rev lst;; k a) =
  (reset (x <-- k nil;;
          ret (reverse0 lst ++ x))
              : Cont a a (list A)).
```

where `reverse0` calculates the reverse of its argument in a standard way (without using shift and reset). The lemma is then proved by induction on `lst`, using the axioms and monad laws, without unfolding monadic operators. Using this lemma, we can obtain the following theorem by instantiating `k` as an identity continuation, that is, `ret`.

```
Theorem theorem1 : forall {A} (lst: list A),
  reverse lst = ret (reverse0 lst).
```

It is not immediately clear how to generalize an equation to hold for an arbitrary context. In the `reverse` case, we needed to add two things. The first one appears to be natural: `k` should be pure (when applied to `nil`). The second one is more complicated: we need to consider how the delimited context is used and embed it into the equation.

In the `reverse` case, the generalized `lemma1` has been derived by considering the following four steps:

(i) We start by trying to prove the main theorem (`theorem1`). We simplify the inductive case by applying `apply-axioms`. For `theorem1`, we obtain the following goal:

```
reset (y <-- rev lst;;
       a0 <-- ret y;;
       ret (a :: a0)) =
ret (reverse0 lst ++ a :: nil)
```

(ii) By comparing the current goal with the original theorem, we identify the context in which recursion occurs. In the current example, the recursion `rev lst` occurs in the context `a0 <-- [];; ret (a :: a0)`.

(iii) We generalize the context as a variable `k`, and use it to obtain the generalized lemma. In our case, we notice that passing `nil` to the above context yields almost what we want in the right-hand side, leading to the main sentence of `lemma1`.

(iv) We prove the generalized lemma. During the proof, we may find that additional condition on the context is needed. In our case, we need the purity of `k`. We add such condition to the lemma.

3.2 Times

The definition of `times`, using shift to return 0 immediately when 0 is found in the argument, is as follows:

```
Fixpoint time (lst: list nat) :=
  match lst with
| nil => ret 1
| 0 :: rest => shift (fun k => ret 0)
| a :: rest => x <-- time rest;; ret (a * x)
  end.

Definition times {A} (lst: list A) :=
  reset (time lst).
```

To prove that this definition actually calculates the product of the argument list, we again need to generalize the equation to hold for contexts other than the identity context:

```
Lemma lemma2 : forall (lst : list nat)
  (k : nat -> Cont nat nat nat),
  reset (k 0) = ret 0 ->
  reset (x <-- time lst;; k x) =
  reset (k (times0 lst)).
```

This time, we needed not only to generalize the context but also to add a condition that `reset (k 0)` evaluates to `ret 0`. We could derive `lemma2` using the same steps as `lemma1`. Once we come up with this lemma, it is straightforward to prove it by induction on `lst`. However, we again needed to devise a nontrivial lemma from the theorem we want to prove.

Using the lemma, we can obtain the following theorem by instantiating `k` as an identity continuation.

```
Theorem theorem2 : forall (lst : list nat),
  times lst = ret (times0 lst).
```

References

[1] Kameyama, Y., and M. Hasegawa "A Sound and Complete Axiomatization of Delimited Continuations," *ICFP'03*, pp. 177–188 (September 2003).

[2] Sozeau, M., and O. Kiselyov "The Proved Program of The Month - Type-safe printf via delimited continuations," http://mattam.org/repos/coq/misc/shiftreset/GenuineShiftReset.html (January 2008).

The Limit of the CPS Hierarchy

Josef Svenningsson

Chalmers University of Technology
josef.svenningsson@chalmers.se

Abstract. When encoding several control feature of a programming language in term of continuations one needs several levels of continuations so as to not mix up the control features. Allowing multiple levels of continuations gives rise to the CPS hierarchy which is a sequence of languages with an increasing number of continuations. However, each language only allows for a fixed number of continuations.
We present a language which we refer to as *the limit of the CPS hierarchy*. It allows for an unbounded number of levels of continuations. We present a semantics in the form of an abstract machine.

1 Introduction

Continuations is one of the most powerful tools in the toolbox of programming language semantics. They can be used to encode numerous control features such as concurrency and backtracking search, not to mention arbitrary jumps. However, care is required when using two or more such encodings for language features. It is important that these features not use the same continuations or they will interact in very unpredictable ways. A solution to this problem is to have multiple levels of continuations so that each language feature can be correctly encoded without interfering the others.

Allowing multiple continuations give rise the *CPS hierarchy*[DF90]. Specifically, the CPS hierarchy consists of a sequence of languages with an increasing number of continuations. Each language in the hierarchy can be given a semantics in terms of its predecessor by means of CPS conversion. Furthermore, each language is equipped with control operators, one set for each level of continuation (in [DF90] they use shift and reset and we will also stick with those for the remainder of this paper). Thus each language in the hierarchy forms a kind of meta language which allows a certain number of control features to be encoded into it.

The CPS hierarchy pose a problem though. Which language in the CPS hierarchy should we choose to work with? How many continuations will we need at most? As always, any choice apart from zero, one and infinity would be an arbitrary choice. Clearly infinity is the choice we would like to make but each language in the CPS hierarchy only allows a fixed number of continuations.

In this paper we present a language which supports an unbounded number of continuations. We refer to this language as *the limit of the CPS hierarchy* because it includes all the languages in the hierarchy. When considering the CPS

hierarchy it is not at all clear that it is possible to define such a language, because a straight forward attempt would require an infinite number of CPS conversion and an infinite number of continuations in the denotational semantics. The key to making this work is to use an operational semantics which introduces new levels of continuations as they are needed.

We will assume familiarity with the CPS hierarchy. We recommend the papers [DF90,BBD05] as a good background to the present paper.

2 An abstract machine for the CPS hierarchy

In this section we will present the syntax and semantics of the limit of the cps hierarchy. It should come as no surprise that we cannot give the semantics in the form of iterated cps conversion, that would require an infinite number of conversions and continuations. Instead we will give the semantics in the form of an abstract machine.

Our presentation is heavily influenced by the abstract machines developed for the CPS hierarchy in [BBD05].

2.1 Syntax

First of all, the way we specify the syntax of our language is somewhat nonstandard. We will use indexed syntactic categories, which varies depending on the index. Variables ranging over the categories will have a subscript which indicates which index in the category it belongs to. This should not be confused with the praxis of using subscripts to denote different variables ranging over the same category.

$$
\begin{array}{ll}
Value \ni V & ::= n \mid \lambda x.N \mid ctxt_n \\
Term \ni N, M & ::= v \mid x \mid N\ M \mid \mathrm{succ}\ N \mid \mathrm{shift}_n\ N \mid \mathrm{reset}_n\ N \\
Context_n \ni ctxt_n & ::= ctxtcont_n ctxt_n | \circ \\
ContextCont_0 \ni ctxtcont_0 & ::= [\cdot]N \mid V[\cdot] \mid \mathrm{succ} \\
ContextCont_{m+1} \ni ctxtcont_{n+1} & ::= ctxtlist_m \\
ContextList_0 \ni ctxtlist_0 & ::= ctxt_0 \\
ContextList_{m+1} \ni ctxtlist_{m+1} & ::= ctxt_{m+1} ctxtlist_m \\
ContextStack_n \ni ctxtstack_n & ::= ctxt_n ctxtstack_{n+1} \mid \bullet
\end{array}
$$

The starting point of our language is the lambda calculus with natural numbers. The first oddity is that a context is counted as a value. The reason for this is that the semantics does not reify a continuation to a function, they are just kept as

they are but can still be applied to values to invoke them. The control operators shift and reset are indexed by which level of continuations they operate on.

The notion of context is what we normally think of as a stack. However, it only contains stack-like elements at index 0. At higher indices it will contain lists of stacks representing the continuations.

The $ContextStack_n$ is really a stack of stacks of continuations. It is the key component in this abstract machine as it allows for the hierarchy of continuations. It is grown as needed by the abstract machine.

2.2 Machine states

The abstract machine which we will present has a number of states which are presented as tuples. Here we describe what these states look like and as also present some guiding intuition about their rôle in the abstract machine.

EVAL	$\langle Term, ContextStack_0 \rangle_{\text{eval}}$ The EVAL state takes care of ordinary evaluation of lambda terms using a call-by-value reduction strategy.
CONT	$\langle n \in Nat, Context_n, Value, ContexStack_{n+1} \rangle_{\text{cont}}$ The CONT state goes through the context stack in search for the next continuation to invoke on the value which was computed in the EVAL state.
CONT0	$\langle Context_0, Value, ContextStack_1 \rangle_{\text{cont0}}$ CONT0 takes care of the actual reductions in the abstract machine. It applies the current continuation to the recently computed value and transfers the control to the appropriate state.

The CONT0 state is not strictly necessary but it factors out a well defined task from the already complex transitions in the Cont state.

2.3 The Abstract Machine

The abstract machine is presented in figure 1. The first transition rule takes care of starting the evaluation in the eval state. The rules for application and the successor function are straightforward, they push the appropriate continuation on the stack and proceeds with the evaluation. Once the evaluation has reached a value the cont state is invoked to find the next continuation ready to accept the value and proceed with the evaluation. Evaluating shift and reset requires a bit of stack fiddling which is handled by the primitive functions shiftStack and resetStack.

The CONT state searches though the stack of stacks to find the next stack ready for proceeding with the evaluation. The first rule matches if the stack we are currently looking at is empty and we have to take the next stack in the stack. The second rule transfers control to the CONT0 state when we have found a continuation of level 0. The third rule takes care of the situation when we have found a continuation higher up in the hierarchy. It needs some massaging to turn it into level zero continuation. The last rule in the CONT state terminates the evaluation when we have come to the end of the stack of stacks.

$$t \Rightarrow \langle t, \circ\bullet\rangle_{\text{eval}}$$

$$\begin{array}{r}
\langle v, ctxt_0 ctxtstack_1\rangle_{\text{eval}} \Rightarrow \langle 0, ctxt_0, v, ctxtstack_1\rangle_{\text{cont}} \\
\langle N\ M, ctxt_0 ctxtstack_1\rangle_{\text{eval}} \Rightarrow \langle N, (([\cdot]M)ctxt_0) ctxtstack_1\rangle_{\text{eval}} \\
\langle succ\ N, ctxt_0 ctxtstack_1\rangle_{\text{eval}} \Rightarrow \langle N, ((succ\ [\cdot])ctxt_0) ctxtstack_1\rangle_{eval} \\
\langle shift_n\ x\ N, ctxtstack_0\rangle_{\text{eval}} \Rightarrow \langle N', ctxtstack'_0\rangle_{eval} \\
\text{where } (ctxt_n, ctxtstack_{n+1}) = shiftStack\ n\ ctxtstack_0 \\
\text{and } N' = [x := ctxt_n]N \\
\text{and } ctxtstack'_0 = pad\ ctxtstack_{n+1} \\
\langle reset_n\ N, ctxtstack_0\rangle_{\text{eval}} \Rightarrow \langle N, ctxtstack'_0\rangle_{eval} \\
\text{where } ctxtstack'_0 = resetStack\ n\ ctxtstack_0
\end{array}$$

$$\begin{array}{r}
\langle n, \circ, v, ctxt_{n+1} ctxtstack_{n+2}\rangle_{\text{cont}} \Rightarrow \langle n+1, ctxt_{n+1}, v, ctxtstack_{n+2}\rangle_{\text{cont}} \\
\langle 0, ctxtcont_0 ctxt_0, v, ctxtstack_1\rangle_{\text{cont}} \Rightarrow \langle ctxtcont_0, v, ctxt_0, ctxtstack_1\rangle_{\text{cont0}} \\
\langle n, ctxtcont_n ctxt_n, v, ctxtstack_{n+1}\rangle_{\text{cont}} \Rightarrow \langle 0, ctxt'_n, v, ctxtstack'_{n+1}\rangle_{\text{cont}} \\
\text{where } ctxt'_n = getZeroContextL\ ctxtcont_n \\
\text{and } ctxtstack'_{n+1} = stack\ n\ (ctxt_n ctxtcont_n)\ ctxtstack_{n+1} \\
\langle n, ctxt_n, v, \bullet\rangle_{\text{cont}} \Rightarrow v
\end{array}$$

$$\begin{array}{r}
\langle [\cdot]V, v, ctxt_0, ctxtstack_1\rangle_{cont0} \Rightarrow \langle V, ((v[\cdot])ctxt_0) ctxtstack_1\rangle_{eval} \\
\langle (\lambda x.N)[\cdot], v, ctxt_0, ctxtstack_1\rangle_{cont0} \Rightarrow \langle [x := v]N, ctxt_0 ctxtstack_1\rangle_{eval} \\
\langle (ctxt_n)[\cdot], v, ctxt_0, ctxtstack_1\rangle_{cont0} \Rightarrow \langle 0, ctxt_0, v, ctxtstack'_1\rangle_{cont} \\
\text{where } (ctxt_0 ctxtstack'_1) = unwind\ ctxt_n (resetStack\ ctxtstack_1\ (n+1))
\end{array}$$

Fig. 1. The abstract machine

As mentioned previous CONT0 administers the reductions in the abstract machine. The first two rules should be familiar from the standard reductions in the lambda calculus. The third rule takes care of applying contexts. The function unwind sprinkles the contents of the continuation over the stack of stacks to restore the control state saved by the context.

2.4 Primitive functions

The abstract machine relies several primitive functions, such as pad and resetStack, which we present separately in figure 2. They are treated as primitives in the same sense as substitution is taken as a primitive in many abstract machines. We could have chosen to model these functions as more machine states with associated reductions in the abstract machine but we feel that this presentation is clearer. Each function will be explained in turn.

resetStack is the workhorse for executing the reset operator. It takes the current context and places it at the right level in the context stack, using the functions **pushContext** and **contextToList** to deal with some of the administration.

shiftStack is used to give semantics to the shift operator. It splits the context stack in two depending on what level the shift operator works on. However, it might be that the context stack is not deep enough because continuations of the level used by the present shift operator hasn't been used before. Therefore the **build** function is used which extends the context stack by adding empty contexts as needed. This is the key step in the abstract machine which makes

it possible at all possible to define the limit of the cps hierarchy. The function **pad** is also used in the semantics of shift to pad the context stack with empty contexts to make it a valid zero level context stack.

unwind deals with the case when a continuation is invoked. It installs the corresponding context at the right place in the context stack with the help of **unwindList**.

Finally, **getZeroContextL** and **stack** is used to find the next context when a value has been computed. The function getZeroContextL is used to retrieve the new zeroth-level context which will be used to reduce the current value while stack realigns the context stack.

Many of the primitive functions searches through the context stack and have a linear time complexity in its size. This means that the transitions in the abstract machine are not constant time which one usually expects. However, if we consider the continuation level n as fixed for the control operators shift and reset then the transitions are indeed constant time. The operations will be more expensive as we use more continuations but the cost will always be bounded. Furthermore, in an implementation of this abstract machine some operations could perhaps be implemented more efficiently by indexing into the contexts and the context stack instead of linearly searching through them.

3 Conclusion and Future work

It is possible to define a language with an unbounded number of control levels which supersedes the whole CPS hierarchy. The semantics is specified using an abstract machine which introduces new levels of continuations as they are needed.

The abstract machine as it is currently specified is very complex and uses many primitive functions. We find this complexity rather unsatisfactory and hope that there are ways to simplify the presentation.

The abstract machine presented in this paper is in desperate need of a correctness proof given how complicated this is. We imagine a theorem stating that every language in the CPS hierarchy can be faithfully evaluated by the abstract machine.

It should be possible, although tedious, to derive a direct-style evaluator for our language by using the syntactic correspondence investigated by Danvy et al. [ABDM03,BD07].

Another interesting thing would be to develop a type system for our language. It should follow rather straightforwardly from Danvy and Filinski's work [DF90] on typing the CPS hierarchy.

Having an unlimited number of continuation levels gives a new interesting possibility. It would be possible to let the indices on the control operators be dynamic. In this paper the control operators have all come with a fixed number which statically fixes which level of continuation they work on. But with the semantics we have given there is really nothing stopping us from making this number an argument which is computed at runtime. This would mean that

the programmer could dynamically choose what control feature to use in any particular situation. On the other hand we're not sure whether this would be a good idea in practice. Programming with continuations is difficult as it is.

References

[ABDM03] M.S. Ager, D. Biernacki, O. Danvy, and J. Midtgaard. A functional correspondence between evaluators and abstract machines. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming*, pages 8–19. ACM, 2003.

[BBD05] Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. An operational foundation for delimited continuations in the cps hierarchy. *Logical Methods in Computer Science*, 1(2), November 2005.

[BD07] M. Biernacka and O. Danvy. A syntactic correspondence between context-sensitive calculi and abstract machines. *Theoretical Computer Science*, 375(1-3):76–108, 2007.

[DF90] O. Danvy and A. Filinski. Abstracting control. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 151–160. ACM, 1990.

$$
\begin{array}{ll}
pad \;:\; ContextStack_n \to ContextStack_0 & \\
pad\; ctxtstack_0 & = ctxtstack_0 \\
pad\; ctxtstack_n & = pad\; (\circ\; ctxtstack_n)
\end{array}
$$

$$
\begin{array}{ll}
resetStack \;:\; ContextStack_0 \to (n : Nat) \to ContextStack_n & \\
resetStack\; ctxtstack_0\; 0 & = ctxtstack_0 \\
resetStack\; ctxtstack_0 (n+1) & = pushContext\; ctxt\; ctxtstack_{n+1} \\
\qquad\qquad where\; (ctxt\; ctxtstack_n) & = resetStack\; ctxtstack_0\; n
\end{array}
$$

$$
\begin{array}{ll}
pushContext \;:\; Context_n \to ContextStack_{n+1} \to ContextStack_{n+1} & \\
pushContext\; ctxt_n\; (ctxt_{n+1} ctxtstack_{n+2}) & = ((contextToList\; n\; ctxt_n) ctxt_{n+1}) ctxtstack_{n+2}
\end{array}
$$

$$
\begin{array}{ll}
contextToList \;:\; Context_n \to ContextList_n & \\
contextToList\; ctxt_0 & = ctxt_0 \\
contextToList\; (ctxtlist_n ctxt_{n+1}) & = ctxt_{n+1}\; ctxtlist_n
\end{array}
$$

$$
\begin{array}{ll}
shiftStack \;:\; (n : Nat) \to ContextStack_0 \to (Context_n, ContextStack_{n+1}) & \\
shiftStack\; 0\; (ctxt_0 ctxtstack_1) & = (ctxt_0, ctxtstack_1) \\
shiftStack\; (n+1)\; ctxtstack_0 & = build\; ctxt_n\; ctxtstack_{n+1} \\
\qquad\qquad where\; (ctxt_n, ctxtstack_{n+1}) & = shiftStack\; n\; ctxtstack_0
\end{array}
$$

$$
\begin{array}{ll}
build \;:\; Context_n \to ContextStack_{n+1} \to (Context_{n+1}, ContextStack_{n+2}) & \\
build\; ctxt_n\; \bullet & = ((contextToList\; ctxt_n) \circ, \bullet) \\
build\; ctxt_n\; (ctxt_{n+1}\; ctxtstack_{n+2}) & = ((contextToList\; ctxt_n) ctxt_{n+1}, ctxtstack_{n+2})
\end{array}
$$

$$
\begin{array}{ll}
unwind \;:\; Context_n \to ContextStack_{n+1} \to ContextStack_0 & \\
unwind\; ctxt_0 ctxtstack_1 & = ctxt_0 ctxtstack_1 \\
unwind\; (ctxtList_n ctxt_{n+1}) ctxtstack_{n+2} & = unwindList\; ctxtList_n (ctxt_{n+1} ctxtstack_{n+2})
\end{array}
$$

$$
\begin{array}{ll}
unwindList \;:\; ContextList_n \to ContextStack_{n+1} \to ContextStack_0 & \\
unwindList\; ctxt_0\; ctxtstack_1 & = ctxt_0 ctxtstack_1 \\
unwindList\; (ctxt_{n+1} ctxtList_n)\; ctxtstack_{n+2} & = unwindList\; ctxtList_n (ctxt_{n+1} ctxtstack_{n+2})
\end{array}
$$

$$
\begin{array}{ll}
getZeroContextL \;:\; ContextList_n \to ContextList_0 & \\
getZeroContextL\; ctxt_0 & = ctxt_0 \\
getZeroContextL\; (ctxt_{n+1} ctxtList_n) & = getZeroContextL\; ctxtList_n
\end{array}
$$

$$
\begin{array}{ll}
stack \;:\; ContextList_n \to ContextStack_{n+1} \to ContextStack_{n+1} & \\
stack\; ctxtList_0\; ctxtstack_1 & = ctxtstack_1 \\
stack\; (ctxt_{n+1}\; ctxtList_n) ctxtstack_{n+2} & = stack\; ctxtList_n\; (ctxt_{n+1}\; ctxtstack_{n+2})
\end{array}
$$

Fig. 2. Primitive functions used in the abstract machine

Non-Deterministic Search Library

Kenichi Asai Chihiro Kaneko
Ochanomizu University
{asai,kaneko.chihiro}@is.ocha.ac.jp

1 Introduction

Non-deterministic programming has been used as a non-trivial application of (delimited) continuations.In particular, in the presence of first-class delimited continuation constructs, such as `shift` and `reset` [1], it becomes possible to write a back-tracking program without converting the program into continuation-passing style (CPS). However, the lack of serious support for first-class delimited continuation constructs (especially in a typed setting) prevented us from writing such applications. In this abstract, we report on our experience of using Caml Shift [2], an extension of Caml Light with (polymorphically typed) `shift` and `reset`, to write a search problem using non-deterministic operators. We provide a library for non-deterministic operations implemented using `shift` and `reset` and show how it enables us to write a search problem in direct style.

2 Interface

The interface of the non-deterministic search library we have implemented is shown in Figure 1. The library consists of five search algorithms: depth first, breadth first, best first, the one that collects all the results, and the one for game search implementing the alpha/beta pruning.

A search process is initiated by applying `start` to a thunk. In the thunk, one can use two non-deterministic constructs, `fail` and `choice`. The former aborts the current search and the latter chooses an element non-deterministically from its argument. There are two kinds of function types: `->` and `=>`. The former denotes a type of standard pure functions. The latter denotes a type of functions that incurs control effects and thus their answer types are not polymorphic. We omit the interface for collecting all the solutions. It is identical to the depth/breadth-first search except for `start` which has type `(unit => 'a) -> 'a list` because it returns all the possible results.

Depth/breadth-first search:

```
start  : (unit => 'a) -> 'a
fail   : unit => 'a
choice : 'a list => 'a
```

Best-first search:

```
start  : (unit => 'a) -> 'a
fail   : unit => 'a
choice : 'a list -> 'b => 'a
```

Alpha-beta search:

```
start      : (unit => 'a) -> 'a
return     : 'a -> int => 'b
choice_max : 'a list => 'a
choice_min : 'a list => 'a
```

Figure 1: Search library interface

Note that the impure types without answer types still describe the library functions informatively. One can forget about answer types to use this library.

3 Implementation Overview

Basically, `start` is implemented as `reset`, `fail` is abort, i.e., `shift (fun k -> ())`, and `choice` is realized by first capturing the current continuation and then applying it to all the possible choices. However, details differ for each kind of search. In particular, all the cases pass a state to maintain a search process by making the context higher-order (effectively implementing the store monad). In the depth-first search, the remaining choices together with a continuation (stored in a stack) is passed. In the breadth-first search, a queue is used instead of a stack. Best-first search reorders the remaining choices according to the cost. However, the details of the implementation are beyond the scope of this abstract.

	tile 1	tile 2	tile 3
row 1	`[M]`	`[M]`	`[M; F; M]`
row 2	`[M; F; F; F]`	`[F; M]`	`[F]`

Figure 2: A party puzzle

4 Application

Using the search library, we have solved party puzzles[1]. We are given three tiles, each consisting of two sequences of people of mixed gender (male `M` or female `F`). Figure 2 shows an example. The task is to find a sequence of tiles where at each column, we have a matching couple (of `M` and `F`), if all the lists of each row are appended in order. The solution to the puzzle in Figure 2 is [3; 1; 2; 3] giving the following two rows:

row 1: `[M; F; M]@[M]@[M]@[M; F; M]`
row 2: `[F]@[M; F; F; F]@[F; M]@[F]`

On the web page, we are given five such puzzles.

Breadth-first program. It is immediately clear that the depth-first search is not feasible for this puzzle. Using the library for the breadth-first search, we can write the following program.

```
(* search : state_t => int list => int list *)
let rec search state solution =
  let num = choice [1; 2; 3] in
  let tile = get_tile num in
  let state' = place tile state in
  let solution' = num :: solution in
  if complete state' then solution'
  else search state' solution' ;;
```

Using `choice` to choose the right tile non-deterministically, we can write a search program in a straightforward manner. By calling this function as follows:

```
start (fun () -> search (State ([], [])) [])
```

we can obtain (the reverse of) the solution for the first three puzzles out of five.

Adding history. For the fourth puzzle, the search space is much larger. To eliminate redundant search, we introduce a history mechanism:

```
(* add_state '_a => unit *)
let add_state state =
  if mem state !history then fail ()
  else history := add state !history ;;
```

[1] `http://wonderfl.net/c/wcnb`

Best-first program. Adding history was not enough to solve the fourth puzzle. Breadth-first search led to a partial solution where one row is much longer than the other row. To this end, we switch to best-first search where we take the length of the unmatched people as the cost. The change of the search strategy is easy. We simply replace the library to be included. With these changes, we could obtain the solution to the fourth puzzle.

```
(* search : state_t => int list => int list *)
let rec search state solution =
  add_state state;
  let diff = state_length state in
  let num = choice [1; 2; 3] diff in
  ... ;;
```

Limiting depth. Finally, the last puzzle searches for a result too deeply even if we use the best-first search. To solve the last puzzle, we needed to limit the depth of the search as follows:

```
(* search : state_t => int list => int =>
            int list *)
let rec search state solution n =
  add_state state;
  if n > 600 then fail () else
  ...
  if complete state' then solution'
  else search state' solution' (n + 1) ;;
```

5 Conclusion

This abstract reported on our experience of using non-deterministic search library to solve party puzzles. Once a proper library is provided, writing a search problem becomes much easier: we are not cluttered by the control flow, but can concentrate on the search process itself. We can switch search strategy by replacing the library. We can also include various kinds of pruning. We will further investigate the usefulness of the library in the future.

References

[1] Danvy, O., and A. Filinski "Abstracting Control," *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pp. 151–160 (June 1990).

[2] Masuko, M., and K. Asai "Caml Light + shift/reset = Caml Shift," *Theory and Practice of Delimited Continuations (TPDC 2011)*, pp. 33–46 (May 2011).