How to CPS Transform a Monad

Annette Bieniusa and Peter Thiemann

Institut für Informatik, Universität Freiburg, Georges-Köhler-Allee 079
79110 Freiburg, Germany
{bieniusa,thiemann}@informatik.uni-freiburg.de

Abstract. CPS transformation is an important tool in the compilation of functional programming languages. For strict languages, such as our web programming language "Rinso" or Microsoft's F#, monadic expressions can help with structuring and composing computations.

To apply a CPS transformation in the compilation process of such a language, we integrate explicit monadic abstraction in a call-by-value source language, present a Danvy-Filinski-style CPS transformation for this extension, and verify that the translation preserves simple typing. We establish the simulation properties of this transformation in an untyped setting and relate it to a two stage transformation that implements the monadic abstraction with thunks and introduces continuations in a second step. Furthermore, we give a direct style translation which corresponds to the monadic translation.

1 Introduction

A monad [21] is a powerful abstraction for a computation that may involve side effects. Programming languages that support monads are often of the lazy functional kind. For example, in Haskell [25] monads serve to integrate side-effecting computations like I/O operations, exceptions, operations on references and mutable arrays, and concurrency primitives [26,27,28,29].

However, monads do not only serve to encapsulate computation but also to structure it. The basic operations of a monad are the creation of a trivial computation (the "return" operator, which just returns a value) and the composition of computations (the "bind" operator). Thus, a computation expressed using a monad can be assembled declaratively (and compositionally) from some primitive computations. This compositionality aspect has proven its relevance, for example, in the Kleisli database query system where a monad abstracts over different collection types and its laws serve as simplification rules for queries [41].

Monadic structure also plays a role in strict languages (see Danvy and Hatcliff's factorization of CPS translations [14], Wadler's marriage of monads and effects [39], or the work on monadic regions [12]) and there are less obvious applications like the monads representing probability distributions in the work of Ramsey and Pfeffer [31] or Park and others [24].

We are currently running two projects in the context of call-by-value functional programming languages that both benefit from the structuring aspect of

O. de Moor and M. Schwartzbach (Eds.): CC 2009, LNCS 5501, pp. 266–280, 2009.

a monad and the compositionality of monadic computations. The first project concerns the implementation of a web programming language inspired by the second author's work on the WASH system [38], the Links project [6], Hop [35], and generally the idea of tierless web programming [22]. The second project deals with the efficient implementation of Park's work [24] on representing probability distributions by sampling functions (of monadic type).

Another indication for the importance of monads in strict languages is the recent addition of *workflow expressions* to the F# language[37]. These workflow expressions (or computation expressions) are nothing but monad comprehensions [40] which admit some additional operators for monads that support them. F# supports list and sequence operations, database operations, asynchronous operations, manipulation of probability distributions as in Ramsey and Pfeffer's work [31], and a few more monadic computations through workflow expressions. Interestingly, the concrete syntax chosen in F# closely matches our calculus in Sec.3.1. Thus, our results are applicable to compiling F#.

The suitability of the CPS transformation for compilation has been disputed [11,5] but is now receiving renewed attention and is successfully competing with other approaches like ANF or monadic languages [17]. Our projects and in particular the work reported here may yield additional evidence in favor of CPS.

The projects have two commonalities. First, both source languages are strict functional languages with linguistic support for monads (see Sec. 2). Both languages restrict side effects to monadic computations, so we are after encapsulation of both, effects and compositionality.[1] Second, both implementations involve a CPS translation, a well-established implementation path for such languages. These two requirements lead directly to the present work.

The main contributions of this work are as follows. We define Λ^M, a call-by-value lambda calculus with explicit monadic constructs (a strict variant of the monadic metalanguage). We specify an optimizing CPS translation from Λ^M to the lambda calculus and prove its simulation and translation properties. We define the corresponding direct-style translation and prove simulation for it. We briefly investigate an alternative transformation that first performs thunkification and then runs a standard CPS transformation. We state a type system based on simple types for Λ^M and prove that the transformation preserves typing.

2 Two Strict Languages with Monads

In two seemingly unrelated projects, we have arrived at using a strict language with a monadic sublanguage as a good match for the problem domain. In both projects there is also the need of applying the CPS transformation to programs. This section briefly introduces the projects and explains the role of the CPS transformation in their implementation.

Rinso. Rinso is an experimental programming language for writing client-side web applications. Rinso compiles to JavaScript and provides convenient monadic

[1] Another option would be to structure side effects using a hierarchy of effect-indexed monads [10], but we stick with the simpler scenario for this paper.

```
// producer : MVar Int * Int * Int -> IO ()
producer (mvar, a, b)
  if (a <= b) {
    exec (putMVar (mvar, a));
    exec (producer (mvar, a+1, b))
  } else {
    return ()
  }
// consumer : MVar Int -> IO ()
consumer (mvar) {
  x = exec (readMVar (mvar));
  exec (print (x));
  consumer (mvar)
}
// main : Unit -> IO ()
main () {
  mvar = exec newEmptyMVar;
  exec (fork (producer (mvar, 1, 100)));
  exec (consumer (mvar))
}
```

Fig. 1. Producer and consumer in Rinso

abstractions to protect programmers from the idiosyncrasies of the target language as much as possible. The current prototype implementation supports a monadic interface to I/O, references, and concurrency via thread creation. Before getting to an actual example program, we take a short digression and explain the underlying concurrency primitives.

Rinso's concurrency library is based on Concurrent Haskell's MVar abstraction [26]. An MVar is a mutable variable with two distinct states. It is either empty or it is full and holds a value of a fixed type. An MVar supports the following operations in the `IO` monad:

```
newEmptyMVar : IO (MVar a)
putMVar      : MVar a * a -> IO ()
readMVar     : MVar a -> IO a
```

An MVar starts its life cycle with an invocation of `newEmptyMVar`, which creates a fresh, empty MVar. The execution of `readMVar (mv)` blocks while `mv` is empty. If `mv` is full, then `readMVar mv` empties `mv` and returns its value. The execution of `putMVar (mv, v)` blocks while `mv` is full. If `mv` is empty, then `putMVar (mv, v)` fills `mv` with `v` and returns the unit value. Multiple `readMVar` (`putMVar`) may block on the same empty (full) MVar, only one will be chosen by the run-time system to proceed. The operations `putMVar` and `readMVar` are atomic.

Figure 1 shows an excerpt of Rinso code implementing a producer/consumer abstraction. Rinso marks monadic computations by curly braces, that is, $\{m\}$ is a computation defined by the statement sequence m (which is quite similar to Haskell's do-notation [25]). A statement can be a binding $x = e$; (where the

```
// bernoulli : double -> P bool
bernoulli (p) {
  x = exec sample;
  return (x <= p)
}
// uniform : double * double -> P double
uniform (a, b) {
  x = exec sample;
  return (a + x * (b-a))
}
// gaussian : double * double -> P double
gaussian (m, sigma) {
  x1 = exec sample;
  x2 = exec sample;
  ...
  x12 = exec sample;
  return (m + sigma * (x1 + x2 + ... + x12 - 6.0))
}
```

Fig. 2. Encodings of distributions

$x =$ part may be omitted) or a return statement `return` e. In both cases, e is evaluated. Ordinary binding is free of side effects, whereas a binding $x =$ `exec` e ; expects e to evaluate to a monadic value which is then executed immediately.

The prototype implementation of Rinso performs lambda lifting, CPS transformation, and closure conversion. The resulting first-order program is translated to JavaScript. The CPS transformation must be involved for two reasons. First, the target technology (your friendly web browser) stops programs that run "too long". Hence, the program has to be chopped in pieces that invoke each others indirectly. Cooper et al. report a similar approach [6].

Second, as Rinso is supposed to be used on the client side of a web application, it needs facilities for implementing user interfaces. One important ingredient here is concurrency where Rinso supports a thread model similar to concurrent Haskell. The implementation of such a thread model is much facilitated if programs are translated to CPS.

A planned extension of Rinso to also include server-side computation would add yet another reason for using the CPS transformation. As Graunke et al. [19] point out, compiling interactive programs for server-side execution requires a CPS transformation.

Stochastic Computation. Our second project concerns sensor-based technical devices. These devices perform stochastic processing of their sensor data close to the sensors themselves to avoid network congestion with bulk data and also to save power by keeping network transmitters powered down as long as possible.

To cut down on power and cost, as well as to lower the likelihood of errors, part of this processing is implemented in hardware. Thus, this hardware implements computation-intensive tasks which remain fixed over the lifetime of the system.

It is often co-designed with the software that performs higher level processing tasks which are more likely to change over time.

Our project investigates an approach to specifying such systems in a single linguistic framework. One core aspect is the modeling of probability distributions using a sampling monad as inspired by Park et al.'s work [24]. One obstacle in putting this work into practice is its limited performance if implemented purely in software. Thus, we aim at implementing the stochastic processing in hardware. The implementation follows one of the standard paths in functional language compilation, CPS transformation and closure conversion, before mapping the program to hardware via VHDL [34].

Figure 2 contains some distributions which are encoded using a Rinso-like syntax. They are transcribed from Park et al. [24]. The basic computation is `sample` of type `P double` (where `P` is the probability monad), which models a 0-1 uniformly distributed random variable. `bernoulli(p)` implements a Bernoulli distribution with probability `p`, `uniform(a,b)` implements a uniform distribution over the interval `(a,b)`, and `gaussian(m,sigma)` implements an (approximation of a) Gaussian distribution using the 12-rule.

3 CPS Transformation

3.1 The Source Language

Figure 3 shows the syntax of Λ^M, a call-by-value lambda calculus extended with monadic expressions. In addition to constants, variables, lambda abstractions, and function applications (marked with infix @) there are also monadic computations $\{m\}$, which open a new binding scope with the $x = \ldots$ statements as binding operations. Side effects can only occur in computations. Computations can be bound to variables as in $x = \{m\}$ because they are values. Their evaluation must be triggered via the keyword `exec`. The monadic statement $x = e\,; m$ behaves like $x =$ `exec` $\{$`return` $e\}\,; m$. We use fv to denote the set of free variables in an expression or computation, and bv for variable bound by a binding operation m_i. The `print` operation which displays integers serves as an example for a side effecting operation.

The figure further defines the semantics of Λ^M. Monadic reduction $\mapsto_m$ is the top-level notion of reduction. $\mathcal{M}$ denotes the evaluation context for monadic statements, $\mathcal{E}$ the corresponding one for expressions. The superscript on the reduction can be i representing the printed value or ε if no output happens. The annotation $\mathcal{A}$ on the transitive closure of reduction stands for a (potentially empty) sequence of integers. Computation stops with `return` v at the top level.

Figure 4 presents a simple type system for Λ^M inspired by Moggi's metalanguage [21]. The unary type constructor T represents the monad. Hence, a computation returning a value of type τ has type $\mathrm{T}\,\tau$.

Theorem 1. *The type system in Fig. 4 is sound with respect to the semantics of Λ^M in Fig. 3.*

Syntax:

$$
\begin{array}{lll}
\text{expressions} & e & ::= c \mid x \mid \lambda x.e \mid e@e \mid \{m\} \\
\text{statements} & m & ::= \texttt{return}\ e \mid x = \texttt{exec}\ e\,; m \mid x = e\,; m \\
\text{constants} & c & ::= \ulcorner i \urcorner \mid \texttt{print} \\
\text{values} & v & ::= \ulcorner i \urcorner \mid \lambda x.e \mid \{m\} \mid \texttt{print} \\
\text{output} & a & ::= \varepsilon \mid i \qquad\qquad i \in \mathbb{Z} \\
\text{variables} & x & \in \mathit{Var}
\end{array}
$$

Evaluation contexts:

$$
\begin{array}{rcl}
\mathcal{M} & ::= & x = \texttt{exec}\ \mathcal{E}\,; m \mid x = \mathcal{E}\,; m \mid \texttt{return}\ \mathcal{E} \\
\mathcal{E} & ::= & [\,] \mid \mathcal{E}@e \mid v@\mathcal{E}
\end{array}
$$

Evaluation:

$$
\begin{array}{c}
(\lambda x.e)@v \;\mapsto_e\; e[x \mapsto v] \\
x = v\,; m \overset{\varepsilon}{\mapsto}_m m[x \mapsto v] \\
x = \texttt{exec}\ (\texttt{print}@\ulcorner i \urcorner)\,; m \overset{i}{\mapsto}_m m[x \mapsto \ulcorner i \urcorner] \\
x = \texttt{exec}\ \{m_1; \ldots; m_n; \texttt{return}\ e\}\,; m \overset{\varepsilon}{\mapsto}_m m_1; \ldots; m_n; x = e\,; m \\
\text{if } \mathit{fv}(m) \cap \mathit{bv}(m_1, \ldots, m_n) = \emptyset
\end{array}
$$

$$
\frac{e \mapsto_e e'}{\mathcal{E}[e] \mapsto_e \mathcal{E}[e']} \qquad e \mapsto_e^* e \qquad \frac{e \mapsto_e^* e' \qquad e' \mapsto_e e''}{e \mapsto_e^* e''} \qquad \frac{e \mapsto_e e'}{\mathcal{M}[e] \overset{\varepsilon}{\mapsto}_m \mathcal{M}[e']}
$$

$$
\frac{m \overset{a}{\mapsto}_m m'}{\mathcal{M}[m] \overset{a}{\mapsto}_m \mathcal{M}[m']} \qquad m \overset{\varepsilon}{\mapsto}_m{}^* m \qquad \frac{m \overset{\mathcal{A},*}{\longrightarrow}_m m' \qquad m' \overset{a}{\mapsto}_m m''}{m \overset{\mathcal{A}a,*}{\longrightarrow}_m m''}
$$

Fig. 3. Syntax and semantics of the source language Λ^M

$$
\begin{array}{lll}
\text{types} & \tau, \sigma & ::= \text{int} \mid \tau \to \tau \mid \mathrm{T}\,\tau \\
\text{contexts} & \Gamma & ::= \cdot \mid \Gamma, x : \tau
\end{array}
$$

Typing rules:

$$
\frac{}{\Gamma \vdash_e \texttt{print} : \text{int} \to \mathrm{T}\ \text{int}} \qquad \frac{}{\Gamma \vdash_e \ulcorner i \urcorner : \text{int}} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash_e x : \tau}
$$

$$
\frac{\Gamma, x : \tau_1 \vdash_e e : \tau_2}{\Gamma \vdash_e \lambda x.e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash_e e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash_e e_2 : \tau_1}{\Gamma \vdash_e e_1@e_2 : \tau_2}
$$

$$
\frac{\Gamma \vdash_m m : \mathrm{T}\,\tau}{\Gamma \vdash_e \{m\} : \mathrm{T}\,\tau} \qquad \frac{\Gamma \vdash_e e : \tau}{\Gamma \vdash_m \texttt{return}\ e : \mathrm{T}\,\tau} \qquad \frac{\Gamma \vdash_e e : \tau \qquad \Gamma, x : \tau \vdash_m m : \mathrm{T}\,\tau'}{\Gamma \vdash_m x = e; m : \mathrm{T}\,\tau'}
$$

$$
\frac{\Gamma \vdash_e e : \mathrm{T}\,\tau \qquad \Gamma, x : \tau \vdash_m m : \mathrm{T}\,\tau'}{\Gamma \vdash_m x = \texttt{exec}\ e; m : \mathrm{T}\,\tau'}
$$

Fig. 4. Simple type system for Λ^M

3.2 CPS Transformation of Monadic Expressions

Our CPS transformation on Λ^M terms extends Danvy and Filinski's one-pass optimizing call-by-value CPS transformation [8] with transformation rules for

Syntax:

$$\begin{array}{lll} \text{expressions} & E, F & ::= C \mid x \mid \lambda x.E \mid F@E \\ \text{constants} & C & ::= \mathtt{print}_c \mid \ulcorner i \urcorner \\ \text{values} & V & ::= C \mid \lambda x.E \end{array}$$

where $i \in \mathbb{Z}$ and $x \in \mathit{Var}$, an infinite set of variables
Reduction contexts for call-by-value ($\mathcal{E}_v$) and call-by-name ($\mathcal{E}_n$):

$$\mathcal{E}_v ::= [\,] \mid \mathcal{E}_v@E \mid V@\mathcal{E}_v \qquad \mathcal{E}_n ::= [\,] \mid \mathcal{E}_n@E$$

Reduction (for $j \in \{v, n\}$):

$$(\lambda x.E)@F \overset{\varepsilon}{\mapsto}_\beta E[F/x] \qquad (\lambda x.E)@V \overset{\varepsilon}{\mapsto}_{\beta V} E[V/x] \qquad (\mathtt{print}_c@\ulcorner i \urcorner)@F \overset{i}{\mapsto}_\gamma F@\ulcorner i \urcorner$$

$$\frac{E \overset{a}{\mapsto}_{\beta V,\gamma} E'}{\mathcal{E}_v[E] \overset{a}{\mapsto}_v \mathcal{E}_v[E']} \qquad \frac{E \overset{a}{\mapsto}_{\beta,\gamma} E'}{\mathcal{E}_n[E] \overset{a}{\mapsto}_n \mathcal{E}_n[E']} \qquad E \overset{\varepsilon,*}{\longmapsto}_j E \qquad \frac{E \overset{\mathcal{A},*}{\longmapsto}_j E' \qquad E' \overset{a}{\longmapsto}_j E''}{E \overset{\mathcal{A}a,*}{\longmapsto}_j E''}$$

Fig. 5. The target language Λ

monadic expressions and statements. The result is a one-pass CPS transformation which does not introduce any administrative β-redexes. In addition, potential η-redexes around tail calls are avoided by using auxiliary transformations $\mathcal{C}'_e$ and $\mathcal{C}'_m$.

The transformation is defined in a two-level lambda calculus [23] which distinguishes between abstractions and applications at transformation time ($\overline{\lambda}x.e$ and $f\overline{@}e$) and at run time ($\underline{\lambda}x.e$ and $f\underline{@}e$). The former reduce during transformation whereas the latter generate target code.

Figure 5 defines syntax and semantics of the target language of the transformation. There are two semantics, call-by-value given by the relation $\mapsto_v$ and call-by-name given by $\mapsto_n$. The print operation is provided in terms of a CPS primitive $\mathtt{print}_c$.

Figure 6 defines the CPS transformation for Λ^M. The result of transforming an expression e to CPS in an empty context is given by $\mathcal{C}_e[\![e]\!]\overline{@}(\overline{\lambda}z.z)$, and in a dynamic context by $\underline{\lambda}k.\mathcal{C}_e[\![e]\!]\overline{@}(\overline{\lambda}z.k\underline{@}z)$. The same holds for the transformation of monadic expressions m. The latter are only transformed in a dynamic context, so the corresponding transformation $\mathcal{C}_m[\![\_]\!]$ for static contexts has been elided.

The type transformation corresponding to our call-by-value CPS transformation is defined in two steps with a value type transformation * and a computation type transformation $\_^\sharp$. The type X is the answer type of all continuations.

$$\begin{aligned} \mathrm{int}^* &= \mathrm{int} \\ (\tau \to \sigma)^* &= \tau^* \to \sigma^\sharp \\ (\mathrm{T}\,\tau)^* &= \tau^\sharp \\ \tau^\sharp &= (\tau^* \to X) \to X \end{aligned}$$

Theorem 2. *If $\Gamma \vdash_e e : \tau$, then $\Gamma^*, k : \tau^* \to X \vdash_{\underline{e}} (\mathcal{C}'_e[\![e]\!])\overline{@}k : \tau^\sharp$.*
If $\Gamma \vdash_m m : \tau$, then $\Gamma^*, k : \tau^* \to X \vdash_{\underline{e}} (\mathcal{C}'_m[\![m]\!])\overline{@}k : \tau^\sharp$.

Danvy and Filinski's optimizing call-by-value CPS transformation [8]

$$\begin{aligned}
\mathcal{C}_e[\![\ulcorner i \urcorner]\!] &= \overline{\lambda}\kappa.\kappa\overline{@}\ulcorner i \urcorner \\
\mathcal{C}_e[\![x]\!] &= \overline{\lambda}\kappa.\kappa\overline{@}x \\
\mathcal{C}_e[\![\lambda x.e]\!] &= \overline{\lambda}\kappa.\kappa\overline{@}(\underline{\lambda}x.\underline{\lambda}k.\mathcal{C}'_e[\![e]\!]\overline{@}k) \\
\mathcal{C}_e[\![e_0 @ e_1]\!] &= \overline{\lambda}\kappa.\mathcal{C}_e[\![e_0]\!]\overline{@}(\overline{\lambda}v_0.\mathcal{C}_e[\![e_1]\!]\overline{@}(\overline{\lambda}v_1.(v_0\underline{@}v_1)\underline{@}(\underline{\lambda}a.\kappa\overline{@}a)))
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}'_e[\![\ulcorner i \urcorner]\!] &= \overline{\lambda}k.k\underline{@}\ulcorner i \urcorner \\
\mathcal{C}'_e[\![x]\!] &= \overline{\lambda}k.k\underline{@}x \\
\mathcal{C}'_e[\![\lambda x.e]\!] &= \overline{\lambda}k.k\underline{@}(\underline{\lambda}x.\underline{\lambda}k.\mathcal{C}'_e[\![e]\!]\overline{@}k) \\
\mathcal{C}'_e[\![e_0 @ e_1]\!] &= \overline{\lambda}k.\mathcal{C}_e[\![e_0]\!]\overline{@}(\overline{\lambda}v_0.\mathcal{C}_e[\![e_1]\!]\overline{@}(\overline{\lambda}v_1.(v_0\underline{@}v_1)\underline{@}k))
\end{aligned}$$

Extension to monadic expressions and statements

$$\begin{aligned}
\mathcal{C}_e[\![\mathtt{print}]\!] &= \overline{\lambda}\kappa.\kappa\overline{@}(\underline{\lambda}x.\underline{\lambda}k.k\underline{@}(\mathtt{print}_c\,\underline{@}x)) \\
\mathcal{C}_e[\![\{m\}]\!] &= \overline{\lambda}\kappa.\kappa\overline{@}(\underline{\lambda}k.\mathcal{C}'_m[\![m]\!]\overline{@}k)
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}'_e[\![\mathtt{print}]\!] &= \overline{\lambda}k.k\underline{@}(\underline{\lambda}x.\underline{\lambda}k.k\underline{@}(\mathtt{print}_c\,\underline{@}x)) \\
\mathcal{C}'_e[\![\{m\}]\!] &= \overline{\lambda}k.k\underline{@}(\underline{\lambda}n.\mathcal{C}'_m[\![m]\!]\overline{@}n)
\end{aligned}$$

$$\begin{aligned}
\mathcal{C}'_m[\![\mathtt{return}\ e]\!] &= \mathcal{C}'_e[\![e]\!] \\
\mathcal{C}'_m[\![x = e\,; m]\!] &= \overline{\lambda}k.\mathcal{C}'_e[\![e]\!]\overline{@}(\underline{\lambda}x.\mathcal{C}'_m[\![m]\!]\overline{@}k) \\
\mathcal{C}'_m[\![x = \mathtt{exec}\ e\,; m]\!] &= \overline{\lambda}k.\mathcal{C}_e[\![e]\!]\overline{@}(\overline{\lambda}v.v\underline{@}(\underline{\lambda}x.\mathcal{C}'_m[\![m]\!]\overline{@}k))
\end{aligned}$$

Fig. 6. CPS transformation

Proof. The proof works by ignoring the annotations, performing induction on the translated terms, and then invoking subject reduction for the simply typed lambda calculus to see that the overlined reductions do not change the type.

3.3 Simulation and Indifference

Danvy and Filinski [8] have shown that the upper half of the rules in Fig. 6 transforms a source term to a result which is $\beta\eta$-equivalent to applying Plotkin's call-by-value CPS transformation to the same source term. Like Plotkin, they prove simulation and indifference results and we follow their lead closely in extending the simulation and indifference results to our setting.

For values v let $\Psi(v) = \mathcal{C}_e[\![v]\!]\overline{@}(\overline{\lambda}x.x)$. It is straightforward to show that $\Psi(v)$ is a value and that the following equations hold:

$$\begin{aligned}
\mathcal{C}_e[\![v]\!]\overline{@}\kappa &= \kappa\overline{@}(\Psi(v)) \\
\mathcal{C}'_e[\![v]\!]\overline{@}k &= k\underline{@}(\Psi(v)) \\
\mathcal{C}_e[\![w]\!]\overline{@}\kappa &= \mathcal{C}'_e[\![w]\!]\overline{@}(\underline{\lambda}n.\kappa\overline{@}n)
\end{aligned}$$

where v denotes a value and w a term that is not a value.

A variable x occurs free in a static continuation κ if for some term p it occurs free in $\kappa\overline{@}p$ but not in p. An expression κ is *schematic* if for any terms p and q and any variable x not occurring free in κ,

$$(\kappa\overline{@}p)[x \mapsto q] = \kappa\overline{@}(p[x \mapsto q]).$$

Lemma 1. *Let p be a term, v a value, x a variable, x' a fresh variable, and let κ be a schematic continuation and k any term. Then*

$$\mathcal{C}_e[\![p[x \mapsto v]]\!]\overline{@}\kappa = (\mathcal{C}_e[\![p[x \mapsto x']]\!]\overline{@}\kappa)[x' \mapsto \Psi(v)]$$
$$\mathcal{C}'_{e/m}[\![p[x \mapsto v]]\!]\overline{@}k = (\mathcal{C}'_{e/m}[\![p[x \mapsto x']]\!]\overline{@}k)[x' \mapsto \Psi(v)]$$

Proof. By induction on p.

The next lemma extends the indifference theorem to Λ^M. All reductions are independent of the choice of the reduction strategy j for the target language: Each argument of an application is a value from the beginning, hence the $V@\mathcal{E}_v$ evaluation context is never needed and the rule βV is sufficient for all reductions. The relation $\overset{a,+}{\mapsto}_j$ denotes the transitive closure of the respective relation $\overset{a}{\mapsto}_j$.

Lemma 2. *Let κ be a schematic continuation and $j \in \{v, n\}$.*
If $p \mapsto_e q$, then $\mathcal{C}_e[\![p]\!]\overline{@}\kappa \overset{\varepsilon,+}{\mapsto}_j \mathcal{C}_e[\![q]\!]\overline{@}\kappa$ and $\mathcal{C}'_e[\![p]\!]\overline{@}k \overset{\varepsilon,+}{\mapsto}_j \mathcal{C}'_e[\![q]\!]\overline{@}k$.
If $p \overset{a}{\mapsto}_m q$, then $\mathcal{C}'_m[\![p]\!]\overline{@}k \overset{a,+}{\mapsto}_j \mathcal{C}'_m[\![q]\!]\overline{@}k$.
Each source reduction gives rise to at most five target reduction steps.

Proof. Induction on the derivation of $\mapsto_e$ and $\overset{i}{\mapsto}_m$. The case for reducing $x =$ `exec` (`print`$@\ulcorner i \urcorner$) ; takes five steps in the target language. All other cases take fewer steps.

Inductive application of Lemma 2 to a multi-step reduction yields the indifference and simulation theorem.

Theorem 3. *Let m be a well-typed term and v be a value such that $m \overset{\mathcal{A}}{\mapsto}_m$* `return` *$v$. Then*

$$\mathcal{C}'_m[\![m]\!]\overline{@}(\underline{\lambda}x.x) \overset{\mathcal{A},*}{\mapsto}_j \Psi(v)$$

in at most five times as many reduction steps for $j \in \{v, n\}$.

4 Alternative CPS Transformation

An obvious alternative to the discussed CPS transformation works in two stages, thunkification followed by CPS transformation. Thunkification defers the evaluation of a monadic expression by wrapping its body into a thunk. The transformation of `exec` forces the thunk's evaluation by providing a dummy argument.

We extend Λ^M (and its CPS transformation) with a new direct-style print operator $\texttt{print}_d$ as indicated in Fig. 7. Figure 8 gives the thunkification as a transformation on Λ^M. It maps `print` to a function that accepts an output

$$
\begin{array}{lll}
c ::= \cdots \mid \mathtt{print}_d & & \mathtt{print}_d@\ulcorner i \urcorner \overset{i}{\mapsto}_e \ulcorner i \urcorner \\
v ::= \cdots \mid \mathtt{print}_d & & \mathcal{C}_e[\![\mathtt{print}_d]\!] \;=\; \overline{\lambda}\kappa.\kappa\overline{@}\mathtt{print}_c
\end{array}
$$

Fig. 7. Extension of the source language

$$
\begin{array}{ll}
\mathcal{T}_e[\![\ulcorner i \urcorner]\!] & = \ulcorner i \urcorner \\
\mathcal{T}_e[\![\mathtt{print}]\!] & = \lambda x.\lambda z.\mathtt{print}_d@x \qquad z \neq x \\
\mathcal{T}_e[\![x]\!] & = x \\
\mathcal{T}_e[\![\lambda x.e]\!] & = \lambda x.\mathcal{T}_e[\![e]\!] \\
\mathcal{T}_e[\![e_1@e_2]\!] & = (\mathcal{T}_e[\![e_1]\!])@(\mathcal{T}_e[\![e_2]\!]) \\
\mathcal{T}_e[\![\{m\}]\!] & = \lambda z.\mathcal{T}_m[\![m]\!] \qquad z \notin fv(m) \\
\mathcal{T}_m[\![\mathtt{return}\ e]\!] & = \mathcal{T}_e[\![e]\!] \\
\mathcal{T}_m[\![x = e\,; m]\!] & = (\lambda x.\mathcal{T}_m[\![m]\!])@(\mathcal{T}_e[\![e]\!]) \\
\mathcal{T}_m[\![x = \mathtt{exec}\ e\,; m]\!] & = (\lambda x.\mathcal{T}_m[\![m]\!])@((\mathcal{T}_e[\![e]\!])@\ulcorner 0 \urcorner)
\end{array}
$$

Fig. 8. Thunkification

value and a dummy argument and calls $\mathtt{print}_d$ if the dummy argument is provided. The value $\ulcorner 0 \urcorner$ serves as a dummy argument to force the evaluation of the expression following an `exec`. The transformed program does not use the monadic constructs anymore.[2]

We now get a one-pass CPS transformation as the combination of two transformations:

$$\tilde{\mathcal{C}}_e[\![p]\!] = \mathcal{C}_e[\![\mathcal{T}_e[\![p]\!]]\!] \quad \text{and} \quad \tilde{\mathcal{C}}'_{e/m}[\![p]\!] = \mathcal{C}'_e[\![\mathcal{T}_{e/m}[\![p]\!]]\!]$$

The result is a set of somewhat more complicated transformation rules for the monadic expressions (all other transformation rules remain unchanged as they are not affected by thunkification).

$$
\begin{aligned}
\tilde{\mathcal{C}}_e[\![\mathtt{print}]\!] &= \overline{\lambda}\kappa.\kappa\overline{@}\underline{\lambda}x.\underline{\lambda}k.k\underline{@}(\underline{\lambda}z.\underline{\lambda}k.(\mathtt{print}_c\underline{@}x)\underline{@}k) \\
\tilde{\mathcal{C}}_e[\![\{m\}]\!] &= \overline{\lambda}\kappa.\kappa\overline{@}\underline{\lambda}z.(\underline{\lambda}k.\tilde{\mathcal{C}}'_m[\![m]\!]\overline{@}k) \\
\tilde{\mathcal{C}}'_m[\![\mathtt{return}\ e]\!] &= \mathcal{C}'_e[\![e]\!] = \mathcal{C}'_m[\![\mathtt{return}\ e]\!] \\
\tilde{\mathcal{C}}'_m[\![x = e\,; m]\!] &= \overline{\lambda}k.\tilde{\mathcal{C}}_e[\![e]\!]\overline{@}(\overline{\lambda}v_1.((\underline{\lambda}x.\underline{\lambda}k.\tilde{\mathcal{C}}'_m[\![m]\!]\overline{@}k)\underline{@}v_1)\underline{@}k) \\
\tilde{\mathcal{C}}'_m[\![x = \mathtt{exec}\ e\,; m]\!] &= \\
&\quad \overline{\lambda}k.\tilde{\mathcal{C}}_e[\![e]\!]\overline{@}(\overline{\lambda}w_0.(w_0\underline{@}\ulcorner 0 \urcorner)\underline{@}(\underline{\lambda}a.((\underline{\lambda}x.\underline{\lambda}k.\tilde{\mathcal{C}}'_m[\![m]\!]\overline{@}k)\underline{@}a)\underline{@}k))
\end{aligned}
$$

As one can easily show, this more intuitive ansatz is $\beta\eta$ equivalent, but less efficient for the monadic constructs as the one in Fig. 6. Indeed, of the most frequently used monadic operations the $x = v$ binding requires one additional reduction step and the $x = \mathtt{exec}\,\{m\}$ binding requires three additional reduction steps.

[2] Park's implementation of the probability monad [24] works in a similar way.

5 Direct-Style Translation

To obtain the direct-style translation in Fig.9 corresponding to the monadic translation in Fig.6, we first have to find a suitable grammar for the resulting CPS terms. The nonterminals cv, cc, and ck stand for CPS values, computations, and continuations. Their definitions are familiar from direct-style translations for the lambda calculus [7]. The last two cases for cv are specific to the monadic case. They involve mc (monadic computations), which in turn involve monadic continuations mk. The translation inserts $\texttt{let}\, x = e\, \texttt{in}\, f$ expressions which are interpreted as $(\lambda x.f)@e$.

The special cases are as follows. The new value $\lambda k.mc$ corresponds to a monadic computation. The computation $cv@mk$ stands for the activation of a delayed computation and is hence mapped to an `exec` statement in the monad.

The direct style transformation is given for each CPS term. To obtain better readability, $\mathcal{D}^e_{mk}[\![\_]\!]$ denotes the translation that results in a monadic binding with `exec`. The expected simulation result holds:

Lemma 3. *Suppose that* $mc \xmapsto{\mathcal{A},*}_j k@cv$. *Then* $\mathcal{D}_{mc}[\![mc]\!] \xmapsto{\mathcal{A},*}_m \mathcal{D}_{mc}[\![k@cv]\!]$.

However, the pair of transformations $\mathcal{C}'_m$ and $\mathcal{D}_{mc}$ does not form an equational correspondence (let alone a reduction correspondence or a reflection) because the source language Λ^M lacks reductions that perform `let` insertion and `let` normalization. Such reductions are added in the work of Sabry, Wadler, and Felleisen [33,32] and lead directly to the existence of such correspondences. The same reductions could be added to Λ^M with the same effect, but we refrained from doing so because it yields no new insights.

6 Related Work

Since Plotkin's seminal paper [30] CPS transformations have been described and characterized in many different flavors. Danvy and Filinski [8] describe an optimizing one-pass transformation for an applied call-by-value lambda calculus that elides administrative reductions by making them static reductions which are performed at transformation time. Our transformation extends their results for a source language with an explicit monad.

Danvy and Hatcliff [9] present a CPS transformation that exploits the results of strictness analysis. Our transformation of the explicit monad is inspired by their treatment of `force` and `delay`, but adds the one-pass machinery.

Hatcliff and Danvy's generic account of continuation-passing styles [14] factorizes CPS transformations in two strata. The first stratum transforms the source language into Moggi's computational meta-language [21] encoding different evaluation strategies. The second stratum "continuation introduction" is independent from the source language and maps the meta-language into the CPS sublanguage of lambda calculus. Our transformation is reminiscent of the second stratum, but our source language is call-by-value lambda calculus with an explicit monad and our transformation optimizes administrative reductions.

Grammar of CPS terms

$$
\begin{array}{rcl}
cv & ::= & \ulcorner i \urcorner \mid x \mid \lambda x.\lambda k.cc \mid \lambda k.mc \mid \texttt{print}_c@x \\
cc & ::= & cv@cv@ck \mid ck@cv \\
ck & ::= & \lambda a.cc \mid k \\
mc & ::= & cv@cv@mk \mid mk@cv \mid cv@mk \\
mk & ::= & \lambda x.mc \mid k
\end{array}
$$

Lambda calculus cases

$$
\begin{array}{ll}
\mathcal{D}_{cv}[\![\ulcorner i \urcorner]\!] = \ulcorner i \urcorner & \mathcal{D}_{cc}[\![ck@cv]\!] = \mathcal{D}_{ck}[\![ck]\!][\mathcal{D}_{cv}[\![cv]\!]] \\
\mathcal{D}_{cv}[\![x]\!] = x & \mathcal{D}_{ck}[\![k]\!] = [\,] \\
\mathcal{D}_{cv}[\![\lambda x.\lambda k.cc]\!] = \lambda x.\mathcal{D}_{cc}[\![cc]\!] & \mathcal{D}_{ck}[\![\lambda a.cc]\!] = \texttt{let}\ a = [\,]\ \texttt{in}\ \mathcal{D}_{cc}[\![cc]\!] \\
\mathcal{D}_{cc}[\![cv_1@cv_2@ck]\!] = \mathcal{D}_{ck}[\![ck]\!][\mathcal{D}_{cv}[\![cv_1]\!]@\mathcal{D}_{cv}[\![cv_2]\!]] &
\end{array}
$$

Monadic cases

$$
\begin{array}{ll}
\mathcal{D}_{cv}[\![\lambda k.mc]\!] = \{\mathcal{D}_{mc}[\![mc]\!]\} & \mathcal{D}^e_{mk}[\![\lambda x.mc]\!] = x = \texttt{exec}\ [\,]\,;\mathcal{D}_{mc}[\![mc]\!] \\
\mathcal{D}_{cv}[\![\texttt{print}_c@x]\!] = \texttt{print}@x & \mathcal{D}^e_{mk}[\![k]\!] = x = \texttt{exec}\ [\,]\,;\texttt{return}\ x \\
\mathcal{D}_{mc}[\![mk@cv]\!] = \mathcal{D}_{mk}[\![mk]\!][\mathcal{D}_{cv}[\![cv]\!]] & \mathcal{D}_{mk}[\![\lambda x.mc]\!] = x = [\,]\,;\mathcal{D}_{mc}[\![mc]\!] \\
\mathcal{D}_{mc}[\![cv@mk]\!] = \mathcal{D}^e_{mk}[\![mk]\!][\mathcal{D}_{cv}[\![cv]\!]] & \mathcal{D}_{mk}[\![k]\!] = \texttt{return}\ [\,] \\
\mathcal{D}_{mc}[\![cv_1@cv_2@mk]\!] = \mathcal{D}_{mk}[\![mk]\!][\mathcal{D}_{cv}[\![cv_1]\!]@\mathcal{D}_{cv}[\![\ cv_2]\!]] &
\end{array}
$$

Fig. 9. Direct style translation

An unoptimized version of our transformation could likely be factored through the computational meta-language, but we have not investigated this issue, yet.

Danvy and Hatcliff [15] study an alternative presentation of the call-by-name CPS transformation by factoring it into a thunkification transformation that inserts `delay`s around all function arguments and `force`s all variables and a call-by-value CPS transformation extended to deal with `delay` and `force`. In addition, the paper also investigates an optimizing one-pass transformation but the details are different because our monadic brackets do not contain expressions but monadic statements.

Ager et al. [2] employ another path for transforming monadic code to CPS, which is a key step in their work to derive an abstract machine from a monadic evaluator. The authors first replace the monadic operations in the interpreter with their functional definitions. Then they transform the resulting monad-free evaluator to CPS using a standard call-by-value CPS transformation. It turns out that our thunkification transformation can be seen as expansion of the monadic operations. In fact, the transformation maps the monad type $(\mathrm{T}\,\tau)^\natural$ to $() \to \tau^\natural$ with the obvious return and bind operations. However, as we have demonstrated in Section 4, the combined transformation misses opportunities for optimization that our one-pass transformation exploits. One way to obtain a better transformation via thunkification might be to apply Millikin's idea of using shortcut deforestation with a normalization pass to create a one-pass transformation [20], but we have not yet explored this idea further.

Sabry and Felleisen [32] describe their source calculus via an axiom set which extends the call-by-value lambda calculus. Using an compactifying CPS transformation they present an inverse mapping which yields equational correspondence

of terms in source and target calculi of Fischer-style call-by-value CPS transformations. Sabry and Wadler [33] show that Plotkin's CPS transformation is a reflection on Moggi's computational lambda calculus. Barthe et al. [4] propose the weaker notion of reduction correspondence for reasoning about translations. An initial investigation shows some promise for embedding our CPS transformation into this framework.

On the practical side, Appel's book [3] presents all the machinery necessary for compiling with continuations and applies it to the full ML language. The main impact for compilation is that CPS names each intermediate value, sequentializes all computations, and yields an evaluation-order independent intermediate representation that is closed under β reduction. The latter is important as it simplifies the optimization phase of the compiler: It can perform unrestricted β reduction wherever that is desirable. Steele [36] was the first to exploit this insight in his Rabbit compiler for Scheme, Kelsey and others [18,16] later extended the techniques to work with procedural languages in general. Unlike some of his precursors, Appel uses a one-pass CPS transformation which reduces some administrative reductions. He relies on another optimizing pass for eliminating η reductions. An optimizing transformation, like ours, avoids this burden and leads to more efficient compilation.

Another point in favor of CPS-based compilation is the ease with which control operators can be supported in the source language. Friedman et al. [13] make a compelling point of this fact. This may be important in the further development of our Rinso language as control operators are well suited to implement cooperative concurrency.

7 Conclusion

There is evidence that a call-by-value language with an explicit monad is a design option for certain applications. Working towards compilation of such a language, we have developed an optimizing one-pass CPS transformation for this language and proven simulation and indifference for it. We present a direct style transformation for the CPS terms. We have demonstrated that our CPS transformation is preferable to an indirect one via thunkification. Finally, the transformation is compatible with simple typing.

References

1. Abadi, M. (ed.): Proc. 32nd ACM Symp. POPL, Long Beach, CA, USA, January 2005. ACM Press, New York (2005)
2. Ager, M.S., Danvy, O., Midtgaard, J.: A functional correspondence between monadic evaluators and abstract machines for languages with computational effects. Theoretical Computer Science 342(1), 149–172 (2005)
3. Appel, A.W.: Compiling with Continuations. Cambridge University Press, Cambridge (1992)

4. Barthe, G., Hatcliff, J., Sørensen, M.H.: Reflections on reflections. In: Hartel, P.H., Kuchen, H. (eds.) PLILP 1997. LNCS, vol. 1292, pp. 241–258. Springer, Heidelberg (1997)
5. Benton, N., Kennedy, A., Russell, G.: Compiling Standard ML to Java bytecodes. In: Hudak, P. (ed.) Proc. ICFP 1998, Baltimore, MD, USA. ACM Press, New York (1998)
6. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web programming without tiers. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2006. LNCS, vol. 4709, pp. 266–296. Springer, Heidelberg (2007)
7. Danvy, O.: Back to direct style. Science of Computer Programming 22, 183–195 (1994)
8. Danvy, O., Filinski, A.: Representing control: A study of the CPS transformation. Mathematical Structures in Computer Science 2, 361–391 (1992)
9. Danvy, O., Hatcliff, J.: CPS transformation after strictness analysis. Letters on Programming Languages and Systems 1(3), 195–212 (1993)
10. Filinski, A.: Representing layered monads. In: Aiken, A. (ed.) Proc. 26th ACM Symp. POPL, San Antonio, Texas, USA, pp. 175–188. ACM Press, New York (1999)
11. Flanagan, C., Sabry, A., Duba, B.F., Felleisen, M.: The essence of compiling with continuations. In: Proc. 1993 PLDI, Albuquerque, NM, USA, pp. 237–247 (June 1993)
12. Fluet, M., Morrisett, G.: Monadic regions. J. Funct. Program. 16(4-5), 485–545 (2006)
13. Friedman, D.P., Wand, M.: Essentials of Programming Languages, 3rd edn. MIT Press and McGraw-Hill (2008)
14. Hatcliff, J., Danvy, O.: A generic account of continuation-passing styles. In: Proc. 1994 ACM Symp. POPL, Portland, OR, USA, pp. 458–471. ACM Press, New York (1994)
15. Hatcliff, J., Danvy, O.: Thunks and the λ-calculus. J. Funct. Program. 7(3), 303–319 (1997)
16. Kelsey, R., Hudak, P.: Realistic compilation by program transformation. In: Proc. 16th ACM Symp. POPL, Austin, Texas, pp. 281–292. ACM Press, New York (1989)
17. Kennedy, A.: Compiling with continuations, continued. In: Ramsey, N. (ed.) Proc. ICFP 2007, Freiburg, Germany, pp. 177–190. ACM Press, New York (2007)
18. Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., Adams, N.: ORBIT: An optimizing compiler for Scheme. SIGPLAN Notices 21(7), 219–233 (1986); Proc. Sigplan 1986 Symp. on Compiler Construction
19. Matthews, J., Findler, R.B., Graunke, P., Krishnamurthi, S., Felleisen, M.: Automatically restructuring programs for the web. Automated Software Engineering 11(4), 337–364 (2004)
20. Millikin, K.: A new approach to one-pass transformations. In: van Eekelen, M. (ed.) Trends in Functional Programming, September 2007, vol. 6 (2007), `intellectbooks.co.uk`
21. Moggi, E.: Notions of computations and monads. Information and Computation 93, 55–92 (1991)
22. Neubauer, M., Thiemann, P.: From sequential programs to multi-tier applications by program transformation. In: Abadi [1], pp. 221–232
23. Nielson, F., Nielson, H.R.: Two-Level Functional Languages. Cambridge Tracts in Theoretical Computer Science, vol. 34. Cambridge University Press, Cambridge (1992)

24. Park, S., Pfenning, F., Thrun, S.: A probabilistic language based upon sampling functions. In Abadi [1], pp. 171–182
25. Peyton Jones, S. (ed.): Haskell 98 Language and Libraries, The Revised Report. Cambridge University Press, Cambridge (2003)
26. Peyton Jones, S., Gordon, A., Finne, S.: Concurrent Haskell. In: Proc. 1996 ACM Symp. POPL, St. Petersburg, FL, USA, pp. 295–308. ACM Press, New York (1996)
27. Peyton Jones, S., Reid, A., Hoare, T., Marlow, S., Henderson, F.: A semantics for imprecise exceptions. In: Proc. 1999 PLDI, Atlanta, Georgia, USA (May 1999); volume 34(5) of SIGPLAN Notices
28. Peyton Jones, S.L.: Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In: Hoare, T., Broy, M., Steinbruggen, R. (eds.) Engineering Theories of Software Construction, pp. 47–96. IOS Press, Amsterdam (2001)
29. Peyton Jones, S.L., Wadler, P.L.: Imperative functional programming. In: Proc. 1993 ACM Symp. POPL, Charleston, South Carolina, pp. 71–84. ACM Press, New York (1993)
30. Plotkin, G.: Call-by-name, call-by-value and the λ-calculus. Theoretical Computer Science 1, 125–159 (1975)
31. Ramsey, N., Pfeffer, A.: Stochastic lambda calculus and monads of probability distributions. In: Mitchell, J. (ed.) Proc. 29th ACM Symp. POPL, Portland, OR, USA. ACM Press, New York (2002)
32. Sabry, A., Felleisen, M.: Reasoning about programs in continuation-passing style. Lisp and Symbolic Computation 6(3/4), 289–360 (1993)
33. Sabry, A., Wadler, P.: A reflection on call-by-value. ACM Trans. Prog. Lang. and Systems 19(6), 916–941 (1997)
34. Saint-Mleux, X., Feeley, M., David, J.-P.: SHard: A Scheme to hardware compiler. In: Proc. 2006 Scheme and Functional Programming Workshop, pp. 39–49. Univ. of Chicago Press (2006)
35. Serrano, M., Gallesio, E., Loitsch, F.: HOP, a language for programming the Web 2.0. In: Proceedings of the First Dynamic Languages Symposium, Portland, OR, USA (October 2006)
36. Steele, G.L.: Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT, Cambridge, MA (1978)
37. Syme, D., Granicz, A., Cisternino, A.: Expert F#. Apress (2007)
38. Thiemann, P.: An embedded domain-specific language for type-safe server-side Web-scripting. ACM Trans. Internet Technology 5(1), 1–46 (2005)
39. Wadler, P., Thiemann, P.: The marriage of monads and effects. ACM Trans. Computational Logic 4(1), 1–32 (2003)
40. Wadler, P.L.: Comprehending monads. In: Proc. ACM Conference on Lisp and Functional Programming, Nice, France, pp. 61–78. ACM Press, New York (1990)
41. Wong, L.: Kleisli, a functional query system. J. Funct. Program. 10(1), 19–56 (2000)