Technical Report No. 545

Proceedings of the Third ACM SIGPLAN Workshop on Continuations (CW'01)

(January 16, 2001 — Royal Society, London.)

by

Amr Sabry (editor)

December 2000

Computer Science Department
Indiana University
Bloomington, Indiana 47405-4101

Preface

This report constitutes the proceedings of the third ACM SIGPLAN workshop on Continuations (CW'01) held on January 16, 2001, at the Royal Society in London, England. The workshop was held as a satellite event of the 28th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL).

The notion of continuations is ubiquitous in many different areas of computer science, including category theory, compilers, logic, operating systems, programming, and semantics. Following on the 1992 and 1997 ACM SIGPLAN Workshops on Continuations, the current workshop provides a forum for the presentation and discussion of new results and work in progress aimed at a better understanding of the nature of continuations, the relation of continuations to other areas of logic and computer science, and exciting new applications of continuations in contexts such as mobile threads, simulation, distributed systems, graphical user interfaces, and education.

The program includes seven technical papers. Christopher Wadsworth (the originator of the term "continuation") and Suresh Jagannathan (at NEC Research and Emphora, Inc.) kindly agreed to deliver invited talks at CW'01.

Review Process

A call for papers was announced on several mailing lists and newsgroups. Thirteen submissions were received and forty six reviews were written. Each paper was "owned" by one committee member who was responsible for summarizing all the reviews and making a recommendation to the rest of the committee. The final decisions were made collectively by the program committee on the basis of the collected reviews. The discussions among the committee members were conducted electronically and lasted for about a week.

Program Committee

Daniel P. Friedman	Indiana University
John Hatcliff	Kansas State University
Richard Kelsey	NEC Research Institute
Amr Sabry	Indiana University
Olin Shivers	Georgia Institute of Technology
Carolyn Talcott	Stanford University
Hayo Thielecke	University of Birmingham

Acknowledgments

Thanks to Olivier Danvy for valuable suggestions regarding the organization of the workshop. Thanks to the program committee for the careful reviews. And of course we extend our thanks to the participants and especially the invited speakers.

Finally we would also like to thank ACM and SIGPLAN, the POPL chairs, David Schmidt and Chris Hankin, and the local organizers for their help and support.

Contents

1 John Reppy:
Local CPS Conversion in a Direct-Style Compiler 1

2 Jung-taek Kim and Kwangkeun Yi:
Interconnecting Between CPS Terms and Non-CPS Terms 7

3 Hayo Thielecke:
Comparing Control Constructs by Typing Double-barrelled CPS Transforms 17

4 Yukiyoshi Kameyama:
Towards Logical Understanding of Delimited Continuations 27

5 Olivier Danvy and Lasse R. Nielsen:
CPS Transformation of Beta-Redexes 35

6 Andrzej Filinski:
An Extensional CPS Transform (Preliminary Report) 41

7 Josh Berdine, Peter W. O'Hearn, Uday S. Reddy, Hayo Thielecke:
Linearly Used Continuations 47

Index

Berdine, Josh, 47

Danvy, Olivier, 35

Filinski, Andrzej, 41

Kameyama, Yukiyoshi, 27
Kim, Jung-taek, 7

Nielsen, Lasse R., 35

O'Hearn, Peter W., 47

Reddy, Uday S., 47
Reppy, John, 1

Thielecke, Hayo, 17, 47

Yi, Kwangkeun, 7

Local CPS conversion in a direct-style compiler

John Reppy
Bell Labs, Lucent Technologies
jhr@research.bell-labs.com

Abstract

Local CPS conversion is a compiler transformation for improving the code generated for nested loops by a direct-style compiler. The transformation consists of a combination of CPS conversion and light-weight closure conversion, which allows the compiler to merge the environments of nested recursive functions. This merging, in turn, allows the backend to use a single machine-level procedure to implement the nested loops. Preliminary experiments with the Moby compiler show the potential for significant reductions in loop overhead as a result of Local CPS conversion.

1 Introduction

Most compilers for functional languages use a λ-calculus based intermediate representation (IR) for their optimization phases. The λ-calculus is a good match for this purpose because, on the one hand, it models surface-language features like higher-order functions and lexical scoping, while, on the other hand, it can be transformed into a form that is quite close to the machine model.

To make analysis and optimization more tractable, compilers typically restrict the IR to a subset of the λ-calculus. One such subset is the so-called *direct style* (DS) representation, where terms are normalized so that function arguments are always atomic (*i.e.*, variables and constants) and intermediate results are bound to variables.[1] The DS representation makes the data-flow of the program explicit by binding all intermediate values to variables. Another common representation is *continuation-passing style* (CPS), where function applications are further restricted to occur only in tail positions and function returns are represented explicitly as the tail-application of continuation functions [Ste78, KKR+86, App92]. In CPS, both the data-flow and control-flow of the program is made explicit, which makes it well suited to optimizing the program's control structures.

While there has been some debate over the relative merits of these two approaches [FSDF93, DD00], it is fair to say that both have their advantages and we do not discuss their relative merits. In the end, the choice of IR is an engineering decision that must be made for each compiler. A discussion of this choice is beyond the scope of this paper; instead, we focus on the idea of exploiting CPS representation in a DS-based optimizer. This exploitation is possible because the CPS terms are a subset of the DS terms (*i.e.*, CPS $\subset$ DS $\subset$ λ-calculus).

This paper describes a transformation and supporting analysis that exemplifies the idea of exploiting CPS representation in a DS-based optimizer. In the next section, we describe a motivating example. We then describe our transformation and an analysis for detecting when it is applicable in Section 3. This transformation should be useful for any DS-based optimizer. We are implementing this transformation in our compiler for the MOBY programming language [FR99] and we present a preliminary indication of its usefulness in Section 4.[2] We discuss related work in Section 5 and then conclude.

2 The problem

It is well known that loops can be represented as tail-recursive functions and many compilers for functional languages use tail recursion to represent loops in their IR. By treating tail-recursive function calls as "gotos with arguments," a compiler can generate code for a loop that is comparable to that generated by a compiler for an imperative language. But when loops are nested, generating efficient code becomes more difficult. For example, consider the following C-code:

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    f (i, j);
```

This kind of nested loop structure is found in many algorithms (*e.g.*, matrix multiplication). Translating this code to a sugared DS representation, with the **`for`**-loops replaced by recursion, results in the code given in Figure 1.[3] While the two loop functions, `lp_i` and `lp_j`, are tail-recursive, the call "`lp_j 0`" from the outer loop (`lp_i`) to the inner loop is not tail recursive. If the compiler directly translates the DS representation to machine code, the two loops will occupy separate procedures with separate stack frames. This structure inhibits loop optimizations, register allocation, and scheduling, as well as adding call/return overhead to the outer loop. On the other hand, the CPS representation of this example, given in Figure 2, makes explicit the connection between the return from `lp_j` and the next iteration of `lp_i`. A simple control-flow analysis will show that the return continuation of `lp_j` is always the known function `k5`, which enables compiling the nested loops into a single machine procedure.

[1] There are a number of different direct-style representations: *e.g.*, Flanagan *et al's A-form* [FSDF93], the TIL compiler's *B-form* [TMC+96], and the RML compiler's *SIL* [OT98].

[2] MOBY is a higher-order typed language that combines support for functional, object-oriented, and concurrent programming. See `www.cs.bell-labs.com/~jhr/moby` for more information.

[3] In this paper we use SML-like syntax as a sugared form of DS representation.

```
fun applyf (f, n) = let
      fun lp_i i = if (i < n)
            then let
              fun lp_j j = if (j < n)
                  then (f(i, j); lp_j(j+1))
                  else ()
              in
                lp_j 0; lp_i(i+1)
              end
            else ()
      in
        lp_i 0
      end
```

Figure 1: A nested loop using tail-recursion

```
fun applyf (f, n, k1) = let
      fun lp_i (i, k2) = if (i < n)
            then let
              fun lp_j (j, k3) = if (j < n)
                  then let
                    fun k4 () = lp_j(i+1, k3)
                    in
                      f(i, j, k4)
                    end
                  else k3()
              fun k5 () = lp_i(i+1, k2)
              in
                lp_j (0, k5)
              end
            else k2()
      in
        lp_i (0, k1)
      end
```

Figure 2: The CPS converted `applyF` example

This example suggests that by making the return continuation of `lp_j` explicit, we can replace the call/return of `lp_j` with direct jumps.

3 The solution

The MOBY compiler performs standard optimizations (*e.g.*, contraction, useless-variable elimination, and CSE) using a DS representation we call BOL. Loops are represented using tail recursion in BOL. After optimization, the compiler performs the *closure* phase, which is responsible for converting the nested functions into a collection of top-level functions with no free variables. Following the closure phase is the *frame* phase, which determines which functions can share the same stack frame; a group of functions that share the same frame is called a *cluster*. Our goal is to have nested loops, like the one in Figure 1, translate into a single cluster (as they would in an imperative language like C). Doing so has many performance advantages. It enables better loop optimizations, register allocation, and scheduling. It also eliminates the overhead of creating a closure for the inner loop, the call to the inner loop, and the heap-limit check on return from the inner loop.

The technique we use to achieve this goal is us a *local CPS* (LCPS) conversion to convert the non-tail calls to the inner loop into tail calls. Once the LCPS transformation has been applied, the frame phase is able to group the functions that comprise the nested loop into the same cluster.

To determine where it is useful to apply the transformation, we need some form of control-flow analysis. The property that we are interested in is when a known function has the same return continuation at all of its call-sites. The MOBY compiler uses a simple syntactic analysis to determine this property. For each function defined in the module being compiled, we conservatively estimate the set of return continuations for the function. If the estimated set is a singleton set, then we apply the transformation.

e	$::=$	$l : t$	labeled term
t	$::=$	x	variable
	$\mid$	**fun** $f\,(\vec{x}) = e_1$ **in** e_2	function binding
	$\mid$	**let** $x = e_1$ **in** e_2	let binding
	$\mid$	**if** x **then** e_1 **else** e_2	conditional
	$\mid$	$f\,(\vec{x})$	application

Figure 3: A simple direct-style IR

3.1 Analysis

The analysis computes an approximation of return continuations of each known function, so a standard control-flow analysis is applicable [NNH99]. In this section, we describe a very simple linear-time analysis. This analysis uses a simple notion of *escaping* function — if a function name is mentioned in a non-application rôle, it is regarded as escaping and we define its return continuation to be $\top$.[4]

To describe the analysis, we use the simple DS IR given in Figure 3. As usual, we assume bound variables are unique so we do not have to worry about unintended name capture when transforming code. In this IR, expressions are uniquely labeled terms. We use the labels to represent abstract continuations in the analysis.

Let LABEL be the set of term labels. Then we define an abstract domain $\text{RCONT} = \text{LABEL} \cup \{\bot, \top\}$. We use ρ to denote elements of RCONT. Intuitively, one can think of RCONT as a squashed powerset domain, with $\bot$ for the empty set, l for the singleton set $\{l\}$, and $\top$ for everything else. We define the partial order $\sqsubseteq$ on RCONT, with $\bot \sqsubseteq l \sqsubseteq \top$ for any $l \in \text{LABEL}$, and we define $\rho_1 \sqcup \rho_2$ to be the least upper bound of ρ_1 and ρ_2 under $\sqsubseteq$.

Given an expression and its abstract continuation, the analysis computes a map Γ from variables (*i.e.*, function names) to abstract continuations.

$$\Gamma \in \text{RENV} = \text{VAR} \xrightarrow{\text{fin}} \text{RCONT}$$

We extend Γ to a total function when applying it to a variable by defining $\Gamma(x) = \bot$ for $x \notin \text{dom}(\Gamma)$. We define the *join* of Γ_1 and Γ_2 by

$$\Gamma_1 \uplus \Gamma_2 = \{x \mapsto \Gamma_1(x) \sqcup \Gamma_2(x) \mid x \in \text{dom}(\Gamma_1) \cup \text{dom}(\Gamma_2)\}$$

The analysis itself has the following type:

$$\mathcal{R} : \text{EXP} \to \text{RCONT} \to \text{RENV}$$

With these definitions, we can describe the analysis, which is presented in Figure 4. We map unknown and escaping functions to $\top$, as can be seen in Rules 1, 2, and 5. Rule 2 shows how we analyse function definitions — first we analyse the uses of f in its scope and then we use the result of that analysis as the return

[4] This definition is the one used by Appel [App92] in his CPS-based framework.

$$\mathcal{R}[\![l : x]\!]\rho = \{x \mapsto \top\} \quad (1)$$

$$\mathcal{R}[\![l : \mathbf{fun}\ f\ (\vec{x}) = e_1\ \mathbf{in}\ e_2]\!]\rho = \mathcal{R}[\![e_1]\!]\rho' \uplus \Gamma \uplus \{\vec{x} \mapsto \top\} \quad (2)$$
where $\Gamma = \mathcal{R}[\![e_2]\!]\rho$ and $\rho' = \Gamma(f)$.

$$\mathcal{R}[\![l : \mathbf{let}\ x = e_1\ \mathbf{in}\ l' : t_2]\!]\rho = \mathcal{R}[\![e_1]\!]l' \uplus \mathcal{R}[\![l' : t_2]\!]\rho \uplus \{x \mapsto \top\} \quad (3)$$

$$\mathcal{R}[\![l : \mathbf{if}\ x\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!]\rho = \mathcal{R}[\![e_1]\!]\rho \uplus \mathcal{R}[\![e_2]\!]\rho \quad (4)$$

$$\mathcal{R}[\![l : f\ (\vec{x})]\!]\rho = \{f \mapsto \rho\} \uplus \{\vec{x} \mapsto \top\} \quad (5)$$

Figure 4: The analysis

continuation for the body of f. For **let** bindings, the body of the **let** is the continuation of the binding. The result of the analysis is the join of the sub-analyses. When analysing a function application (Rule 5), we map the applied function to the application's abstract continuation and treat the arguments as escaping. To analyse a complete program, we use $\top$ as the return continuation and define $\textsc{Anal}(e) = \mathcal{R}[\![e]\!]\top$.

This analysis can be extended to handle mutually recursive functions by computing a fixed-point at function bindings. Since such a brute-force approach may prove expensive in practice, a better solution may be to first compute the approximate call-graph and then use the call-graph to guide the analysis.

3.2 The LCPS transformation

If the analysis has determined that all call sites of a function `f` have the same return continuation (*i.e.*, $\textsc{Renv}(\mathtt{f}) = l$), then we apply the LCPS transformation to `f`. Transforming `f` has two parts: we must reify the continuation of `f`, creating an *explicit* continuation function $k_\mathtt{f}$, and we must introduce calls to $k_\mathtt{f}$ at the return sites of `f`. For example, consider the following code fragment:

```
fun f () = ... in ...
  fun g () = ... let y = f() in e
```

Assuming that f is eligible for the LCPS transformation, this fragment is converted to

```
fun f (k) = (... k()) in ...
  fun g () = ... fun k_f (y) = e in f(k_f)
```

Here we have made the return continuation of `f` explicit by modifying `f` to take its continuation as an argument (k), which it calls at its return sites. We have also split the body of `g` to create the explicit representation of `f`'s return continuation ($k_\mathtt{f}$) and have modified the non-tail call site of `f` to pass $k_\mathtt{f}$ to `f`.

To understand how the LCPS transformation works, it is instructive to examine the global CPS conversion. Figure 5 gives the transformation for the simple direct-style IR of Figure 3 (ignoring the labels). For the LCPS transformation, we only want to apply the CPS conversion under certain conditions. Assuming that Γ is the result of analysing the program, let the set of eligible function be defined as $\mathcal{E} = \{f \mid \Gamma(f) \in \textsc{Label}\}$. We then specialize the rules of Figure 5 as follows:

Rule 6: When the expression x is in the tail position of a function $f \in \mathcal{E}$, then we apply the CPS conversion (*i.e.*, we transform the implicit return into an explicit application of the tail continuation k).

Rule 7: When $f \in \mathcal{E}$ we apply the CPS conversion.

Rule 8: When the expression e_1 contains an application of a function $f \in \mathcal{E}$, we apply the CPS transformation.

Rule 9: Since this rule does not transform the expression, there is no modification. Note that we do not have to worry about code duplication, since k is a variable and not a λ-term.

Rule 10: If the function $f \in \mathcal{E}$, then we apply the CPS conversion. In this situation, k will either be an explicit return continuation (introduced by Rule 8) or a return continuation parameter (introduced by Rule 7).

It is interesting to examine what happens when the call-site of `f` is buried in the branch of a conditional. For example, consider the following fragment:

```
let x = if y then (let w = f() in e1) else e2
  in e3
```

Applying LCPS to `f` results in the following code:

```
fun k_if (x) = e3 in
  if y
    then fun k_f (w) = C[[e1]]k_if in f(k_f)
    else C[[e2]]k_if
```

Here we have introduced two continuation functions: $k_\mathtt{if}$ for the join-point following the **if** and $k_\mathtt{f}$ for the return continuation of `f`.

Up to now, we have been passing the return continuation $k_\mathtt{f}$ to `f` as an additional argument (as is standard in CPS conversion), but since our analysis has already told us that `f` has exactly one return continuation, we should specialize the return sites of `f` to call $k_\mathtt{f}$ directly. To do so requires lifting the definition of $k_\mathtt{f}$ up to the binding site of `f`.[5] The difficulty with this lifting is that the return continuation and its functions have different free variables, so we need to *close* the function over those variables that will be out of scope at the destination of the function. While a closure object would be sufficient for this purpose, we chose to augment our transformation with a simple form of *light-weight closure conversion* [SW97], which turns the free variables into function parameters. Having these variables as parameters in the final DS representation means that they will get mapped to machine registers. To illustrate closure conversion, consider the following fragment:

```
fun f () = ... z in
...
  let y = ... in
  ...
    let x = f() in
    let w = x + y in
      w
```

In this code, the return continuation of the call to `f` has `y` as a free variable, so we add `y` to the parameters of `f` and the transformed `f` passes `y` to its return continuation. The result of the transformation is:

[5] We also need to extend the IR to support mutually recursive bindings.

$$\mathcal{C}[\![x]\!]k = k\,(x) \quad (6)$$

$$\mathcal{C}[\![\textbf{fun}\ f\,(\vec{x}) = e_1\ \textbf{in}\ e_2]\!]k = \textbf{fun}\ f\,(k',\ \vec{x}) = \mathcal{C}[\![e_1]\!]k'\ \textbf{in}\ \mathcal{C}[\![e_2]\!]k \quad \text{where } k' \text{ is fresh} \quad (7)$$

$$\mathcal{C}[\![\textbf{let}\ x = e_1\ \textbf{in}\ e_2]\!]k = \textbf{fun}\ k'\,(x) = \mathcal{C}[\![e_2]\!]k\ \textbf{in}\ \mathcal{C}[\![e_1]\!]k' \quad \text{where } k' \text{ is fresh} \quad (8)$$

$$\mathcal{C}[\![\textbf{if}\ x\ \textbf{then}\ e_1\ \textbf{else}\ e_2]\!]k = \textbf{if}\ x\ \textbf{then}\ \mathcal{C}[\![e_1]\!]k\ \textbf{else}\ \mathcal{C}[\![e_2]\!]k \quad (9)$$

$$\mathcal{C}[\![f\,(\vec{x})]\!]k = f\,(k,\ \vec{x}) \quad (10)$$

Figure 5: A global CPS conversion

```
fun applyf (f, n) = let
      fun lp_i (i, f, n) = if (i < n)
            then lp_j (0, i, f, n)
            else ()
      and lp_j (j, i, f, n) =
          if (j < n)
            then (
              f(i, j);
              lp_j(j+1, i, f, n))
            else k (i, f, n)
      and k (i, f, n) =
          lp_i(i+1, f, n)
      in
        lp_i (0, f, n)
      end
```

Figure 6: The `applyF` function after the LCPS transformation

```
fun f (y) = ... k_f(z, y)
and k_f (x, y) = let w = x + y in w
...
  let y = ... in
  ...
    f(y)
```

We define `f` and k_f in the same binding because, in general, they may be mutually recursive. In general, the LCPS transformation migrates the converted function f and its explicit continuation to the binding site of f's *non-tail* caller. By doing so, we guarantee that all these functions will be compiled into a single machine procedure.

Revisiting our original motivating example from Figure 1, the result of the analysis will identify `lp_j` as a candidate for the LCPS transformation (*i.e.*, all of its call sites have the same return continuation). The code resulting from applying the transformation to `lp_j` is given in Figure 6. Notice that light-weight closure conversion has been applied to all of the functions in the body of `applyF`. When translated to the target machine code, the functions `applyF`, `lp_i`, `lp_j`, and `k` will all be in the same cluster and share the same stack frame.

4 Experience

We have written a prototype implementation of the analysis and LCPS transformation for the language of Figure 3. Our implementation is currently organized into the analysis plus four transformation passes. The first pass marks those **`let`** bindings for which Rule 8 applies and the second pass performs the actual CPS transformation guided by the marks.[6] The second pass also determines which functions should share the same binding site. The third pass then computes the free variables of these functions and the fourth pass migrates the function definitions and performs the light-weight closure conversion.

We are in the process of implementing the LCPS transformation in the MOBY compiler. For the most part, this is straightforward adaptation of our prototype, although we need a more sophisticated analysis to handle mutually recursive functions. We are also integrating the third and fourth passes of the transformation with the existing closure phase. This integration has the added benefit that we can use light-weight closure conversion for functions that represent local control-flow (*e.g.*, loops). By moving variables from closures into parameters, we expose them to the register allocator and reduce heap allocation,

To judge the effectiveness of the transformation, we applied it as a source-language transformation to the MOBY version of the `applyF` function (essentially, we compared the MOBY versions of Figure 1 and Figure 6). Running the `applyF` program on the null function with `n` set to 10000, the LCPS transformation results in a 25% reduction in execution time (2.34s vs. 3.11s on a 733MHz PIII). While the performance improvements on real workloads is yet to be determined, these preliminary measurements strongly suggest that the LCPS transformation is a useful tool in a DS optimizer.

5 Related work

Most of the literature about compiler optimizations for strict functional languages uses CPS as a representation. Tarditi's thesis [Tar96] is probably the most detailed description of a DS-based optimizer for strict functional languages, but he does not collapse nested loops. We are not aware of any direct-style compiler that implements the LCPS transformation (the OCAML [Ler00], TIL [TMC+96, Tar96], and RML [OT98] compilers do not). The OCAML system provides **`for`**-loops over integer intervals as a language feature. These loops are preserved in the IR and result in code that is similar to that produced by a C compiler [Ler97], but the OCAML compiler (Version 3.00) does not flatten nested loops when they are expressed using recursion.

Kelsey describes a technique for combining functions in a CPS-based framework [Kel95]. In his framework, he annotates λ abstractions with either *proc*, *cont*, or *jump*, where jump is used to mark control transfers that occur within the same machine procedure. He describes an analysis and transformation for converting a λ_{proc} into a λ_{jump}. The MOBY compiler's frame phase (mentioned in Section 3) performs a similar analysis and transformation when grouping functions into clusters. Since the BOL IR is direct-style,

[6] We use two passes for this part of the transformation because it greatly simplifies the bookkeeping.

we rely on the LCPS transformation to enable the clustering of nested loops in the frame phase.

The MLton compiler is a CPS-based compiler for Standard ML, which uses a transformation called *contification* to group functions into the same machine procedure. This transformation very similar to Kelsey's approach, but it has not been described in the literature.

Kim, Yi, and Danvy have used *selective CPS transformation* as a technique for replacing SML's exception raising and handling mechanisms with continuation operations [KYD98]. Unlike LCPS, their transformation does not move code or do closure conversion. Selective CPS transformation is another example of using the CPS representation in a direct-style compiler (although their experiments were done using SML/NJ for their backend, which is a CPS-based compiler).

6 Conclusion

We have presented a local CPS transformation that can be used in a direct-style compiler to improve the performance of nested loops. Preliminary measurements show a 25% reduction in loop overhead for a simple nested loop. While we have only presented fairly simple examples, the LCPS transformation can handle complicated looping structures, such as multiple inner loops and loops expressed as mutually recursive functions (*e.g.*, as you would get when encoding state machines). The LCPS transformation is one example of exploiting the advantages of CPS in a direct-style compiler; we plan to explore other opportunities for exploiting CPS in our compiler.

Acknowledgements

Kathleen Fisher, Lal George, and Jon Riecke provided useful comments on drafts of this paper. Stephen Weeks provided a detailed description of the MLton contification analysis and transformation.

References

[App92] Appel, A. W. *Compiling with Continuations*. Cambridge University Press, New York, N.Y., 1992.

[DD00] Damian, D. and O. Danvy. Syntactic accidents in program analysis: On the impact of the CPS transformation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, September 2000, pp. 209–220.

[FR99] Fisher, K. and J. Reppy. The design of a class mechanism for Moby. In *Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation*, May 1999, pp. 37–49.

[FSDF93] Flanagan, C., A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, June 1993, pp. 237–247.

[Kel95] Kelsey, R. A. A correspondence between continuation passing style and static single assignment form. In *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations*, January 1995, pp. 13–22.

[KKR+86] Kranz, D., R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. Orbit: An optimizing compiler for Scheme. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*, July 1986, pp. 219–233.

[KYD98] Kim, J., K. Yi, and O. Danvy. Assessing the overhead of ML exceptions by selective CPS transformation. In *Proceedings of the 1998 ACM SIGPLAN Workshop on ML*, September 1998, pp. 103–114.

[Ler97] Leroy, X. The effectiveness of type-based unboxing. In *Workshop Types in Compilation '97*. Technical report BCCS-97-03, Boston College, Computer Science Department, June 1997.

[Ler00] Leroy, X. *The Objective Caml System (release 3.00)*, April 2000. Available from `http://caml.inria.fr`.

[NNH99] Nielson, F., H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, New York, NY, 1999.

[OT98] Oliva, D. P. and A. P. Tolmach. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, **8**(4), July 1998, pp. 367–412.

[Ste78] Steele Jr., G. L. Rabbit: A compiler for Scheme. Master's dissertation, MIT, May 1978.

[SW97] Steckler, P. A. and M. Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, **19**(1), January 1997, pp. 48–86.

[Tar96] Tarditi, D. *Design and implementation of code optimizations for a type-directed compiler for Standard ML*. Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1996. Available as Technical Report CMU-CS-97-108.

[TMC+96] Tarditi, D., G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed compiler optimizing compiler for ML. In *Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation*, May 1996, pp. 181–192.

Interconnecting Between CPS Terms and Non-CPS Terms *

Jung-taek Kim and Kwangkeun Yi †

ROPAS ‡
Department of Computer Science
KAIST §

Abstract

We present a type-based partial CPS transformation (CPS-transforming only a sub-part of a program) and its correctness proof. A program's sub-parts which need to be CPS-transformed are initially annotated as such. Partial CPS-transformation scans the expressions and selectively apply the transformation depending on the annotation. Wherever an interface between CPS-transformed and direct-style expressions is needed, proper conversion code is padded based on expression types. The correctness of the partial CPS transformation is proven similarly to that of Plotkin's simulation theorem[Plo75].

1 This Work

Is it possible to CPS-transform only a sub-part of a program, leaving others in direct-style? Such partial CPS-transformation is needed, for example, to link CPS-transformed modules with non-CPS modules or to minimize the performance overhead by the transformation[KYD98].

In this article we present a type-based partial CPS transformation. The partial CPS transformation transforms only parts of a program and connects the results to the rest, direct-style parts. A program's sub-parts which need to be CPS-transformed is initially annotated as such. Partial CPS-transformation scans the expressions and selectively apply the transformation depending on the annotation. Wherever an interface between CPS-transformed and direct-style expressions is needed, proper conversion code is padded based on expression types.

Let's consider an example code:

$$((\lambda x.x)\,1) + 1$$

Assume that we want to CPS-transform only the function part - $\lambda x.x$. A CPS transformed version of the function part is: $\lambda K.K\,(\lambda x.\lambda K.K\,x)$. To use the CPS-transformed function inside the non-CPS context $([]\,1) + 1$, we wrap the CPS-function position with a conversion code as follows:

$$\lambda y.([]\,(\lambda x.x)\;y\;(\lambda x.x))$$

where the first identity continuation is for the function expression itself and the second identity continuation is for the function's result. By merging all components, we get a partially CPS transformed code :

$$((\lambda y.(\lambda K.K\,(\lambda x.\lambda K.K\,x))\;(\lambda x.x)\;y\;(\lambda x.x))1) + 1.$$

2 Source Language and CPS Transformation

We consider ML's core language with annotations. Its abstract syntax reads as follows. (κ denotes a data constructor)

$$\begin{array}{llcl}
\textit{expressions} & e & ::= & t^a \\
 & t & ::= & 1 \\
 & & | & \lambda x.e \\
 & & | & \texttt{fix}\, f\, \lambda x.e \\
 & & | & x \\
 & & | & e_1 @ e_2 \\
 & & | & \texttt{con}\, \kappa\, e \\
 & & | & \texttt{decon}\, e \\
 & & | & \texttt{case}\, e_1\, \kappa\, e_2\, e_3 \\
\textit{annotation} & a & ::= & N \mid C
\end{array}$$

Annotation C denotes that the expression needs CPS transformation and N otherwise. Value $\kappa \cdot v$ is constructed by "$\texttt{con}\,\kappa\,e$" where evaluating e yields v. Symmetrically, value $\kappa \cdot v$ denoted by e is deconstructed into v by evaluating "$\texttt{decon}\,e$". Evaluating the case expression "$\texttt{case}\,e_1\,\kappa\,e_2\,e_3$" yields the value of e_2 if the value of e_1 is $\kappa \cdot v$; otherwise, it yields the value of e_3.

We assume that the source programs are type-checked before processing. Types range over τ :

$$\begin{array}{llcl}
\textit{types} & \tau & ::= & \iota \\
 & & | & \tau \to \tau \\
 & & | & \alpha \\
 & & | & \mu\,\alpha.\tau_{\kappa_1} + \tau_{\kappa_2}
\end{array}$$

ι denotes basic types, $\tau \to \tau$ denotes function types and $\mu\,\alpha.\tau_{\kappa_1} + \tau_{\kappa_2}$ denotes user-defined recursive types where data constructor κ_1 has type $\tau_{\kappa_1} \to (\mu\,\alpha.\tau_{\kappa_1} + \tau_{\kappa_2})$ and κ_2 has type $\tau_{\kappa_2} \to (\mu\,\alpha.\tau_{\kappa_1} + \tau_{\kappa_2})$. And we use the conventional mono-morphic type system for the ML core language.

Figure 1 shows the conventional call-by-value CPS transformation [Plo75] which transforms the whole program(we ignore the annotations).

*This work is supported by Creative Research Initiatives of Korean Ministry of Science and Technology.

†email:{judaigi,kwang}@ropas.kaist.ac.kr

‡Research On Program Analysis System (http://ropas.kaist.ac.kr)

§Korea Advanced Institute of Science and Technology (http://www.kaist.ac.kr)

$$
\begin{array}{rcl}
[\![\mathtt{1}]\!] & = & \lambda K.K@\mathtt{1} \\
[\![x]\!] & = & \lambda K.K@x \\
[\![\mathtt{con}\ \kappa\ e]\!] & = & \lambda K.[\![e]\!]@(\lambda v.K@(\mathtt{con}\ \kappa\ v)) \\
[\![\mathtt{decon}\ e]\!] & = & \lambda K.[\![e]\!]@(\lambda v.K@(\mathtt{decon}\ v)) \\
[\![\lambda x.e]\!] & = & \lambda K.K@(\lambda x.[\![e]\!]) \\
[\![\mathtt{fix}\ f\ \lambda x.e]\!] & = & \lambda K.K@(\mathtt{fix}\ f\ \lambda x.[\![e]\!]) \\
[\![e_1@e_2]\!] & = & \lambda K.[\![e_1]\!]@(\lambda f.[\![e_2]\!]\ (\lambda v.f@v@K)) \\
[\![\mathtt{case}\ e_1\ \kappa\ e_2\ e_3]\!] & = & \\
\multicolumn{3}{r}{\lambda K.[\![e_1]\!]@(\lambda v.\mathtt{case}\ v\ \kappa\ ([\![e_2]\!]@K)\ ([\![e_3]\!]@K))}
\end{array}
$$

Figure 1: Base CPS transformation $[\![\cdot]\!]$

3 Conversion Functions and Partial CPS Transformation

Partial CPS transformation receives an annotated program and outputs a partially CPS transformed version of it. C-annotated expressions are targets for CPS transformation.

Not all annotations lead to correct translations. If a variable occurs at multiple positions, they all should have the same annotation. In case that a variable is bound, its annotation should match with that of its binding expression. Annotated programs satisfying these constraints are called consistent:

Definition 1 (Consistency of Annotations) *Annotations in an expression e is called consistent whenever the following holds :*

- *If x^a and $x^{a'}$ are free variables in e then a and a' are equal.*
- *If x^a is bounded by a sub-expression $(\lambda x.e)^{a'}$, $(\mathtt{fix} f \lambda x.e)^{a'}$, or $(\mathtt{fix}\ x\ \lambda y.e)^{a'}$ in e then a and a' are equal.*

In partial CPS transformation, we need conversion codes that interface CPS transformed expressions with non-CPS contexts and vice versa. We have to consider three cases according to the types of the expressions to be converted:

- For expressions of base type ι :

  No conversion is necessary. CPS version of a base-typed expression just passes to the continuation the value computed by the non-CPS version.

- For expressions of function type $\tau_1 \to \tau_2$:

  We eta-expand them to set up conversion codes around their arguments and results. For example, let's assume a CPS function is used in a non-CPS part. Then its argument should be a non-CPS value. So, we convert the argument to a CPS expression and apply it to the function. Because function's result is a CPS expression, we convert the result to a non-CPS expression.

- For expressions of user-defined type $\mu\,\alpha.\tau_{\kappa_1}+\tau_{\kappa_2}$:

  An expression of this user-defined type will be evaluated to $\kappa_1 \cdot v$ or $\kappa_2 \cdot v$. In order to convert the argument value v, because the argument value v can also be $\kappa_1 \cdot v'$ or $\kappa_2 \cdot v'$ (due to recursive type), we should make a recursive conversion function. This conversion function deconstructs the value, converts the argument value recursively, and reconstructs.

The conversion codes are generated by two mutually recursive functions `c2n` (CPS to non-CPS) and `n2c` (non-CPS to CPS) which are inductively defined over the types as follows:

- The `c2n` function, given the type of a non-CPS expression, generates a conversion code that converts the values of the type into their CPS versions, to be used inside a CPS context:

$$
\begin{array}{lcl}
\mathtt{c2n}(\iota) & = & \lambda v.v \\
\mathtt{c2n}(\tau_1 \to \tau_2) & = & \lambda v.\lambda x.v@(\mathtt{n2c}(\tau_1)@x)@\mathtt{c2n}(\tau_2) \\
\mathtt{c2n}(\mu\,\alpha.\tau_{\kappa_1}+\tau_{\kappa_2}) & = & \\
\multicolumn{3}{r}{(\mathtt{fix}\ f\ \lambda y.\ \mathtt{case}\ y\ \kappa_1} \\
\multicolumn{3}{r}{(\mathtt{con}\ \kappa_1\ (\mathtt{c2n}([\alpha^f/\alpha]\tau_{\kappa_1})@(\mathtt{decon}\ y)))} \\
\multicolumn{3}{r}{(\mathtt{con}\ \kappa_2\ (\mathtt{c2n}([\alpha^f/\alpha]\tau_{\kappa_2})@(\mathtt{decon}\ y))))} \\
\mathtt{c2n}(\alpha^f) & = & f
\end{array}
$$

  Note that the conversion code for a recursive type $\mu\,\alpha.\tau_{\kappa_1}+\tau_{\kappa_2}$ is a recursive function "$\mathtt{fix}\ f\ \lambda x.-$." The code for a recursive call to f is generated when `n2c` is called with the recursive type α^f.

- The `n2c` receives the original type of CPS expression. The output is a conversion code that will convert the CPS values into their non-CPS versions, to be used inside a non-CPS context.

$$
\begin{array}{lcl}
\mathtt{n2c}(\iota) & = & \lambda v.v \\
\mathtt{n2c}(\tau_1 \to \tau_2) & = & \\
\multicolumn{3}{r}{\lambda v.\lambda x.\lambda K.K@(\mathtt{n2c}(\tau_2)@(v@(\mathtt{c2n}(\tau_1)@x)))} \\
\mathtt{n2c}(\mu\,\alpha.\tau_{\kappa_1}+\tau_{\kappa_2}) & = & \\
\multicolumn{3}{r}{(\mathtt{fix}\ f\ \lambda y.\ \mathtt{case}\ y\ \kappa_1} \\
\multicolumn{3}{r}{(\mathtt{con}\ \kappa_1\ (\mathtt{n2c}([\alpha^f/\alpha]\tau_{\kappa_1})@(\mathtt{decon}\ y)))} \\
\multicolumn{3}{r}{(\mathtt{con}\ \kappa_2\ (\mathtt{n2c}([\alpha^f/\alpha]\tau_{\kappa_2})@(\mathtt{decon}\ y))))} \\
\mathtt{n2c}(\alpha^f) & = & f
\end{array}
$$

The conversion codes can be generated at compile time because sizes of types are finite.

The partial CPS transformation $[\![\cdot]\!]_N$ and $[\![\cdot]\!]_C$ by using the two conversion-code-generators (`c2n` and `n2c`) are defined in Figure 2. For a term t, we write τ_t for its type.

4 Correctness of Partial CPS transformation

To prove the correctness of our partial CPS transformation, we extend our annotated language to have annotation sequences, not just a single annotation. These annotation sequences are used to trace the changes of annotations (hence the conversions between CPS and non-CPS styles) during the evaluation.

$$
\begin{array}{rcl}
[\![\mathtt{1}^N]\!]_N & = & \mathtt{1} \\
[\![(\lambda x.e)^N]\!]_N & = & \lambda x.[\![e]\!]_N \\
[\![(\mathtt{fix}\ f\ \lambda x.e)^N]\!]_N & = & \mathtt{fix}\ f\ \lambda x.[\![e]\!]_N \\
[\![x^N]\!]_N & = & x \\
[\![(e_1@e_2)^N]\!]_N & = & [\![e_1]\!]_N@[\![e_2]\!]_N \\
[\![(\mathtt{con}\ \kappa\ e)^N]\!]_N & = & \mathtt{con}\ \kappa\ [\![e]\!]_N \\
[\![(\mathtt{decon}\ e)^N]\!]_N & = & \mathtt{decon}\ [\![e]\!]_N \\
[\![(\mathtt{case}\ e_1\ \kappa\ e_2\ e_3)^N]\!]_N & = & \mathtt{case}\ [\![e_1]\!]_N\ \kappa\ [\![e_2]\!]_N\ [\![e_3]\!]_N \\
[\![t^C]\!]_N & = & [\![t^C]\!]_C@\mathtt{c2n}(\tau_t) \\
 & & \\
[\![\mathtt{1}^C]\!]_C & = & \lambda K.K@\mathtt{1} \\
[\![(\lambda x.e)^C]\!]_C & = & \lambda K.K@\lambda x.[\![e]\!]_C \\
[\![(\mathtt{fix}\ f\ \lambda x.e)^C]\!]_C & = & \lambda K.K@\mathtt{fix}\ f\ \lambda x.[\![e]\!]_C \\
[\![x^C]\!]_C & = & \lambda K.K@x \\
[\![(e_1@e_2)^C]\!]_C & = & \lambda K.[\![e_1]\!]_C@ \\
 & & \quad (\lambda f.[\![e_2]\!]_C@(\lambda v.f@v@K)) \\
[\![(\mathtt{con}\ \kappa\ e)^C]\!]_C & = & \lambda K.[\![e]\!]_C@(\lambda v.K@(\mathtt{con}\ \kappa\ v)) \\
[\![(\mathtt{decon}\ e)^C]\!]_C & = & \lambda K.[\![e]\!]_C@(\lambda v.K@(\mathtt{decon}\ v)) \\
[\![(\mathtt{case}\ e_1\ \kappa\ e_2\ e_3)^C]\!]_C & = & \lambda K.[\![e_1]\!]_C@(\lambda v.\mathtt{case}\ v\ \kappa \\
 & & \quad ([\![e_2]\!]_C@K)\ ([\![e_2]\!]_C@K)) \\
[\![t^N]\!]_C & = & \lambda K.K@(\mathtt{n2c}(\tau_t)@[\![t^N]\!]_N)
\end{array}
$$

Figure 2: Partial CPS transformation

$$
\begin{array}{lrcl}
\textit{annotated expressions} & & & \\
 & e & ::= & t^a \mid e^a \\
 & t & ::= & \mathtt{1} \\
 & & \mid & \lambda x.e \\
 & & \mid & \mathtt{fix}\ f\ \lambda x.e \\
 & & \mid & \kappa \cdot v \\
 & & \mid & x \\
 & & \mid & e_1@e_2 \\
 & & \mid & \mathtt{con}\ \kappa\ e \\
 & & \mid & \mathtt{decon}\ e \\
 & & \mid & \mathtt{case}\ e_1\ \kappa\ e_2\ e_3 \\
\textit{annotated values} & & & \\
 & v & ::= & w^a \mid v^a \\
 & w & ::= & \mathtt{1} \\
 & & \mid & \lambda x.e \\
 & & \mid & \mathtt{fix}\ f\ \lambda x.e \\
 & & \mid & \kappa \cdot v \\
\textit{annotation} & & & \\
 & a & ::= & N \mid C
\end{array}
$$

For convenience, we use the following notation:

$$
\begin{array}{lrcl}
\textit{annotation sequence} & l & ::= & a \mid la \\
\textit{annotated expressions} & t^{la} & = & (t^l)^a \\
\textit{reversal of annotations} & \overline{a} & = & a \\
 & \overline{al} & = & \overline{l}a
\end{array}
$$

We now define the semantics of annotated expressions. This semantics is equivalent to that of ML core language except that we book-keep the annotations during reduction. The annotated evaluation context E is defined by the following grammar:

$$
\begin{array}{lrcl}
\textit{annotated contexts} & & & \\
 & E & ::= & D^a \\
 & D & ::= & [\,] \\
 & & \mid & E@e \\
 & & \mid & v@E \\
 & & \mid & \mathtt{con}\ \kappa\ E \\
 & & \mid & \mathtt{decon}\ E \\
 & & \mid & \mathtt{case}\ E\ \kappa\ e_1\ e_2
\end{array}
$$

This context defines a left-to-right, call-by-value reduction. As usual, we write $E[e]$ if the hole in context E is filled with e. We use this context to define the reduction rules for arbitrary expressions :

$$\frac{e \mapsto e'}{E[e] \to E[e']}$$

The single reduction step $e \mapsto e'$ for a redex e is defined as :

$$
\begin{array}{rcll}
((\lambda x.e)^l@v)^a & \mapsto & [v^{a\overline{l}}/x]e^{la} & \\
((\mathtt{fix}\ f\ \lambda x.e)^l@v)^{a'} & \mapsto & [(\mathtt{fix}\ f\ \lambda x.e)^a/f][v^{a'\overline{l}}/x]e^{la'} & \\
 & & & (\text{head of } l \text{ is } a) \\
(\mathtt{con}\ \kappa\ v)^a & \mapsto & (\kappa \cdot v)^a & \\
(\mathtt{decon}\ (\kappa \cdot v)^l)^a & \mapsto & (v)^{la} & \\
(\mathtt{case}\ (\kappa \cdot v)^l\ \kappa\ e_1\ e_2)^a & \mapsto & e_1^a & \\
(\mathtt{case}\ (\kappa \cdot v)^l\ \kappa'\ e_1\ e_2)^a & \mapsto & e_2^a & (\kappa \neq \kappa')
\end{array}
$$

Notation $[v/x]e$ denotes, as usual, the new expression that results from substituting v for every free occurrence of x in e.

Definition 2 *The semantics of a closed annotated expression e is defined to be the sequence of reduction steps*

$$e \to e_1 \to e_2 \to \cdots.$$

If the sequence terminates with a value v after zero or more reductions, we write

$$e \xrightarrow{*} v$$

Now we can define a correctness theorem for our partial CPS transformation similarly to the simulation theorem in [Plo75].

Theorem 1 (Correctness of Partial CPS transformation) *For any closed expression e with consistent annotations,*

$$e \xrightarrow{*} v \iff [\![e]\!]_N \xrightarrow{*} \Psi_N(v)$$

where the auxiliary functions Ψ_N and Ψ_C coerces direct-style values to partial CPS values:

$$
\begin{array}{rcll}
\Psi_N(\mathtt{1}^N) & = & \mathtt{1} & \\
\Psi_N((\lambda x.e)^N) & = & \lambda x.[\![e]\!]_N & \\
\Psi_N((\mathtt{fix}\ f\ \lambda x.e)^N) & = & \mathtt{fix}\ f\ \lambda x.[\![e]\!]_N & \\
\Psi_N((\kappa \cdot v)^N) & = & \kappa \cdot \Psi_N(v) & \\
\Psi_N(v^N) & = & \Psi_N(v) & \\
\Psi_N(w^C) & = & w' & (\mathtt{c2n}(\tau_w)@\Psi_C(w^C) \xrightarrow{*} w') \\
\Psi_N(v^C) & = & v' & (\mathtt{c2n}(\tau_v)@\Psi_C(v^C) \xrightarrow{*} v') \\
 & & & \\
\Psi_C(\mathtt{1}^C) & = & \mathtt{1} & \\
\Psi_C((\lambda x.e)^C) & = & \lambda x.[\![e]\!]_C & \\
\Psi_C((\mathtt{fix}\ f\ \lambda x.e)^C) & = & \mathtt{fix}\ f\ \lambda x.[\![e]\!]_C & \\
\Psi_C((\kappa \cdot v)^C) & = & \kappa \cdot \Psi_C(v) & \\
\Psi_C(v^C) & = & \Psi_C(v) & \\
\Psi_C(w^N) & = & w' & (\mathtt{n2c}(\tau_w)@\Psi_N(w^N) \xrightarrow{*} w') \\
\Psi_C(v^N) & = & v' & (\mathtt{n2c}(\tau_v)@\Psi_N(v^N) \xrightarrow{*} v')
\end{array}
$$

$$
\begin{array}{rcl}
[\![v]\!]_N & = & \Psi_N(v) \\
[\![(\lambda x.e)^N]\!]_N & = & \lambda x.[\![e]\!]_N \\
[\![(\mathtt{fix}\ f\ \lambda x.e)^N]\!]_N & = & \mathtt{fix}\ f\ \lambda x.[\![e]\!]_N \\
[\![x^N]\!]_N & = & x \\
[\![(e_1@e_2)^N]\!]_N & = & [\![e_1]\!]_N@[\![e_2]\!]_N \\
[\![(\mathtt{con}\ \kappa\ e)^N]\!]_N & = & \mathtt{con}\ \kappa\ [\![e]\!]_N \\
[\![(\mathtt{decon}\ e)^N]\!]_N & = & \mathtt{decon}\ [\![e]\!]_N \\
[\![(\mathtt{case}\ e_1\ \kappa\ e_2\ e_3)^N]\!]_N & = & \mathtt{case}\ [\![e_1]\!]_N\ \kappa\ [\![e_2]\!]_N\ [\![e_3]\!]_N \\
[\![e^N]\!]_N & = & [\![e]\!]_N \\
[\![t^C]\!]_N & = & [\![t^C]\!]_C@\mathtt{c2n}(\tau_t) \\
[\![e^C]\!]_N & = & [\![e^C]\!]_C@\mathtt{c2n}(\tau_e) \\
 & & \\
[\![v]\!]_C & = & \lambda K.K@\Psi_C(v) \\
[\![(\lambda x.e)^C]\!]_C & = & \lambda K.K@\lambda x.[\![e]\!]_C \\
[\![(\mathtt{fix}\ f\ \lambda x.e)^C]\!]_C & = & \lambda K.K@\mathtt{fix}\ f\ \lambda x.[\![e]\!]_C \\
[\![x^C]\!]_C & = & \lambda K.K@x \\
[\![(e_1@e_2)^C]\!]_C & = & \lambda K.[\![e_1]\!]_C@ \\
 & & \quad (\lambda f.[\![e_2]\!]_C@(\lambda v.f@v@K)) \\
[\![(\mathtt{con}\ \kappa\ e)^C]\!]_C & = & \lambda K.[\![e]\!]_C@(\lambda v.K@(\mathtt{con}\ \kappa\ v)) \\
[\![(\mathtt{decon}\ e)^C]\!]_C & = & \lambda K.[\![e]\!]_C@(\lambda v.K@(\mathtt{decon}\ v)) \\
[\![(\mathtt{case}\ e_1\ \kappa\ e_2\ e_3)^C]\!]_C & = & \lambda K.[\![e_1]\!]_C@(\lambda v.\mathtt{case}\ v\ \kappa \\
 & & \quad ([\![e_2]\!]_C@K)\ ([\![e_2]\!]_C@K)) \\
[\![e^C]\!]_C & = & [\![e]\!]_C \\
[\![t^N]\!]_C & = & \lambda K.K@(\mathtt{n2c}(\tau_t)@[\![t^N]\!]_N) \\
[\![e^N]\!]_C & = & \lambda K.K@(\mathtt{n2c}(\tau_e)@[\![e^N]\!]_N)
\end{array}
$$

Figure 3: Extended Partial CPS transformation

We use a similar approach to [Plo75] to prove the above theorem.

At first, we extend the partial CPS transformation to be defined also for the intermediate expressions during reduction. Then we define two colon operators : and $:K$ to represent the expressions after the administrative reduction. Operator : is used for $[\![\cdot]\!]_N$ and $:K$ for $[\![\cdot]\!]_C$. In partially CPS-transforming the intermediate expressions, computed values (closed expressions without redices) are transformed by a special transformers Ψ_N and Ψ_C. The extended partial CPS transformation is defined in Figure 3.

Figure 4 defines the two colon operators (: and $:K$) which represent the transformed terms (respectively from the $[\![\cdot]\!]_N$ and $[\![\cdot]\!]_C$) after the administrative reductions.

Value transformation preserves the structure of the values:

Lemma 1

$$\Psi_N((\kappa\cdot v)^l) = \kappa\cdot\Psi_N(v^l) \quad \textit{and} \quad \Psi_C((\kappa\cdot v)^l) = \kappa\cdot\Psi_C(v^l)$$

Proof. Proof by induction on the length of l. The details of the proof are in Appendix A. □

Following three Lemmas correspond to the Plotkin's ones [Plo75]. Substitution Lemma enables substitions to penetrate $[\![\cdot]\!]_N$ and $[\![\cdot]\!]_C$:

Lemma 2 (Substitution) *For any expression e with consistent annotation and any value v, if free variable x is annotated as N then*

$$
\begin{array}{rcl}
[\Psi_N(v)/x][\![e]\!]_N & = & [\![[v/x]e]\!]_N \\
[\Psi_N(v)/x][\![e]\!]_C & = & [\![[v/x]e]\!]_C
\end{array}
$$

$$
\begin{array}{rcl}
v: & = & \Psi_N(v) \\
(v_1@v_2)^N: & = & (\Psi_N(v_1))@(\Psi_N(v_2)) \\
(v@e)^N: & = & (\Psi_N(v))@(e:) \\
(e_1@e_2)^N: & = & (e_1:)@[\![e_2]\!]_N \\
(\mathtt{con}\ \kappa\ v)^N: & = & \mathtt{con}\ \kappa\ (\Psi_N(v)) \\
(\mathtt{con}\ \kappa\ e)^N: & = & \mathtt{con}\ \kappa\ (e:) \\
(\mathtt{decon}\ v)^N: & = & \mathtt{decon}\ (\Psi_N(v)) \\
(\mathtt{decon}\ e)^N: & = & \mathtt{decon}\ (e:) \\
(\mathtt{case}\ v\ \kappa\ e_1\ e_2)^N: & = & \mathtt{case}\ (\Psi_N(v))\ \kappa\ [\![e_1]\!]_N\ [\![e_2]\!]_N \\
(\mathtt{case}\ e_1\ \kappa\ e_2\ e_3)^N: & = & \mathtt{case}\ (e_1:)\ \kappa\ [\![e_2]\!]_N\ [\![e_3]\!]_N \\
e^N: & = & e: \\
t^C: & = & t^C{:}\mathtt{c2n}(\tau_t) \\
e^C: & = & e^C{:}\mathtt{c2n}(\tau_e) \\
 & & \\
v{:}K & = & K@\Psi_C(v) \\
(v_1@v_2)^C{:}K & = & \Psi_C(v_1)@\Psi_C(v_2)@K \\
(v@e)^C{:}K & = & e{:}\lambda v.\Psi_C(v)@v@K \\
(e_1@e_2)^C{:}K & = & e_1{:}\lambda f.[\![e_2]\!]_C@(\lambda v.f@v@K) \\
(\mathtt{con}\ \kappa\ v)^C{:}K & = & K@(\mathtt{con}\ \kappa\ (\Psi_C(v))) \\
(\mathtt{con}\ \kappa\ e)^C{:}K & = & e{:}\lambda v.K@(\mathtt{con}\ \kappa\ v) \\
(\mathtt{decon}\ v)^C{:}K & = & K@(\mathtt{decon}\ (\Psi_C(v))) \\
(\mathtt{decon}\ e)^C{:}K & = & e{:}\lambda v.K@(\mathtt{decon}\ v) \\
(\mathtt{case}\ v\ \kappa\ e_1\ e_2)^C{:}K & = & \mathtt{case}\ (\Psi_C(v))\ \kappa \\
 & & \quad ([\![e_1]\!]_C@K)\ ([\![e_2]\!]_C@K) \\
(\mathtt{case}\ e_1\ \kappa\ e_2\ e_3)^C{:}K & = & e_1{:}(\lambda v.\mathtt{case}\ v\ \kappa \\
 & & \quad ([\![e_2]\!]_C@K)([\![e_3]\!]_C@K)) \\
e^C{:}K & = & e{:}K \\
t^N{:}K & = & K@(\mathtt{n2c}(\tau_t)@(t^N{:})) \\
e^N{:}K & = & K@(\mathtt{n2c}(\tau_e)@(e^N{:}))
\end{array}
$$

Figure 4: The Colon Operators

and if free variable x is annotated as C then

$$
\begin{array}{rcl}
[\Psi_C(v)/x][\![e]\!]_N & = & [\![[v/x]e]\!]_N \\
[\Psi_C(v)/x][\![e]\!]_C & = & [\![[v/x]e]\!]_C
\end{array}
$$

Proof. Proof by mutual structural induction on e with a case for type of expression. The details of the proof are in Appendix B. □

Static reduction Lemma establishes the relation between partial CPS transformation and colon operator:

Lemma 3 (Static Reduction) *For any closed expression e with consistent annotation and any closed value K,*

$$
\begin{array}{rcl}
[\![e]\!]_N & \xrightarrow{*} & e: \\
[\![e]\!]_C@K & \xrightarrow{*} & e{:}K
\end{array}
$$

Proof. Proof by mutual structural induction on e with a case for type of expression. The details of the proof are in Appendix C. □

Single-Step Simulation Lemma shows that a reduction from an original expression corresponds to reductions between partially CPS transformed expressions:

Lemma 4 (Single-Step Simulation) *For any closed expression e with consistent annotations and any closed value K,*

$$e \to e' \implies e{:} \xrightarrow{+} e'{:} \quad \textit{and} \quad e{:}K \xrightarrow{+} e'{:}K$$

Proof. Proof by mutual structural induction on the reduction $e \to e'$ with a case for type of reduction rules. The details of the proof are in Appendix D. □

Proof of Theorem 1. Using the lemmas, we can prove the correctness theorem easily. If the original program evaluates to a value, that is , $e \xrightarrow{*} v$ then

$$\begin{array}{lll} [\![e]\!]_N & & \\ \xrightarrow{*} & e{:} & \text{(Lemma 3)} \\ \xrightarrow{*} & v{:} & \text{(repeated use of Lemma 4)} \\ = & \Psi_N(v) & \text{(def. of :)} \end{array}$$

Note that non-termination is also preserved by the single step simulation.

□

5 Conclusion

We presented a type-based partial CPS transformation. The partial CPS transformation transforms only parts of a program and connects the results to the rest, direct-style parts. A program's sub-parts which need to be CPS-transformed is initially annotated as such. Partial CPS-transformation scans the expressions and selectively apply the transformation depending on the annotation. Wherever an interface between CPS-transformed and direct-style expressions is needed, proper conversion code is padded based on expression types. The correctness of the partial CPS transformation is proven similarly to that of Plotkin's simulation theorem[Plo75].

There are analogous works in literature. Our `n2c` and `c2n` functions can be seen as the retraction pairs in the setting of Meyer and Wand[MW85]. Danvy, Dussart, and Hatcliff [DD95, DH93a, DH93b]'s works are analogous to ours. They switch between call-by-value and call-by-name CPS transformations interconnecting the two transformations by "delay" and "force" operators. Another analogous one is Leroy's "box" and "unbox" operators[Ler92] that are used to handle mixed representations of values.

References

[DD95] Olivier Danvy and Dirk Dussart. CPS transformation after binding-time analysis. Unpublished note, Department of Computer Science, University of Aarhus, April 1995.

[DH93a] Olivier Danvy and John Hatcliff. CPS transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, 1(3):195–212, 1993.

[DH93b] Olivier Danvy and John Hatcliff. On the transformation between direct and continuation semantics. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Proceedings of the 9th Conference on Mathematical Foundations of Programming Semantics*, number 802 in Lecture Notes in Computer Science, pages 627–648, New Orleans, Louisiana, April 1993. Springer-Verlag.

[KYD98] Jung-taek Kim, Kwangkeun Yi, and Olivier Danvy. Assessing the overhead of ML exceptions by selective CPS transformation. In Greg Morrisett, editor, *Record of the 1998 ACM SIGPLAN Workshop on ML and its Applications*, Baltimore, Maryland, September 1998. Also appears as BRICS technical report RS-98-15.

[Ler92] Xavier Leroy. Unboxed objects and polymorphic typing. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 177–188, January 1992.

[MW85] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi. In R. Parikh, editor, *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 219–224. Springer, Brooklyn, June 1985.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

A Proof of Lemma 1

Lemma 1

$$\Psi_N((\kappa{\cdot}v)^l) = \kappa{\cdot}\Psi_N(v^l) \quad \textit{and} \quad \Psi_C((\kappa{\cdot}v)^l) = \kappa{\cdot}\Psi_C(v^l)$$

Proof.

Proof by induction on the length of l.

In proof, we abbreviate some expressions as "$(\cdots)$" when they are omittable.

For Ψ_N cases,

- base case $(\kappa{\cdot}v)^N$

$$\begin{array}{lll} & \Psi_N((\kappa{\cdot}v)^N) & \\ = & \kappa{\cdot}\Psi_N(v) & \text{(def. of } \Psi_N) \\ = & \kappa{\cdot}\Psi_N(v^N) & \text{(def. of } \Psi_N) \end{array}$$

- base case $(\kappa{\cdot}v)^C$

$$\begin{array}{lll} & \texttt{c2n}(\mu\,\alpha.\tau_\kappa{+}\tau_{\kappa'}){@}\Psi_C((\kappa{\cdot}v)^C) & \\ = & (\texttt{fix}\,f\,\lambda y.\texttt{case}\,y\,\kappa & \\ & \quad (\texttt{con}\,\kappa\,\texttt{c2n}([\alpha^f/\alpha]\tau_\kappa){@}(\texttt{decon}\,y))\,(\cdots)) & \\ & \quad {@}(\kappa{\cdot}\Psi_C(v)) & \text{(def. of } \texttt{n2c} \text{ and } \Psi_N) \\ \to & \texttt{case}\,(\kappa{\cdot}\Psi_C(v))\,\kappa & \\ & \quad (\texttt{con}\,\kappa\,(\texttt{c2n}([\alpha^{f'}/\alpha]\tau_\kappa) & \\ & \quad\quad {@}(\texttt{decon}\,(\kappa{\cdot}\Psi_C(v)))))\,(\cdots) & \\ & \quad \text{(where } f' \text{ is } \texttt{fix}\,f\,\cdots) & \\ \xrightarrow{+} & \texttt{con}\,\kappa\,(\texttt{c2n}([\alpha^{f'}/\alpha]\tau_\kappa){@}(\Psi_C(v))) & \\ = & \texttt{con}\,\kappa\,(\texttt{c2n}([\alpha^{f'}/\alpha]\tau_\kappa){@}(\Psi_C(v^C))) & \\ & & \text{(def. of } \Psi_N) \\ \xrightarrow{*} & \texttt{con}\,\kappa\,(\Psi_N(v^C)) & \\ & \quad \text{(Because } [\alpha^{f'}/\alpha]\tau_\kappa \text{ is indeed the type of } v, & \\ & \quad \texttt{c2n}([\alpha^{f'}/\alpha]\tau_\kappa){@}\Psi_C(v^C) \xrightarrow{*} v' = \Psi_N(v^C) & \\ & \quad \text{by definition of } \Psi_N.) & \\ \to & \kappa{\cdot}(\Psi_N(v^C)) & \end{array}$$

By above and definition of Ψ_N,

$$\begin{array}{lll} & \Psi_N((\kappa{\cdot}v)^C) & \\ = & \kappa{\cdot}\Psi_N(v^C) & \end{array}$$

- induction step case $(\kappa{\cdot}v)^{lN}$

$$\begin{array}{lll}\Psi_N((\kappa{\cdot}v)^{lN}) & & \\ = & \Psi_N((\kappa{\cdot}v)^{l}) & \text{(def. of } \Psi_N) \\ = & \kappa{\cdot}\Psi_N((v)^{l}) & \text{(I.H.)} \\ = & \kappa{\cdot}\Psi_N((v)^{lN}) & \text{(def. of } \Psi_N)\end{array}$$

- induction step case $(\kappa{\cdot}v)^{lC}$

$$\begin{array}{lll}\mathtt{c2n}(\mu\ \alpha.\tau_\kappa{+}\tau_{\kappa'}){@}\Psi_C((\kappa{\cdot}v)^{lC}) & & \\ = & \mathtt{c2n}(\mu\ \alpha.\tau_\kappa{+}\tau_{\kappa'}){@}\Psi_C((\kappa{\cdot}v)^{l}) & \text{(def. of } \Psi_C) \\ = & (\mathtt{fix}\ f\ \lambda y.\mathtt{case}\ y\ \kappa & \\ & \quad (\mathtt{con}\ \kappa\ \mathtt{c2n}([\alpha^{f}/\alpha]\tau_\kappa){@}(\mathtt{decon}\ y))\ (\cdots)) & \\ & {@}(\kappa{\cdot}\Psi_C(v^{l})) & \text{(def. of } \mathtt{n2c} \text{ and I.H.)} \\ \to & \mathtt{case}\ (\kappa{\cdot}\Psi_C(v^{l}))\ \kappa & \\ & \quad (\mathtt{con}\ \kappa\ (\mathtt{c2n}([\alpha^{f'}/\alpha]\tau_\kappa) & \\ & \quad\quad {@}(\mathtt{decon}\ (\kappa{\cdot}\Psi_C(v^{l}))))\ (\cdots) & \\ & \text{(where } f' \text{ is } \mathtt{fix}\ f\ \cdots) & \\ \xrightarrow{+} & \mathtt{con}\ \kappa\ (\mathtt{c2n}([\alpha^{f'}/\alpha]\tau_\kappa){@}(\Psi_C(v^{l}))) & \\ = & \mathtt{con}\ \kappa\ (\mathtt{c2n}([\alpha^{f'}/\alpha]\tau_\kappa){@}(\Psi_C(v^{lC}))) & \\ & & \text{(def. of } \Psi_C) \\ \xrightarrow{*} & \mathtt{con}\ \kappa\ (\Psi_N(v^{lC})) & \\ & \text{(Because } [\alpha^{f'}/\alpha]\tau_\kappa \text{ is indeed the type of } v, & \\ & \mathtt{c2n}([\alpha^{f'}/\alpha]\tau_\kappa){@}\Psi_C(v^{lC}) \xrightarrow{*} v' = \Psi_N(v^{lC}) & \\ & \text{by definition of } \Psi_N.) & \\ \to & \kappa{\cdot}(\Psi_N(v^{lC})) & \end{array}$$

By above and definition of Ψ_N,

$$\begin{array}{lll}\Psi_N((\kappa{\cdot}v)^{lC}) & & \\ = & \kappa{\cdot}\Psi_N(v^{lC}) & \text{(def. of } \Psi_N)\end{array}$$

Because Ψ_T and Ψ_N are duals, we can prove Ψ_T cases similarly to Ψ_N cases. □

B Proof of Substitution Lemma

Lemma 2 (Substitution) *For any expression e with consistent annotation and any value v, if free variable x is annotated as N then*

$$\begin{array}{lll}[\Psi_N(v)/x][\![e]\!]_N & = & [\![[v/x]e]\!]_N \\ [\Psi_N(v)/x][\![e]\!]_C & = & [\![[v/x]e]\!]_C\end{array}$$

and if free variable x is annotated as C then

$$\begin{array}{lll}[\Psi_C(v)/x][\![e]\!]_N & = & [\![[v/x]e]\!]_N \\ [\Psi_C(v)/x][\![e]\!]_C & = & [\![[v/x]e]\!]_C\end{array}$$

Proof.

Proof by mutual structural induction on e, firstly taking the $[\![\cdot]\!]_N$ for the N annotated expressions and $[\![\cdot]\!]_C$ for the C annotated expressions.

For $[\![\cdot]\!]_N$ cases,

- case v, $(\lambda x.e_1)^N$, $(\mathtt{fix}\ f\ \lambda x.e_1)^N$, or $(\mathtt{fix}\ x\ \lambda y.e_1)^N$

$$\begin{array}{lll}[\Psi_a(v)/x][\![e]\!]_N & & \\ = & [\![e]\!]_N & (x \text{ is not free}) \\ = & [\![[v/x]e]\!]_N & (x \text{ is not free})\end{array}$$

  a is N or C according to the annotation of the free variable x.

- case $(\lambda y.e)^N \quad (x \neq y)$

$$\begin{array}{lll}[\Psi_a(v)/x][\![(\lambda y.e)^N]\!]_N & & \\ = & [\Psi_a(v)/x]\lambda y.[\![e]\!]_N & \text{(def. of } [\![\cdot]\!]_N) \\ = & \lambda y.[\Psi_a(v)/x][\![e]\!]_N & \text{(def. of } [/]) \\ = & \lambda y.[\![[v/x]e]\!]_N & \text{(I.H.)} \\ = & [\![(\lambda y.[v/x]e)^N]\!]_N & \text{(def. of } [\![\cdot]\!]_N) \\ = & [\![[v/x](\lambda y.e)^N]\!]_N & \text{(def. of } [/])\end{array}$$

  a is N or C according to the annotation of the free variable x.

- case $(\mathtt{fix}\ f\ \lambda y.e)^N \quad (x \neq y, x \neq f)$

  We can prove similarly to the above $(\lambda y.e)^N$ case.

- case x^N

  Because the variable x is annotated as N, we need to prove only for Ψ_N.

$$\begin{array}{lll}[\Psi_N(v)/x][\![x^N]\!]_N & & \\ = & [\Psi_N(v)/x]x & \text{(def. of } [\![\cdot]\!]_N) \\ = & \Psi_N(v) & \text{(def. of } [/]) \\ = & \Psi_N(v^N) & \text{(def. of } \Psi_N) \\ = & [\![v^N]\!]_N & \text{(def. of } [\![\cdot]\!]_N) \\ = & [\![[v/x]x^N]\!]_N & \text{(def. of } [/])\end{array}$$

- case $(e_1@e_2)^N$

  The annotations of the free variable x in both e_1 and e_2 are equal.

$$\begin{array}{lll}[\Psi_a(v)/x][\![(e_1@e_2)^N]\!]_N & & \\ = & [\Psi_a(v)/x]([\![e_1]\!]_N@[\![e_2]\!]_N) & \text{(def. of } [\![\cdot]\!]_N) \\ = & ([\Psi_a(v)/x][\![e_1]\!]_N)@([\Psi_a(v)/x][\![e_2]\!]_N) & \\ & & \text{(def. of } [/]) \\ = & [\![[v/x]e_1]\!]_N@[\![[v/x]e_2]\!]_N & \text{(I.H. twice)} \\ = & [\![([[v/x]e_1)@([v/x]e_2))^N]\!]_N & \text{(def. of } [\![\cdot]\!]_N) \\ = & [\![[v/x](e_1@e_2)^N]\!]_N & \text{(def. of } [/])\end{array}$$

  a is N or C according to the annotation of the free variable x.

- case $(\mathtt{con}\ \kappa\ e)^N$, $(\mathtt{decon}\ e)^N$, $(\mathtt{case}\ e_1\ \kappa\ e_2\ e_3)^N$, or e^N

  We can prove similarly to the above $(e_1@e_2)^N$ case.

- case e^C

$$\begin{array}{lll}[\Psi_a(v)/x][\![e^C]\!]_N & & \\ = & [\Psi_a(v)/x]([\![e^C]\!]_C@\mathtt{c2n}(\tau_e)) & \text{(def. of } [\![\cdot]\!]_N) \\ = & ([\Psi_a(v)/x][\![e^C]\!]_C)@\mathtt{c2n}(\tau_e) & \text{(def. of } [/]) \\ = & [\![[v/x]e^C]\!]_C@\mathtt{c2n}(\tau_e) & \text{(I.H. about } [\![\cdot]\!]_C) \\ = & [\![[v/x]e^C]\!]_N & \text{(def. of } [\![\cdot]\!]_N)\end{array}$$

- case t^C

  We can prove similarly to the above e^C case.

For $[\![\cdot]\!]_C$ cases,

- case v, $(\lambda x.e_1)^C$, $(\mathtt{fix}\ f\ \lambda x.e_1)^C$, or $(\mathtt{fix}\ x\ \lambda y.e_1)^C$

$$\begin{array}{lll}[\Psi_a(v)/x][\![e]\!]_C & & \\ = & [\![e]\!]_C & (x \text{ is not free}) \\ = & [\![[v/x]e]\!]_C & (x \text{ is not free})\end{array}$$

  a is N or C according to the annotation of the free variable x.

- case $(\lambda y.e)^C \quad (x \neq y)$

$[\Psi_a(v)/x][\![(\lambda y.e)^C]\!]_C$
$= [\Psi_a(v)/x]\lambda K.K@\lambda x.[\![e]\!]_C$ (def. of $[\![\cdot]\!]_C$)
$= \lambda K.K@\lambda x.[\Psi_a(v)/x][\![e]\!]_C$ (def. of $[/]$)
$= \lambda K.K@\lambda x.[\![[v/x]e]\!]_C$ (I.H.)
$= [\![(\lambda y.[v/x]e)^C]\!]_C$ (def. of $[\![\cdot]\!]_C$)
$= [\![[v/x](\lambda y.e)^C]\!]_C$ (def. of $[/]$)

a is N or C according to the annotation of the free variable x.

- case $(\mathtt{fix}\, f\, \lambda y.e)^C \quad (x \neq y, x \neq f)$

  We can prove similarly to the above $(\lambda y.e)^C$ case.

- case x^C

  Because the variable x is annotated as C, we need to prove only for Ψ_C.

  $[\Psi_C(v)/x][\![x^C]\!]_C$
  $= [\Psi_C(v)/x]\lambda K.K@x$ (def. of $[\![\cdot]\!]_C$)
  $= \lambda K.K@\Psi_C(v)$ (def. of $[/]$)
  $= \lambda K.K@\Psi_C(v^C)$ (def. of Ψ_C)
  $= [\![v^C]\!]_C$ (def. of $[\![\cdot]\!]_C$)
  $= [\![[v/x]x^C]\!]_C$ (def. of $[/]$)

- case $(e_1@e_2)^C$

  The annotations of the free variable x in both e_1 and e_2 are equal.

  $[\Psi_a(v)/x][\![(e_1@e_2)^C]\!]_C$
  $= [\Psi_a(v)/x]\lambda K.[\![e_1]\!]_C@(\lambda f.[\![e_2]\!]_C@(\lambda v.f@v@K))$ (def. of $[\![\cdot]\!]_C$)
  $= \lambda K.([\Psi_a(v)/x][\![e_1]\!]_C)@(\lambda f.([\Psi_a(v)/x][\![e_2]\!]_C)@(\lambda v.f@v@K))$ (def. of $[/]$)
  $= \lambda K.[\![[v/x]e_1]\!]_C@(\lambda f.[\![[v/x]e_2]\!]_C@(\lambda v.f@v@K))$ (I.H.)
  $= [\![(([v/x]e_1)@([v/x]e_2))^C]\!]_C$ (def. of $[\![\cdot]\!]_C$)
  $= [\![[v/x](e_1@e_2)^C]\!]_C$ (def. of $[/]$)

  a is N or C according to the annotation of the free variable x.

- case $(\mathtt{con}\,\kappa\, e)^C$, $(\mathtt{decon}\, e)^C$, $(\mathtt{case}\, e_1\, \kappa\, e_2\, e_3)^C$, or e^C

  We can prove similarly to the above $(e_1@e_2)^C$ case.

- case e^N

  $[\Psi_a(v)/x][\![e^N]\!]_C$
  $= [\Psi_a(v)/x](\lambda K.K@(\mathtt{n2c}(\tau_e)@[\![e^N]\!]_N))$ (def. of $[\![\cdot]\!]_C$)
  $= \lambda K.K@(\mathtt{n2c}(\tau_e)@([\Psi_a(v)/x][\![e^N]\!]_N))$ (def. of $[/]$)
  $= \lambda K.K@(\mathtt{n2c}(\tau_e)@[\![[v/x]e^N]\!]_N)$ (I.H. about $[\![\cdot]\!]_N$)
  $= [\![[v/x]e^N]\!]_C$ (def. of $[\![\cdot]\!]_C$)

  a is N or C according to the annotation of the free variable x.

- case t^N

  We can prove similarly to the above e^N case.

□

C Proof of Static Reduction Lemma

Lemma 3 (Static Reduction) *For any closed expression e with consistent annotation and any closed value K,*

$$[\![e]\!]_N \xrightarrow{*} e{:}$$
$$[\![e]\!]_C@K \xrightarrow{*} e{:}K$$

Proof.

Proof by mutual structural induction on e, firstly taking the $[\![\cdot]\!]_N$ for the N annotated expressions and $[\![\cdot]\!]_C$ for the C annotated expressions.

For $[\![\cdot]\!]_N$ cases,

- case v

  $[\![v]\!]_N$
  $= \Psi_N(v)$ (def. of $[\![\cdot]\!]_N$)
  $= v{:}$ (def. of :)

- case $(v_1@v_2)^N$

  $[\![(v_1@v_2)^N]\!]_N$
  $= [\![v_1]\!]_N@[\![v_2]\!]_N$ (def. of $[\![\cdot]\!]_N$)
  $= \Psi_N(v_1)@\Psi_N(v_2)$ (def. of $[\![\cdot]\!]_N$)
  $= (v_1@v_2)^N{:}$ (def. of :)

- case $(v@e)^N$

  $[\![(v@e)^N]\!]_N$
  $= [\![v]\!]_N@[\![e]\!]_N$ (def. of $[\![\cdot]\!]_N$)
  $= (\Psi_N(v))@[\![e]\!]_N$ (def. of $[\![\cdot]\!]_N$)
  $\xrightarrow{*} (\Psi_N(v))@(e{:})$ (I.H.)
  $= (v@e)^N{:}$ (def. of :)

- case $(e_1@e_2)^N$, $(\mathtt{con}\,\kappa\, v)^N$, $(\mathtt{con}\,\kappa\, e)^N$, $(\mathtt{decon}\, v)^N$, $(\mathtt{decon}\, e)^N$, $(\mathtt{case}\, v\, \kappa\, e_1\, e_2)^N$, $(\mathtt{case}\, e_1\, \kappa\, e_2\, e_3)^N$, or e^N

  We can prove similarly to one of the above two cases - $(v_1@v_2)^N$, $(v@e)^N$.

- case e^C

  $[\![e^C]\!]_N$
  $= [\![e^C]\!]_C@\mathtt{c2n}(\tau_e)$ (def. of $[\![\cdot]\!]_N$)
  $\xrightarrow{*} [\![e^C]\!]_C{:}\mathtt{c2n}(\tau_e)$ (I.H. about $[\![\cdot]\!]_C$)
  $= e^C{:}$ (def. of :)

- case t^C

  We can prove similarly to the above e^C case.

For C-annotated expressions and $[\![\cdot]\!]_C$ cases,

- case v

  $[\![v]\!]_C@K$
  $\to K@\Psi_C(v)$ (def. of $[\![\cdot]\!]_C$)
  $= v{:}K$ (def. of $:K$)

- case $(v_1@v_2)^C$

  $[\![(v_1@v_2)^C]\!]_C@K$
  $\to [\![v_1]\!]_C@(\lambda f.[\![v_2]\!]_C@(\lambda v.f@v@K))$ (def. of $[\![\cdot]\!]_C$)
  $= (\lambda K.K@\Psi_C(v_1))@(\lambda f.[\![v_2]\!]_C@(\lambda v.f@v@K))$ (def. of $[\![\cdot]\!]_C$)
  $\xrightarrow{+} [\![v_2]\!]_C@(\lambda v.\Psi_C(v_1)@v@K)$
  $= (\lambda K.K@\Psi_C(v_2))@(\lambda v.\Psi_C(v_1)@v@K)$ (def. of $[\![\cdot]\!]_C$)
  $\xrightarrow{+} \Psi_C(v_1)@\Psi_C(v_2)@K$
  $= (v_1@v_2)^C{:}K$ (def. of $:K$)

- case $(v@e)^C$

  $[\![(v@e)^C]\!]_C@K$
  $\to$ $[\![v]\!]_C@(\lambda f.[\![e]\!]_C@(\lambda v.f@v@K))$ (def. of $[\![\cdot]\!]_C$)
  $=$ $(\lambda K.K@\Psi_C(v))@(\lambda f.[\![e]\!]_C@(\lambda v.f@v@K))$ (def. of $[\![\cdot]\!]_C$)
  $\xrightarrow{+}$ $[\![e]\!]_C@(\lambda v.\Psi_C(v)@v@K)$
  $\xrightarrow{*}$ $e{:}\lambda v.\Psi_C(v)@v@K$ (I.H.)
  $=$ $(v@e)^C{:}K$ (def. of $:K$)

- case $(e_1@e_2)^C$, $(\mathtt{con}\ \kappa\ v)^C$, $(\mathtt{con}\ \kappa\ e)^C$, $(\mathtt{decon}\ v)^C$, $(\mathtt{decon}\ e)^C$, $(\mathtt{case}\ v\ \kappa\ e_1\ e_2)^C$, $(\mathtt{case}\ e_1\ \kappa\ e_2\ e_3)^C$, or$e^C$

  We can prove similarly to one of the above two cases - $(v_1@v_2)^C$, $(v@e)^C$.

- case e^N

  $[\![e^N]\!]_C@K$
  $\to$ $K@(\mathtt{n2c}(\tau_e)@[\![e^N]\!]_N)$ (def. of $[\![\cdot]\!]_C$)
  $\xrightarrow{*}$ $K@(\mathtt{n2c}(\tau_e)@(e^N{:}))$ (I.H. about $[\![\cdot]\!]_N$)
  $=$ $e^N{:}K$ (def. of $:K$)

- case t^N

  We can prove similarly to the above e^N case.

□

D Proof of Single Step Simulation Lemma

Lemma 4 (Single Step Simulation) *For any closed expression e with consistent annotations and any closed value K,*

$$e \to e' \implies e{:} \xrightarrow{+} e'{:} \quad \textit{and} \quad e{:}K \xrightarrow{+} e'{:}K$$

Proof.

Proof by mutual structural induction on e, firstly taking $:$ for the N annotated expressions and $:K$ for the C annotated expressions.

For $:$ cases,

- $((\lambda x.e)^l@v)^N \to [v^{N\overline{l}}/x]e^{lN}$

  Prove this case by induction on the length of l.

  - $((\lambda x.e)^N@v)^N \to [v^{NN}/x]e^{NN}$

    By the consistency, free variable x in e should be annotated as N.

    $((\lambda x.e)^N@v)^N{:}$
    $=$ $\Psi_N((\lambda x.e)^N)@(\Psi_N(v))$ (def. of $:$)
    $=$ $(\lambda x.[\![e]\!]_N)@(\Psi_N(v))$ (def. of Ψ_N)
    $\to$ $[\Psi_N(v)/x][\![e]\!]_N$
    $=$ $[\Psi_N(v^{NN})/x][\![e^{NN}]\!]_N$ (def. of Ψ_N and $[\![\cdot]\!]_N$)
    $=$ $[\![[v^{NN}/x]e^{NN}]\!]_N$ (Lemma 2)
    $\xrightarrow{*}$ $[v^{NN}/x]e^{NN}{:}$ (Lemma 3)

  - $((\lambda x.e)^C@v)^N \to [v^{NT}/x]e^{CN}$

    By the consistency, free variable x in e should be annotated as C. And $\lambda x.e$'s type should be $\tau_1 \to \tau_2$.

    $((\lambda x.e)^C@v)^N{:}$
    $=$ $\Psi_N((\lambda x.e)^C)@\Psi_N(v)$ (def. of $:$)
    $=$ $(\lambda x.\Psi_C((\lambda x.e)^C)@(\mathtt{n2c}(\tau_1)@x)@\mathtt{c2n}(\tau_2))@\Psi_N(v^N)$ (def. of Ψ_N)
    $\to$ $\Psi_C((\lambda x.e)^C)@(\mathtt{n2c}(\tau_1)@\Psi_N(v^N))@\mathtt{c2n}(\tau_2)$
    $\xrightarrow{*}$ $\Psi_C((\lambda x.e)^C)@\Psi_C(v^N)@\mathtt{c2n}(\tau_2)$ (def. of Ψ_C)
    $=$ $(\lambda x.[\![e]\!]_C)@\Psi_C(v^N)@\mathtt{c2n}(\tau_2)$ (def. of Ψ_C)
    $\to$ $([\Psi_C(v^N)/x][\![e]\!]_C)@\mathtt{c2n}(\tau_2)$
    $=$ $([\Psi_C(v^{NC})/x][\![e^C]\!]_C)@\mathtt{c2n}(\tau_2)$ (def. of Ψ_C and $[\![\cdot]\!]_C$)
    $=$ $[\![[v^{NC}/x]e^C]\!]_C@\mathtt{c2n}(\tau_2)$ (Lemma 2)
    $=$ $[\![[v^{NC}/x]e^C]\!]_N$ (def. of $[\![\cdot]\!]_N$)
    $\xrightarrow{*}$ $[v^{NC}/x]e^C{:}$ (Lemma 3)
    $=$ $[v^{NC}/x]e^{CN}{:}$ (def. of $:$)

  - $((\lambda x.e)^{lN}@v)^N \to [v^{NN\overline{l}}/x]e^{lNN}$

    $((\lambda x.e)^{lN}@v)^N{:}$
    $=$ $\Psi_N((\lambda x.e)^{lN})@(\Psi_N(v))$ (def. of $:$)
    $=$ $\Psi_N((\lambda x.e)^l)@(\Psi_N(v^N))$ (def. of Ψ_N)
    $=$ $((\lambda x.e)^l@v^N)^N{:}$ (def. of $:$)
    $\xrightarrow{+}$ $[v^{NN\overline{l}}/x]e^{lN}{:}$ (I.H. about the length of l)
    $=$ $[v^{NN\overline{l}}/x]e^{lNN}{:}$ (def. of $:$)

  - $((\lambda x.e)^{lT}@v)^N \to [v^{NT\overline{l}}/x]e^{lTN}$

    $\lambda x.e$'s type should be $\tau_1 \to \tau_2$.

    $((\lambda x.e)^{lT}@v)^N{:}$
    $=$ $\Psi_N((\lambda x.e)^{lT})@(\Psi_N(v))$ (def. of $:$)
    $=$ $(\lambda x.\Psi_C((\lambda x.e)^{lT})@(\mathtt{n2c}(\tau_1)@x)@\mathtt{c2n}(\tau_2))@(\Psi_N(v^N))$ (def. of Ψ_N)
    $\to$ $\Psi_C((\lambda x.e)^{lT})@(\mathtt{n2c}(\tau_1)@\Psi_N(v^N))@\mathtt{c2n}(\tau_2)$
    $\xrightarrow{*}$ $\Psi_C((\lambda x.e)^l)@\Psi_C(v^N)@\mathtt{c2n}(\tau_2)$ (def. of Ψ_C)
    $=$ $((\lambda x.e)^l@v^N)^C{:}\mathtt{c2n}(\tau_2)$ (def. of $:K$)
    $\xrightarrow{+}$ $([v^{NT\overline{l}}/x]e^{lT}){:}\mathtt{c2n}(\tau_2)$ (I.H. about the length of l)
    $=$ $([v^{NT\overline{l}}/x]e^{lT}){:}$
    $=$ $([v^{NT\overline{l}}/x]e^{lTN}){:}$ (def. of $:$)

  By induction, $(((\lambda x.e)^l@v)^N{:}) \xrightarrow{+} ([v^{N\overline{l}}/x]e^{lN}{:})$

- $((\mathtt{fix}\ f\ \lambda x.e)^{al}@v)^N \to [(\mathtt{fix}\ f\ \lambda x.e)^a/f][v^{N\overline{l}a}/x]e^{alN}$ (head of l is a)

  We can prove similarly to the above case except that f should annotated as the head of l.

- $(\mathtt{con}\ \kappa\ v)^N \to (\kappa{\cdot}v)^N$

  $(\mathtt{con}\ \kappa\ v)^N{:}$
  $=$ $\mathtt{con}\ \kappa\ (\Psi_N(v))$ (def. of $:$)
  $\to$ $\kappa{\cdot}(\Psi_N(v))$
  $=$ $\Psi_N(\kappa{\cdot}v)$ (def. of Ψ_N)
  $=$ $(\kappa{\cdot}v){:}$ (def. of $:$)
  $=$ $(\kappa{\cdot}v)^N{:}$ (def. of $:$)

- $(\mathtt{decon}\ (\kappa{\cdot}v)^l)^N \to v^{lN}$

$(\mathtt{decon}\ (\kappa{\cdot}v)^l)^N$:
$= \quad \mathtt{decon}\ (\Psi_N((\kappa{\cdot}v)^l))$ (def. of :)
$= \quad \mathtt{decon}\ (\kappa{\cdot}\Psi_N(v^l))$ (Property 1)
$\to \quad \Psi_N(v^l)$
$= \quad v^l$: (def. of :)
$= \quad v^{lN}$: (def. of :)

- $(\mathtt{case}\ (\kappa{\cdot}v)^l\ \kappa\ e_1\ e_2)^N \to e_1^N$

  $(\mathtt{case}\ (\kappa{\cdot}v)^l\ \kappa\ e_1\ e_2)^N$:
  $= \quad \mathtt{case}\ (\Psi_N((\kappa{\cdot}v)^l))\ \kappa\ [\![e_1]\!]_N\ [\![e_2]\!]_N$ (def. of :)
  $= \quad \mathtt{case}\ (\kappa{\cdot}\Psi_N(v^l))\ \kappa\ [\![e_1]\!]_N\ [\![e_2]\!]_N$ (Property 1)
  $\to \quad [\![e_1]\!]_N$
  $= \quad [\![e_1^N]\!]_N$ (def. of $[\![\cdot]\!]_N$)
  $\xrightarrow{*} \quad e_1^N$: (Lemma 3)

- $(\mathtt{case}\ (\kappa{\cdot}v)^l\ \kappa'\ e_1\ e_2)^N \to e_2^N \quad (\kappa \neq \kappa')$

  We can prove similarly to the above case.

- $(e_1@e_2)^N \to (e_1'@e_2)^N \qquad (e_1 \to e_1')$

  $(e_1@e_2)^N$:
  $= \quad (e_1{:})@[\![e_2]\!]_N$ (def. of :)
  $\xrightarrow{+} \quad (e_1'{:})@[\![e_2]\!]_N$ (I.H.)
  $= \quad (e_1'@e_2)^N$: (def. of :)

- $(v@e)^N \to (v@e')^N$, $(\mathtt{con}\ \kappa\ e)^N \to (\mathtt{con}\ \kappa\ e')^N$,
  $(\mathtt{decon}\ e)^N \to (\mathtt{decon}\ e')^N$,
  $(\mathtt{case}\ e_1\ \kappa\ e_2\ e_3)^N \to (\mathtt{case}\ e_1'\ \kappa\ e_2\ e_3)^N$,

  We can prove similarly to the above case.

- $e^N \to e'^N$

  Then $e \to e'$.
  e^N:
  $= \quad e$: (def. of :)
  $\xrightarrow{+} \quad e'$: (I.H.)
  $= \quad e'^N$: (def. of :)

- $e^C \to e'^C$

  e^C:
  $= \quad e^C{:}\mathtt{c2n}(\tau_e)$ (def. of :)
  $\xrightarrow{+} \quad e'^C{:}\mathtt{c2n}(\tau_e)$ (I.H. about $:K$)
  $= \quad e'^C$: (def. of :)

- $t^C \to t'^C$

  We can prove similarly to the above e^C case.

For $:K$ cases,

- $((\lambda x.e)^l@v)^C \to [v^{C\overline{l}}/x]e^{lC}$

  Prove this case by induction on the length of l.

  - $((\lambda x.e)^C@v)^C \to [v^{CC}/x]e^{CC}$

    By the consistency, free variable x in e should be annotated as C.
    $((\lambda x.e)^C@v)^C{:}K$
    $= \quad \Psi_C((\lambda x.e)^C)@\Psi_C(v)@K$ (def. of $:K$)
    $= \quad (\lambda x.[\![e]\!]_C)@\Psi_C(v)@K$ (def. of Ψ_C)
    $\to \quad ([\Psi_C(v)/x][\![e]\!]_C)@K$
    $= \quad ([\Psi_C(v^{CC})/x][\![e^{CC}]\!]_C)@K$ (def. of Ψ_C and $[\![\cdot]\!]_C$)
    $= \quad ([\![[v^{CC}/x]e^{CC}]\!]_C)@K$ (Lemma 2)
    $\xrightarrow{*} \quad [v^{CC}/x]e^{CC}{:}K$ (Lemma 3)

  - $((\lambda x.e)^N@v)^C \to [v^{CN}/x]e^{NC}$

    By the consistency, free variable x in e should be annotated as N. And $\lambda x.e$'s type should be $\tau_1 \to \tau_2$.
    $((\lambda x.e)^N@v)^C{:}K$
    $= \quad \Psi_C((\lambda x.e)^N)@\Psi_C(v)@K$ (def. of $:K$)
    $= \quad (\lambda x.\lambda K.K@(\mathtt{n2c}(\tau_2)@(\Psi_N((\lambda x.e)^N)$
    $\qquad @(\mathtt{c2n}(\tau_1)@x))))@\Psi_C(v^C)@K$ (def. of Ψ_C)
    $\xrightarrow{+} \quad K@(\mathtt{n2c}(\tau_2)@(\Psi_N((\lambda x.e)^N)$
    $\qquad @(\mathtt{c2n}(\tau_1)@\Psi_C(v^C))))$
    $\xrightarrow{*} \quad K@(\mathtt{n2c}(\tau_2)@(\Psi_N((\lambda x.e)^N)@\Psi_N(v^C)))$ (def. of Ψ_N)
    $= \quad K@(\mathtt{n2c}(\tau_2)@((\lambda x.[\![e]\!]_N)@\Psi_N(v^C)))$ (def. of Ψ_N)
    $\to \quad K@(\mathtt{n2c}(\tau_2)@([\Psi_N(v^C)/x][\![e]\!]_N))$
    $= \quad K@(\mathtt{n2c}(\tau_2)@([\Psi_N(v^{CN})/x][\![e^N]\!]_N))$ (def. of Ψ_N and $[\![\cdot]\!]_N$)
    $= \quad K@(\mathtt{n2c}(\tau_2)@([\![[v^{CN}/x]e^N]\!]_N))$(Lemma 2)
    $\xrightarrow{*} \quad K@(\mathtt{n2c}(\tau_2)@([v^{CN}/x]e^N{:}))$ (Lemma 3)
    $= \quad [v^{CN}/x]e^N{:}K$ (def. of $:K$)
    $= \quad [v^{CN}/x]e^{NC}{:}K$ (def. of $:K$)

  - $((\lambda x.e)^{lC}@v)^C \to [v^{CC\overline{l}}/x]e^{lCC}$

    $((\lambda x.e)^{lC}@v)^C{:}K$
    $= \quad \Psi_C((\lambda x.e)^{lC})@\Psi_C(v)@K$ (def. of $:K$)
    $= \quad \Psi_C((\lambda x.e)^l)@\Psi_C(v^C)@K$ (def. of Ψ_C)
    $= \quad ((\lambda x.e)^l@v^C)^C{:}K$ (def. of $:K$)
    $\xrightarrow{+} \quad [v^{CC\overline{l}}/x]e^{lC}{:}K$ (I.H. about the length of l)
    $= \quad [v^{CC\overline{l}}/x]e^{lCC}{:}K$ (def. of $:K$)

  - $((\lambda x.e)^{lN}@v)^C \to [v^{CN\overline{l}}/x]e^{lNC}$

    $\lambda x.e$'s type should be $\tau_1 \to \tau_2$.
    $((\lambda x.e)^{lN}@v)^C{:}K$
    $= \quad \Psi_C((\lambda x.e)^{lN})@\Psi_C(v)@K$ (def. of $:K$)
    $= \quad (\lambda x.\lambda K.K@(\mathtt{n2c}(\tau_2)@(\Psi_N((\lambda x.e)^{lN})$
    $\qquad @(\mathtt{c2n}(\tau_1)@x))))@\Psi_C(v)@K$ (def. of $\mathtt{n2c}()$)
    $\xrightarrow{+} \quad K@(\mathtt{n2c}(\tau_2)@(\Psi_N((\lambda x.e)^{lN})$
    $\qquad @(\mathtt{c2n}(\tau_1)@\Psi_C(v))))$
    $\xrightarrow{*} \quad K@(\mathtt{n2c}(\tau_2)@(\Psi_N((\lambda x.e)^l)@\Psi_N(v^C)))$ (def. of Ψ_N)
    $= \quad K@(\mathtt{n2c}(\tau_2)@(((\lambda x.e)^l@v^C)^N{:}))$(def. of :)
    $= \quad ((\lambda x.e)^l@v^C)^N{:}K$ (def. of $:K$)
    $\xrightarrow{+} \quad [v^{CN\overline{l}}/x]e^{lN}{:}K$(I.H. about the length of l)
    $= \quad [v^{CN\overline{l}}/x]e^{lNC}{:}K$ (def. of $:K$)

  By induction, $((\lambda x.e)^l@v)^C{:}K \xrightarrow{+} [v^{C\overline{l}}/x]e^{lC}{:}K$

- $((\mathtt{fix}\ f\ \lambda x.e)^l@v)^C \to [(\mathtt{fix}\ f\ \lambda x.e)^a/f][v^{C\overline{l}}/x]e^{lC}$
  (head of l is a)

  We can prove similarly to the above case except that f should annotated as the head of l.

- $(\mathtt{con}\ \kappa\ v)^C \to (\kappa{\cdot}v)^C$

$(\mathtt{con}\ \kappa\ v)^C{:}K$
$= \quad K@(\mathtt{con}\ \kappa\ \Psi_C(v))$ (def. of $:K$)
$\to \quad K@(\kappa{\cdot}\Psi_C(v))$
$= \quad K@(\Psi_C((\kappa{\cdot}v)^C))$ (def. of Ψ_C)
$= \quad (\kappa{\cdot}v)^C{:}K$ (def. of $:K$)

- $(\mathtt{decon}\ (\kappa{\cdot}v)^l)^C \to v^{lC}$

$(\mathtt{decon}\ (\kappa{\cdot}v)^l)^C{:}K$
$= \quad K@(\mathtt{decon}\ \Psi_C((\kappa{\cdot}v)^l))$ (def. of $:K$)
$= \quad K@(\mathtt{decon}\ (\kappa{\cdot}\Psi_C(v^l)))$ (Property 1)
$\to \quad K@\Psi_C(v^l)$
$= \quad v^l{:}K$ (def. of $:K$)
$= \quad v^{lC}{:}K$ (def. of $:K$)

- $(\mathtt{case}\ (\kappa{\cdot}v)^l\ \kappa\ e_1\ e_2)^C \to e_1^C$

$(\mathtt{case}\ (\kappa{\cdot}v)^l\ \kappa\ e_1\ e_2)^C{:}K$
$= \quad \mathtt{case}\ (\Psi_C((\kappa{\cdot}v)^l))\ \kappa\ ([\![e_1]\!]_C@K)\ ([\![e_2]\!]_C@K)$ (def. of :)
$= \quad \mathtt{case}\ (\kappa{\cdot}\Psi_C(v^l))\ \kappa\ ([\![e_1]\!]_C@K)\ ([\![e_2]\!]_C@K)$ (Property 1)
$\to \quad [\![e_1]\!]_C@K$
$= \quad [\![e_1^C]\!]_C@K$ (def. of $[\![\cdot]\!]_C$)
$\overset{*}{\to} \quad e_1^C{:}K$ (Lemma 3)

- $(\mathtt{case}\ (\kappa{\cdot}v)^l\ \kappa'\ e_1\ e_2)^C \to e_1^C \quad (\kappa \neq \kappa')$

We can prove similarly to the above case.

- $(e_1@e_2)^C \to (e_1'@e_2)^C \qquad (e_1 \to e_1')$

$(e_1@e_2)^C{:}K$
$= \quad e_1{:}\lambda f.[\![e_2]\!]_C@(\lambda v.f@v@K)$ (def. of :)
$\overset{+}{\to} \quad e_1'{:}\lambda f.[\![e_2]\!]_C@(\lambda v.f@v@K)$ (I.H.)
$= \quad (e_1'@e_2)^C{:}K$ (def. of :)

- $(v@e)^C \to (v@e')^C$, $(\mathtt{con}\ \kappa\ e)^C \to (\mathtt{con}\ \kappa\ e')^C$,
$(\mathtt{decon}\ e)^C \to (\mathtt{decon}\ e')^C$,
$(\mathtt{case}\ e_1\ \kappa\ e_2\ e_3)^C \to (\mathtt{case}\ e_1'\ \kappa\ e_2\ e_3)^C$,

We can prove similarly to the above case.

- $e^C \to e'^C$

Then $e \to e'$.
$e^C{:}K$
$= \quad e{:}K$ (def. of $:K$)
$\overset{+}{\to} \quad e'{:}K$ (I.H.)
$= \quad e'^C{:}K$ (def. of $:K$)

- $e^N \to e'^N$

$e^N{:}K$
$= \quad K@(\mathtt{n2c}(\tau_e)@(e^N{:}))$ (def. of $:K$)
$\overset{+}{\to} \quad K@(\mathtt{n2c}(\tau_e)@(e'^N{:}))$ (I.H.)
$= \quad e'^N{:}K$ (def. of $:K$)

- $t^N \to t'^N$

We can prove similarly to the above e^N case.

□

Comparing Control Constructs by Typing Double-barrelled CPS Transforms

Hayo Thielecke*
School of Computer Science
University of Birmingham
Birmingham, United Kingdom
H.Thielecke@cs.bham.ac.uk

ABSTRACT

We investigate continuation-passing style transforms that pass two continuations. Altering a single variable in the translation of λ-abstraction gives rise to different control operators: first-class continuations; dynamic control; and (depending on a further choice of a variable) either the `return` statement of C; or Landin's **J**-operator. In each case there is an associated simple typing. For those constructs that allow upward continuations, the typing is classical, for the others it remains intuitionistic, giving a clean distinction independent of syntactic details.

1. INTRODUCTION

Control operators come in bewildering variety. Sometimes the same term is used for distinct constructs, as with `catch` in early Scheme or `throw` in Standard ML of New Jersey, which are very unlike the `catch` and `throw` in Lisp whose names they borrow. On the other hand, this Lisp `catch` is fundamentally similar to exceptions despite their dissimilar and much more ornate appearance.

Fortunately it is sometimes possible to glean some high-level "logical" view of a programming language construct by looking only at its type. Specifically for control operations, Griffin's discovery [3] that `call/cc` and related operators can be ascribed classical types gives us the fundamental distinction between languages that have such classical types and those that do not, even though they may still enjoy some form of control. This approach complements comparisons based on contextual equivalences [10, 14].

Such a comparison would be difficult unless we blot out complication. In particular, exceptions are typically tied in with other, fairly complicated features of the language which are not relevant to control as such: in ML with the datatype mechanism, in Java with object-orientation. In order to simplify, we first strip down control operators to the bare essentials of labelling and jumping, so that there are no longer any distracting syntactic differences between them. The grammar of our toy language is uniformly this:

$$M ::= x \mid \lambda x.M \mid MM \mid \texttt{here}\, M \mid \texttt{go}\, M.$$

The intended meaning of `here` is that it labels a "program point" or expression without actually naming any particular label—just uttering the demonstrative "here", as it were. Correspondingly, `go` jumps to a place specified by a `here`, without naming the "to" of a `goto`.

Despite the simplicity of the language, there is still scope for variation: not by adding bells and whistles to `here` and `go`, but by varying the meaning of λ-abstraction. Its impact can be seen quite clearly in the distinction between exceptions and first-class continuations. The difference between them is as much due to the meaning of λ-abstraction as due to the control operators themselves, since λ-abstraction determines what is statically put into a closure and what is passed dynamically. Readers familiar with, say, Scheme implementations will perhaps not be surprised about the impact of what becomes part of a closure. But the point of this paper is twofold:

- small variations in the meaning of λ completely change the meaning of our control operators;
- we can see these differences at an abstract, logical level, without delving into the innards of interpreters.

We give meaning to the λ-calculus enriched with `here` and `go` by means of continuations in Section 2, examining in Sections 3–5 how variations on λ-abstraction determine what kind of control operations `here` and `go` represent. For each of these variations we present a simple typing, which agrees with the transform (Section 6). We conclude by explaining the significance of these typings in terms of classical and intuitionistic logic (Section 7).

2. DOUBLE-BARRELLED CPS

Our starting point is a continuation-passing style (CPS) transform. This transform is double-barrelled in the sense that it always passes *two* continuations. Hence the clauses start with $\lambda kq.\ldots$ instead of $\lambda k.\ldots$. Other than that, this CPS transform is in fact a very mild variation on the usual call-by-value one [8]. As indicated by the $\boxed{?}$, we leave one variable, the extra continuation passed to the body of a λ-abstraction, unspecified.

*Partially supported by the EPSRC

To appear in *The Third ACM SIGPLAN Workshop on Continuations* (*CW* '01), 16 January 2001, London, United Kingdom.

$$
\begin{array}{rcl}
[\![x]\!] & = & \lambda kq.kx \\
[\![\lambda_? x.M]\!] & = & \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{?}) \\
[\![MN]\!] & = & \lambda kq.[\![M]\!](\lambda m.[\![N]\!](\lambda n.mnkq)q)q \\
[\![\texttt{here}\, M]\!] & = & \lambda kq.[\![M]\!]kk \\
[\![\texttt{go}\, M]\!] & = & \lambda kq.[\![M]\!]qq
\end{array}
$$

The extra continuation may be seen as a jump continuation, in that its manipulation accounts for the labelling and jumping. This is done symmetrically: `here` makes the jump continuation the same as the current one k, whereas `go` sets the current continuation of its argument to the jump continuation q. The clauses for variables and applications do not interact with the additional jump continuation: the former ignores it, while the latter merely distributes it into the operator, the operand and the function call.

Only in the clause for λ-abstraction do we face a design decision. Depending on which continuation (static s, dynamic d, or the return continuation r) we fill in for "?" in the clause for λ, there are three different flavours of λ-abstraction.

$$
\begin{array}{rcl}
[\![\lambda_{\mathsf{s}} x.M]\!] & = & \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{s}) \\
[\![\lambda_{\mathsf{d}} x.M]\!] & = & \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{d}) \\
[\![\lambda_{\mathsf{r}} x.M]\!] & = & \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{r})
\end{array}
$$

The lambdas are subscripted to distinguish them, and the box around the last variable is meant to highlight that this is the crucial difference between the transforms. Formally there is also a fourth possibility, the outer continuation k, but this seems less meaningful and would not fit into simple typing.

For all choices of λ, the operation `go` is always a jump to a place specified by a `here`. For example, for any M, the term $\texttt{here}\,((\lambda x.M)(\texttt{go}\,N))$ should be equivalent to N, as the `go` jumps past the M. But in more involved examples than this, there may be different choices *where* `go` can go to among several occurrences of `here`. In particular, if s is passed as the second continuation argument to M in the transform of $\lambda x.M$, then a `go` in M will refer to the `here` that was in scope at the point of definition (unless there is an intervening `here`, just as one binding of a variable x can shadow another). By contrast, if d is passed to M in $\lambda x.M$, then the `here` that is in scope at the point of definition is forgotten; instead `go` in M will refer to the `here` that is in scope at the point of call when $\lambda x.M$ is applied to an argument. In fact, depending upon the choice of variable in the clause for λ as above, `here` and `go` give rise to different control operations:

- first-class continuations like those given by `call/cc` in Scheme [4];
- dynamic control in the sense of Lisp, and typeable in a way reminiscent of checked exceptions;
- a `return`-operation, which can be refined into the **J**-operator invented by Landin in 1965 and ancestral to `call/cc` [4, 6, 7, 13].

We examine these constructs in turn, giving a simple type system in each case. An unusual feature of these type judgements is that, because we have two continuations, there are two types in the succedent on the right of the turnstile, as in

$$\Gamma \vdash M : A, B.$$

The first type on the right accounts for the case that the term returns a value; it corresponds to the current continuation. The second type accounts for the jump continuation. In logical terms. the comma on the right may be read as a disjunction. It makes a big difference whether this disjunction is classical or intuitionistic. That is our main criterion of comparing and contrasting the control constructs.

3. FIRST-CLASS CONTINUATIONS

The first choice of which continuation to pass to the body of a function is arguably the cleanest. Passing the static continuation s gives control the same static binding as ordinary λ-calculus variables. In the static case, the transform is this:

$$
\begin{array}{rcl}
[\![x]\!] & = & \lambda kq.kx \\
[\![\lambda_{\mathsf{s}} x.M]\!] & = & \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{s}) \\
[\![MN]\!] & = & \lambda kq.[\![M]\!](\lambda m.[\![N]\!](\lambda n.mnkq)q)q \\
[\![\texttt{here}\, M]\!] & = & \lambda kq.[\![M]\!]kk \\
[\![\texttt{go}\, M]\!] & = & \lambda kq.[\![M]\!]qq
\end{array}
$$

We type our source language with `here` and `go` as in Figure 1.

In logical terms, both `here` and `go` are a combined right weakening and contraction. By themselves, weakening and contraction do not amount to much; but it is the combination with the rule for $\rightarrow$-introduction that makes the calculus "classical", in the sense that there are terms whose types are propositions of classical, but not of intuitionistic, minimal logic.

To see how $\rightarrow$-introduction gives classical types, consider λ-abstracting over `go`.

$$\frac{x : A \vdash_{\mathsf{s}} \texttt{go}\, x : A, B}{\vdash_{\mathsf{s}} \lambda_{\mathsf{s}} x.\texttt{go}\, x : A \rightarrow B, A}$$

If we read the comma as "or", and $A \rightarrow B$ for arbitrary B as "not A", then this judgement asserts the classical excluded middle, "not A or A". We build on the classical type of $\lambda_{\mathsf{s}} x.\texttt{go}\, x$ for another canonical example: Scheme's `call-with-current-continuation` (`call/cc` for short) operator [4]. It is syntactic sugar in terms of static `here` and `go`:

$$\texttt{call/cc} = \lambda_{\mathsf{s}} f.(\texttt{here}\,(f\,(\lambda_{\mathsf{s}} x.\texttt{go}\, x))).$$

As one would expect [3], the type of `call/cc` is Peirce's law "if not A implies A, then A". We derive the judgement

$$\vdash_{\mathsf{s}} \lambda_{\mathsf{s}} f.(\texttt{here}\,(f\,(\lambda_{\mathsf{s}} x.\texttt{go}\, x))) : ((A \rightarrow B) \rightarrow A) \rightarrow A, C$$

in Figure 2.

As a further example, we show that right exchange is admissible. Let Γ be any context, and assume we have

$$\Gamma \vdash_{\mathsf{s}} M : A, B.$$

Then, by the derivation in Figure 3, we also have

$$\Gamma \vdash_{\mathsf{s}} M : B, A.$$

In the typing of `call/cc`, a `go` is (at least potentially, depending on f) exported from its enclosing `here`. Conversely,

Figure 1: Typing for static `here` and `go`

$$\frac{}{\Gamma, x : A, \Gamma' \vdash_{\mathsf{s}} x : A, C}$$

$$\frac{\Gamma \vdash_{\mathsf{s}} M : B, B}{\Gamma \vdash_{\mathsf{s}} \mathtt{here}\, M : B, C} \qquad \frac{\Gamma \vdash_{\mathsf{s}} M : B, B}{\Gamma \vdash_{\mathsf{s}} \mathtt{go}\, M : C, B}$$

$$\frac{\Gamma, x : A \vdash_{\mathsf{s}} M : B, C}{\Gamma \vdash_{\mathsf{s}} \lambda_{\mathsf{s}} x.M : A \to B, C} \qquad \frac{\Gamma \vdash_{\mathsf{s}} M : A \to B, C \qquad \Gamma \vdash_{\mathsf{s}} N : A, C}{\Gamma \vdash_{\mathsf{s}} MN : B, C}$$

Figure 2: Derivation of `call/cc` in the static case

$$\dfrac{\dfrac{}{f : (A \to B) \to A \vdash_{\mathsf{s}} f : (A \to B) \to A, A} \qquad \dfrac{\dfrac{\dfrac{}{f : (A \to B) \to A, x : A \vdash_{\mathsf{s}} x : A, A}}{f : (A \to B) \to A, x : A \vdash_{\mathsf{s}} \mathtt{go}\, x : B, A}}{f : (A \to B) \to A \vdash_{\mathsf{s}} \lambda_{\mathsf{s}} x.\mathtt{go}\, x : A \to B, A}}{\dfrac{\dfrac{f : (A \to B) \to A \vdash_{\mathsf{s}} (f\, (\lambda_{\mathsf{s}} x.\mathtt{go}\, x)) : A, A}{f : (A \to B) \to A \vdash_{\mathsf{s}} \mathtt{here}\, (f\, (\lambda_{\mathsf{s}} x.\mathtt{go}\, x)) : A, C}}{\vdash_{\mathsf{s}} \lambda_{\mathsf{s}} f.(\mathtt{here}\, (f\, (\lambda_{\mathsf{s}} x.\mathtt{go}\, x))) : ((A \to B) \to A) \to A, C}}$$

Figure 3: Derivation of right exchange in the static case

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, f : A \to B \vdash_{\mathsf{s}} f : A \to B, B} \qquad \Gamma, f : A \to B \vdash_{\mathsf{s}} M : A, B}{\Gamma, f : A \to B \vdash_{\mathsf{s}} (f\, M) : B, B}}{\Gamma, f : A \to B \vdash_{\mathsf{s}} \mathtt{here}\, (f\, M) : B, A}}{\Gamma \vdash_{\mathsf{s}} \lambda_{\mathsf{s}} f.\mathtt{here}\, (f\, M) : (A \to B) \to B, A} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma, x : A \vdash_{\mathsf{s}} x : A, A}}{\Gamma, x : A \vdash_{\mathsf{s}} \mathtt{go}\, x : B, A}}{\Gamma \vdash_{\mathsf{s}} \lambda_{\mathsf{s}} x.\mathtt{go}\, x : A \to B, A}}{\Gamma \vdash_{\mathsf{s}} (\lambda_{\mathsf{s}} f.\mathtt{here}\, (f\, M))\, (\lambda_{\mathsf{s}} x.\mathtt{go}\, x) : B, A}$$

in the derivation of right exchange, a `go` is imported into a `here` from without. What makes everything work is static binding.

4. DYNAMIC CONTROL

Next we consider the dynamic version of `here` and `go`. The word "dynamic" is used here in the sense of dynamic binding and dynamic control in Lisp. In the dynamic case, the transform is this:

$$\begin{array}{rcl}
[\![x]\!] & = & \lambda kq.kx \\
[\![\lambda_{\mathsf{d}} x.M]\!] & = & \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{d}) \\
[\![MN]\!] & = & \lambda kq.[\![M]\!](\lambda m.[\![N]\!](\lambda n.mnkq)q)q \\
[\![\mathtt{here}\, M]\!] & = & \lambda kq.[\![M]\!]kk \\
[\![\mathtt{go}\, M]\!] & = & \lambda kq.[\![M]\!]qq
\end{array}$$

In this transform, the jump continuation acts as a handler continuation; since it is passed as an extra argument on each call, the dynamically enclosing handler is chosen. Hence under the dynamic semantics, `here` and `go` become a stripped-down version of Lisp's `catch` and `throw` with only a single catch tag. These `catch` and `throw` operation are themselves a no-frills version of exceptions with only identity handlers. We can think of `here` and `go` as a special case of these more elaborate constructs:

$$\begin{array}{rcl}
\mathtt{here}\, M & \equiv & \mathtt{(catch\ 'e}\ \ M\mathtt{)} \\
\mathtt{go}\, M & \equiv & \mathtt{(throw\ 'e}\ \ M\mathtt{)}
\end{array}$$

Because the additional continuation is administered dynamically, we cannot fit it into our simple typing without annotating the function type. So for dynamic control, we write the function type as $A \to B \vee C$, which should be read as a single operator with the three arguments in mixfix; it is not quite the same as $A \to (B \vee C)$, and neither $\to$ nor $\vee$ exist on their own. This annotated arrow can be seen as an idealization of the Java `throws` clause in method definitions, in that $A \to B \vee C$ could be written as $B(A)$ `throws` C in a more Java-like syntax. A function of type $A \to B \vee C$ may throw things of type C, so it may only be called inside a `here` with the same type. Our typing for the language with dynamic `here` and `go` is presented in Figure 4.

We do not attempt to idealize the ML way of typing exceptions because ML uses a universal type `exn` for exceptions, in effect allowing a carefully delimited area of untypedness into the language. The typing of ML exceptions is therefore much less informative than that of checked exceptions.

Note that `here` and `go` are still the same weakening and contraction hybrid as in the static setting. But here their significance is a completely different one because the $\to$-introduction is coupled with a sort of $\vee$-introduction. To see the difference, recall that in the static setting λ-abstracting over a `go` reifies the jump continuation and thereby, at the type level, gives rise to classical disjunction. This is not possible with the version of λ that gives `go` the dynamic semantics. Consider the following inference:

$$\frac{x : A \vdash_{\mathsf{d}} \mathtt{go}\, x : B, A}{\vdash_{\mathsf{d}} \lambda_{\mathsf{d}} x.\mathtt{go}\, x : A \to B \vee A, C}$$

The C-accepting continuation at the point of definition is not accessible to the `go` inside the λ_{d}. Instead, the `go` refers only to the A-accepting continuation that will be available at the point of call. Far from the excluded middle, the type of $\lambda_{\mathsf{d}} x.\mathtt{go}\, x$ is thus "A implies A or B; or anything".

In the same vein, as a further illustration how fundamentally different the dynamic `here` and `go` are from the static variety, we revisit the term that, in the static setting, gave rise to `call/cc` with its classical type:

$$\lambda f.\mathtt{here}\,(f\,(\lambda x.\mathtt{go}\, x)).$$

Now in the dynamic case, we can only derive an intuitionistic formula as the type of this term:

$$((A \to B \vee A) \to A \vee A) \to A \vee C, D.$$

See Figure 5 for the derivation.

5. RETURN CONTINUATION

Our last choice is passing the return continuation as the extra continuation to the body of a λ-abstraction. So the CPS transform is this:

$$\begin{array}{rcl}
[\![x]\!] & = & \lambda kq.qx \\
[\![\lambda_{\mathsf{r}} x.M]\!] & = & \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{r}) \\
[\![MN]\!] & = & \lambda kq.[\![M]\!](\lambda m.[\![N]\!](\lambda n.mnkq)q)q \\
[\![\mathtt{here}\, M]\!] & = & \lambda kq.[\![M]\!]kk \\
[\![\mathtt{go}\, M]\!] & = & \lambda kq.[\![M]\!]qq
\end{array}$$

This transform grants λ_{r} the additional role of a continuation binder. The original operator for this purpose, `here`, is rendered redundant, since $\mathtt{here}\, M$ is now equivalent to $(\lambda_{\mathsf{r}} x.M)(\lambda_{\mathsf{r}} y.y)$ where x is not free in M. At first sight, binding continuations seems an unusual job for a λ; but it becomes less so if we think of `go` as the `return` statement of C or Java.

5.1 Non-first class return

Because the enclosing λ determines which continuation `go` jumps to with its argument, the `go`-operator has the same effect as a `return` statement. The type of extra continuation assumed by `go` needs to agree with the return type of the nearest enclosing λ:

$$\frac{\Gamma, x : A \vdash_{\mathsf{r}} M : B, B}{\Gamma \vdash_{\mathsf{r}} \lambda_{\mathsf{r}} x.M : A \to B, C}$$

The whole type system for the calculus with λ_{r} is in Figure 6.

The agreement between `go` and the enclosing λ_{r} is comparable with the typing in C, where the expression featuring in a `return` statement must have the return type declared by the enclosing function. For instance, M needs to have type `int` in the definition:

$$\mathtt{int\ f()\{}\ldots\mathtt{return}\ M\mathtt{;}\ldots\mathtt{\}}$$

With λ_{r}, the special form `go` cannot be made into a first-class function. If we try to λ-abstract over $\mathtt{go}\, x$ by writing $\lambda_{\mathsf{r}} x.\mathtt{go}\, x$ then `go` will refer to that λ_{r}.

The failure of λ_{r} to give first-class returning can be seen logically as follows. In order for λ_{r} to be introduced, both types on the right have to be the same:

$$\frac{x : A \vdash_{\mathsf{r}} \mathtt{go}\, x : A, A}{\vdash_{\mathsf{r}} \lambda_{\mathsf{r}} x.\mathtt{go}\, x : A \to A, C}$$

Rather than the classical "not A or A" this asserts merely the intuitionistic "A implies A; or anything".

Figure 4: Typing for dynamic `here` and `go`

$$\frac{}{\Gamma, x : A, \Gamma' \vdash_{\mathsf{d}} x : A, C}$$

$$\frac{\Gamma \vdash_{\mathsf{d}} M : B, B}{\Gamma \vdash_{\mathsf{d}} \mathtt{here}\, M : B, C} \qquad \frac{\Gamma \vdash_{\mathsf{d}} M : B, B}{\Gamma \vdash_{\mathsf{d}} \mathtt{go}\, M : C, B}$$

$$\frac{\Gamma, x : A \vdash_{\mathsf{d}} M : B, C}{\Gamma \vdash_{\mathsf{d}} \lambda_{\mathsf{d}} x.M : A \to B \vee C, D} \qquad \frac{\Gamma \vdash_{\mathsf{d}} M : A \to B \vee C, C \quad \Gamma \vdash_{\mathsf{d}} N : A, C}{\Gamma \vdash_{\mathsf{d}} MN : B, C}$$

Figure 5: A derivation in the dynamic case
Let $\Gamma \equiv f : (A \to B \vee A) \to A \vee A$.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_{\mathsf{d}} f : (A \to B \vee A) \to A \vee A, A} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma, x : A \vdash_{\mathsf{d}} x : A, A}}{\Gamma, x : A \vdash_{\mathsf{d}} \mathtt{go}\, x : B, A}}{\Gamma \vdash_{\mathsf{d}} \lambda_{\mathsf{d}} x.\mathtt{go}\, x : A \to B \vee A, A}}{\Gamma \vdash_{\mathsf{d}} (f\, (\lambda_{\mathsf{d}} x.\mathtt{go}\, x)) : A, A}}{\Gamma \vdash_{\mathsf{d}} \mathtt{here}\, (f\, (\lambda_{\mathsf{d}} x.\mathtt{go}\, x)) : A, C}}{\vdash_{\mathsf{d}} \lambda_{\mathsf{d}} f.\mathtt{here}\, (f\, (\lambda_{\mathsf{d}} x.\mathtt{go}\, x)) : ((A \to B \vee A) \to A \vee A) \to A \vee C, D}$$

Figure 6: Typing for `go` as a `return`-operation

$$\frac{}{\Gamma, x : A, \Gamma' \vdash_{\mathsf{r}} x : A, C} \qquad \frac{\Gamma \vdash_{\mathsf{r}} M : B, B}{\Gamma \vdash_{\mathsf{r}} \mathtt{go}\, M : C, B}$$

$$\frac{\Gamma, x : A \vdash_{\mathsf{r}} M : B, B}{\Gamma \vdash_{\mathsf{r}} \lambda_{\mathsf{r}} x.M : A \to B, C} \qquad \frac{\Gamma \vdash_{\mathsf{r}} M : A \to B, C \quad \Gamma \vdash_{\mathsf{r}} N : A, C}{\Gamma \vdash_{\mathsf{r}} MN : B, C}$$

One has a similar situation in Gnu C, which has both the `return` statement and nested functions, without the ability to refer to the return address of another function. If we admit `go` as a first-class function, it becomes a much more powerful form of control, Landin's **JI**-operator.

5.2 The JI-operator

Keeping the meaning of λ_{r} as a continuation binder, we now consider a control operator **JI** that always refers to the statically enclosing λ_{r}, but which, unlike the special form `go`, is a first-class expression, so that we can pass the return continuation to some other function f by writing $f(\mathbf{JI})$. The CPS of this operator is this:

$$[\![\mathbf{JI}]\!] = \lambda ks.k(\lambda xrd.\boxed{s}x)$$

That is almost, but not quite, the same as if we tried to define **JI** as $\lambda_{\mathsf{r}}x.\mathtt{go}\,x$:

$$\begin{array}{rcl}[\![\mathbf{JI}]\!] & = & [\![\lambda_{\mathsf{r}}x.\mathtt{go}\,x]\!] \\ & = & \lambda ks.k(\lambda xrd.\boxed{r}x)\end{array}$$

We can, however, define **JI** in terms of `go` if we use the static λ_{s}, that is $\mathbf{JI} = \lambda_{\mathsf{s}}x.\mathtt{go}\,x$, as this does not inadvertently shadow the continuation s that we want **JI** to refer to.

The whole transform for the calculus with **JI** is this:

$$\begin{array}{rcl}[\![x]\!] & = & \lambda kq.qx \\ [\![\lambda_{\mathsf{r}}x.M]\!] & = & \lambda ks.k(\lambda xrd.[\![M]\!]r\boxed{r}) \\ [\![MN]\!] & = & \lambda kq.[\![M]\!](\lambda m.[\![N]\!](\lambda n.mnkq)q)q \\ [\![\mathbf{JI}]\!] & = & \lambda ks.k(\lambda xrd.\boxed{s}x)\end{array}$$

Recall that the role of `here` has been usurped by λ_{r}, and we replaced `go` by its first-class cousin **JI**.

In the transform for **JI**, the jump continuation is the current "dump" in the sense of the SECD-machine. The dump in the SECD-machine is a sort of call stack, which holds the return continuation for the procedure whose body is currently being evaluated. Making the dump into a first-class object was precisely how Landin invented first-class control, embodied by the **J**-operator.

The typing for the language with **JI** is given in Figure 7. In particular, the type of **JI** is the classical disjunction

$$\frac{}{\Gamma \vdash_{\mathsf{j}} \mathbf{JI} : B \to C, B}$$

As an example of the type system for the calculus with the **JI**-operator, we see that Reynolds's [9] definition of `call/cc` in terms of **JI** typechecks. (Strictly speaking, Reynolds used `escape`, the binding-form cousin of `call/cc`, but `call/cc` and `escape` are syntactic sugar for each other.) In Figure 8, we infer the type of $\mathtt{call/cc} \equiv \lambda_{\mathsf{r}}f.((\lambda_{\mathsf{r}}k.f\,k)(\mathbf{JI}))$ to be:

$$\vdash_{\mathsf{j}} \lambda_{\mathsf{r}}f.((\lambda_{\mathsf{r}}k.f\,k)(\mathbf{JI})) : ((A \to B) \to A) \to A), C.$$

Because **JI** has such evident logical meaning as classical disjunction, we have considered it as basic. Landin [6] took another operator, called **J**, as primitive, while **JI** was derived as the special case of **J** applied to the identity combinator:

$$\mathbf{J}\,\mathbf{I} = \mathbf{J}\,(\lambda x.x)$$

This explains the name "**JI**", as "**J**" stands for "jump" and **I** for "identity". We were able to start with **JI**, since (as noted by Landin) the **J**-operator is syntactic sugar for **JI** by virtue of:

$$\mathbf{J} = (\lambda_{\mathsf{r}}r.\lambda_{\mathsf{r}}f.\lambda_{\mathsf{r}}x.r(fx))\,(\mathbf{JI}).$$

To accommodate **J** in our typing, we use this definition in terms of **JI** to derive the following type for **J**:

$$\vdash_{\mathsf{j}} \mathbf{J} : (A \to B) \to (A \to C), B$$

See Figure 9. This type reflects the behaviour of the **J**-operator in the SECD machine. When **J** is evaluated, it captures the B-accepting current dump continuation; it can then be applied to a function of type $A \to B$. This function is composed with the captured dump, yielding a non-returning function of type $A \to C$, for arbitrary C. By analogy with `call-with-current-continuation`, we may read the**J**-operator as "`compose-with-current-dump`" [13].

The logical significance, if any, of the extra function types in the general **J** seems unclear. There is a curious, though vague, resemblance to exception handlers in dynamic control, since they too are functions only to be applied on jumping. This feature of **J** may be historical, as it arose in a context where greater emphasis was given to attaching dumps to functions than to dumps as first-class continuations in their own right.

6. TYPE PRESERVATION

The typings agree with the transforms in that they are preserved in the usual way for CPS transforms. The only complication is that we need (at least ML-style) polymorphism in the target λ-calculus to type the dynamic continuation in those transforms that ignore it. Let α be the answer type (which could, but need not, be a free type variable). The annotated and the ordinary function type are transformed as follows:

$$\begin{array}{rcl}[\![A \to B \vee C]\!] & = & [\![A]\!] \to ([\![B]\!] \to \alpha) \to ([\![C]\!] \to \alpha) \to \alpha \\ [\![A \to B]\!] & = & \forall\beta.[\![A]\!] \to ([\![B]\!] \to \alpha) \to \beta \to \alpha\end{array}$$

For all the transforms we have preservation of the respective typing: if $\Gamma \vdash_? M : A, B$, then

$$[\![\Gamma]\!] \vdash [\![M]\!] : ([\![A]\!] \to \alpha) \to ([\![B]\!] \to \alpha) \to \alpha.$$

7. CONCLUSIONS

As a summary of the four control constructs we have considered, we present their typings in Figure 10, omitting the terms for conciseness. As logical systems, these toy logics may seem a little eccentric, with two succedents that can only be manipulated in a slightly roundabout way. But they are sufficient for our purposes here, which is to illustrate the correspondence of first-class continuations with classical logic and weaker control operation with intuitionistic logic, and the central role of the arrow type in this dichotomy.

Recall the following fact from proof theory (see for example [15]). Suppose one starts from a presentation of intuitionistic logic with sequents of the form $\Gamma \vdash \Delta$. If a rule like the following is added that allows $\to$-introduction even if there are multiple succedents, the logic becomes classical.

$$\frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \to B, \Delta}$$

In continuation terms, the significance of this rule is that the function closure of type $A \to B$ may contain any of the continuations that appear in Δ; to use the jargon, these continuations become "reified". The fact that the logic becomes classical means that once we can have continuations in function closures, we gain first-class continuations and

Figure 7: Typing for JI

$$\frac{}{\Gamma, x : A, \Gamma' \vdash_{\mathsf{j}} x : A, C} \qquad \frac{}{\Gamma \vdash_{\mathsf{j}} \mathbf{JI} : B \to C, B}$$

$$\frac{\Gamma, x : A \vdash_{\mathsf{j}} M : B, B}{\Gamma \vdash_{\mathsf{j}} \lambda_{\mathsf{r}} x.M : A \to B, C} \qquad \frac{\Gamma \vdash_{\mathsf{j}} M : A \to B, C \quad \Gamma \vdash_{\mathsf{j}} N : A, C}{\Gamma \vdash_{\mathsf{j}} MN : B, C}$$

Figure 8: Derivation of `call/cc` from JI
Let $\Gamma \equiv f : (A \to B) \to A, k : (A \to B)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_{\mathsf{j}} f : (A \to B) \to A, A} \quad \dfrac{}{\Gamma \vdash_{\mathsf{j}} k : (A \to B), A}}{\Gamma \vdash_{\mathsf{j}} f\, k : A, A}}{f : (A \to B) \to A \vdash_{\mathsf{j}} \lambda_{\mathsf{r}} k.f k : (A \to B) \to A, A} \quad \dfrac{}{f : (A \to B) \to A \vdash_{\mathsf{j}} \mathbf{JI} : A \to B, A}}{f : (A \to B) \to A \vdash_{\mathsf{j}} (\lambda_{\mathsf{r}} k.f\, k)(\mathbf{JI}) : A, A}}{\vdash_{\mathsf{j}} \lambda_{\mathsf{r}} f.((\lambda_{\mathsf{r}} k.f\, k)(\mathbf{JI})) : ((A \to B) \to A) \to A), C}$$

Figure 9: Derivation of J from JI
Let $\Gamma \equiv x : A, r : B \to C, f : A \to B$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_{\mathsf{j}} r : B \to C, C} \quad \dfrac{\dfrac{}{\Gamma \vdash_{\mathsf{j}} f : A \to B, C} \quad \dfrac{}{\Gamma \vdash_{\mathsf{j}} x : A, C}}{\Gamma \vdash_{\mathsf{j}} fx : B, C}}{\Gamma \vdash_{\mathsf{j}} r(fx) : C, C}}{r : B \to C, f : A \to B \vdash_{\mathsf{j}} \lambda_{\mathsf{r}} x.r(fx) : A \to C, A \to C}}{r : B \to C \vdash_{\mathsf{j}} \lambda_{\mathsf{r}} f.\lambda_{\mathsf{r}} x.r(fx) : (A \to B) \to (A \to C), (A \to B) \to (A \to C)}}{\vdash_{\mathsf{j}} \lambda_{\mathsf{r}} r.\lambda_{\mathsf{r}} f.\lambda_{\mathsf{r}} x.r(fx) : (B \to C) \to (A \to B) \to (A \to C), B} \quad \dfrac{}{\vdash_{\mathsf{j}} \mathbf{JI} : B \to C, B}}{\vdash_{\mathsf{j}} (\lambda_{\mathsf{r}} r.\lambda_{\mathsf{r}} f.\lambda_{\mathsf{r}} x.r(fx))\,(\mathbf{JI}) : (A \to B) \to (A \to C), B}$$

Figure 10: Comparison of the type systems as logics

Static `here` and `go`, implies `call/cc`

$$\frac{\Gamma \vdash_{\mathsf{s}} B, B}{\Gamma \vdash_{\mathsf{s}} B, C} \qquad \frac{\Gamma \vdash_{\mathsf{s}} B, B}{\Gamma \vdash_{\mathsf{s}} C, B} \qquad \frac{}{\Gamma, A, \Gamma' \vdash_{\mathsf{s}} A, C}$$

$$\frac{\Gamma, A \vdash_{\mathsf{s}} B, C}{\Gamma \vdash_{\mathsf{s}} A \to B, C} \qquad \frac{\Gamma \vdash_{\mathsf{s}} A \to B, C \quad \Gamma \vdash_{\mathsf{s}} A, C}{\Gamma \vdash_{\mathsf{s}} B, C}$$

Dynamic `here` and `go`, like checked exceptions

$$\frac{\Gamma \vdash_{\mathsf{d}} B, B}{\Gamma \vdash_{\mathsf{d}} B, C} \qquad \frac{\Gamma \vdash_{\mathsf{d}} B, B}{\Gamma \vdash_{\mathsf{d}} C, B} \qquad \frac{}{\Gamma, A, \Gamma' \vdash_{\mathsf{d}} A, C}$$

$$\frac{\Gamma, A \vdash_{\mathsf{d}} B, C}{\Gamma \vdash_{\mathsf{d}} A \to B \vee C, D} \qquad \frac{\Gamma \vdash_{\mathsf{d}} A \to B \vee C, C \quad \Gamma \vdash_{\mathsf{d}} A, C}{\Gamma \vdash_{\mathsf{d}} B, C}$$

Non-first class `return`-operation

$$\frac{\Gamma \vdash_{\mathsf{r}} B, B}{\Gamma \vdash_{\mathsf{r}} C, B} \qquad \frac{}{\Gamma, A, \Gamma' \vdash_{\mathsf{r}} A, C}$$

$$\frac{\Gamma, A \vdash_{\mathsf{r}} B, B}{\Gamma \vdash_{\mathsf{r}} A \to B, C} \qquad \frac{\Gamma \vdash_{\mathsf{r}} A \to B, C \quad \Gamma \vdash_{\mathsf{r}} A, C}{\Gamma \vdash_{\mathsf{r}} B, C}$$

Landin's **JI**-operator

$$\frac{}{\Gamma \vdash_{\mathsf{j}} B \to C, B} \qquad \frac{}{\Gamma, A, \Gamma' \vdash_{\mathsf{j}} A, C}$$

$$\frac{\Gamma, A \vdash_{\mathsf{j}} B, B}{\Gamma \vdash_{\mathsf{j}} A \to B, C} \qquad \frac{\Gamma \vdash_{\mathsf{j}} A \to B, C \quad \Gamma \vdash_{\mathsf{j}} A, C}{\Gamma \vdash_{\mathsf{j}} B, C}$$

thereby the same power as `call/cc`. We have this form of rule for static `here` and `go`; though not for **JI**, since **JI** as the excluded middle is already blatantly classical by itself.

But the logic remains intuitionistic if the $\rightarrow$-introduction is restricted. The rule for this case typically admits only a single formula on the right:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B, \Delta}$$

Considered as a restriction on control operators, this rule prohibits λ-abstraction for terms that contain free continuation variables. There are clearly other possibilities how we can prevent assumptions from Δ to become hidden (in that they can be used in the derivation of $A \rightarrow B$ without showing up in this type itself). We could require these assumptions to remain explicit in the arrow type, by making Δ a singleton that either coincides with the B on the right of the arrow, or is added to it:

$$\frac{\Gamma, A \vdash_{\mathsf{r}} B, B}{\Gamma \vdash_{\mathsf{r}} A \rightarrow B, C} \qquad \frac{\Gamma, A \vdash_{\mathsf{d}} B, C}{\Gamma \vdash_{\mathsf{d}} A \rightarrow B \vee C, D}$$

These are the rules for $\rightarrow$-introduction in connection with the `return`-operation, and dynamic `here` and `go`, respectively. Neither of which gives rise to first-class continuations, corresponding to the fact that with these restrictions on $\rightarrow$-introduction the logics remain intuitionistic.

The distinction between static and dynamic control in logical terms appears to be new, as is the logical explanation of Landin's **JI**-operator.

7.1 Related work

Following Griffin [3], there has been a great deal of work on classical types for control operators, mainly on `call/cc` or minor variants thereof. A similar CPS transforms for dynamic control (exceptions) has appeared in [5], albeit for a very different purpose. Felleisen describes the **J**-operator by way of CPS, but since his transform is not double-barrelled, **J** means something different in each λ [2]. Variants of the `here` and `go` operators are even older than the notion of continuation itself: the operations `valof` and `resultis` from BCPL appeared in Strachey and Wadsworth's report on continuations [11, 12]. These operators led to the modern `return` in C. As we have shown here, they lead to much else besides if combined with different flavours of λ.

7.2 Further work

In this paper, control constructs were compared by CPS transforms and typing of the *source*. A different, but related approach compares them by typing in the *target* of the CPS [1] . On the source, we have the dichotomy between intuitionistic and classical typing, whereas on the target, the distinction is between linear and intuitionistic. We hope to relate these in further work.

8. REFERENCES

[1] J. Berdine, P. W. O'Hearn, U. Reddy, and H. Thielecke. Linearly used continuations. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Continuations*, 2001.

[2] M. Felleisen. Reflections on Landin's J operator: a partly historical note. *Computer Languages*, 12(3/4):197–207, 1987.

[3] T. G. Griffin. A formulae-as-types notion of control. In *Proc. 17th ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, CA USA, 1990.

[4] R. Kelsey, W. Clinger, and J. Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(3):7–105, 1998.

[5] J. Kim, K. Yi, and O. Danvy. Assessing the overhead of ML exceptions by selective CPS transformation. In *Proceedings of the 1998 ACM SIGPLAN Workshop on ML*, 1998.

[6] P. J. Landin. A generalization of jumps and labels. Report, UNIVAC Systems Programming Research, Aug. 1965.

[7] P. J. Landin. A generalization of jumps and labels. *Higher-Order and Symbolic Computation*, 11(2), 1998. Reprint of [6].

[8] G. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.

[9] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the 25th ACM National Conference*, pages 717–740. ACM, Aug. 1972.

[10] J. G. Riecke and H. Thielecke. Typed exceptions and continuations cannot macro-express each other. In J. Wiedermann, P. van Emde Boas, and M. Nielsen, editors, *Proceedings 26th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1644 of *LNCS*, pages 635–644. Springer Verlag, 1999.

[11] C. Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, UK, 1974.

[12] C. Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation*, 13(1/2):135–152, April 2000. Reprint of [11].

[13] H. Thielecke. An introduction to Landin's "A generalization of jumps and labels". *Higher-Order and Symbolic Computation*, 11(2):117–124, 1998.

[14] H. Thielecke. On exceptions versus continuations in the presence of state. In G. Smolka, editor, *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000,*, number 1782 in LNCS, pages 397–411. Springer Verlag, 2000.

[15] A. S. Troelstra and D. van Dalen. *Constructivism in Mathematics*, volume 2. North Holland, 1988.

Towards Logical Understanding of Delimited Continuations

Yukiyoshi Kameyama
Graduate School of Informatics, Kyoto University
kameyama@kuis.kyoto-u.ac.jp

ABSTRACT

The aim of this paper is to give a *logical* understanding of control operators for delimited (partial) continuations, in particular, Danvy and Filinski's shift/reset. Starting with a simple type system for a static analogue of the shift/reset operators, we analyze the type structure in a logical way, which gives rise to control operators for delimited continuations. The resulting control operators nicely behave in the sense that it satisfies strong normalization and other nice properties, and we can use them with a higher order types. We finally compare our control operators with the original shift/reset operators.

KEYWORDS

Delimited Continuations, Type system, Classical Logic

1 Introduction

Delimited continuation (often called *partial continuation*) is a notion representing a *partial* rest of the computation. There have been many proposals of control operators for delimited continuations, but our understanding for them does not seem to reach at a satisfactory level as in the case of the `call/cc` operator.

This paper tries to give a better understanding of delimited continuations. To do so, we use types and their logical meaning as our guide, namely the Curry-Howard isomorphism is our guiding principle. We start with a simple type system for control operators of delimited continuations, then study its type structure and logical meaning, obtain suitable representation (encoding) of control operators, and finally obtain reduction rules which are induced from the types of the encoded terms.

Our result is that, the naturally arising control operators are not exactly the same as any of existing ones, but they have the same operational meaning if there is only one invocation of delimited continuations.[1] Since our study is done in the framework of well understood logical machinery with nice properties, the resulting operator is ensured to have a number of desirable properties, such as subject reduction and strong normalizability, and also a type-preserving CPS-translation is obtained without restricting the type of the reset operator to atomic.

This paper is organized as follows: Section 2 gives the calculus for the catch/throw mechanism which will be used in later sections. In Sections 3, we give a simple type-theoretic formulation of Danvy and Filinski's shift/reset operators, and in Section 4, we analyze the logical structure of their static analogue to obtain two views for them. In Section 5, we obtain two computational interpretations corresponding to the two views, and show one interpretation is a control operator for delimited continuations. In Section 6, we study the properties of the obtained control operator. Section 7 gives concluding remarks.

2 Preliminary: the Catch/Throw Calculus

This section briefly introduces the static catch/throw calculi developed by Nakano, Sato, and the author [10, 12, 8]. The catch/throw calculus is used to interpret the calculus for delimited continuations in later sections.

Remark 1. The use of the word "static" in this paper may be different from [2].

In this paper, "static" means that the corresponding `catch`-construct for a `throw`-construct is determined in a static way, that is, at the time when they are written. On the contrary, the actual catch/throw mechanism in Common Lisp (and the exception mechanism in Standard ML) is "dynamic" in the sense that the corresponding `catch`-construct is determined when `throw` is evaluated. The difference is illustrated by the following example:

[1] Of course, the invoked continuation objects can be used arbitrary times.

©2001, Yukiyoshi Kameyama

(Static)

$$\begin{aligned} & \mathtt{catch}_\alpha((\lambda x.1 + \mathtt{catch}_\alpha(x0))(\lambda y.\mathtt{throw}_\alpha y)) \\ \to\ & \mathtt{catch}_\alpha(1 + \mathtt{catch}_\beta((\lambda y.\mathtt{throw}_\alpha y)0)) \\ \to\ & \mathtt{catch}_\alpha(1 + \mathtt{catch}_\beta(\mathtt{throw}_\alpha 0)) \\ \to\ & \mathtt{catch}_\alpha(\mathtt{throw}_\alpha 0) \\ \to\ & 0 \end{aligned}$$

Note that, in the static calculus the tag variable α was renamed to β to avoid the unwanted capture of the free tag variable. The corresponding `catch`-construct for `throw` is the outer one.

(Dynamic)

$$\begin{aligned} & \mathtt{catch}_\alpha((\lambda x.1 + \mathtt{catch}_\alpha(x0))(\lambda y.\mathtt{throw}_\alpha y)) \\ \to\ & \mathtt{catch}_\alpha(1 + \mathtt{catch}_\alpha((\lambda y.\mathtt{throw}_\alpha y)0)) \\ \to\ & \mathtt{catch}_\alpha(1 + \mathtt{catch}_\alpha(\mathtt{throw}_\alpha 0)) \\ \to\ & \mathtt{catch}_\alpha(1 + 0) \\ \stackrel{*}{\to}\ & 1 \end{aligned}$$

In this case, the `throw`-expression is captured by the inner `catch`-construct.

Although our terminology differ from Danvy and Filinski's, ours coincides with the terminology of the dynamic/static binding of variables. Later we shall see the shift/reset operators of Danvy and Filinski are dynamic in our sense. **(End of Remark 1.)**

2.1 Type System of $\mathrm{LK}_{c/t}$ and $\mathrm{LK}_{c/t}^{\mathrm{CBV}}$

Among several variants of static catch/throw calculi, we take here $\mathrm{LK}_{c/t}$ in [8] and its call-by-value subcalculus $\mathrm{LK}_{c/t}^{\mathrm{CBV}}$. The formulation of $\mathrm{LK}_{c/t}$ is essentially due to Nakano [10], but since he wanted to confine himself to intuitionistic logic, he put a restriction on the formation rule of λ (the implication-introduction rule). In our earlier works, we showed that the calculus without the restriction is more natural and has more applications in higher-order programming and program extraction from classical proofs.

The type system of $\mathrm{LK}_{c/t}$ and that of $\mathrm{LK}_{c/t}^{\mathrm{CBV}}$ are the same, which we will be given below.

Types are given by the grammar:

$$A, B ::= K \mid A \to B$$

where K ranges over atomic types. Raw terms are

$$M, N ::= x \mid \lambda x.M \mid MN \mid \mathtt{catch}_\alpha M \mid \mathtt{throw}_\alpha M$$

where x ranges over individual (normal) variables, and α ranges over tag variables.[2] The individual variable x is bound in $\lambda x.M$ and the tag variable α is bound in $\mathtt{catch}_\alpha M$. The bound/free occurrences of individual/tag variables are determined from these notions in a usual way. We identify two raw terms which are the same modulo the renaming of bound individual/tag variables. For instance, $\lambda x.\mathtt{catch}_\alpha(\mathtt{throw}_\alpha x)$ and $\lambda y.\mathtt{catch}_\beta(\mathtt{throw}_\beta y)$ are identified. $FV(M)$ and $FTV(M)$ denote the set of free individual/tag variables in M, respectively. $M\{x := N\}$ and $M\{\alpha := \beta\}$ denote the usual capture-avoiding substitutions.

A judgement is in the form $\Gamma \vdash M : A \ ; \ \Delta$ where M is a raw term, A is a type, Γ is a finite set of declarations of the form $x : A$, and Δ is a set of declarations of the form $\alpha : A$. Note that Γ denotes the set of free individual variables, and Δ denotes that of free tag variables. We write Δ in the righthand side of the judgement because, when we consider the Curry-Howard correspondence, this form is more natural than the form $\Gamma \ ; \ \Delta \vdash M : A$.

We now give type inference rules in the natural-deduction style which are used to infer judgements.

$$\frac{}{\Gamma \vdash x : A \ ; \ \Delta} \ (\text{if} \quad x : A \in \Gamma)$$

$$\frac{\Gamma \vdash M : B \ ; \ \Delta}{\Gamma - \{x : A\} \vdash \lambda x.M : A \to B \ ; \ \Delta}$$

$$\frac{\Gamma \vdash M : A \to B \ ; \ \Delta \quad \Gamma \vdash N : A \ ; \ \Delta}{\Gamma \vdash MN : B \ ; \ \Delta}$$

$$\frac{\Gamma \vdash M : A \ ; \ \Delta}{\Gamma \vdash \mathtt{throw}_\alpha M : B \ ; \ \Delta \cup \{\alpha : A\}}$$

$$\frac{\Gamma \vdash M : A \ ; \ \Delta}{\Gamma \vdash \mathtt{catch}_\alpha M : A \ ; \ \Delta - \{\alpha : A\}}$$

If $\Gamma \vdash M : A \ ; \ \Delta$ is inferred using the above inference rules, we say M is a term of type A under Γ and Δ.

2.2 The Curry-Howard Isomorphism

The Curry-Howard isomorphism relates the simply typed lambda calculus with the intuitionistic propositional logic.[3] This isomorphism can be extended to the above type system and the classical propositional logic.

To see this, let us rewrite the type inference rules for catch and throw where (i) terms are omitted, (ii) $A; \Delta$ is written as $A, B_1, \cdots, B_n$ if $\Delta = \{B_1, \cdots, B_n\}$.

$$\frac{\Gamma \vdash A, \Delta}{\Gamma \vdash B, A, \Delta} \ (\text{throw})$$

$$\frac{\Gamma \vdash A, \cdots A, \Delta}{\Gamma \vdash A, \Delta} \ (\text{catch})$$

The first (throw) rule is the weakening rule, and the second (catch) rule is (n-times application of) the contraction rule. Namely, the essence of the catch/throw mechanism can be captured as the well known logical inference rules; weakening and contraction. Parigot's $\lambda\mu$-calculus [11] has essentially the same inference rules as above.

2.3 Operational Semantics of $\mathrm{LK}_{c/t}$

The operational semantics of $\mathrm{LK}_{c/t}^{\mathrm{CBV}}$ is given in the evaluation-context semantics style as follows.

First, values are defined as usual:

[2] We assume that the set of individual variables and the set of tag variables are disjoint.

[3] Strictly speaking, the corresponding logic is minimal logic, since we do not have the bottom-elimination rule.

$$V \quad ::= \quad x \mid \lambda x.M$$

An evaluation context (in the call-by-value semantics) is defined as follows:

$$E \quad ::= \quad [\,] \mid EM \mid VE \mid \mathtt{throw}_\alpha E \mid \mathtt{catch}_\alpha E$$

$E[M]$ denotes the term obtained by replacing the hole $[\,]$ in E by M.

The 1-step reduction in $\mathrm{LK}_{c/t}^{\mathrm{CBV}}$ is defined as the union of the following relations:

$$\begin{array}{rcl}
E[(\lambda x.M)V] & \to & E[M\{x := V\}] \\
E[\mathtt{catch}_\alpha V] & \to & E[V] \quad \text{if } \alpha \notin FTV(V) \\
E[\mathtt{catch}_\alpha(\mathtt{throw}_\alpha V)] & \to & E[\mathtt{catch}_\alpha V] \\
E[(\mathtt{throw}_\alpha V)M] & \to & E[\mathtt{throw}_\alpha V] \\
E[V_1(\mathtt{throw}_\alpha V_2)] & \to & E[\mathtt{throw}_\alpha V_2] \\
E[\mathtt{throw}_\beta(\mathtt{throw}_\alpha V)] & \to & E[\mathtt{throw}_\alpha V] \\
E[\mathtt{catch}_\beta(\mathtt{throw}_\alpha V)] & \to & E[\mathtt{throw}_\alpha V] \quad \text{if } \beta \notin FTV(V)
\end{array}$$

where in the last two rules we assumed $\alpha \not\equiv \beta$.

The calculus $\mathrm{LK}_{c/t}^{\mathrm{CBV}}$ gives a simple formulation of the static catch/throw calculus. It is a very weak calculus in the sense that the actual (dynamic) catch/throw mechanism cannot be simulated, but we can recover the lost expressiveness to some extent by introducing the abstraction mechanism of tag variables [8]. Also, the calculus cannot reduce non-value terms such as $(\mathtt{catch}_\alpha \lambda x.M)V$ if M contains free occurrences of α, but this defect can be compromised by introducing the structural reduction in the $\lambda\mu$-calculus. In the scope of this paper, the weakness of the reduction rules do not have problems (in particular, we can give encoding of control operators in this weak calculus).

$\mathrm{LK}_{c/t}^{\mathrm{CBV}}$ can be considered as a subcalculus of (the call-by-value variant of) Parigot's $\lambda\mu$-calculus [11], so our interpretation in later sections can be done within the $\lambda\mu$-calculus. We do not do so in this paper, since there are several variants of the call-by-value $\lambda\mu$-calculus, and few studies on the non-deterministic version of $\lambda\mu$-calculus have been studied (which contains the call-by-name and the call-by-value fragments), while the catch/throw calculus is sufficiently simple and its properties have been already studied intensively.

2.4 Operational Semantics of $\mathrm{LK}_{c/t}$

The operational semantics of $\mathrm{LK}_{c/t}$ is given by changing the reduction rules of $\mathrm{LK}_{c/t}^{\mathrm{CBV}}$ as follows:

- All the occurrences of V, V_1, V_2 are replaced by arbitrary terms M, M_1, M_2.
- The following rule is added:

$$E[\lambda x.\mathtt{throw}_\alpha M] \to E[\mathtt{throw}_\alpha M] \quad \text{if } x \notin FV(M)$$

Apparently the computation of $\mathrm{LK}_{c/t}$ is non-deterministic. Nevertheless it is strongly normalizing [8].

3 Shift/Reset Operators and Their Static Analogue

3.1 Shift/Reset Operators

Danvy and Filinski's shift/reset operators [1, 2] are the most well known (and probably the most widely used) control operators for delimited continuations. The reset operator (the delimiter) is denoted as $\#M$ (or $\langle M \rangle$ in their original notation), and the shift operator is $\xi k.M$ where the variable k is bound by this ξ. Although the operational semantics of shift/reset is defined via the CPS-transformation, we can consider the following reduction rule for shift/reset:

$$E[\#(E'[\xi k.M])] \to E[\#M\{k := \lambda u.\#(E'[u])\}]$$

The continuation up to the closest delimiter $\#$ is captured as a functional object $\lambda u.\#(E[u])$ and can be used in later computation. Note that, unlike the standard (full) continuation generated by `call/cc` in Scheme and SML/NJ, the created continuation is a partial rest of computation (it is not the whole rest of computation like $\lambda u.E[E'[u]]$). Another important difference between delimited/full continuations is that the full continuation is abortive (not composable) while the delimited continuation can be composed. It is illustrated by the following example:

$$\begin{array}{rcl}
\#(1 + (\xi k.k(k(0)))) & \to & \#(k(k(0)))\{k := \lambda u.\#1 + u\} \\
& \stackrel{*}{\to} & 1 + 1 + 0 \\
& \stackrel{*}{\to} & 2
\end{array}$$

On the contrary, the term $1 + \mathbf{call/cc}\lambda k.k(k(k(0)))$ evaluates to 1, since the current continuation is aborted when k is applied to 0, and $k(k(k(0)))$ and $k(0)$ result in the same answer.

Note that, the shift/reset operators are *dynamic* in our sense, since $(\lambda x.\#(xy))(\lambda z.\xi k.M)$ reduces to $\#((\lambda z.\xi k.M)y)$, and $\#((\#\lambda x.\xi k.M)y)$ reduces to $\#((\lambda x.\xi k.M)y)$. In both cases, the reset operator corresponding to the shift operator is different from the lexically determined one.

Remark 2. In [1], the shift/reset operators are said to be *static* in comparison with Felleisen's *dynamic* operators. The functional object generated by an invocation of the shift operator is $\lambda u.\#E[u]$, while Felleisen's operator generates a functional object like $\lambda u.E[u]$. Suppose E contains another occurrence of shift. Then it is delimited by a fixed delimiter in the former case, and by an unknown outer delimiter in the latter case. This difference is the reason why Danvy and Filinski's operator enjoy a simple, elegant CPS-transformation. **(End of Remark 2)**

3.2 Type System

We then try to give types to the shift/reset operators. One may think that the following typing rules are appropriate for the shift/reset operators:

$$\frac{\Gamma, k : A \to B \vdash M : B \;;\; \Delta}{\Gamma \vdash \xi\, k.M : A \;;\; \Delta \cup \{B\}} \text{ (shift)}$$

$$\frac{\Gamma \vdash M : B \;;\; \Delta}{\Gamma \vdash \#M : B \;;\; \Delta - \{B\}} \text{ (reset)}$$

However, this type system does not work since the shift/reset operators are dynamic in our sense. In other words, this type system does not enjoy the subject reduction property, which we think is the minimum requirement for a type system being sound.

In order to overcome this problem, Murthy [9] introduced a rather elaborated type system for the hierarchy of shift/reset's. He changed the function type $A \to B$ to $((A \to B) \to T_i) \to T_i$ where T_i is the type of i-th delimiter, that is, he annotates every function type by T_i, which complicates the type structures a bit, and his type system is not a (simple) extension of the simply typed lambda calculus. Thielecke [13] also considers a type system to cope with dynamic features of control operators where he uses the notation $A \to B \vee C$, which is essentially the same as $((A \to B) \to C) \to C$.

Since our aim is to find an essential logical structure of delimited continuation operators, we do not want to treat such a complicated system. Instead, we stick to a simple type system, and by analyzing it in a purely logical way, we are interested in seeing what control operators arise from the type system.

In summary, we address the problem to find a computational interpretation of the following typing rules.

$$\frac{\Gamma, A \to B \vdash B \ ; \ \Delta}{\Gamma \vdash A \ ; \ \Delta \cup \{B\}} \ \text{(shift)} \qquad \frac{\Gamma \vdash B \ ; \ \Delta}{\Gamma \vdash B \ ; \ \Delta - \{B\}} \ \text{(reset)}$$

Remark 3. Since we take the Curry-Howard isomorphism as the guide of our study, the corresponding calculus always becomes static (in our sense). Hieb et al [6] proposed control operators for subcontinuations, which is another variant of delimited continuation operators, but are static in our sense. Hence, our study is closer to subcontinuations than shift/reset.

4 Logical Analysis

We first note that the first (shift) type inference rule plays two roles at the same time; (i) it discharges the type $A \to B$ and (ii) it introduces the type A in the righthand side, that is, it works as the weakening rule. Reflecting these two roles, we split the shift rule into the two rules and obtain the following set of rules:

$$\frac{\Gamma, A \to B \vdash A \ ; \ \Delta \cup \{B\}}{\Gamma \vdash A \ ; \ \Delta \cup \{B\}} \ (*)$$

$$\frac{\Gamma \vdash B \ ; \ \Delta}{\Gamma \vdash A \ ; \ \Delta \cup \{B\}} \ \text{(throw)}$$

$$\frac{\Gamma \vdash B \ ; \ \Delta}{\Gamma \vdash B \ ; \ \Delta - \{B\}} \ \text{(catch)}$$

In this formulation the shift rule splits into the first and the second rules by which the shift rule is derivable.

Now it is easy to see that the second (throw) and the third (catch) rules are the same as the catch and the throw rules given in the last section, namely, the weakening and the contraction rules. These rules are well known inference rules in formulating classical propositional logic, so, we can say that the logical meaning of these rules are well understood.

Our remaining task is to understand the first rule (marked with *). Since we use the Curry-Howard isomorphism as the guiding principle, we should have a consistent logical system. If A is proved by the above set of inference rules, namely, $\{\} \vdash A \ ; \ \{\}$ is derived, then A must be valid in some suitable logic, say, classical logic. By considering how to ensure this property, we can think of the following two views:

(First View) The (*) rule is a stand-alone valid rule in classical logic. Namely, we can regard it as a natural extension of Peirce's inference rule below:

$$\frac{\Gamma, A \to B \vdash A}{\Gamma \vdash A}$$

This rule is admissible in classical logic, that is, derivable from the catch/throw rules. We give a proof of the (*) rule using the catch/throw rules as follows (where we suppress to write Γ and Δ for readability):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A \vdash A;}}{A \vdash B; A} \quad \begin{array}{c}\vdots\\ A \to B \vdash A;\end{array}}{\vdash A \to B; A} \quad \dfrac{}{\vdash (A \to B) \to A;}}{\dfrac{\vdash A; A}{\vdash A;}}$$

(Second View) The (*) rule is not valid as a stand-alone rule, but with the combination of the catch rule it is valid. The situation is similar to the throw rule, which is not valid as a stand-alone rule, but is valid with the combination of the catch rule. In order to have a proof of $\{\} \vdash A \ ; \ \{\}$, if there is an application of the (*) rule with B introduced in the righthand side, then some application of the catch rule must exist under it which discharges the same B. So, each occurrence of the (*) rule must be accompanied with an occurrence of the catch rule:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma, A \to B \vdash A \ ; \ \{B\} \cup \Delta\end{array}}{\Gamma \vdash A \ ; \ \{B\} \cup \Delta}}{\begin{array}{c}\vdots\\ \Gamma' \vdash B \ ; \ \{B\} \cup \Delta'\end{array}}}{\Gamma' \vdash B \ ; \ \Delta'}$$

In [7], we showed that this combination is provable in intuitionistic logic. There could be many ways to give its proof, but the presumably simplest way is given as follows (where we again suppress to write Γ, Δ and so on):

$$\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\\ A \to B \vdash A; B\end{array}}{\vdash (A \to B) \to A; B} \quad \begin{array}{c}A \vdash A; B\\ \vdots\\ A \vdash B; B\end{array}}{\vdash A \to B; B}}{\vdash A; B}}{\begin{array}{c}\vdots\\ \vdash B; B\end{array}}}{\vdash B;}$$

Note that, the right-upper subproof of this proof is taken from the lower subproof of the same proof, so that part is duplicated. Note also that this proof is written by intuitionistic inference rules only, so it is valid in intuitionistic logic.

Corresponding to these two views, we obtain two different computational interpretations of the (*) rule, which we will exploit in the next section.

5 Computational Interpretations

So far we have been concentrating on the types of (the static analogue of) the delimited continuation operators. In this section we consider the term assignment which arises from the interpretation of typing rules, and see what would be their natural reduction rules.

As in the case of the catch/throw calculi, we attach an individual variable (denoted as $x, y, \cdots$) to each element of Γ, and a tag variable (denoted as $\alpha, \beta, \cdots$) to each element of Δ. For brevity, we use the same symbols ξ and $\#$ to the control operators obtained in this analysis, but here these operators are annotated by tag variables, hence we will have terms like $\xi_\alpha k.M$ and $\#_\alpha M$.

5.1 The First View

From the first view, we obtain the following definitions:

$$\begin{array}{rcl} \xi_\alpha k.M & \triangleq & \mathtt{catch}_{\alpha'}((\lambda k.M)(\lambda x.\mathtt{throw}_{\alpha'} x)) \\ \#_\alpha M & \triangleq & \mathtt{catch}_\alpha M \end{array}$$

where α' is a new tag.

This interpretation is not interesting; there are no throw's with the tag α in the representation of the term $\xi_\alpha k.M$, so this operator becomes the simple `call/cc`-operator, and the reset operator becomes identity.

A slightly more interesting interpretation is the following:

$$\begin{array}{rcl} \xi_\alpha k.M & \triangleq & \mathtt{catch}_{\alpha'}(\mathtt{throw}_\alpha((\lambda k.M)(\lambda x.\mathtt{throw}_{\alpha'} x))) \\ \#_\alpha M & \triangleq & \mathtt{catch}_\alpha M \end{array}$$

At a first sight, this may look interesting, since $\#_\alpha(E[\xi_\alpha k.M])$ reduces to $\#_\alpha M\{k := \lambda x.\mathtt{throw}_\alpha E[x]\}$, which captures the delimited continuation E. However, it turns out another representation of the `call/cc` mechanism, since the reified continuation-like object $\lambda x.\mathtt{throw}_\alpha E[x]$ contains the throw operator in it, and it discards the current continuation (recall that an important property of the delimited continuation is that it is composable like normal functions on the contrary to the abortive object generated by **call/cc**.

In either case, the first view does not lead to a proper computational interpretation for delimited continuations.

5.2 The Second View

Our second view implies the following definition:

$$\begin{array}{rcl} \xi_\alpha k.M & \triangleq & \alpha(\lambda k.\mathtt{throw}_{\alpha'} M) \\ \#_\alpha M & \triangleq & \mathtt{catch}_{\alpha'}(\lambda v.v(\lambda m.m\lambda u.v(\lambda y.u)))(\lambda\alpha.M) \end{array}$$

where α' is a new tag.

Let E be an evaluation context which contains exactly one instance of a hole, namely the term $E[\xi_\alpha k.M]$ contains exactly one occurrence of ξ. Let $\mathtt{R}$ be $\lambda v.v(\lambda m.m\lambda u.v(\lambda y.u))$. Then we have the following reduction sequence:

$$\begin{array}{rcl} & & \#_\alpha E[\xi_\alpha k.M] \\ & \equiv & \mathtt{catch}_{\alpha'}\mathtt{R}(\lambda\alpha.E[\alpha(\lambda k.\mathtt{throw}_{\alpha'} M)]) \\ & \xrightarrow{*} & \mathtt{catch}_{\alpha'}(\lambda\alpha.E[\alpha(\lambda k.\mathtt{throw}_{\alpha'} M)])(\lambda m.m\lambda u.E[u]) \\ & \to & \mathtt{catch}_{\alpha'} E[(\lambda m.m\lambda u.E[u])(\lambda k.\mathtt{throw}_{\alpha'} M)] \\ & \xrightarrow{*} & \mathtt{catch}_{\alpha'} E[\mathtt{throw}_{\alpha'} M\{k := \lambda u.E[u]\}] \end{array}$$

Let N be the final term of the above reduction sequence. Suppose $M\{k := \lambda u.E[u]\}$ reduces to a value V, then N reduces to $\mathtt{catch}_{\alpha'} E[\mathtt{throw}_{\alpha'} V]$, and then reduces to V. Suppose $M\{k := \lambda u.E[u]\}$ reduces to $\mathtt{throw}_\beta V$. Then N also reduces to $\mathtt{throw}_\beta V$. Consequently, two terms N and $M\{k := \lambda u.E[u]\}$ are observationally equivalent. In summary, the behavior of the term $\#_\alpha E[\xi_\alpha k.M]$ is the same as that of $M\{k := \lambda u.E[u]\}$.

This result shows that the computational behavior of our new control operators are the same as that of the original shift/reset operators. In particular, the result of the above computation contains a functional object $\lambda u.E[u]$, which represents the intended delimited continuation object, thus our encoding of $\#$ and ξ can be thought as another representation of control operators for delimited continuations.

If E contains more than one hole, the result may not be the same as that of the original shift/reset. That is, our encoding generated a similar, but different control operator than the existing one.

Yet, we can argue that a large number of examples contains only one occurrence of the shift operator; Danvy's flip-flop, Danvy-Filinski's CPS-translation [2], and Thiemann's partial evaluator [14]; for these examples, our encoding can be thought as providing a sound logical foundations, by which we can reason about properties of controlful programs.

In this analysis, we used the static catch/throw calculus, but the encoding can be done by the call-by-value version of Parigot's $\lambda\mu$-calculus.

6 Implications of Our Encoding

Since our encoding of control operators uses only logically established machinery which enjoys a number of good properties, the following are obtained straightforwardly:

1. Higher-orderness; types of the delimiters were limited to atomic types for shift/reset in order to obtain a type-preserving CPS-translation, while arbitrary types are allowed in our case without loss of the type-preserving CPS-translation and the strong normalization.
2. Properties such as subject reduction, confluence, and strong normalization are obtained automatically from those for the static catch/throw calculus.
3. A simple type-preserving CPS-translation is obtained by the type-preserving CPS-translation of the call-by-value version of $\lambda\mu$-calculus.

Since most of these results are apparent, we only mention a few properties and the CPS-translation in what follows.

6.1 Subject Reduction and Strong Normalization

Our encoding of control operators is within the scope of $\mathrm{LK}^{\mathrm{CBV}}_{c/t}$, they inherit nice properties from the calculus. In [8],

we proved the subject reduction and the strong normalization of the non-deterministic version of the catch/throw calculi. This version contains the call-by-value version and the call-by-name version of catch/throw calculi, so these properties for $\mathrm{LK}^{\mathrm{CBV}}_{c/t}$ is guaranteed.

Theorem 1 *The calculus for the control operators (encoded using the catch/throw operators) enjoys the subject reduction property.*

Theorem 2 *The calculus for the control operators (encoded using the catch/throw operators) are strongly normalizing.*

The latter result implies that our operators differ from the original shift/reset control operators, since we can write non-terminating reduction sequence using the shift/reset operators [3].

6.2 CPS-translation

A CPS-translation of $\mathrm{LK}^{\mathrm{CBV}}_{c/t}$ is given in a standar way. First, we give translation of a value V (denoted as V^*).

$$\begin{array}{rcl} x^* & \equiv & x \\ (\lambda x.M)^* & \equiv & \lambda x.[[M]] \end{array}$$

Then the CPS-transform of a term M (denoted as $[[M]]$) is given by:

$$\begin{array}{rcl} [[V]] & \equiv & V* \\ [[MN]] & \equiv & \lambda k.[[M]]\lambda m.[[N]]\lambda n.mnk \\ [[\mathtt{catch}_\alpha M]] & \equiv & \lambda\alpha.[[M]]\alpha \\ [[\mathtt{throw}_\alpha M]] & \equiv & \lambda k.[[M]]\alpha \end{array}$$

Considering these definitions and by simple calculation, we can easily obtain the CPS-transform of obtained control operators.

$$[[\xi_\alpha c.M]] \quad \stackrel{*}{\to} \quad \lambda k.\alpha(\lambda c.\lambda k'.[[M]]\alpha')k$$

$$\begin{array}{l} [[\#_\alpha M]] \\ \stackrel{*}{\to} \quad \lambda\alpha'.[[M]]\{\alpha := (\lambda m.m\lambda u.(\lambda\alpha.[[M]])(\lambda yL.Lu))\}\alpha' \end{array}$$

Compared to the simple CPS-translation given to the shift/reset operators, the result of ours is very complex, but we can obtain the following information:

(i) Our reset operator (#) creates a functional object $\lambda u.(\lambda\alpha.[[M]])(\lambda yL.Lu)$ which is roughly an interpretation of $\lambda u.M\{\alpha := u\}$.

(ii) $\lambda yL.Lu$ will be substituted for the α in the second occurrence of $[[M]]$, which means that the second call of our shift operator (ξ) does not really invoke the reified delimited continuation. Rather, it simply ignores it. (Recall that our encoding coincides with the real shift/reset operators only when there is only one occurrence of the shift operator.)

7 Conclusion: What did We Obtain ?

Our conclusion of this expository work is that, a static aspect of delimited continuation operators can be grasped using the sound logical system plus some encoding. We used the static catch/throw calculus as the base system in this paper, but the call-by-value variant of $\lambda\mu$-calculus can be also used (and can be better, since it has stronger reductions).

Apparently, our result does not give full understanding for shift/reset operators; our result applies only to the static variants (such as Hieb et al's subcontinuations), and also is limited to side-effect free calculi (since we duplicated the context). On the other hand, the static nature can be compromised by using the abstraction mechanism of the tag α, and we believe that this work gives a first step towards logical understanding of control operators for delimited continuations.

Acknowledgement The author would like to thank Olivier Danvy, Andrzej Filinski and Hayo Thielecke for helpful discussions. He also thanks Amr Sabry and anonymous referees of CW'01 for constructive comments. This work is supported in part by Inamori Foundation and Grant-in-Aid for Scientific Research by Ministry of Education, Japan, No. 11780213.

REFERENCES

[1] O. Danvy and A. Filinski, "Abstracting Control", Proc. 1990 ACM Conference on Lisp and Functional Programming, pp. 151-160, 1990.

[2] O. Danvy and A. Filinski, "Representing Control: a Study of the CPS Transformation", Mathematical Structures in Computer Science 2(4), pp. 361-391, 1992.

[3] O. Danvy, private communication, 2000.

[4] P. de Groote, "A CPS-Translation of the $\lambda\mu$-Calculus", Lecture Notes in Computer Science **787**, pp. 85-99, 1994.

[5] M. Felleisen, "The Theory and Practice of First-Class Prompts", Proc. 15th ACM Symp. on Principles of Programming Languages, pp. 180-190, 1988.

[6] R. Hieb, R. Dybvig, and C. W. Anderson, "Subcontinuations", Lisp and Symbolic Computation 6, pp. 453–484, 1993.

[7] Y. Kameyama, "A Type-theoretic Study on Partial Continuations", Proc. IFIP International Conference on Theoretical Computer Science (IFIP TCS2000), Sendai, Japan, Lecture Notes in Computer Science **1872**, pp.489-504, 2000.

[8] Y. Kameyama and M. Sato, "Strong Normalizability of the Non-deterministic Catch/Throw Calculi", *Theoretical Computer Science*, to appear.

[9] C. Murthy, "Control Operators, Hierarchies, and Pseudo-Classical Type Systems: A-Translation at Work", Proc. ACM Workshop on Continuations, pp. 49–71, 1992.

[10] H. Nakano, "A Constructive Formalization of the Catch and Throw Mechanism", *Conf. Rec. IEEE Symposium on Logic in Computer Science*, pp. 82–89, 1992.

[11] M. Parigot, "$\lambda\mu$-Calculus: An Algorithmic Interpretation of Classical Natural Deduction", Proc. Intl Conf. on Logic Programming and Automated Reasoning, *Lecture Notes in Computer Science* **624** pp. 199–201, 1992.

[12] M. Sato, "Intuitionistic and Classical Natural Deduction Systems with the Catch and the Throw Rules", *Theoretical Computer Science* 175 (1), pp. 75-92, 1997.

[13] H. Thielecke, private communication, 2000.

[14] P. Thiemann, "Cogen in Six Lines", International Conference on Functional Programming, pp. 180-189, 1995.

CPS Transformation of Beta-Redexes

Olivier Danvy and Lasse R. Nielsen

BRICS *
Department of Computer Science
University of Aarhus †

Abstract

The extra compaction of Sabry and Felleisen's transformation is due to making continuations occur first in CPS terms and classifying more redexes as administrative. We show that the extra compaction is actually independent of the relative positions of values and continuations and furthermore that it is solely due to a context-sensitive transformation of beta-redexes. We stage the more compact CPS transformation into a first-order uncurrying phase and a context-insensitive CPS transformation. We also define a context-insensitive CPS transformation that is just as compact. This CPS transformation operates in one pass and is dependently typed.

Keywords

Continuation-passing style (CPS), Plotkin, Fischer, one-pass CPS transformation, two-level λ-calculus, generalized reduction.

1 Introduction

1.1 Continuation-passing style (CPS)

The meaning of a λ-term, in general, depends on its evaluation order. Evaluation-order independence was one of the motivations for continuations [14, 21], and continuation-passing style was developed as an evaluation-order independent λ-encoding of λ-terms [4, 13]. In this λ-encoding, each evaluation context is represented by a λ-abstraction, called a *continuation*, and each λ-abstraction is passed a continuation in addition to its usual argument. All intermediate results are sent to a continuation and thus all calls are tail-calls. This λ-encoding gives rise to a variety of continuation-passing styles, whose structure is a subject of study in itself [8, 15, 20].

*Basic Research in Computer Science (`http://www.brics.dk/`), Centre of the Danish National Research Foundation.
†Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark. E-mail: {`danvy,lrn`}`@brics.dk`

1.2 The CPS transformation

The format of CPS λ-terms was soon noticed to be of interest for the compiler writer [18], which in turn fostered interest in automating the transformation of λ-terms into CPS. Over the last twenty years, a wide body of CPS transformations has thus been developed for various purposes, e.g., to compile and to analyze programs, and to generate compilers [1, 6, 10, 17, 18, 22].

The naïve λ-encoding into CPS, however, generates a quite impressive inflation of lambdas, most of which form *administrative redexes* that can be safely reduced. Administrative reductions yield CPS terms corresponding to what one could write by hand. It has therefore become a challenge to eliminate as many administrative redexes as possible, at CPS-transformation time.

1.3 Sabry and Felleisen's optimization

In their article "Reasoning about Programs in Continuation-Passing Style" [16], Sabry and Felleisen present a CPS transformation that yields more compact terms than existing CPS transformations. For example [16, Footnote 6], CPS-transforming

$$((\lambda x.\lambda y.x)\, a)\, b$$

where a and b are variables, yields the term

$$\lambda k.((\lambda x.((\lambda y.k\, x)\, b))\, a)$$

whereas earlier transformations, such as Steele's [18] or Danvy and Filinski's [3], yield the more voluminous term

$$\lambda k.((\lambda x.\lambda k_1.(k_1\, (\lambda y.\lambda k_2.k_2\, x)))\, a\, (\lambda m.m\, b\, k)).$$

Sabry and Felleisen's optimization relies on using Fischer's CPS (where continuations occur first, as in $\lambda k.\lambda x.e$), whereas earlier transformations use Plotkin's CPS (where values occur first, as in $\lambda x.\lambda k.e$).

1.4 This article

Section 2 reviews administrative reductions in the CPS transformation and characterizes Sabry and Felleisen's optimization, independent of the relative positions of values and continuations in CPS terms (i.e., both for Fischer's and Plotkin's CPS). Section 3 constructs a similarly compact CPS transformation by composing an uncurrying phase and an ordinary CPS transformation. Section 4 integrates the optimization in a context-insensitive, one-pass CPS transformation. Section 5 concludes.

2 Administrative reductions in the CPS transformation

2.1 Context-insensitive administrative reductions

Appel, Danvy and Filinski, and Wand each independently developed a "one-pass" CPS transformation for call by value [1, 3, 22]. This CPS transformation relies on a context-free characterization of administrative reductions, i.e., a characterization that is independent of any source term. This one-pass transformation, shown below for Plotkin's CPS, is formulated with a static, context-free distinction between (translation-time) administrative reductions and (run-time) reductions, using a two-level λ-calculus [3, 12].

$$\begin{array}{rcl}
[\![\cdot]\!]_{\mathrm{p}} & : & \Lambda \to (\Lambda \to \Lambda) \to \Lambda \\
[\![x]\!]_{\mathrm{p}} & = & \overline{\lambda}\kappa.\kappa\,\overline{@}\,x \\
[\![\lambda x.e]\!]_{\mathrm{p}} & = & \overline{\lambda}\kappa.\kappa\,\overline{@}\,(\underline{\lambda}x.\underline{\lambda}k.[\![e]\!]_{\mathrm{p}}\,\overline{@}\,(\overline{\lambda}t.k\,\underline{@}\,t)) \\
[\![e_0\,e_1]\!]_{\mathrm{p}} & = & \overline{\lambda}\kappa.[\![e_0]\!]_{\mathrm{p}}\,\overline{@}\,(\overline{\lambda}t_0.[\![e_1]\!]_{\mathrm{p}}\,\overline{@}\,(\overline{\lambda}t_1.(t_0\,\underline{@}\,t_1)\,\underline{@}\,(\underline{\lambda}v.\kappa\,\overline{@}\,v)))
\end{array}$$

"$\underline{\lambda}$" and "$\underline{@}$" denote hygienic abstract-syntax constructors and "$\overline{\lambda}$" and "$\overline{@}$" denote translation-time abstractions and (infix) applications, respectively. A λ-term $e : \Lambda$ is CPS-transformed with

$$\underline{\lambda}k.[\![e]\!]_{\mathrm{p}}\,\overline{@}\,(\overline{\lambda}t.k\,\underline{@}\,t).$$

The corresponding one-pass transformation for Fischer's CPS is as follows.

$$\begin{array}{rcl}
[\![\cdot]\!]_{\mathrm{f}} & : & \Lambda \to (\Lambda \to \Lambda) \to \Lambda \\
[\![x]\!]_{\mathrm{f}} & = & \overline{\lambda}\kappa.\kappa\,\overline{@}\,x \\
[\![\lambda x.e]\!]_{\mathrm{f}} & = & \overline{\lambda}\kappa.\kappa\,\overline{@}\,(\underline{\lambda}k.\underline{\lambda}x.[\![e]\!]_{\mathrm{f}}\,\overline{@}\,(\overline{\lambda}t.k\,\underline{@}\,t)) \\
[\![e_0\,e_1]\!]_{\mathrm{f}} & = & \overline{\lambda}\kappa.[\![e_0]\!]_{\mathrm{f}}\,\overline{@}\,(\overline{\lambda}t_0.[\![e_1]\!]_{\mathrm{f}}\,\overline{@}\,(\overline{\lambda}t_1.(t_0\,\underline{@}\,(\underline{\lambda}v.\kappa\,\overline{@}\,v))\,\underline{@}\,t_1))
\end{array}$$

A λ-term $e : \Lambda$ is CPS-transformed with

$$\underline{\lambda}k.[\![e]\!]_{\mathrm{f}}\,\overline{@}\,(\overline{\lambda}t.k\,\underline{@}\,t).$$

2.2 Context-sensitive administrative reductions

Sabry and Felleisen (1) tag all the "new" lambdas introduced by the CPS transformation, (2) reduce systematically the β-redexes with a tagged lambda, and (3) untag the remaining tagged lambdas:

$$\begin{array}{rcl}
[\![x]\!] & = & \overline{\lambda}k.k\,x \\
[\![\lambda x.e]\!] & = & \overline{\lambda}k.k\,(\overline{\lambda}k.\lambda x.[\![e]\!]\,k) \\
[\![e_0\,e_1]\!] & = & \overline{\lambda}k.[\![e_0]\!]\,(\overline{\lambda}t_0.[\![e_1]\!]\,(\overline{\lambda}t_1.t_0\,k\,t_1))
\end{array}$$

A λ-term e is CPS-transformed with

$$[\![e]\!].$$

An administrative reduction amounts to reducing a β-redex where the λ-abstraction is tagged.

This three-pass CPS transformation resembles much the Fischer-style one-pass CPS transformation of Section 2.1, with three exceptions:

Form: it does not use @ for applications, is more implicit by not underlining abstract-syntax constructors, and η-reduces continuations.

Content: it is a first-order rewriting system whereas the one-pass transformation is a higher-order one.

Plus: it contains one more overlined λ-abstraction, namely the one declaring the continuation of a λ-abstraction.

The extra overline makes administrative reductions context-sensitive, as illustrated below:

$$\begin{array}{l}
[\![\lambda x.((\lambda y.y)\,x)]\!] = \\
\overline{\lambda}k.k\,(\overline{\lambda}k.\lambda x.(\overline{\lambda}k.k\,(\overline{\lambda}k.\lambda y.(\overline{\lambda}k.k\,y)\,k))\,\overline{\lambda}t_0.(\overline{\lambda}k.k\,x)\,\overline{\lambda}t_1.t_0\,k\,t_1) \\
\longrightarrow_{\beta^+} \overline{\lambda}k.k\,(\overline{\lambda}k.\lambda x.(\overline{\lambda}k.\lambda y.k\,y)\,k\,x) \\
\longrightarrow_{\overline{\beta}} \overline{\lambda}k.k\,(\overline{\lambda}k.\lambda x.(\lambda y.k\,y)\,x)
\end{array}$$

The term $\overline{\lambda}k.\lambda x....$ arises from the transformation of $\lambda x....$ and cannot be administratively reduced. The term $\overline{\lambda}k.\lambda y....$ arises from the transformation of $\lambda y....$ and can be administratively reduced.

In contrast, in a context-insensitive one-pass CPS transformation, all overlined λ-abstractions are guaranteed to occur in an overlined application (and thus there is no need for post-erasure). A context-sensitive CPS transformation thus can perform more administrative reductions than a context-insensitive one.

Furthermore, we can precisely locate the extra gain: for source β-redexes. Given a source β-redex, one can actually substitute the continuation of the application for the continuation of the abstraction:

$$(\lambda x.e[c/k])\,t_1$$

thereby enabling further administrative reductions inside e.

This reduction is not accounted for in a (say, Plotkin-style) one-pass CPS transformation, since in the particular case where t_0 denotes $\underline{\lambda}x.\underline{\lambda}k.e$, one does not simplify

$$(t_0\,\underline{@}\,t_1)\,\underline{@}\,c$$

into

$$(\underline{\lambda}x.e[c/k])\,\underline{@}\,t_1\,.$$

The reduction thus yields more compact CPS counterparts of source β-redexes, in that the translated λ-abstractions are not explicitly passed any continuation when they occur in a β-redex.[1]

On the other hand, a similar phenomenon occurs for let expressions, as reviewed next.

2.3 CPS transformation of let expressions

The CPS transformation of let expressions reads as follows:

$$[\![\text{let } x = e' \text{ in } e]\!] \quad = \quad \overline{\lambda}\kappa.[\![e']\!]\overline{\lambda}t'.\underline{\text{let}}\ x = t'\ \underline{\text{in}}\ [\![e]\!]\kappa$$

In words, e is in tail-position in the let expression, and is CPS-transformed with respect to the same κ as the let expression. This technique is instrumental in continuation-based partial evaluation [11].

Seeing let expressions as syntactic sugar for β-redexes, it appears clearly that the context-sensitive administrative reduction includes the standard let optimization, independently of whether continuations are put first or last. This administrative reduction, however, yields more.

[1] As Shivers puts it [17] and can be read off their type, the translated λ-abstractions are promoted to continuations.

2.4 CPS transformation of embedded β-redexes

Extra mileage is obtained for fully applied (curried) λ-abstractions. CPS-transforming the curried application of a "n-ary" λ-abstraction to n arguments relocates the continuation of the application to the body of the λ-abstraction.

$$\begin{array}{l}[\![(\lambda x_1 .... \lambda x_n . e)\, e_1 ... \, e_n]\!] = \\ \underline{\lambda}\kappa.[\![e_1]\!]\, \overline{@}\, (\overline{\lambda} t_1 .... [\![e_n]\!]\, \overline{@}\, (\overline{\lambda} t_n . (\underline{\lambda} x_n .... (\underline{\lambda} x_1 . [\![e]\!]\, \overline{@}\, \kappa)\, \underline{@}\, t_1 ...)\, \underline{@}\, t_n) ...)\end{array}$$

This extra mileage is independent of whether continuations are put first or last.

As a net effect, a term such as

$$(\lambda f.\lambda g.\lambda x.f\, x\, (g\, x))\, (a\, b)\, c\, (d\, e)$$

where a, b, c, d, and e are variables, is CPS transformed into (letting continuations occur last)

$$\lambda k.a\, b\, (\lambda f.(\lambda g.d\, e\, (\lambda x.f\, x\, (\lambda v_1 .g\, x\, (\lambda v_2 .v_1\, v_2\, k))))\, c).$$

Observe how the λ-abstractions λf.... and λx.... end up as the continuations of the applications $(a\, b)$ and $(d\, e)$, and how the application of λg.... to c survives in the CPS term.

Letting continuations occur first would yield a similar term:

$$\lambda k.a\, (\lambda f.(\lambda g.d\, (\lambda x.f\, (\lambda v_1 .g\, (\lambda v_2 .v_1\, k\, v_2)\, x)\, x)\, e)\, c)\, b.$$

2.5 Summary and conclusion

A CPS transformation with context-sensitive administrative reductions yields more compact CPS terms because it exposes more administrative redexes. The extra administrative reductions affect nested β-redexes corresponding to fully applied curried λ-abstractions, and reduce continuation-passing by promoting the inner λ-abstractions to continuations. These extra administrative reductions can be carried out independently of whether continuations occur first or last in CPS terms.

3 Staging the more compact CPS transformation

Sabry and Felleisen [16, Definition 7, page 306] identify a reduction β_{lift} moving the context of a β-redex into the body of the corresponding λ-abstraction:[2]

$$E[(\lambda x.M)N] \longrightarrow (\lambda x.E[M])N \quad \text{where } E \neq [\,] \text{ and } x \notin FV(E) \qquad (\beta_{lift})$$

They also pointed out that CPS-transforming a term e and mapping the result back to direct style yields a term in β_{lift}-normal form.

But a term in β_{lift}-normal form does not give rise to the extra context-sensitive administrative reduction of Section 2. Therefore, the extra power of the context-sensitive CPS transformation is solely due to β_{lift}.

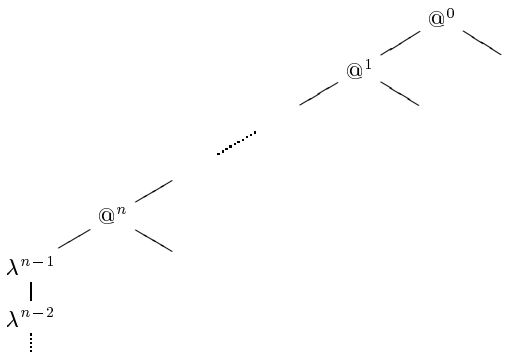
The more compact CPS transformation can thus be staged as follows:

1. a phase uncurrying (and appropriately renaming, if need be) all β-redexes $(\lambda x_1 .... \lambda x_n . e)\, e_1 ... \, e_n$ into embedded let expressions $\begin{array}{l}\text{let } x_1 = e_1 \\ \text{in let } x_2 = e_2 \\ \text{in } ... \text{ let } x_n = e_n \\ \text{in } e\end{array}$
2. an ordinary, context-insensitive CPS transformation (either à la Plotkin or à la Fischer) handling let expressions.

The benefit of this staging, we believe, is three-fold: (1) it clarifies the extra compaction; (2) it extends a context-insensitive, one-pass CPS transformation; and (3) it suggests how to obtain even more compact terms. Indeed, in the same fashion as control-flow analysis can be used to locate the application sites of curried λ-abstractions in order to uncurry them [1, 7], the CPS transformation can benefit from control-flow information to promote more functions to continuations.

4 More compact CPS transformations in one pass

Promoting functions into continuations compromises context independence in the CPS transformation, since how to CPS-transform a λ-abstraction depends on whether it occurs in a β-redex or not. Fortunately, it does so in a very regular way, which makes it possible to derive a *family* of one-pass CPS transformations indexed by positions in the current context.



Indexing the transformation functions with the lexical position of their argument yields the one-pass CPS transformation à la Plotkin of Figure 1. A λ-term $e : \Lambda$ is CPS-transformed with

$$\underline{\lambda}k.[\![e]\!]^0\, \overline{@}\, (\overline{\lambda} t.k\, \underline{@}\, t).$$

Similarly, a one-pass CPS transformation à la Fischer is displayed in Figure 2.

$[\![\cdot]\!]^0$ is applied to the root of a term (i.e., to the body of a λ-abstraction or to the expression in position of argument in an application). For $n > 0$, $[\![\cdot]\!]^n$ is applied to an expression in position of function in an application. n is the depth of the expression since the closest root, as in the picture above. Ψ (resp. Φ) coerces a syntactic object into a translation-time one.

The transformation based on these families of functions can be proven correct by a simulation theorem similar to Plotkin's [13]. The correctness criterion is a relation between the transformation of the result of an expression and the result of the transformation of it, i.e., (noting contextual equivalence with $\sim$)

$$e \longrightarrow^* v \quad \text{implies} \quad [\![e]\!]^0\, \lambda a.a \longrightarrow^* v' \quad \text{and} \quad v' \sim [\![v]\!]^0\, \lambda a.a$$

as well as preservation of non-termination and of getting stuck.

[2] The transitive closure of β_{lift} is a generalized reduction in the sense of Bloo, Kamareddine, and Nederpelt [2].

$$
\begin{array}{rcl}
\Psi^0\, v & = & v \;:\; \tau_0 \\
 & & \quad \text{where } \tau_0 = \Lambda. \\
\Psi^{n+1}\, v & = & \overline{\lambda} t.\overline{\lambda}\kappa.(t\,\underline{@}\,v)\,\underline{@}\,(\underline{\lambda} v'.\kappa\,\overline{@}\,(\Psi^n\, v')) \;:\; \tau_{n+1} \\
 & & \quad \text{where } \tau_{n+1} = \Lambda \to (\tau_n \to \Lambda) \to \Lambda. \\
 & & \\
[\![\cdot]\!]^n & : & \Lambda \to (\tau_n \to \Lambda) \to \Lambda \\
[\![x]\!]^n & = & \overline{\lambda}\kappa.\kappa\,\overline{@}\,(\Psi^n\, x) \\
[\![\lambda x.e]\!]^0 & = & \overline{\lambda}\kappa.\kappa\,\overline{@}\,(\underline{\lambda} x.\underline{\lambda} k.[\![e]\!]^0\,\overline{@}\,(\overline{\lambda} t.k\,\underline{@}\,t)) \\
[\![\lambda x.e]\!]^{n+1} & = & \overline{\lambda}\kappa.\kappa\,\overline{@}\,(\overline{\lambda} t.\overline{\lambda}\kappa'.(\underline{\lambda} x.[\![e]\!]^n\,\overline{@}\,\kappa')\,\underline{@}\,t) \\
[\![e_0\, e_1]\!]^n & = & \overline{\lambda}\kappa.[\![e_0]\!]^{n+1}\,\overline{@}\,(\overline{\lambda} t_0.[\![e_1]\!]^0\,\overline{@}\,(\overline{\lambda} t_1.(t_0\,\overline{@}\,t_1)\,\overline{@}\,\kappa))
\end{array}
$$

Figure 1: A family of one-pass, call-by-value CPS transformations à la Plotkin

$$
\begin{array}{rcl}
\Phi^0\, v & = & v \;:\; \tau_0 \\
 & & \quad \text{where } \tau_0 = \Lambda. \\
\Phi^{n+1}\, v & = & \overline{\lambda}\kappa.v\,\underline{@}\,(\underline{\lambda} v'.\kappa\,\overline{@}\,(\Phi^n\, v')) \;:\; \tau_{n+1} \\
 & & \quad \text{where } \tau_{n+1} = (\tau_n \to \Lambda) \to \Lambda. \\
 & & \\
[\![\cdot]\!]^n & : & \Lambda \to (\tau_n \to \Lambda) \to \Lambda \\
[\![x]\!]^n & = & \overline{\lambda}\kappa.\kappa\,\overline{@}\,(\Phi^n\, x) \\
[\![\lambda x.e]\!]^0 & = & \overline{\lambda}\kappa.\kappa\,\overline{@}\,(\underline{\lambda} k.\underline{\lambda} x.[\![e]\!]^0\,\overline{@}\,(\overline{\lambda} t.k\,\underline{@}\,t)) \\
[\![\lambda x.e]\!]^{n+1} & = & \overline{\lambda}\kappa.\kappa\,\overline{@}\,(\overline{\lambda}\kappa'.\underline{\lambda} x.[\![e]\!]^n\,\overline{@}\,\kappa') \\
[\![e_0\, e_1]\!]^n & = & \overline{\lambda}\kappa.[\![e_0]\!]^{n+1}\,\overline{@}\,(\overline{\lambda} t_0.[\![e_1]\!]^0\,\overline{@}\,(\overline{\lambda} t_1.(t_0\,\overline{@}\,\kappa)\,\underline{@}\,t_1))
\end{array}
$$

Figure 2: A family of one-pass, call-by-value CPS transformations à la Fischer

Reflecting the context dependence of both CPS transformations, the two-level specifications in Figures 1 and 2 are not themselves simply typed. Instead, they are dependently typed and define two families of simply typed two-level specifications. Each of these families produces simply-typed two-level λ-terms, that can be statically (i.e., administratively) reduced in one pass.

Figures 1 and 2 can be programmed in a dependently typed language and also in Scheme, if one treats the indices as arguments.

5 Conclusion and issues

In their study of CPS programs [16], Sabry and Felleisen needed a CPS transformation that would perform more administrative reductions than the ones already available [1, 3, 6, 22]. We have identified the extra power of this CPS transformation: a context-sensitive administrative reduction enabling a more effective treatment of β-redexes which corresponds to Bloo, Kamareddine, and Nederpelt's notion of generalized reduction. This treatment turns out to be independent of the relative positions of values and continuations. The resulting CPS transformation can be factored into (1) a first-order uncurrying phase and (2) a CPS transformation with context-insensitive administrative reductions. We have also presented two one-pass CPS transformations embodying the extra compaction and generalizing the corresponding one-pass CPS transformations à la Plotkin and à la Fischer. They can be adapted *mutatis mutandis* for encoding λ-terms into monadic normal form [9], A-normal form [5, Figure 9], nqCPS, etc., including β_{lift}.

Acknowledgements

The first author is grateful to Matthias Felleisen, Andrzej Filinski, John Hatcliff, and Amr Sabry for discussions and comments on this topic and these transformations in June and July 1993, at CMU. Kristoffer Rose wanted to see the dependent types of Figures 1 and 2 spelled out. Thanks are also due to the reviewers and to Julia Lawall for perceptive comments.

References

[1] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, New York, 1992.

[2] Roel Bloo, Fairouz Kamareddine, and Rob Nederpelt. The Barendregt cube with definitions and generalised reduction. *Information and Computation*, 126(2):123–143, 1996.

[3] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[4] Michael J. Fischer. Lambda-calculus schemata. In Talcott [19], pages 259–288. An earlier version appeared in an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No. 1, January 1972.

[5] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In David W. Wall, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 28, No 6, pages 237–247, Albuquerque, New Mexico, June 1993. ACM Press.

[6] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. The MIT Press and McGraw-Hill, 1991.

[7] John Hannan and Patrick Hicks. Higher-order un-Currying. *Higher-Order and Symbolic Computation*, 13(3):179–218, 2000.

[8] John Hatcliff. *The Structure of Continuation-Passing Styles*. PhD thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas, June 1994.

[9] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 458–471, Portland, Oregon, January 1994. ACM Press.

[10] David Kranz, Richard Kesley, Jonathan Rees, Paul Hudak, Jonathan Philbin, and Norman Adams. Orbit: An optimizing compiler for Scheme. In *Proceedings of the ACM SIGPLAN'86 Symposium on Compiler Construction*, pages 219–233, Palo Alto, California, June 1986. ACM Press.

[11] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In Carolyn L. Talcott, editor, *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, Orlando, Florida, June 1994. ACM Press.

[12] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[13] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[14] John C. Reynolds. The discoveries of continuations. In Talcott [19], pages 233–247.

[15] Amr Sabry. *The Formal Relationship between Direct and Continuation-Passing Style Optimizing Compilers: A Synthesis of Two Paradigms*. PhD thesis, Computer Science Department, Rice University, Houston, Texas, August 1994. Technical report TR94-242.

[16] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In Talcott [19], pages 289–360.

[17] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

[18] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.

[19] Carolyn L. Talcott, editor. *Special issue on continuations (Part I)*, Lisp and Symbolic Computation, Vol. 6, Nos. 3/4, December 1993.

[20] Hayo Thielecke. *Categorical Structure of Continuation Passing Style*. PhD thesis, University of Edinburgh, Edinburgh, Scotland, 1985. ECS-LFCS-97-376.

[21] Christopher P. Wadsworth. Continuations revisited. *Higher-Order and Symbolic Computation*, 13(1/2):131–133, 2000.

[22] Mitchell Wand. Correctness of procedure representations in higher-order assembly language. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Proceedings of the 7th International Conference on Mathematical Foundations of Programming Semantics*, number 598 in Lecture Notes in Computer Science, pages 294–311, Pittsburgh, Pennsylvania, March 1991. Springer-Verlag.

An Extensional CPS Transform
(Preliminary Report)

Andrzej Filinski

BRICS*, Department of Computer Science, University of Aarhus
Ny Munkegade, bldg. 540, DK-8000 Aarhus C, Denmark
andrzej@brics.dk

Abstract

We show that, in a language with general continuation-effects, the syntactic, or *intensional*, CPS transform is mirrored by a semantic, or *extensional*, functional term. In other words, from only the observable behavior any direct-style term (possibly containing the usual first-class continuation primitives), we can uniformly extract the observable behavior of its CPS counterpart. As a consequence of this result, we show that the computational lambda-calculus is complete for observational equivalence of pure, simply typed lambda-terms in Scheme-like contexts.

1 Introduction

CPS transformations are usually defined on the syntax of terms. For example, Plotkin's CBV transform [Plo75] defines, for any direct-style term E, the CPS term $\overline{E}$ by structural induction on E. Variations are possible, such as one-pass optimizing transforms [DF92], but all have in common that they are intensional: they act on representations of the program texts, rather than on the meanings of the programs themselves.

The continuation-passing form of a term seems to contain strictly more semantic information than its direct-style counterpart: Meyer and Wand showed that, in the typed case, there exists for any type τ, a term D_τ such that for all closed $E : \tau$, $\mathsf{D}_\tau\,\overline{E} =_{\beta\eta} E$ [MW85]. In fact, we can even obtain $\mathsf{D}_\tau\,\overline{E} =_{\lambda_c} E$ [Kuc98], where λ_c is the computational lambda-calculus [Mog89]. On the other hand, functionally extracting the CPS meaning of a term from its direct meaning seems impossible. Indeed, Meyer and Riecke showed that there can be no lambda-term C_τ such that $\mathsf{C}_\tau\,E =_{\beta\eta} \overline{E}$ [MR88]. Adding recursion and similar operations does not help in defining C. The primary reason is that – assuming the usual left-to-right evaluation order – terms such as $f\,x\,(g\,x)$ and $(\lambda y.\,f\,x\,y)\,(g\,x)$ are indistinguishable (because they have the same denotation in set- and domain-theoretic models), yet their CPS counterparts can be easily distinguished.

With additional effects in the language, such as first-class continuations, we can distinguish the above terms. However, even the call/cc operator apparently cannot distinguish $f\,x$ and $(\lambda y.\,f\,x)\,(f\,x)$. On the other hand, Sitaram and Felleisen showed that, after adding the further control operators "abort" and "prompt", all terms with different continuation-passing denotations actually become observationally distinguishable [SF90]. This opens – at least in principle – the possibility of defining a suitable C_τ in terms of sufficiently powerful control primitives. In this note we will define such a term and outline some of its properties and applications.

*Basic Research in Computer Science (www.brics.dk), Centre of the Danish National Research Foundation.

2 Extensional CPS transformation by monadic reflection

In the following we first present a unified, effect-typed language expressive enough to embed both the source and target languages of the CPS transform, and then define the extensional CPS transform and show its correctness with respect to an equational theory for the unified language.

2.1 A unified language

We consider a simply typed language L^{p} of pure, direct-style lambda-terms. Based on this, we define a language L^{c} with SML/NJ-style first-class continuations. The type structure of L^{c} is thus:

$$\tau ::= a \mid \tau_1 \to \tau_2 \mid \neg\tau$$

(a ranges over atomic types and $\neg\tau$ is the type of τ-accepting continuations, $\tau\,\mathit{cont}$.) The term syntax of L^{c} consists of the usual constructs for function abstraction and application, as well as two constant families $\mathit{callcc}_\alpha : (\neg\alpha \to \alpha) \to \alpha$ and $\mathit{throw}_{\alpha,\beta} : \neg\alpha \to \alpha \to \beta$ (we will usually omit the explicit type tags):

$$\begin{aligned} V &::= x \mid \lambda x.E \mid \mathit{callcc} \mid \mathit{throw} \\ E &::= V \mid E_1\,E_2 \end{aligned}$$

The typed CBV CPS-transform wrt. an atomic answer type o is based on the following transformation of L^{c}-types to L^{p}-types:

$$\begin{aligned} \overline{a} &= a \\ \overline{\tau_1 \to \tau_2} &= \overline{\tau_1} \to (\overline{\tau_2} \to o) \to o \\ \overline{\neg\tau} &= \overline{\tau} \to o \end{aligned}$$

The term transform is most conveniently expressed with separate translations for values and for general expressions:

$$\begin{aligned} \overline{x} &= x \\ \overline{\lambda x.E} &= \lambda x.\,\overline{\overline{E}} \\ \overline{\mathit{callcc}} &= \lambda f.\lambda k.\,f\,k\,k \\ \overline{\mathit{throw}} &= \lambda c.\lambda k.\,k\,(\lambda a.\lambda k'.\,c\,a) \end{aligned}$$

$$\begin{aligned} \overline{\overline{V}} &= \lambda k.\,k\,\overline{V} \\ \overline{\overline{E_1\,E_2}} &= \lambda k.\,\overline{\overline{E_1}}\,(\lambda f.\,\overline{\overline{E_2}}\,(\lambda a.\,f\,a\,k)) \end{aligned}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau \,!\, \mathbf{n}} \qquad \frac{\Gamma, x{:}\,\tau_1 \vdash E : \tau_2 \,!\, e}{\Gamma \vdash \lambda x.\, E : (\tau_1 \xrightarrow{e} \tau_2) \,!\, \mathbf{n}} \qquad \frac{\Gamma \vdash E_0 : (\tau_1 \xrightarrow{e} \tau_2) \,!\, e \qquad \Gamma \vdash E_1 : \tau_1 \,!\, e}{\Gamma \vdash E_0\, E_1 : \tau_2 \,!\, e}$$

$$\frac{\Gamma \vdash E_1 : \tau_1 \,!\, e \qquad \Gamma, x{:}\,\tau_1 \vdash E_2 : \tau_2 \,!\, e}{\Gamma \vdash \mathbf{let}\ x = E_1\ \mathbf{in}\ E_2 : \tau_2 \,!\, e} \qquad \frac{\Gamma \vdash E : \tau \,!\, \mathbf{k}}{\Gamma \vdash [E] : (\tau \xrightarrow{\mathbf{p}} o) \xrightarrow{\mathbf{p}} o \,!\, \mathbf{n}} \qquad \frac{\Gamma \vdash E : (\tau \xrightarrow{\mathbf{p}} o) \xrightarrow{\mathbf{p}} o \,!\, \mathbf{n}}{\Gamma \vdash \mu(E) : \tau \,!\, \mathbf{k}} \qquad \frac{\Gamma \vdash E : \tau \,!\, e \qquad \tau \,!\, e \leq \tau' \,!\, e'}{\Gamma \vdash E : \tau' \,!\, e'}$$

$$\frac{\tau \leq \tau' \qquad e \preceq e'}{\tau \,!\, e \leq \tau' \,!\, e'} \qquad \frac{}{b \leq b} \qquad \frac{\tau_1' \leq \tau_1 \qquad \tau_2 \,!\, e \leq \tau_2' \,!\, e'}{\tau_1 \xrightarrow{e} \tau_2 \leq \tau_1' \xrightarrow{e'} \tau_2'} \qquad \frac{}{e \preceq e} \qquad \frac{e \preceq e' \qquad e' \preceq e''}{e \preceq e'} \qquad \frac{}{\mathbf{n} \preceq \mathbf{p}} \qquad \frac{}{\mathbf{p} \preceq \mathbf{k}}$$

Figure 1: Effect-typing and subtyping rules for $L^{\mathbf{k}}$.

It is easy to check that the transformation is type-preserving, in the sense that if $\Gamma \vdash_{L^{\mathbf{c}}} E : \tau$ then $\overline{\Gamma} \vdash_{L^{\mathbf{p}}} \overline{\overline{E}} : (\overline{\overline{\tau}} \to o) \to o$, where $\overline{\Gamma}$ is the pointwise type-transformation of the typing context Γ. Note also that both $\overline{V}$ and $\overline{\overline{E}}$ are $L^{\mathbf{p}}$-values.

To express the transform extensionally, we embed both $L^{\mathbf{p}}$ and $L^{\mathbf{c}}$ into a unified language $L^{\mathbf{k}}$, with both "pure" and "impure" function spaces. Applying the extensional transformation to a value V from the $L^{\mathbf{c}}$-fragment will then give the $L^{\mathbf{k}}$-term $\mathsf{C}_\tau V$, which can be shown equivalent to the term $\overline{V}$ from the $L^{\mathbf{p}}$-fragment.

The typing rules of $L^{\mathbf{k}}$ are given in Figure 1; they are essentially a specialization of the multi-effect monadic metalanguage of [Fil99a] to continuation-effects. To accommodate both notions of function space in a single language, we introduce a simple effect-typing discipline, with a typing judgment $\Gamma \vdash E : \tau \,!\, e$, where e is either $\mathbf{k}$ (signifying potential control effects), $\mathbf{p}$ (absence of control effects), or $\mathbf{n}$ (absence of any effects). Likewise, we annotate arrows as $\xrightarrow{e}$, according to the potential effects of the function body. We also allow a simple notion of effect-subtyping, where any $\mathbf{n}$-computation can be seen as a $\mathbf{p}$-computation, and any $\mathbf{p}$-computation as a $\mathbf{k}$-computation. This subtyping extends in the usual covariant-contravariant way to function types.

There is a one-to-one correspondence between general $L^{\mathbf{k}}$-terms of type $\tau \,!\, \mathbf{k}$ and $L^{\mathbf{k}}$-values of type $(\tau \xrightarrow{\mathbf{p}} o) \xrightarrow{\mathbf{p}} o$. This correspondence is made explicit by two special $L^{\mathbf{k}}$-constructs, *monadic reification* $[E]$ and *reflection* $\mu(E)$, expressing the isomorphism.

The embedding of $L^{\mathbf{c}}$ into $L^{\mathbf{k}}$ is now straightforward: values are treated as $\mathbf{n}$-effect terms, non-values as terms with $\mathbf{k}$-effects. Also, the single function type of $L^{\mathbf{c}}$ is annotated as $\xrightarrow{\mathbf{k}}$. The additional type constructor and terms of $L^{\mathbf{c}}$ are desugared as follows:

$$\begin{aligned} \neg\tau &= \tau \xrightarrow{\mathbf{p}} o \\ \mathit{callcc} &= \lambda f.\, \mu(\lambda k.\, [f\, k]\, k) \\ \mathit{throw} &= \lambda c.\, \lambda a.\, \mu(\lambda k'.\, c\, a) \end{aligned}$$

(We do not actually need to assume that $\neg\tau$ and $\tau \xrightarrow{p} o$ are exactly the same type; it suffices that there exist $L^{\mathbf{k}}$-isomorphisms between them.) Similarly, $L^{\mathbf{p}}$ is embedded into $L^{\mathbf{k}}$ by assigning all non-value terms the effect $\mathbf{p}$, and annotating all function spaces as $\xrightarrow{\mathbf{p}}$.

We can extend the CPS-transform to all of $L^{\mathbf{k}}$ by taking the translation of terms with $\mathbf{p}$- and $\mathbf{n}$-effects to be the identity, while $\mathbf{k}$-effects are CPS-translated away as defined above; the details can be found in [Fil99a]. Note that this translation is defined by induction on effect-typing derivations, not on the raw $L^{\mathbf{k}}$-terms themselves; however, it is still observationally coherent, as remarked below.

The extended translation assigns an $L^{\mathbf{p}}$-meaning to all $L^{\mathbf{k}}$-programs, and can thus be taken as the definition of the semantics of $L^{\mathbf{k}}$. We will not actually need the definition of the translation for all of $L^{\mathbf{k}}$ to establish correctness of the extensional transform, however; instead, a sound axiomatization of the equational theory induced by the full translation will suffice.

Note that the reification and reflection operators of $L^{\mathbf{k}}$ also allow us to define the operations *reset* (also known as *prompt*) and *abort*, with typing rules:

$$\frac{\Gamma \vdash E : o \,!\, \mathbf{k}}{\Gamma \vdash \#E : o \,!\, \mathbf{p}} \qquad \frac{}{\Gamma \vdash \mathit{abort}_\tau : (o \xrightarrow{\mathbf{k}} \tau) \,!\, \mathbf{n}}$$

The definitions are very simple:

$$\begin{aligned} \#E &= [E]\,(\lambda r.\, r) \\ \mathit{abort} &= \lambda r.\, \mu(\lambda k.\, r) \end{aligned}$$

That is, $\#E$ evaluates E with the identity continuation, while $\mathit{abort}\, r$ discards the current continuation and returns r as the answer. As shown by Sitaram and Felleisen [SF90], adding these operators, together with *callcc*/*throw* and parallel-if, to CBV PCF makes the usual domain-theoretic continuation semantics fully abstract. Even without parallel-if, however, the control operators have the same expressive power as reification and reflection, because it is easy to show that, in the equational theory of $L^{\mathbf{k}}$,

$$\begin{aligned} [E] &= \lambda k.\, \#(k\, E) \\ \mu(E) &= \mathit{callcc}\,(\lambda c.\, \mathit{abort}\,(E\,(\lambda v.\, \#(\mathit{throw}\, c\, v)))) \end{aligned}$$

These equations are in fact exploited directly in the ML implementation of reflection and reification below.

Finally, let us note that the effect-separated language $L^{\mathbf{k}}$ can actually be implemented by a *direct* embedding into a single-effect language $L^{\mathbf{cs}}$ with first-class continuations and state, such as Scheme or SML/NJ. In other words, the exact choice of a typing derivation for an $L^{\mathbf{k}}$-program does not affect its observable result.

More precisely, we can define an operation $|{-}| : L^{\mathbf{k}} \to L^{\mathbf{cs}}$ that erases all effect-annotations from types and terms, and replaces $\mu(E)$ and $[E]$ with $\mathit{reflect}\, E$ and $\mathit{reify}\,(\lambda().\, E)$, respectively, where

$$\begin{aligned} \mathit{reflect}_\alpha &: ((\alpha \to o) \to o) \to \alpha \\ \mathit{reify}_\alpha &: (1 \to \alpha) \to (\alpha \to o) \to o \end{aligned}$$

are fixed $L^{\mathbf{cs}}$ term families. Then for any closed term $\vdash_{L^{\mathbf{k}}} E : a \,!\, \mathbf{p}$, $\mathit{Eval}^{\mathbf{cs}}(|E|) = \mathit{Eval}^{\mathbf{k}}(E) \stackrel{\text{def}}{=} \mathit{Eval}^{\mathbf{p}}(\overline{E})$, where the

$$\frac{\Gamma \vdash E_1 : \tau_1 \,!\, \mathbf{n} \qquad \Gamma, x{:}\,\tau_1 \vdash E_2 : \tau_2 \,!\, e}{\Gamma \vdash (\lambda x.\, E_2)\, E_1 = E_2\{E_1/x\} : \tau_2 \,!\, e} \qquad \frac{\Gamma \vdash E : \tau_1 \xrightarrow{e} \tau_2 \,!\, \mathbf{n}}{\Gamma \vdash \lambda x.\, E\, x = E : \tau_1 \xrightarrow{e} \tau_2 \,!\, \mathbf{n}} \qquad \frac{\Gamma \vdash E_1 : \tau_1 \,!\, \mathbf{n} \qquad \Gamma, x{:}\,\tau_1 \vdash E_2 : \tau_2 \,!\, e}{\Gamma \vdash \mathbf{let}\ x = E_1\ \mathbf{in}\ E_2 = E_2\{E_1/x\} : \tau_2 \,!\, e}$$

$$\frac{\Gamma \vdash E : \tau \,!\, e}{\Gamma \vdash \mathbf{let}\ x = E\ \mathbf{in}\ x = E : \tau \,!\, e} \qquad \frac{\Gamma \vdash E_1 : \tau_1 \,!\, e \qquad \Gamma, x_1{:}\,\tau_1 \vdash E_2 : \tau_2 \,!\, e \qquad \Gamma, x_2{:}\,\tau_2 \vdash E_3 : \tau_3 \,!\, e}{\Gamma \vdash \mathbf{let}\ x_2 = (\mathbf{let}\ x_1 = E_1\ \mathbf{in}\ E_2)\ \mathbf{in}\ E_3 = \mathbf{let}\ x_1 = E_1\ \mathbf{in}\ \mathbf{let}\ x_2 = E_2\ \mathbf{in}\ E_3 : \tau_3 \,!\, e}$$

$$\frac{\Gamma \vdash E_0 : \tau_1 \xrightarrow{e} \tau_2 \,!\, e \qquad \Gamma \vdash E_1 : \tau_1 \,!\, e}{\Gamma \vdash E_0\, E_1 = \mathbf{let}\ f = E_0\ \mathbf{in}\ f\, E_1 : \tau_2 \,!\, e} \qquad \frac{\Gamma \vdash E_0 : \tau_1 \xrightarrow{e} \tau_2 \,!\, \mathbf{n} \qquad \Gamma \vdash E_1 : \tau_1 \,!\, e}{\Gamma \vdash E_0\, E_1 = \mathbf{let}\ a = E_1\ \mathbf{in}\ E_0\, a : \tau_2 \,!\, e}$$

$$\frac{\Gamma \vdash E : \tau \,!\, \mathbf{p}}{\Gamma \vdash [\![E]\!] = \lambda k.\, k\, E : (\tau \xrightarrow{\mathbf{p}} o) \xrightarrow{\mathbf{p}} o \,!\, \mathbf{n}} \qquad \frac{\Gamma \vdash E_1 : \tau_1 \,!\, \mathbf{k} \qquad \Gamma, x{:}\,\tau_1 \vdash E_2 : \tau_2 \,!\, \mathbf{k}}{\Gamma \vdash [\![\mathbf{let}\ x = E_1\ \mathbf{in}\ E_2]\!] = \lambda k.\, [\![E_1]\!]\, (\lambda x.\, [\![E_2]\!]\, k) : (\tau_2 \xrightarrow{\mathbf{p}} o) \xrightarrow{\mathbf{p}} o \,!\, \mathbf{n}}$$

$$\frac{\Gamma \vdash E : (\tau \xrightarrow{\mathbf{p}} o) \xrightarrow{\mathbf{p}} o \,!\, \mathbf{n}}{\Gamma \vdash [\![\mu(E)]\!] = E : (\tau \xrightarrow{\mathbf{p}} o) \xrightarrow{\mathbf{p}} o \,!\, \mathbf{n}} \qquad \frac{\Gamma \vdash E : \tau \,!\, \mathbf{k}}{\Gamma \vdash \mu([\![E]\!]) = E : \tau \,!\, \mathbf{k}}$$

Figure 2: The equational theory of $L^{\mathbf{k}}$.

*Eval*s are partial functions denoting evaluation of complete programs to base-type results. (Note that even though E may not contain *top-level* control effects, it may still use control operators internally.)

In fact, this embedding result also holds for extensions of $L^{\mathbf{k}}$ with recursion and arithmetic [Fil99a]. Moreover, wrt. all $L^{\mathbf{cs}}$-contexts, $|callcc| \cong callcc$ and $|throw| \cong throw$. That is, the defined versions of *callcc* and *throw* in $L^{\mathbf{k}}$ are observationally equivalent to the native versions already in $L^{\mathbf{cs}}$.

The SML/NJ definitions of reflection and reification can be written as follows:

```
functor Control (type ans) =
struct
  local
    open SMLofNJ.Cont
    val mc : ans cont ref =
      callcc (fn q => (callcc (fn c => throw q (ref c));
                       raise Fail "missing reset"))
    fun abort x = throw (!mc) x
    fun reset t =
      let val m = !mc
          val r = callcc (fn c => (mc := c; abort (t ())))
      in mc := m; r end
  in
    type ans = ans
    fun reify t = fn k => reset (fn () => k (t ()))
    fun reflect h =
      callcc (fn c =>
        abort (h (fn v => reset (fn () => throw c v))))
  end
end;
```

(This is a slightly streamlined variant of the code from [Fil99a].)

2.2 The extensional CPS transform

Reification and reflection establish a "local" correspondence between direct-style and continuation-passing views of a term. We can extend these isomorphisms to arbitrary $L^{\mathbf{c}}$-types τ, by defining the following term families:

$$\begin{aligned}
\mathsf{C}_\tau &: \tau \xrightarrow{\mathbf{n}} \overline{\tau} \\
\mathsf{C}_a &= \lambda x.\, x \\
\mathsf{C}_{\tau_1 \to \tau_2} &= \lambda f.\, \lambda a.\, \lambda k.\, [\![\mathsf{C}_{\tau_2}\, (f\, (\mathsf{D}_{\tau_1}\, a))]\!]\, k \\
\mathsf{C}_{\neg\tau} &= \lambda c.\, \lambda a.\, [\![throw\, c\, (\mathsf{D}_\tau\, a)]\!]\, (\lambda r.\, r) \\
\mathsf{D}_\tau &: \overline{\tau} \xrightarrow{\mathbf{n}} \tau
\end{aligned}$$

$$\begin{aligned}
\mathsf{D}_a &= \lambda x.\, x \\
\mathsf{D}_{\tau_1 \to \tau_2} &= \lambda f.\, \lambda a.\, \mathsf{D}_{\tau_2}\, (\mu(\lambda k.\, f\, (\mathsf{C}_{\tau_1}\, a)\, k)) \\
\mathsf{D}_{\neg\tau} &= \lambda k.\, mkcont\, (\lambda a.\, \mu(\lambda k'.\, k\, (\mathsf{C}_\tau\, a))) \\
&\quad \text{where } mkcont(q) = callcc\, (\lambda c.\, q\, (callcc\, (throw\, c)))
\end{aligned}$$

(The cases for $\neg\tau$ appear somewhat messy due to the asymmetry of introduction and elimination constructs at that type; explicit first-class continuation abstractions would simplify things.).

To show that these functions correctly mirror the intensional CPS transform, we extend the equational theory of the computational lambda-calculus λ_c with a few axioms for monadic reflection and reification from [Fil99a]. The complete set of axioms is given in Figure 2. This calculus is easily checked to be sound in the sense that if $\vdash_{L^{\mathbf{k}}} E = E'$ then $\overline{\overline{E}} =_{\lambda_c} \overline{\overline{E'}}$, and hence, in particular, E and E' are observationally congruent wrt. all $L^{\mathbf{k}}$-program contexts. Let us now establish that the extensional CPS transform of an $L^{\mathbf{c}}$-term is $L^{\mathbf{k}}$-equivalent to the original transform.

First, we note that the fairly elaborate $L^{\mathbf{c}}$-based constructions of $\mathsf{C}_{\neg\tau}$ and $\mathsf{D}_{\neg\tau}$ are equivalent to much simpler $L^{\mathbf{k}}$ constructions:

Lemma 1 *The transformation functions at continuation types satisfy the following equations:*

$$\begin{aligned}
\vdash_{L^{\mathbf{k}}} \mathsf{C}_{\neg\tau} &= \lambda c.\, \lambda a.\, c(\mathsf{D}_\tau a) : \neg\tau \xrightarrow{\mathbf{n}} \overline{\tau} \xrightarrow{\mathbf{p}} o \,!\, \mathbf{n} \\
\vdash_{L^{\mathbf{k}}} \mathsf{D}_{\neg\tau} &= \lambda k.\, \lambda a.\, k(\mathsf{C}_\tau a) : (\overline{\tau} \xrightarrow{\mathbf{p}} o) \xrightarrow{\mathbf{n}} \neg\tau \,!\, \mathbf{n}
\end{aligned}$$

Proof: by simple equational reasoning. ■

Then, we show that the extensional CPS transform is indeed an isomorphism:

Lemma 2 C_τ *and* D_τ *are inverses for any* $L^{\mathbf{c}}$*-type* τ*:*

$$\begin{aligned}
a{:}\,\tau \vdash_{L^{\mathbf{k}}} \mathsf{D}_\tau(\mathsf{C}_\tau a) &= a : \tau \,!\, \mathbf{n} \\
a{:}\,\overline{\tau} \vdash_{L^{\mathbf{k}}} \mathsf{C}_\tau(\mathsf{D}_\tau a) &= a : \overline{\tau} \,!\, \mathbf{n}
\end{aligned}$$

Proof: by straightforward induction on τ, using Lemma 1 for $\neg\tau$-types. ■

And finally, we can state the main result:

Theorem 1 *Let* $\Gamma \vdash_{L^c} V : \tau$ *and* $\Gamma \vdash_{L^c} E : \tau$. *Then*

$$\Gamma \vdash_{L^k} \mathsf{C}_\tau V = \overline{V}\{\mathsf{C}_\Gamma\} : \overline{\tau}\,!\,\mathbf{n}$$
$$\Gamma \vdash_{L^k} [\mathsf{C}_\tau E] = \overline{\overline{E}}\{\mathsf{C}_\Gamma\} : (\overline{\tau} \xrightarrow{\mathbf{p}} o) \xrightarrow{\mathbf{p}} o\,!\,\mathbf{n}$$

where C_Γ *is the pointwise substitution* $\mathsf{C}_{\tau_i} x_i / x_i$, *for each* $x_i : \tau_i$ *in* Γ. *In particular, for closed* V *and* E, $\mathsf{C}_\tau V = \overline{V}$ *and* $[\mathsf{C}_\tau V] = \overline{\overline{E}}$

Proof: by structural induction on V and E, using Lemmas 1 and 2. ■

We can also express the extensional transforms in ML by defining the pair $(\mathsf{C}_\tau, \mathsf{D}_\tau)$ simultaneously for each type τ: (See [Yan98] for a general discussion of this technique.)

```
functor Iso (type ans) =
struct
  local
    structure C = Control (type ans = ans)
    open SMLofNJ.Cont C
    fun mkcont q = callcc (fn c => q (callcc (throw c)))
  in
    datatype ('a,'ac) cd = CD of ('a -> 'ac) * ('ac -> 'a)

    val base = CD (fn x => x, fn x => x)
    fun func (CD (c1,d1), CD (c2,d2)) =
      CD (fn f => fn a =>
            fn k => reify (fn () => c2 (f (d1 a))) k,
          fn f => fn a =>
            d2 (reflect (fn k => f (c1 a) k)))
    fun cont (CD (ca,da)) =
      CD (fn c => fn a =>
            reify (fn () => throw c (da a)) (fn r => r),
          fn k =>
            mkcont (fn a => reflect (fn _ => k (ca a))))
    fun ctrans (CD (ca,da)) = ca
    fun dtrans (CD (ca,da)) = da
  end
end;
```

The extensional transformations C_τ and D_τ thus allow us to link together direct-style and CPS functions in the same program – even if either body of code makes use of non-local control transfers. In fact, they also allow us to "decompile" some direct-style lambda-terms into their CPS source code, as we will see next.

3 Visualizing the transform through normalization by evaluation

In a functional language with a base type Λ of syntactic lambda-terms, we can define, for any simple type τ built out of Λ and function spaces, a term $\mathsf{N}_\tau : \tau \to \Lambda$, such that for any closed, pure lambda-term $E : \tau$, $Eval^{\mathbf{p}}(\mathsf{N}_\tau E) =_{\beta\eta} E$ [BS91]. (In fact, the result of $Eval^{\mathbf{p}}(\mathsf{N}_\tau E)$ is exactly the $\beta\eta$-long normal form of E, hence the label "normalization by evaluation".)

This type-indexed family of terms can be coded in ML using same idea as before:

```
structure NBE =
struct
  local
    val g = ref 0
    fun gensym s = (g := !g + 1; s ^ Int.toString (!g))
  in
    datatype exp =
      VAR of string | LAM of string * exp | APP of exp * exp
    datatype 'a nw =
      NW of ('a -> exp) * (exp -> 'a) * string

    fun base s = NW (fn x => x, fn x => x, s)
    fun func (NW (n1, w1, s1), NW (n2, w2, s2)) =
      NW (fn f => let val v = gensym s1
                  in LAM (v, n2 (f (w1 (VAR v)))) end,
          fn e => fn a => w2 (APP (e, n1 a)),
          "f")

    fun name s (NW (na,wa,_)) = NW (na,wa,s)
    fun nbe (NW (na,_,_)) a = (g := 0; na a)
  end
end;
```

(We use a gensym-based variable-name generator for conciseness only; a purely functional definition of `nbe` is also possible [BS91, Fil99b].) Note that, in ML, we can also apply N_τ to polymorphic lambda-terms: their type variables will be automatically instantiated to Λ by the typing rule for function application. The extra string component of `nw` is used to supply "preferred" names for gensym'd variables of that type, as in [Dan96]. This is for improving readability only – since names are generated uniquely anyway, we could just use `"x"` in all cases. For example, we get:

```
infixr 5 -->
val op --> = NBE.func
val bt = NBE.base;

val test0 =
  NBE.nbe ((bt "a" --> bt "b") --> bt "a" --> bt "a")
          (fn f => fn x => (fn y => x) (f x));
(*
val test0 = LAM ("f1",LAM ("a2",VAR "a2")) : NBE.exp
*)
```

Note that the source application `f x` does not appear in the output, since its result is discarded; the decompilation works only up to $\beta\eta$-conversion. But we can now also combine NBE and the extensional CPS transform to decompile the CPS counterpart of a term:

```
structure NBECPS =
struct
  local
    structure N = NBE
    structure I = Iso (type ans = N.exp)
  in
    datatype ('a,'ca) cn = CN of ('a,'ca) I.cd * 'ca N.nw

    fun base s = CN (I.base, N.base s)
    fun func (CN (ca, ra), CN (cb,rb)) =
      CN (I.func (ca,cb),
          N.func (ra,
                  N.func (N.name "k" (N.func (rb,
                                              N.base "o")),
                          N.base "o")))
    fun cont (CN (ca,ra)) =
      CN (I.cont ca,
          N.name "c" (N.func (ra, N.base "o")))

    fun dcps (CN (cd, nw)) x = N.nbe nw (I.ctrans cd x)
  end
end;
```

For example:

```
val op --> = NBECPS.func
val bt = NBECPS.base
val ct = NBECPS.cont;

val test1 =
  NBECPS.dcps ((bt "a" --> bt "b") --> bt "a" --> bt "a")
              (fn f => fn x => (fn y => x) (f x))
val test2 =
```

```
  NBECPS.dcps ((ct (bt "a") -->  bt "a") --> bt "a")
              SMLofNJ.Cont.callcc
val test3 =
  NBECPS.dcps (ct (bt "a") --> bt "a" --> bt "b")
              SMLofNJ.Cont.throw;
(*
val test1 =
  LAM
    ("f1",
     LAM
       ("k2",
        APP
          (VAR "k2",
           LAM
             ("a3",
              LAM
                ("k4",
                 APP
                   (APP (VAR "f1",VAR "a3"),
                    LAM ("b5",APP (VAR "k4",VAR "a3")))))))))
  : NBE.exp
val test2 =
  LAM
    ("f1",
     LAM
       ("k2",
        APP
          (APP (VAR "f1",LAM ("a3",APP (VAR "k2",VAR "a3"))),
           LAM ("a4",APP (VAR "k2",VAR "a4"))))) : NBE.exp
val test3 =
  LAM
    ("c1",
     LAM
       ("k2",
        APP
          (VAR "k2",
           LAM ("a3",LAM ("k4",APP (VAR "c1",VAR "a3"))))))
  : NBE.exp
*)
```

(Note the similarity between the decompiled SML/NJ continuation primitives and the definitions of their CPS transforms in Section 2.)

A related way of regenerating CPS code from meanings is to adapt the output of a CBV version of NBE [Dan96, Section 6]. However, the combined correctness argument for effect-free NBE and the extensional CPS transformation is significantly simpler than for CBV NBE directly.

4 A completeness result

In a minimally realistic functional programming language (including, at least, recursion and integer arithmetic), observational equivalence of terms is not decidable, or even recursively enumerable. One can show, however, that for PCF-like languages [Plo77], *pure* (i.e., constant-free) lambda-terms are observationally equivalent in any PCF context iff they are $\beta\eta$-convertible. A simple proof of this result based on NBE is given in [BS91]. (Note that both the result and its proof rely heavily on the terms being simply typed.)

For CBV languages, $\beta\eta$-conversion is of course not sound for observational equivalence. Convertibility in the computational lambda-calculus is sound, but may not be complete, even for equivalence between pure terms. This is again because there are λ_c-unequal pure CBV-PCF terms (such as the ones from Section 1) that cannot be distinguished by any CBV-PCF context. But with sufficiently powerful effects, we can in fact distinguish all such terms:

Theorem 2 *In $L^{\mathbf{cs}}$ (CBV PCF with Scheme-style effects, i.e., call/cc and mutable state), two pure lambda-terms are observationally indistinguishable iff they are provably equal in the computational lambda-calculus.*

Proof. The "if" direction (soundness) is standard. For "only if", let $V, V' : \tau$ be two $L^{\mathbf{cs}}$-observationally indistinguishable, closed $L^{\mathbf{c}}$-values. (We will reduce the general case to this one, below.) That is, for all $L^{\mathbf{cs}}$-program contexts $C[-]$ with τ-typed holes (i.e., such that $C[V]$ and $C[V']$ are closed terms of base type), $Eval^{\mathbf{cs}}(C[V]) = Eval^{\mathbf{cs}}(C[V'])$. (As usual, it actually suffices to only require coincidence of termination.)

By Theorem 1, we know that $\vdash_{L^{\mathbf{k}}} \mathsf{C}_\tau V = \overline{V} : \overline{\tau}$, and hence (by soundness of $L^{\mathbf{k}}$'s equational theory) $\mathsf{C}_\tau V \cong \overline{V}$ wrt. $L^{\mathbf{k}}$-contexts; likewise, $\mathsf{C}_\tau V' \cong \overline{V'}$. Now consider the $L^{\mathbf{cs}}$-program context $C[-] \stackrel{\text{def}}{=} |\mathsf{N}_{\overline{\tau}}|\,(|\mathsf{C}_\tau|\,-)$: for any closed $L^{\mathbf{cs}}$-term $E : \tau$, $C[E]$ is a closed $L^{\mathbf{cs}}$-term of base type Λ. (We can take Λ to be the type of integers, with a suitable Gödel-coding; we still identify a syntactic lambda-term with its representation as a Λ-value.) Since $\overline{V}$ is a pure lambda-term, we now get

$$\begin{aligned}\overline{V} &=_{\beta\eta} Eval^{\mathbf{p}}(\mathsf{N}_{\overline{\tau}}\overline{V}) = Eval^{\mathbf{k}}(\mathsf{N}_{\overline{\tau}}\overline{V})\\ &= Eval^{\mathbf{k}}(\mathsf{N}_{\overline{\tau}}(\mathsf{C}_\tau V)) = Eval^{\mathbf{cs}}(|\mathsf{N}_{\overline{\tau}}(\mathsf{C}_\tau V)|)\\ &= Eval^{\mathbf{cs}}(C[|V|]) = Eval^{\mathbf{cs}}(C[V]) = Eval^{\mathbf{cs}}(C[V'])\\ &= \cdots =_{\beta\eta} \overline{V'}\end{aligned}$$

And thus, from Sabry and Felleisen's equational correspondence between $\beta\eta$-conversion on CPS terms and λ_c-conversion on their direct-style counterparts [SF93], we get $V =_{\lambda_c} V'$.

Finally, let $E \cong E'$ be arbitrary, contextually equivalent $L^{\mathbf{c}}$-terms – not necessarily values, and not necessarily closed. Let $\vec{x} = (x_1, \ldots, x_n)$ be a finite list including at least all the variables occurring free in E or E'; wlog. we can assume that $n \geq 1$. We write $\lambda\vec{x}.E$ for $\lambda x_1.\cdots.\lambda x_n.E$, and $E\vec{x}$ for $E\,x_1 \cdots x_n$.

Since $\cong$ is a congruence, we must have $\lambda\vec{x}.E \cong \lambda\vec{x}.E'$. By the result for closed values above, we obtain $\lambda\vec{x}.E =_{\lambda_c} \lambda\vec{x}.E'$, and then, since $=_{\lambda_c}$ is also a congruence and includes β_v-conversion,

$$E =_{\lambda_c} (\lambda\vec{x}.E)\,\vec{x} =_{\lambda_c} (\lambda\vec{x}.E')\,\vec{x} =_{\lambda_c} E'$$

which is the desired result. ■

5 Conclusions

Although CPS transforms are usually viewed as syntax-to-syntax transformations, the meanings of direct-style terms in Scheme-like languages (i.e., with first-class continuations and mutable state) actually contain enough information to define the transform as semantics-to-semantics. Practically, this means that we can write fragments of code in either direct style or in CPS as most convenient, and use the extensional CPS- and DS-transforms to glue these fragments together without recompilation.

Moreover, combining the extensional CPS-transform result with the normalization-by-evaluation principle, we obtain a new observational-completeness result for λ_c-convertibility of pure, simply typed lambda-terms in Scheme-like contexts.

The account of the extensional CPS transform reported in this note should be considered preliminary: although not covered here, the transformation result actually generalizes directly to the full language $L^{\mathbf{k}}$, and further to arbitrary monadic effects, as long as their reflection and reification

operations can be expressed within the language. Also, a formal treatment of constants and constant families (such as fixed-point combinators), especially with recursive types, requires further investigation. It is expected, however, that the results presented here will in fact scale up gracefully to such extensions.

Acknowledgments

The author wishes to thank Olivier Danvy and Zhe Yang for many fruitful discussions about the subject matter of this note, as well as the CW'01 reviewers for their constructive comments.

References

[BS91] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991.

[Dan96] Olivier Danvy. Pragmatics of type-directed partial evaluation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, pages 73–94, Dagstuhl, Germany, February 1996. Springer-Verlag. Extended version available as the technical report BRICS RS-96-15.

[DF92] Olivier Danvy and Andrzej Filinski. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[Fil99a] Andrzej Filinski. Representing layered monads. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, January 1999.

[Fil99b] Andrzej Filinski. A semantic account of type-directed partial evaluation. In Gopalan Nadathur, editor, *International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 378–395, Paris, France, September 1999.

[Kuc98] Jakov Kucan. Retraction approach to CPS transform. *Higher-Order and Symbolic Computation*, 11(2):145–175, 1998.

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, June 1989. IEEE.

[MR88] Albert R. Meyer and Jon G. Riecke. Continuations may be unreasonable (preliminary report). In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 63–71, 1988.

[MW85] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Logics of Programs – Proceedings*, number 193 in Lecture Notes in Computer Science, pages 219–224, Brooklyn, June 1985.

[Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, December 1977.

[SF90] Dorai Sitaram and Matthias Felleisen. Reasoning with continuations II: Full abstraction for models of control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 161–175, Nice, France, June 1990.

[SF93] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):289–360, November 1993. (An earlier version appeared in *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*).

[Yan98] Zhe Yang. Encoding types in ML-like languages. In *ACM SIGPLAN International Conference on Functional Programming*, pages 289–300, Baltimore, Maryland, September 1998.

Linearly Used Continuations

Josh Berdine Peter W. O'Hearn
Queen Mary, University of London
{berdine,ohearn}@dcs.qmw.ac.uk

Uday S. Reddy Hayo Thielecke
The University of Birmingham
{U.Reddy,H.Thielecke}@cs.bham.ac.uk

1. INTRODUCTION

Continuations are the raw material of control. They can be used to explain a wide variety of control behaviours, including calling/returning (procedures), raising/handling (exceptions), labelled jumping (`goto` statements), process switching (coroutines), and backtracking. In the most powerful form, represented by `callcc` and its cousins, the programmer can manipulate continuations as first-class values. It can be argued, however, that unrestricted use of continuations, especially when combined with state, can give rise to intractable higher-order spaghetti code. Hence, few languages give the user direct, reified, access to continuations; rather, they are "behind the scenes", implementing other control behaviours, and their use is highly stylised.

But just what is this stylised usage? Remarkably, as we will argue, in many forms of control, continuations are used *linearly* [6]. This is true for a wide range of effects, including procedure call and return, exceptions, `goto` statements, and coroutines.

Formally, for a number of control behaviours, we present continuation-passing-style (CPS) transformations into a language with both intuitionistic and linear function types. We also remark on combinations of features which break linearity. Interestingly, the presence of named labels, by itself, does not. And neither does *backward* jumping, which is different in character from *backtracking*.

This is essentially an attempt to formalise ideas about continuation usage, some of which have been hinted at in the literature. Indeed, part of what we say is known or suspected amongst continuation insiders; however, we have not found any of the linear typings we give stated anywhere.

The basic idea can be seen in the type used to interpret untyped call-by-value (CBV) λ-calculus. Just as Scott gave a typed explanation of untyped λ-calculus using a domain equation

$$D \cong D \to D$$

we can understand linear use of continuations in terms of the domain equation

$$D \cong (D \to \mathbf{R}) \multimap (D \to \mathbf{R}),$$

where $\mathbf{R}$ is a type of results. If we were to change the principal $\multimap$ to an $\to$, then this type could accept `callcc`; but, as we later discuss, `callcc` duplicates the current continuation, and is ruled out by this typing. Thus, even though one might claim that the linear usage of continuations has nothing to do with typing in the *source* language, we can nonetheless use types in the *target* language to analyse control behaviour.

An essential point is that it is *continuation transformers*, rather than continuations, which are linear. This is the reason we say that continuations are *used* linearly. All of the interpretations we present are variations on this idea.

Our chief concern in this paper is to describe the main conceptual aspects of linearly used continuations in a way that keeps the technical discussion as simple as possible. So we concentrate on soundness only. A comprehensive analysis of completeness properties of our transforms, or variants, represents a challenge for future work, and in stating the transforms for a variety of features we hope to make clear what some of the challenges are. Several of these problems are discussed at the end of the paper.

2. THE TARGET LANGUAGE

We need a formulation of linear type theory built from the connectives $\multimap$, $\to$ and &, and use one based on DILL [3], which is a presentation of linear typing that allows a pleasantly direct description of $\to$ (which does not rely on decomposition through !).

$$P ::= \mathbf{R} \mid P \multimap P \mid A \to P \mid P \& P \mid X \mid \mu X.\, P$$
$$A ::= \rho \mid P$$

Here, X ranges over type variables, μ builds recursive types, and ρ ranges over primitive types. Types P are *pointed* while types A are not necessarily. Pointed types are those for which recursion is allowed. In particular, primitive types used to treat atomic data (such as a type for integers) should not be pointed in a CPS language. It would be possible to add type constructors for sums, !, and so on, but we will not need them.

The distinction between pointed and non-pointed types is especially vivid in the standard predomain model of the system, where a pointed type denotes a pointed cpo (one with bottom) and a type denotes a possibly bottomless cpo. The type $\mathbf{R}$ of results denotes the two-point lattice, & is cartesian

product, $\multimap$ is strict function space, and $\to$ is lifting. μ is interpreted using the inverse limit construction. This is not an especially accurate model of the language, because the interpretation of $\multimap$ validates Contraction and because the abstractness of the result type is not accounted for. But the model is certainly adequate for any reasonable operational semantics, and so serves as a useful reference point.

There is also a predomain variant of the original coherence space model of linear logic. In this model a type denotes a family of coherence spaces and a pointed type denotes a singleton such family [2]. Then $\multimap$ is the usual linear function type, and $\{A_i\}_{i\in I} \to P$ is a product $\prod_{i\in I} A_i \Rightarrow P$ where $\Rightarrow$ is stable function space and $\prod_{i\in I}$ is the direct product of coherence spaces; this gives us a singleton family (that is, a pointed type).

The system uses typing judgements of the form

$$\Gamma; \Delta \vdash M : A$$

where the context consists of an intuitionistic zone Γ and a linear zone Δ. Intuitionistic and linear zones are sets of associations $x : A$ pairing variables with types. Since we use sets, the Exchange rules are built in.

$$\frac{}{\Gamma, x : A; _ \vdash x : A} \qquad \frac{}{\Gamma; x : P \vdash x : P}$$

$$\frac{\Gamma; \Delta, x : P \vdash M : Q}{\Gamma; \Delta \vdash \delta x.\, M : P \multimap Q} \qquad \frac{\Gamma; \Delta_1 \vdash M : P \multimap Q \quad \Gamma; \Delta_2 \vdash N : P}{\Gamma; \Delta_1, \Delta_2 \vdash M_{\sqcup} N : Q}$$

$$\frac{\Gamma, x : A; \Delta \vdash M : P}{\Gamma; \Delta \vdash \lambda x.\, M : A \to P} \qquad \frac{\Gamma; \Delta \vdash M : A \to P \quad \Gamma; _ \vdash N : A}{\Gamma; \Delta \vdash M\, N : P}$$

$$\frac{\Gamma; \Delta \vdash M : P \qquad \Gamma; \Delta \vdash N : Q}{\Gamma; \Delta \vdash \langle M, N \rangle : P \& Q} \qquad \frac{\Gamma; \Delta \vdash M : P_1 \& P_2}{\Gamma; \Delta \vdash \pi_{i\sqcup} M : P_i}$$

We will frequently consider situations where some number of continuations are held in a single &-tuple in the linear zone. We introduce the following syntactic sugar for them, using the evident n-ary form of &-product, rather than the binary form.

- $\Gamma; \Delta, \langle x_1, \ldots, x_n \rangle : P_1 \& \cdots \& P_n \vdash M : P$ stands for $\Gamma; \Delta, y : P_1 \& \cdots \& P_n \vdash M[\pi_{1\sqcup} y / x_1, \ldots, \pi_{n\sqcup} y / x_n] : P$.
- $\delta\langle x_1, \ldots, x_n \rangle.\, M$ stands for $\delta y.\, M[\pi_{1\sqcup} y / x_1, \ldots, \pi_{n\sqcup} y / x_n]$.

For simplicity in presenting the transforms, we handle recursive types using the "equality approach", where $\mu X.\, P$ and its unfolding $P[\mu X.\, P / X]$ are equal, yielding the typing rule

$$\frac{\Gamma; \Delta \vdash M : P}{\Gamma; \Delta \vdash M : Q}\, P = Q$$

We omit the details and instead refer to [1] for a comprehensive treatment.

3. CALL/RETURN

A prominent historic explanation of procedure call/return in terms of continuations is the Fischer (continuation-first) CPS transform. Here we transform untyped CBV (operator-first) λ-calculus into the linear target language.

$$\begin{aligned} \overline{x} &= \delta k.\, k\, x \\ \overline{\lambda x.\, M} &= \delta k.\, k\, (\delta k'.\lambda x.\, \overline{M}_{\sqcup} k') \\ \overline{M\, N} &= \delta k.\, \overline{M}_{\sqcup} (\lambda m.\, \overline{N}_{\sqcup} (\lambda n.\, (m_{\sqcup} k)\, n)) \end{aligned}$$

Here we see that, as mentioned in the introduction, source language procedures are interpreted by continuation transformers: terms which accept a continuation and yield another continuation based upon the argument continuation, and thus have type

$$\mu X.\, (X \to \mathbf{R}) \multimap (X \to \mathbf{R}),$$

abbreviated D. (Henceforth, we do not explicitly define the types and type abbreviations corresponding to domains.) So a continuation transformer is effectively the difference, or *delta*, between two continuations, and δ is used to form such abstractions.

PROPOSITION 1. *If $x_1, \ldots, x_n$ contains the free variables of M, then*

$$x_1 : D, \ldots, x_n : D; _ \vdash \overline{M} : (D \to \mathbf{R}) \multimap \mathbf{R}.$$

Here, notice that source variables always get sent to the intuitionistic zone, where they can be duplicated or discarded freely. Continuation arguments, on the other hand, always show up in the linear zone in the course of typing a target term.

A common area of confusion is the relationship between linearity and recursion. Since recursion can be defined via self application in the source language, will we not have to use a continuation many times, or not at all, in the target? The short answer is no: continuations do not need to be used more than once since we use recursive continuation *transformers* to construct non-recursive continuations, and these continuation transformers can be used many times.

We explain this by concentrating on the transform of the most basic self-application:

$$\overline{f\, f} = \delta k.\, f_{\sqcup} k\, f.$$

This makes clear that, in the target language, recursion is effected by a sort of self-application in which a continuation transformer f is passed to a continuation $f_{\sqcup} k$ which is obtained from f itself. If we were to uncurry the type of continuation transformers, a call to f would directly pass itself as one of the arguments. The important point here is that self application in the source only breaks linearity of continuations in the target, not continuation transformers; that is, it is entirely possible for the continuation $f_{\sqcup} k$ to be non-linear, without violating linearity of f. The typing derivation of self-application in the target language (see Figure 1) shows how the recursive type must be "unwound" once to type the operand occurrence of f.

Finally, it is essential to note that linearity does not arise because of any linear λs in the source, but because continuations are not reified and hence cannot be wrapped into closures. This is similar to O'Hearn and Reynolds's work [9], where linearity and polymorphism arise in the target of a translation from Algol; this prevents the state from being treated, semantically, as if it were first-class.

4. EXCEPTIONS

$$\dfrac{\dfrac{\dfrac{\overline{f : D; _ \vdash f : D}}{f : D; _ \vdash f : (D \to \mathbf{R}) \multimap (D \to \mathbf{R})}\, D = (D \to \mathbf{R}) \multimap (D \to \mathbf{R}) \qquad \overline{f : D; k : D \to \mathbf{R} \vdash k : D \to \mathbf{R}}}{f : D; k : D \to \mathbf{R} \vdash f_{\sqcup} k : D \to \mathbf{R}} \qquad \overline{f : D; _ \vdash f : D}}{f : D; k : D \to \mathbf{R} \vdash f_{\sqcup} k\, f : \mathbf{R}}}{f : D; _ \vdash \delta k.\, f_{\sqcup} k\, f : (D \to \mathbf{R}) \multimap \mathbf{R}}$$

Figure 1: Typing derivation of self-application in the target language

Exceptions are a powerful, and useful, jumping construct. But their typing properties are rather complex, and vary somewhat from language to language. To study the jumping aspect of exceptions we focus on an untyped source language with `raise` and `handle` primitives.

We proceed as before, but now using a domain equation

$$D \cong (D \to \mathbf{R}) \& (D \to \mathbf{R}) \multimap (D \to \mathbf{R}).$$

A typed version of this can even be derived from a direct semantics, following Moggi. That is, we start with

$$(A \to B)^* = A^* \to B^* + E,$$

followed by a standard CPS semantics which gives us

$$\overline{A^* \to B^* + E} = ((\overline{B^*} + E) \to \mathbf{R}) \multimap (\overline{A^*} \to \mathbf{R}),$$

and finally a manipulation using the isomorphism

$$(\overline{B^*} + E) \to \mathbf{R} \cong (\overline{B^*} \to \mathbf{R}) \& (E \to \mathbf{R}).$$

In this double-barrelled CPS two continuations are manipulated: current and handler [17].

$$\begin{aligned}
\overline{x} &= \delta\langle k, h\rangle.\, k\, x \\
\overline{\lambda x.\, M} &= \delta\langle k, h\rangle.\, k\, (\delta\langle k', h'\rangle.\, \lambda x.\, \overline{M}_{\sqcup}\langle k', h'\rangle) \\
\overline{M\, N} &= \delta\langle k, h\rangle.\, \overline{M}_{\sqcup}\langle(\lambda m.\, \overline{N}_{\sqcup}\langle m_{\sqcup}\langle k, h\rangle, h\rangle), h\rangle \\
\overline{\mathtt{raise}\, M} &= \delta\langle k, h\rangle.\, \overline{M}_{\sqcup}\langle h, h\rangle \\
\overline{\mathtt{handle}\, M\, \lambda e.\, H} &= \delta\langle k, h\rangle.\, \overline{M}_{\sqcup}\langle k, (\lambda e.\, \overline{H}_{\sqcup}\langle k, h\rangle)\rangle
\end{aligned}$$

PROPOSITION 2. *If $x_1, \ldots, x_n$ contains the free variables of M, then*

$$x_1 : D, \ldots, x_n : D; _ \vdash \overline{M} : (D \to \mathbf{R}) \& (D \to \mathbf{R}) \multimap \mathbf{R}.$$

Note that the first three cases do not manipulate the handler continuation, just pass it along. The transform of $\mathtt{raise}\, M$ indicates that M is evaluated and the resulting value is thrown to the handler continuation, and if the evaluation of M results in an exception being `raise`d, the current handler continuation is used. Correspondingly, the transform of $\mathtt{handle}\, M\, \lambda e.\, H$ evaluates M with the same return continuation but installs a new handler continuation which given e, evaluates H with ($\mathtt{handle}\, M\, \lambda e.\, H$)'s continuations.

At first sight, this treatment of handling looks like duplication of the current continuation. But it is not: the use of & to the left of $\multimap$ indicates that a program uses either the current continuation or the handler continuation, but not both. Thus, linear typing succinctly summarises an important aspect of the jumping behaviour of exceptions, which draws a sharp distinction with `callcc` and its non-linear usage of continuations.

5. DUPLICATING CONTINUATIONS

A crucial reason Propositions 1 and 2 hold is that in the application of an intuitionistic function, the argument cannot have any free linear variables. This has the effect of precluding upward continuations, where a continuation is wrapped in a closure and returned, or passed as the argument to another function. Concretely, this is demonstrated by the term (which does not typecheck)

$$\delta k.\, k\, (\delta h.\lambda x.\, k\, (\delta l.\lambda y.\, l\, x))$$

in which k is an upward continuation, that is, wrapped in a closure which is thrown to another continuation; in this case, k itself. This term, which corresponds to

$$\mathtt{callcc}\, \lambda k.\lambda x.\, \mathtt{throw}\, k\, \lambda y.\, x$$

in the source language, exhibits the backtracking behaviour leading to the higher-order spaghetti code associated with `callcc`. We use the typed variant, `callcc`, rather than the untyped, `call/cc`, since the latter would require modification of the interpretation of procedures. The CPS transform of `callcc` shows how continuations are duplicated, breaking linearity.

$$\overline{\mathtt{callcc}} = \delta k.\, k\, (\delta h.\lambda f.\, (f_{\sqcup} h)\, h)$$

This fails to typecheck since h, which is δ-bound and hence linear, is passed to f as both its return continuation and argument.

Similar backtracking behaviour can be seen in SNOBOL and Prolog, and their continuation semantics do not obey a discipline of linearly used continuations [14, 7].

6. REIFIED CONTINUATIONS, AND UPWARD VERSUS DOWNWARD

It might be expected that the reason continuations are used linearly in the call/return and exceptions cases is that they are not *reified*, which is to say directly named by program variables, as `callcc` achieves. After all, source language variables may appear any number of times in a term. This reasoning is only partially valid. To explain this, we consider a language where continuations are reified, but still used linearly.

We consider a language of arithmetic expressions, with a means of labelling a subexpression.

$$E ::= x \mid n \mid E + E \mid l : E \mid \mathtt{goto}\, l\, E$$

A `goto` statement sends a value to the position where the indicated label resides. As an example,

$$2 + (l : (3 + (\mathtt{goto}\, l\, 7)))$$

evaluates to 9, as evaluation jumps past $3 + []$, effectively sending 7 to the hole in $2 + []$. Labelling an expression and

sending to it with `goto` is effectively a first-order version of naming a continuation with `callcc` and `throw`ing to it.

Following this analogy, labelling an expression associates the current continuation of the expression with the label name, and $\texttt{goto}\, l$ effects a throw to the continuation associated with l. The crucial point is that although continuations are reified, they cannot escape the context in which they are originally defined. That is, in

$$l : E$$

l cannot escape out of E. On the other hand, in the analogous term in the language with first-class continuations

$$\texttt{callcc}\, \lambda k.\, M$$

k can indeed escape out of M, as the example in the previous section demonstrated. This means that continuations are not *upward* in the language of forward jumps, only *downward*.

Unlike the previous cases, this language is not higher-order; so we interpret expressions with the (non-recursive) types

$$(\mathbf{N} \to \mathbf{R}) \& \cdots \& (\mathbf{N} \to \mathbf{R}) \multimap \mathbf{R},$$

where $\mathbf{N}$ is a primitive type of natural numbers. The first continuation in the &-tuple is the current continuation, and the others represent the labels free in the source expression.

$$\begin{aligned}
\overline{x}_{\vec{l}} &= \delta\langle k, \vec{l}\rangle.\, k\, x \\
\overline{n}_{\vec{l}} &= \delta\langle k, \vec{l}\rangle.\, k\, n \\
\overline{E + F}_{\vec{l}} &= \delta\langle k, \vec{l}\rangle.\, \overline{E}_{\vec{l}\sqcup}\langle(\lambda e.\, \overline{F}_{\vec{l}\sqcup}\langle(\lambda f.\, k\,(e + f)), \vec{l}\rangle), \vec{l}\rangle \\
\overline{l_{n+1} : E}_{\vec{l}} &= \delta\langle k, \vec{l}\rangle.\, \overline{E}_{\vec{l}, l_{n+1}\sqcup}\langle k, \vec{l}, k\rangle \\
\overline{\texttt{goto}\, l_i\, E}_{\vec{l}} &= \delta\langle k, \vec{l}\rangle.\, \overline{E}_{\vec{l}\sqcup}\langle l_i, \vec{l}\rangle
\end{aligned}$$

$\vec{l}$ is a list of labels $l_1, \ldots, l_n$. For precision, the transform of E is parameterised by $\vec{l}$ containing the labels free in E.

In the $l : E$ clause, since the two occurrences of k are within a &-tuple, linearity is not violated.

PROPOSITION 3. *If $x_1, \ldots, x_m$ contains the free variables of E, and $\vec{l}$ $(= l_1, \ldots, l_n)$ contains the free labels of E, then*

$$x_1 : \mathbf{N}, \ldots, x_m : \mathbf{N}; _ \vdash \overline{E}_{\vec{l}} : \underbrace{(\mathbf{N} \to \mathbf{R}) \& \cdots \& (\mathbf{N} \to \mathbf{R})}_{n} \multimap \mathbf{R}.$$

The moral of this story is that we cannot attribute the failure of linearity in the treatment of `callcc` *only* to the ability to name continuations (in the presence of Contraction and Weakening of source language variables). However, these features together with *upward* continuations, which arise from higher-order procedures, suffice to break linearity.

7. BACKWARD JUMPS

Next, one might think that the linear use of continuations in the previous section is due to the absence of *backward* jumps. That is, if one has backward jumps, cannot one jump to the same continuation multiple times, thus violating linearity?

The answer is no, backward jumping does not require duplication of continuations. In fact, this point has already been made in the treatment of untyped λ-calculus, which involves self application, but it is helpful to look at it in a setting where jumping is effected by explicit manipulation of reified continuations rather than by the call/return mechanism's implicit manipulation of non-reified continuations.

In order to bring the central issues out with a minimum of distraction, we begin with an informal discussion of how to define a single recursive label, before giving a precise treatment of a full language.

Suppose we have a simple language of commands, with command continuations

$$K = \mathbf{S} \to \mathbf{R}$$

where $\mathbf{S}$ is the type of stores. (When performing backward jumps it is necessary to communicate information, if one is not to always loop indefinitely. So it is reasonable here to consider state; alternatively, we could consider labels that accept a number of arguments.) We suppose that we have a command C of type

$$K \& K \multimap K.$$

The first argument is the current continuation, which represents the effect of executing the rest of the program, and the second is the denotation of the (single) label l. We will show how to interpret a construct $l : C$ where jumps to l within C go back to the beginning of $l : C$. This construct effectively binds l, and will result in a continuation transformer of type

$$K \multimap K$$

which accepts a toplevel (current) continuation. We use a standard fixed-point combinator

$$\Upsilon : (P \to P) \to P.$$

At first sight the desired transform appears to be incompatible with linearity. Indeed, were we not restricting the use of continuations, we could interpret $l : C$ with the type

$$K \to K$$

and define the transform as

$$\overline{l : C} = \lambda k.\, \Upsilon\, \lambda h.\, \overline{C}_{\sqcup}\langle k, h\rangle$$

This approach, in which a recursive continuation is defined directly using $\Upsilon : (K \to K) \to K$, is the one typically taken in the continuation semantics of `goto`.

However, by moving up a level in the types we can tie the label l up in a recursion.

$$\overline{l : C} = \Upsilon\, \lambda t.\, \delta k.\, \overline{C}_{\sqcup}\langle k, t_{\sqcup} k\rangle$$

Note that the term we take a fixed-point of has type $(K \multimap K) \to (K \multimap K)$, so the definition of a program makes use a recursive continuation transformer, but continuations are not themselves recursive. The upshot is that different backward jumps to l correspond to *different* continuations, which may be viewed as being generated in fixed-point unwinding. (This is very similar to the handling of recursion in untyped λ-calculus where continuation transformers are self-applied to unwind to a fixed-point, but continuations are not recursive. The only difference here is that we explicitly take a fixed-point, rather than rely on self-application.)

It is curious how linearity forces fixed-points to be taken at higher types here.

With this as background we move on to a full language, the "small continuation language" of Strachey and

Wadsworth [13]. We emphasise that our treatment of recursive labels is not identical to that of Strachey and Wadsworth, as we must go up a level in the types to accommodate linearity (it is again curious, however, that the entirety of [13] is compatible with linear continuation usage).

The source language consists of expressions, E, and commands, C.

$$C ::= p \mid \mathtt{dummy} \mid C_0; C_1 \mid E \to C_0, C_1 \mid \mathtt{goto}\, E$$
$$\mid \S\, C_0; l_1 : C_1; \ldots; l_n : C_n \,\S \mid \mathtt{resultis}\, E$$
$$E ::= x \mid l \mid \mathtt{true} \mid \mathtt{false} \mid E_0 \to E_1, E_2 \mid \mathtt{valof}\, C$$

Here p is a primitive statement, x is a variable, l is a label. Note that we do not include explicit loops since they are redundant, though they could be easily added.

We extend the target language with a primitive type of booleans, $\mathbf{B}$.

$$\frac{}{\Gamma; _ \vdash \mathtt{tt} : \mathbf{B}} \qquad \frac{}{\Gamma; _ \vdash \mathtt{ff} : \mathbf{B}}$$

$$\frac{\Gamma; \Delta \vdash M : \mathbf{B} \qquad \Gamma; \Delta' \vdash N : A \qquad \Gamma; \Delta' \vdash O : A}{\Gamma; \Delta, \Delta' \vdash \mathtt{if}\, M\, \mathtt{then}\, N\, \mathtt{else}\, O : A}$$

Primitive commands are mapped to their interpretations in the target language by

$$[\![p]\!] : K \multimap K.$$

Commands are interpreted with the types

$$K \& (\mathbf{B} \to K) \& K \& K \& \cdots \& K \multimap K$$

The first argument in the &-tuple is the current command continuation. Next, the current return continuation is the expression continuation to which a `resultis` command will deliver a value. After that, the failure continuation is a constant command continuation invoked when a `valof` command "falls off the end" without performing a `resultis` command. Finally, the remaining command continuations are the denotations of the labels in scope.

Similarly, expressions are interpreted with the types

$$(\mathbf{B} \to K) \& K \& K \& \cdots \& K \multimap K.$$

Here the first argument in the &-tuple, the current expression continuation, is the expression continuation to which the value of the expression will be delivered. The remaining arguments: the failure continuation and command continuations, are handled as above.

The transforms, given in Figure 2, make use of a divergent term

$$\mathtt{diverge} = \mathtt{Y}\, \lambda x.\, x : P$$

and are parameterised by a sequence of labels, $l_1, \ldots, l_n$, which contains the labels free in the term being transformed. In defining the transforms, we use the notation $,_{i=1}^{n} M$ as a shorthand for $M[1/i], M[2/i], \ldots, M[n/i]$.

Strachey and Wadsworth's semantics of $\mathtt{goto}\, E$ uses a current continuation which "projects" its argument, performing a sort of dynamic type-checking. But they do not specify what happens if the check fails. Here we specify that execution diverges, but other choices are possible: the failure continuation which is being carried around could be used, for instance.

The interpretation of a `valof` expression

$$\overline{\mathtt{valof}\, C}_{\vec{l}} = \delta\langle k, f, \vec{l}\rangle.\, \overline{C}_{\vec{l} \sqcup} \langle f, k, f, \vec{l}\rangle$$

installs the failure continuation as the current continuation, and installs the current expression continuation as the return continuation, and executes C. The interpretation of a `resultis` command

$$\overline{\mathtt{resultis}\, E}_{\vec{l}} = \delta\langle k, r, f, \vec{l}\rangle.\, \overline{E}_{\vec{l} \sqcup} \langle r, f, \vec{l}\rangle$$

evaluates expression E with the current return continuation as the expression continuation, ignoring the current continuation.

PROPOSITION 4. *1. If $x_1, \ldots, x_m$ contains the free variables of C, and $\vec{l}$ $(= l_1, \ldots, l_n)$ contains the free labels of C, then*

$$x_1 : A_1, \ldots, x_m : A_m; _ \vdash \overline{C}_{\vec{l}}$$
$$: K \& (\mathbf{B} \to K) \& K \& \underbrace{K \& \cdots \& K}_{n} \multimap K$$

2. If $x_1, \ldots, x_m$ contains the free variables of E, and $\vec{l}$ $(= l_1, \ldots, l_n)$ contains the free labels of E, then

$$x_1 : A_1, \ldots, x_m : A_m; _ \vdash \overline{E}_{\vec{l}}$$
$$: (\mathbf{B} \to K) \& K \& \underbrace{K \& \cdots \& K}_{n} \multimap K$$

8. COROUTINES

One view of a continuation is as the state of a process, and it has been known for some time that the combination of state and labels can be used to implement coroutines [11].

To design a continuation semantics of coroutines we do not, however, need the full power of the features used in these encodings; namely, first-class control and higher-order store. But we need to do more than simply have several continuations, one for each coroutine, and swap them. The extra ingredient that is needed is the ability to pass the saved state of one coroutine to another, so the other coroutine can then swap back; this is implemented using a recursive type. For simplicity, we concentrate on the case of having two coroutines, and we work with the language of arithmetic expressions.

The language consists of arithmetic expressions, E, enriched with a construct for swapping to the other coroutine, and programs, P, which set up two global coroutines.

$$E ::= x \mid n \mid E + E \mid \mathtt{swap}\, E$$
$$P ::= E \parallel (x)E$$

Execution begins with the left E. On the first `swap`, the value sent is bound to x, and the right coroutine is executed. A subsequent `swap` from one coroutine sends a value into the place of the last `swap` executed by the other. `swap`ping then continues until the left coroutine terminates. For example,

$$2 + \mathtt{swap}\, 99 \parallel (x)(x + \mathtt{swap}\, (x + 2)) + 33$$

returns 103. The $x + 2$ part of the right coroutine gets executed, $(x+[])+33$ does not. (We later discuss two options for coroutine termination.)

The transform uses two continuations: current and saved. The current continuation is where a result is delivered on normal termination, and the saved continuation records the

$$
\begin{aligned}
\overline{x}_{\vec{l}} &= \delta\langle k, f, \vec{l}\rangle.\, k\, x \\
\overline{l}_{\vec{l}} &= \delta\langle k, f, \vec{l}\rangle.\, l \\
\overline{\mathtt{true}}_{\vec{l}} &= \delta\langle k, f, \vec{l}\rangle.\, k\, \mathtt{tt} \\
\overline{\mathtt{false}}_{\vec{l}} &= \delta\langle k, f, \vec{l}\rangle.\, k\, \mathtt{ff} \\
\overline{E_0 \to E_1, E_2}_{\vec{l}} &= \delta\langle k, f, \vec{l}\rangle.\, \overline{E_0}_{\vec{l}\sqcup}\langle \lambda x.\, (\mathtt{if}\ x\ \mathtt{then}\ \overline{E_1}_{\vec{l}}\ \mathtt{else}\ \overline{E_2}_{\vec{l}})_{\sqcup}\langle k, f, \vec{l}\rangle, f, \vec{l}\rangle \\
\overline{\mathtt{valof}\, C}_{\vec{l}} &= \delta\langle k, f, \vec{l}\rangle.\, \overline{C}_{\vec{l}\sqcup}\langle f, k, f, \vec{l}\rangle \\
\\
\overline{p}_{\vec{l}} &= \delta\langle k, r, f, \vec{l}\rangle.\, [\![p]\!]_{\sqcup} k \\
\overline{\mathtt{dummy}}_{\vec{l}} &= \delta\langle k, r, f, \vec{l}\rangle.\, k \\
\overline{C_0; C_1}_{\vec{l}} &= \delta\langle k, r, f, \vec{l}\rangle.\, \overline{C_0}_{\vec{l}\sqcup}\langle \overline{C_1}_{\vec{l}\sqcup}\langle k, r, f, \vec{l}\rangle, r, f, \vec{l}\rangle \\
\overline{E \to C_0, C_1}_{\vec{l}} &= \delta\langle k, r, f, \vec{l}\rangle.\, \overline{E}_{\vec{l}\sqcup}\langle \lambda x.\, (\mathtt{if}\ x\ \mathtt{then}\ \overline{C_0}_{\vec{l}}\ \mathtt{else}\ \overline{C_1}_{\vec{l}})_{\sqcup}\langle k, r, f, \vec{l}\rangle, f, \vec{l}\rangle \\
\overline{\mathtt{goto}\, E}_{\vec{l}} &= \delta\langle k, r, f, \vec{l}\rangle.\, \overline{E}_{\vec{l}\sqcup}\langle \mathtt{diverge}_{\sqcup} k, f, \vec{l}\rangle \\
\overline{\S\, C_0; l_1 : C_1; \ldots; l_n : C_n\, \S}_{\vec{l}} &= \delta\langle k, r, f, \vec{l}\rangle.\, (\lambda\langle ,_{i=1}^{n} t_i\rangle.\, \overline{C_0}_{\vec{l}\sqcup}\langle t_{1\sqcup}\langle k, r, f, \vec{l}\rangle, r, f, \vec{l}, ,_{i=1}^{n} t_{i\sqcup}\langle k, r, f, \vec{l}\rangle\rangle) \\
&\qquad (\mathtt{Y}\, \lambda\langle ,_{i=1}^{n} t_i\rangle.\, \langle ,_{i=1}^{n-1} \delta\langle k, r, f, \vec{l}\rangle.\, \overline{C_i}_{\vec{l}, ,_{i=1}^{n} l_i \sqcup}\langle t_{i+1\sqcup}\langle k, r, f, \vec{l}\rangle, r, f, \vec{l}, ,_{i=1}^{n} t_{i\sqcup}\langle k, r, f, \vec{l}\rangle\rangle \\
&\qquad\qquad , \delta\langle k, r, f, \vec{l}\rangle.\, \overline{C_n}_{\vec{l}, ,_{i=1}^{n} l_i \sqcup}\langle k, r, f, \vec{l}, ,_{i=1}^{n} t_{i\sqcup}\langle k, r, f, \vec{l}\rangle\rangle\rangle) \\
\overline{\mathtt{resultis}\, E}_{\vec{l}} &= \delta\langle k, r, f, \vec{l}\rangle.\, \overline{E}_{\vec{l}\sqcup}\langle r, f, \vec{l}\rangle
\end{aligned}
$$

Figure 2: Transforms of expressions and commands

suspended state of the other coroutine. The domain of continuations is

$$C \cong \mathbf{N} \to C \multimap \mathbf{R},$$

and the type of expressions (coroutines) is

$$C \multimap C \multimap \mathbf{R}.$$

The transform of expressions is defined as follows.

$$
\begin{aligned}
\overline{x} &= \delta k.\delta s.\, k\, x_{\sqcup} s \\
\overline{n} &= \delta k.\delta s.\, k\, n_{\sqcup} s \\
\overline{E + F} &= \delta k.\delta s.\, \overline{E}_{\sqcup}(\lambda e.\delta t.\, \overline{F}_{\sqcup}(\lambda f.\delta r.\, k\, (e + f)_{\sqcup} r)_{\sqcup} t)_{\sqcup} s \\
\overline{\mathtt{swap}\, E} &= \delta k.\delta s.\, \overline{E}_{\sqcup}(\lambda e.\delta t.\, t\, e_{\sqcup} k)_{\sqcup} s
\end{aligned}
$$

The idea behind the transform for the `swap` construct is that E is evaluated, and then the saved continuation is invoked. In this invocation, t is used instead of s in case the other coroutine changed state (by `swap`ping and `swap`ping back) during evaluation of E. The current continuation k is saved as the suspended state of the current coroutine.

Proposition 5. *If $x_1, \ldots, x_n$ contains the free variables of E, then*

$$x_1 : \mathbf{N}, \ldots, x_n : \mathbf{N}; _ \vdash \overline{E} : C \multimap C \multimap \mathbf{R}.$$

When it comes to interpreting toplevel programs there are a number of alternatives, which revolve around the choice of what to do when one or the other coroutine terminates.

The first, purest, possibility is to simply have two toplevel continuations, and to "terminate" by passing an answer to one of the continuations, along with the state of the other coroutine. A program is also given type $C \multimap C \multimap \mathbf{R}$, and the transform is

$$\overline{E \parallel (x)F} = \delta p.\delta q.\, \overline{E}_{\sqcup} p_{\sqcup}(\lambda x.\delta s.\, \overline{F}_{\sqcup} q_{\sqcup} s).$$

In this alternative, when one of the coroutines finishes the other might still proceed further, if it is jumped back into from a toplevel continuation (p or q).

Proposition 6. *If $x_1, \ldots, x_n$ contains the free variables of $E \parallel (x)F$, then*

$$x_1 : \mathbf{N}, \ldots, x_n : \mathbf{N}; _ \vdash \overline{E \parallel (x)F} : C \multimap C \multimap \mathbf{R}.$$

In a second alternative, one coroutine's termination makes it impossible to jump back into the other, and the toplevel continuation will just have type $\mathbf{N} \to \mathbf{R}$. The two coroutines "race" until one of them finishes.

$$\overline{E \parallel (x)F} = \lambda p.\, \overline{E}_{\sqcup}(\lambda e.\delta s.\, p\, e)_{\sqcup}(\lambda x.\delta t.\, \overline{F}_{\sqcup}(\lambda e.\delta s.\, p\, e)_{\sqcup} t)$$

With this semantics, if either coroutine finishes it delivers its result to p.

There are two subtle points in this interpretation. First, if either coroutine terminates then it will *discard* the saved continuation of the other coroutine. This is necessary if the toplevel continuation is to have type $\mathbf{N} \to \mathbf{R}$. Thus, at this point we must consider an *affine* system. This is achieved by replacing the two typing rules for variables with

$$\frac{}{\Gamma, x : A; \Delta \vdash x : A} \qquad \frac{}{\Gamma; \Delta, x : P \vdash x : P}$$

The extra Δ components here are tantamount to Weakening. Technically, the need for Weakening can be seen in the fact that the continuation $(\lambda e.\delta s.\, p\, e)$ ignores s.

The second subtle point is that, since p is used in both arguments to $\overline{E}$, we must type toplevel programs using $(\mathbf{N} \to \mathbf{R}) \to \mathbf{R}$ rather than $(\mathbf{N} \to \mathbf{R}) \multimap \mathbf{R}$.

PROPOSITION 7. *If $x_1, \ldots, x_n$ contains the free variables of $E \parallel (x)F$, then*

$$x_1 : \mathbf{N}, \ldots, x_n : \mathbf{N}; _ \vdash \overline{E \parallel (x)F} : (\mathbf{N} \to \mathbf{R}) \to \mathbf{R}$$

in the affine variant.

At this point we have come up against the limitations of linear (or affine) typing. Intuitively, we *should* be able to type programs using $(\mathbf{N} \to \mathbf{R}) \multimap \mathbf{R}$ because only one of the two coroutines will finish first, and so the toplevel continuation will only be used once. Put another way, we have a harmless use of Contraction in the interpretation of $\parallel$. This limitation is perhaps not completely unexpected, since linear typing is only an approximation to linear behaviour. But it also illustrates that the problem of joining coroutines raises type-theoretic subtleties, apart from the treatment of the coroutines themselves.

9. CONCLUSIONS AND RELATED WORK

There are (at least) two main reasons why restricted type systems for CPS are of interest. The first is pragmatic, and current: when CPS is used in a compiler, we can leverage types to communicate information from the source through to intermediate and even back-end languages. A more constrained type system naturally captures more properties expected of source programs than less restricted type systems.

The second reason is conceptual. If control constructs use continuations in a stylised way, then we may hope to better understand these constructs by studying the typing properties of their semantics. An example of this is contained in the observation that `callcc` breaks linear typing, while exceptions do not.

For the case of pure simply-typed λ-calculus, the soundness result we have given—the fact that the CPS target adheres to an linear typing discipline—is well known amongst continuation experts. Surprisingly, we have not been able to find the transform stated in the literature. But, as we have emphasised, it is much more than call/return that obeys linearity. There have certainly been hints of this in the literature, especially in the treatment of coroutines using one-shot continuations [4]. Our focus on linearity grew out of a study of expressiveness, where the distinguishing power of control constructs was found to be intimately related to the number of times a continuation could be used [15, 16].

It is important to note that our approach is very different from Filinski's linear continuations [5]. In our transforms is is *continuation transformers*, rather than continuations themselves, that are linear. Also, since Filinski used a linear target language, he certainly could have accounted for linearly used continuations as we have; but his CBV transform has an additional !, which essentially turns the principal $\multimap$ we use into $\to$.

In a different line [10], Polakow and Pfenning have also investigated substructural properties of the range of CPS, and obtained excellent results. Their approach is quite different than that here in both aims and techniques; generally speaking, one might say that we take a somewhat semantic tack (focusing on use), where their approach is more exact and implementation-oriented. Compared to the approach here, an important point is their use of ordered contexts to account for "stackability". It is difficult to see how we would do the same without further analysis, because in our approach (except for coroutines) there is only ever one continuation, or a &-tuple of continuations, in the linear zone. On the other hand, the typing rules in [10] treat different occurrences of continuations differently, some linearly and some not. As a result, it is not obvious to us how the type system there might be reconstructed or explained, starting from a domain equation.

We have obtained some preliminary completeness results for linearly used continuations, but currently our analysis there is incomplete. For example, we have identified sublanguages for the procedure call and exception cases, together with syntactic completeness results, to the effect that each term in the target is $\beta\eta$-equal to terms that come from transform. But, presently, we use different "carved out" sublanguages (similar to [12]) for each source language, obtained by restricting the types in the target; these languages obviously embed into the larger one here, but there is a question as to whether these embeddings preserve completeness, and whether the transforms themselves preserve contextual equivalence relations (reflection, or soundness, is not problematic).

Besides these syntactic questions, there are a number of challenges for denotational models. For example, given a model of (CBV) λ-calculus, one might conjecture that there is a linear CPS model that is equivalent to it; here, by "equivalent" we would ask for isomorphism, or a full and faithful embedding, and not just an adequacy correspondence. For lower-order source languages we have been able to obtain completeness results based on the coherence space model, but this analysis does not extend to higher order. A good place to try to proceed further might be game models, which have been used by Laird to give very exact models of control [8], and where the linear usage of continuations is to some extend visible.

Of course, one can ask similar questions for classes of models described categorically, as well as for specific, concrete models.

In conclusion, we have displayed that many of the simple control constructs use continuations linearly. The most important remaining conceptual question is *why* linearity keeps turning up. A partial answer might be contained in the observation that each of these control constructs has a simple direct semantics. For example, procedures as functions or coroutines as resumptions. But this answer is incomplete.

ACKNOWLEDGEMENTS

This research was partially supported by the EPSRC.

10. REFERENCES

[1] M. Abadi and M. P. Fiore. Syntactic considerations on recursive types. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, 1996.

[2] S. Abramsky and G. McCusker. Call by value games. In M. Nielsen and W. Thomas (eds), *Proceedings of CSL '97*., 1997.

[3] A. Barber and G. Plotkin. Dual intuitionistic linear logic. Tech Report, Univ of Edinburgh, 1997.

[4] C. Bruggeman, O. Waddell, and R. K. Dybvig. Representing control in the presence of one-shot continuations. In *Proceedings of the ACM*

SIGPLAN'96 Conference on Programming Language Design and Implementation, 1996.

[5] A. Filinski. Linear continuations. In *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages*, 1992.

[6] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, pages 1–102, 1987.

[7] C. T. Haynes. Logic continuations. *Journal of Logic Programming*, 4:157–176, 1987.

[8] J. Laird. *A semantic analysis of control*. PhD thesis, University of Edinburgh, 1998.

[9] P. W. O'Hearn and J. C. Reynolds. From Algol to polymorphic linear lambda-calculus. *J. ACM*, 47(1):167–223, January 2000.

[10] J. Polakow and F. Pfenning. Properties of terms in continuation-passing style in an ordered logical framework, 2000. Workshop on Logical Frameworks and Meta-Languages.

[11] J. C. Reynolds. GEDANKEN—a simple typeless language based on the principle of completeness and the reference concept. In *Communications of the ACM*, pages 308–319. ACM, 1970.

[12] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, November 1992.

[13] C. Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory. Reprinted in *Higher-Order and Symbolic Computation, 13(1/2), pp135–152*, April, 2000., 1974.

[14] R. D. Tennent. Mathematical semantics of SNOBOL4. In *Conference Record of the First Annual ACM Symposium on Principles of Programming Languages*, pages 95–107. ACM, 1973.

[15] H. Thielecke. Using a continuation twice and its implications for the expressive power of `call/cc`. *Higher-Order and Symbolic Computation*, 12(1):47–74, 1999.

[16] H. Thielecke. On exceptions versus continuations in the presence of state. In G. Smolka, editor, *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000,*, number 1782 in LNCS, pages 397–411. Springer Verlag, 2000.

[17] H. Thielecke. Comparing control constructs by typing double-barrelled CPS transforms. In *Proceedings of the Third ACM SIGPLAN Workshop on Continuations*, 2001.