

A Generic Account of Continuation-Passing Styles *

John Hatcliff

Department of Computing and Information Sciences
Kansas State University †
hatcliff@cis.ksu.edu

Olivier Danvy

Department of Computer Science
Aarhus University ‡
danvy@daimi.aau.dk

Abstract

We unify previous work on the continuation-passing style (CPS) transformations in a generic framework based on Moggi's computational meta-language. This framework is used to obtain CPS transformations for a variety of evaluation strategies and to characterize the corresponding administrative reductions and inverse transformations. We establish generic formal connections between operational semantics and equational theories. Formal properties of transformations for specific evaluation orders follow as corollaries.

Essentially, we factor transformations through Moggi's computational meta-language. Mapping λ -terms into the meta-language captures computational properties (*e.g.*, partiality, strictness) and evaluation order explicitly in both the term and the type structure of the meta-language. The CPS transformation is then obtained by applying a generic transformation from terms and types in the meta-language to CPS terms and types, based on a typed term representation of the continuation monad. We prove an adequacy property for the generic transformation and establish an equational correspondence between the meta-language and CPS terms.

These generic results generalize Plotkin's seminal theorems, subsume more recent results, and enable new uses of CPS transformations and their inverses. We discuss how to apply these results to compilation.

1 Introduction

There is a variety of continuation-passing styles — one for each evaluation order (call-by-name, *etc.*) and for each sequencing order (left-to-right, *etc.*). In each style, continuations get passed from function to function — resulting in a strikingly similar structure for all styles. However, in the literature, the formal properties of each style are established independently. For example, in his seminal paper *Call-by-name, call-by-value, and the λ -calculus* [32], Plotkin first presents call-by-value continuation-passing style (CPS) along with a set of correctness proofs and then he presents call-by-name CPS along with another set of correctness proofs. Both styles have similar structure but they are not identical. Their correctness proofs are also structurally similar but they are not identical. We propose to exploit these

* This work was partly supported by NSF under grant CCR-9102625.

† Manhattan, Kansas 66506, USA.

‡ Ny Munkegade, 8000 Aarhus C, Denmark. — This work was initiated at Kansas State University, continued at Carnegie Mellon University in spring 1993, and completed at Aarhus University.

Revised version, December 1, 1993. This version contains minor corrections to the version appearing in the Twenty-First Annual ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages, January 1994, Portland, Oregon.

similarities to factor the CPS transformations and their correctness proofs.

It appears that many CPS transformations are built from common building blocks. We represent these building blocks abstractly by constructs of Moggi's computational meta-language (which we refer to as Λ_{ml}) [26].¹ By formally connecting the language of abstract building blocks and the language of CPS terms, we obtain a generic framework for constructing CPS transformations and for reasoning about CPS terms — as opposed to dealing with each transformation individually. To connect the (operational) semantics of the two languages, we show an adequacy property for a generic transformation \mathcal{C} from Λ_{ml} to CPS terms. To connect equational theories, we show that the transformation \mathcal{C} (continuation introduction), along with its inverse \mathcal{C}^{-1} (continuation elimination), establishes an equational correspondence between Λ_{ml} and CPS terms. The diagram of Figure 1 summarizes the situation.

The result is that, given a correct encoding into Λ_{ml} , the construction and correctness of the corresponding CPS transformation follow as corollaries. Establishing a correct encoding into Λ_{ml} is much simpler than working directly with CPS terms.

This approach generalizes Plotkin's construction and correctness proofs for his call-by-value and call-by-name CPS transformations [32]. It also generalizes similar results for Reynolds's call-by-value CPS transformation [34], and more recently for CPS transformations capturing mixed evaluation orders based on strictness and totality information [3, 8, 9].

Practical use of CPS transformations requires one to characterize “administrative reductions” [7, 32, 36]. Again, administrative reductions are usually characterized for each CPS transformation individually. For example, Plotkin gives a “colon-translation” that performs administrative reductions for call-by-value CPS and then another colon-translation that performs administrative reductions for call-by-name CPS. In contrast, a certain subset of Λ_{ml} reductions generically characterizes the administrative reductions on CPS terms.

For each CPS transformation, a corresponding direct-style (DS) transformation exists that maps CPS terms and types to direct-style terms and types. DS transformations have stirred interest recently [6, 10, 13, 36]. However, just like the CPS transformations, they have been studied individually. In contrast, the continuation elimination \mathcal{C}^{-1} serves as the core of a generic DS transformation.

Finally, in situations where explicit continuations are not needed (*e.g.*, for compiling programs without jumps), Λ_{ml} stands as an alternative language to CPS — very close to CPS but without continuations.

¹Note that Moggi's computational meta-language [26] is a different language than Moggi's computational λ -calculus λ_c [23, 24].

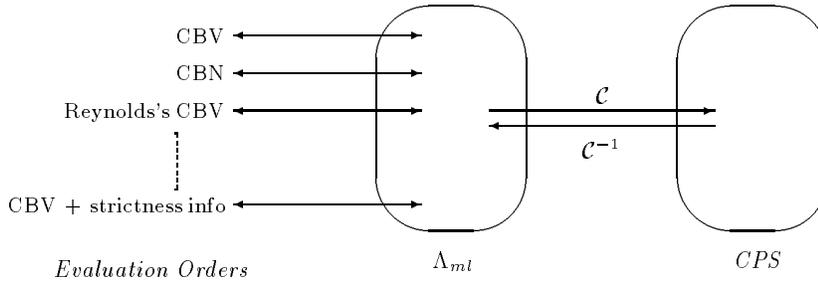


Figure 1: Factoring transformations through the computational meta-language

In Moggi’s work, Λ_{ml} (without recursive functions) is given a categorical semantics. The interpretation can be parameterized with categorical structures called *monads* that abstractly capture various notions of computation. In essence, we point out that by giving a *term representation* of the *continuation monad*, Λ_{ml} forms the basis of an elegant framework capturing the construction, correctness, equational properties, and optimizations of CPS and DS transformations for a variety of evaluation orders. Because the meta-language is typed, our development also gives a generic account of the typing of CPS transformations. Section 9 relates the present work to other recent applications of Moggi’s framework [36, 41, 42].

The rest of the paper is organized as follows. Section 2 addresses the representation of computational properties of λ -terms with Λ_{ml} terms. We consider in detail the standard call-by-name and call-by-value reduction strategies. Section 3 describes the mappings between Λ_{ml} terms and CPS terms. Section 4 illustrates the framework with a variety of CPS transformations. Section 5 formalizes administrative reductions. Section 6 addresses DS transformations. Section 7 describes how data structures are dealt within the framework. Section 8 applies the framework to compilation. Section 9 addresses related work. Section 10 concludes.

2 Representing Computational Properties of λ -terms

2.1 The language Λ of typed λ -terms

2.1.1 Syntax and notation

Figure 2 presents the syntax of the language Λ — simply-typed λ -terms with recursive functions.² Throughout the paper, type annotations are omitted when no ambiguity results. In such cases, the terms are assumed to be type correct. The same meta-variables (Γ for type assumptions, τ for types, e for terms) are used in different languages. Ambiguity is avoided by giving a different subscript for the typing judgement symbol \vdash in each language. We use the meta-variable f to distinguish identifiers of recursive functions. For other identifiers (and where no distinction is necessary) we use the meta-variable x . To simplify substitution, we follow Barendregt’s variable convention³ and consider the quotient of Λ under α -equivalence [2]. We write $e_1 \equiv e_2$ for α -equivalent terms e_1 and e_2 . The notation $FV(e)$ denotes

²We follow Tennent’s presentation of abstract syntax based on derivability of sequents of the form $\Gamma \vdash e : \tau$ [40].

³In terms occurring in definitions and proofs *etc.*, all bound variables are chosen to be different from free variables [2, page 26].

the set of free variables in e and $e_1[x := e_2]$ denotes the result of the capture-free substitution of all free occurrences of x in e_1 by e_2 . Closed terms of base type are called *programs*.

2.1.2 Values and computations

Certain terms of Λ are designated as *values*. Intuitively, values correspond to terms that are irreducible according to the operational semantics for Λ given below. The sets $Values_n[\Lambda]$ and $Values_v[\Lambda]$ represent the set of values from the language Λ under call-by-name and call-by-value reduction respectively.

$$\begin{aligned} Values_n[\Lambda] &::= c \mid \lambda x. e \mid rec f(x). e \mid f \\ Values_v[\Lambda] &::= c \mid \lambda x. e \mid rec f(x). e \mid x \end{aligned}$$

Note that all identifiers are included in $Values_v[\Lambda]$ since only values are substituted for identifiers under call-by-value reduction. Only identifiers f for recursive functions are included in $Values_n[\Lambda]$ (based on the reductions for *rec* in the following section), since arbitrary terms are substituted for identifiers in general. We use v to represent values and, where no ambiguity results, we ignore the distinction between call-by-name and call-by-value values.

Computations are non-values — terms requiring additional computational steps before their meaning can be determined. Not all computations reduce to values due to the possibility of non-termination introduced by recursive functions.

2.1.3 Operational semantics and equational reasoning

The rules of Figure 3 define single-step reduction functions over Λ programs. The meaning functions $eval_n$ and $eval_v$ are defined in terms of the reflexive, transitive closure (denoted \mapsto^*) of the single-step reduction functions.

$$\begin{aligned} eval_n(e) = v &\text{ iff } e \mapsto_n^* v \\ eval_v(e) = v &\text{ iff } e \mapsto_v^* v \end{aligned}$$

For reasoning about the meaning of Λ programs, we consider calculi generated by the usual reductions β , β_v , η_v [2] as well as the reductions *rec* and *rec_v* defined below.

$$\begin{aligned} (rec f(x). e_0) e_1 &\longrightarrow_{rec} e_0[f := rec f(x). e_0, x := e_1] \\ (rec f(x). e) v &\longrightarrow_{rec_v} e[f := rec f(x). e, x := v] \end{aligned}$$

For a notion of reduction r , \longrightarrow_r also denotes compatible one-step reduction, \longrightarrow_r^* is the reflexive, transitive closure of

$$\begin{array}{c}
\Gamma \vdash c : \iota \quad \Gamma \vdash x : \Gamma(x) \quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x. e : \tau_1 \rightarrow \tau_2} \quad \frac{\Gamma, f : \tau_1 \rightarrow \tau_2, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \text{rec } f(x). e : \tau_1 \rightarrow \tau_2} \quad \frac{\Gamma \vdash e_0 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash e_0 e_1 : \tau_2} \\
\tau ::= \iota \mid \tau_1 \rightarrow \tau_2 \\
\Gamma ::= \cdot \mid \Gamma, x : \tau
\end{array}$$

Figure 2: Abstract syntax of Λ , the language of simply-typed λ -terms

Call-by-name:

$$\begin{array}{c}
(\lambda x. e_0) e_1 \mapsto_n e_0[x := e_1] \quad (\text{rec } f(x). e_0) e_1 \mapsto_n e_0[f := \text{rec } f(x). e_0, x := e_1] \\
\frac{e_0 \mapsto_n e'_0}{e_0 e_1 \mapsto_n e'_0 e_1}
\end{array}$$

Call-by-value:

$$\begin{array}{c}
(\lambda x. e) v \mapsto_v e[x := v] \quad (\text{rec } f(x). e) v \mapsto_v e[f := \text{rec } f(x). e, x := v] \\
\frac{e_0 \mapsto_v e'_0}{e_0 e_1 \mapsto_v e'_0 e_1} \quad \frac{e_1 \mapsto_v e'_1}{(\lambda x. e_0) e_1 \mapsto_v (\lambda x. e_0) e'_1} \\
\frac{e_1 \mapsto_v e'_1}{(\text{rec } f(x). e_0) e_1 \mapsto_v (\text{rec } f(x). e_0) e'_1}
\end{array}$$

Figure 3: Single-step reductions for Λ programs

\rightarrow_r , and $=_r$ is the smallest equivalence relation generated by \rightarrow_r [2]. The calculus generated by $R_n = \{\beta, \eta_v, \text{rec}\}$ ⁴ (i.e., the relation $=_{R_n}$) is sound for reasoning about programs under call-by-name evaluation.⁵ The calculus generated by $R_v = \{\beta_v, \eta_v, \text{rec}_v\}$ is sound for reasoning about programs under call-by-value evaluation. When a property P holds for both β and β_v , indifferently, we say that P holds for β_i (similarly for rec_i , R_i , and eval_i).

2.2 The computational meta-language Λ_{ml}

2.2.1 Syntax and notation

Figure 4 presents the syntax of the language Λ_{ml} based on Moggi’s *computational meta-language* [26]. The key feature of the language is that its typing system captures the distinction between computations and values (or, in Reynolds’s terminology, *serious terms* and *trivial terms* [33]) which has often been used to justify the structure of CPS programs intuitively [32, 33]. Types of the form ι and $\tau_1 \rightarrow \tau_2$ are called *value types*. Accordingly, the rules for constants and abstractions are among the introduction rules for value types. Types of the form $\tilde{\tau}$ are called *computation types*.

The *monadic constructs*⁶ are used to make the computational process explicit. Intuitively, $[e]$ simply returns the value of e while $\text{let } x \leftarrow e_1 \text{ in } e_2$ first evaluates e_1 and binds the result to x , and then evaluates e_2 .

⁴The reason for considering only η_v instead of η is explained in Section 3.

⁵Soundness of calculi is formalized via a standard presentation of *operational equivalence* [32] which we omit here and in similar situations throughout the paper.

⁶For the explicit connection to the structure of a monad see [26, page 61].

2.2.2 Operational semantics and equational reasoning

Figure 5 presents the set of reductions R_{ml} for the computational meta-language Λ_{ml} . Note that when a Λ_{ml} -abstraction parameter is of value type, β_{ml} corresponds to β_v — Λ_{ml} typing ensures that only values will be substituted for the parameter. When an Λ_{ml} abstraction parameter is of computation type, both values (coerced to trivial computations by $[\cdot]$) and computations may be substituted for the parameter. rec_{ml} can be viewed in a similar manner.

The *monadic reductions* $R_{mon} = \{\text{let.}\beta, \text{let.}\eta, \text{let.assoc}\}$ are used to structure computation. In fact, an important property of the computational meta-language is that the reductions R_{ml} can describe the evaluation patterns of both direct style and continuation-passing style Λ programs. We exploit this property when constructing correctness proofs for transformations factored through the meta-language.

To capture the property formally, Figure 6 gives two sets of single-step reduction rules which are used to define operational semantics for Λ_{ml} programs (closed terms of type $\tilde{\iota}$).⁷ In both sets of rules, the R_{ml} reductions β_{ml} , rec , and $\text{let.}\beta$ are used to express computation. Adding the let.assoc reduction gives an evaluation pattern reminiscent of continuation-passing style, where the next redex is lifted out of a context before it is contracted. Omitting the let.assoc reduction gives the characteristic “direct-style” evaluation pattern where the evaluator descends into a context to pick the next redex to contract [11].

The following property captures the fact that the “direct-style” and the “continuation-passing style” evaluation patterns give the same results for Λ_{ml} programs.

⁷Moggi gives a categorical semantics for the meta-language. However, an operational semantics is sufficient here. The rules of Figure 6 describe two *leftmost*, *outermost* reduction strategies over Λ_{ml} reductions.

$$\begin{array}{c}
\frac{\Gamma, x:\tau_1 \vdash_{ml} e : \tilde{\tau}_2}{\Gamma \vdash_{ml} \lambda x. e : \tau_1 \rightarrow \tilde{\tau}_2} \quad \frac{\Gamma \vdash_{ml} c : \iota \quad \Gamma \vdash_{ml} x : \Gamma(x)}{\Gamma, f:\tau_1 \rightarrow \tilde{\tau}_2, x:\tau_1 \vdash_{ml} e : \tilde{\tau}_2} \quad \frac{\Gamma \vdash_{ml} e_0 : \tau_1 \rightarrow \tilde{\tau}_2 \quad \Gamma \vdash_{ml} e_1 : \tau_1}{\Gamma \vdash_{ml} e_0 e_1 : \tilde{\tau}_2} \\
\text{(monadic constructs)} \quad \frac{\Gamma \vdash_{ml} e : \tau}{\Gamma \vdash_{ml} [e] : \tilde{\tau}} \quad \frac{\Gamma \vdash_{ml} e_1 : \tilde{\tau}_1 \quad \Gamma, x:\tau_1 \vdash_{ml} e_2 : \tilde{\tau}_2}{\Gamma \vdash_{ml} \text{let } x \Leftarrow e_1 \text{ in } e_2 : \tilde{\tau}_2} \\
\tau ::= \iota \mid \tau_1 \rightarrow \tilde{\tau}_2 \mid \tilde{\tau} \\
\Gamma ::= \cdot \mid \Gamma, x:\tau
\end{array}$$

Figure 4: Abstract syntax of Λ_{ml} , the computational meta-language

$$\begin{array}{l}
(\lambda x. e_0) e_1 \longrightarrow_{\beta_{ml}} e_0[x := e_1] \\
(\text{rec } f(x). e_0) e_1 \longrightarrow_{\text{rec}_{ml}} e_0[f := \text{rec } f(x). e_0, x := e_1] \\
\text{let } x \Leftarrow [e_1] \text{ in } e_2 \longrightarrow_{\text{let}, \beta} e_2[x := e_1] \\
\text{let } x \Leftarrow e \text{ in } [x] \longrightarrow_{\text{let}, \eta} e \\
\text{let } x_2 \Leftarrow (\text{let } x_1 \Leftarrow e_1 \text{ in } e_2) \text{ in } e_3 \longrightarrow_{\text{let}, \text{assoc}} \text{let } x_1 \Leftarrow e_1 \text{ in } (\text{let } x_2 \Leftarrow e_2 \text{ in } e_3) \quad x_1 \notin FV(e_3)
\end{array}$$

Figure 5: The set of reductions R_{ml} for Λ_{ml}

“Direct-Style” Evaluation Pattern:

$$\begin{array}{l}
(\lambda x. e_0) e_1 \mapsto_{ml.D} e_0[x := e_1] \quad (\text{rec } f(x). e_0) e_1 \mapsto_{ml.D} e_0[f := \text{rec } f(x). e_0, x := e_1] \\
\text{let } x \Leftarrow [e_1] \text{ in } e_2 \mapsto_{ml.D} e_2[x := e_1] \quad \frac{e_1 \mapsto_{ml.D} e_1'}{\text{let } x \Leftarrow e_1 \text{ in } e_2 \mapsto_{ml.D} \text{let } x \Leftarrow e_1' \text{ in } e_2}
\end{array}$$

“Continuation-Passing Style” Evaluation Pattern:

$$\begin{array}{l}
(\lambda x. e_0) e_1 \mapsto_{ml.C} e_0[x := e_1] \quad (\text{rec } f(x). e_0) e_1 \mapsto_{ml.C} e_0[f := \text{rec } f(x). e_0, x := e_1] \\
\text{let } x \Leftarrow [e_1] \text{ in } e_2 \mapsto_{ml.C} e_2[x := e_1] \quad \frac{e_1 \mapsto_{ml.C} e_1'}{\text{let } x \Leftarrow e_1 \text{ in } e_2 \mapsto_{ml.C} \text{let } x \Leftarrow e_1' \text{ in } e_2} \text{ where } e_1 \not\equiv \text{let } y \Leftarrow e_a \text{ in } e_b. \\
\text{let } x_2 \Leftarrow (\text{let } x_1 \Leftarrow e_1 \text{ in } e_2) \text{ in } e_3 \mapsto_{ml.C} \text{let } x_1 \Leftarrow e_1 \text{ in } (\text{let } x_2 \Leftarrow e_2 \text{ in } e_3)
\end{array}$$

Figure 6: Single-step reductions for Λ_{ml} programs

$$\begin{array}{l}
\mathcal{E}_n\langle[\cdot]\rangle : \Lambda \rightarrow \Lambda_{ml} \quad \mathcal{E}_n\langle\cdot\rangle : \text{Values}_n[\Lambda] \rightarrow \Lambda_{ml} \\
\mathcal{E}_n\langle v \rangle = [\mathcal{E}_n\langle v \rangle] \quad \dots \text{where } v \in \text{Values}_n[\Lambda]. \quad \mathcal{E}_n\langle c \rangle = c \\
\mathcal{E}_n\langle [e_0 e_1] \rangle = \text{let } x_0 \Leftarrow \mathcal{E}_n\langle [e_0] \rangle \quad \mathcal{E}_n\langle \lambda x. e \rangle = \lambda x. \mathcal{E}_n\langle [e] \rangle \\
\quad \text{in } x_0 \mathcal{E}_n\langle [e_1] \rangle \quad \mathcal{E}_n\langle \text{rec } f(x). e \rangle = \text{rec } f(x). \mathcal{E}_n\langle [e] \rangle \\
\mathcal{E}_n\langle [x] \rangle = x \quad \mathcal{E}_n\langle f \rangle = f \\
\mathcal{E}_n\langle [\tau] \rangle = \widetilde{\mathcal{E}_n\langle \tau \rangle} \quad \mathcal{E}_n\langle [\Gamma, x:\tau] \rangle = \mathcal{E}_n\langle [\Gamma], x:\mathcal{E}_n\langle [\tau] \rangle \rangle \\
\mathcal{E}_n\langle \iota \rangle = \iota \quad \mathcal{E}_n\langle [\Gamma, f:\tau] \rangle = \mathcal{E}_n\langle [\Gamma], f:\mathcal{E}_n\langle \tau \rangle \rangle \\
\mathcal{E}_n\langle \tau_1 \rightarrow \tau_2 \rangle = \mathcal{E}_n\langle [\tau_1] \rangle \rightarrow \mathcal{E}_n\langle [\tau_2] \rangle
\end{array}$$

Figure 7: Call-by-name encoding into Λ_{ml}

$$\begin{array}{ll}
\mathcal{E}_v \langle \cdot \rangle : \Lambda \rightarrow \Lambda_{ml} & \mathcal{E}_v \langle \cdot \rangle : \text{Values}_v[\Lambda] \rightarrow \Lambda_{ml} \\
\mathcal{E}_v \langle v \rangle = [\mathcal{E}_v \langle v \rangle] \quad \dots \text{where } v \in \text{Values}_v[\Lambda]. & \mathcal{E}_v \langle c \rangle = c \\
\mathcal{E}_v \langle [e_0 e_1] \rangle = \text{let } x_0 \Leftarrow \mathcal{E}_v \langle [e_0] \rangle & \mathcal{E}_v \langle \lambda x . e \rangle = \lambda x . \mathcal{E}_v \langle [e] \rangle \\
\quad \text{in let } x_1 \Leftarrow \mathcal{E}_v \langle [e_1] \rangle & \mathcal{E}_v \langle \text{rec } f(x) . e \rangle = \text{rec } f(x) . \mathcal{E}_v \langle [e] \rangle \\
\quad \text{in } x_0 x_1 & \mathcal{E}_v \langle x \rangle = x \\
\\
\mathcal{E}_v \langle [\tau] \rangle = \widetilde{\mathcal{E}_v \langle \tau \rangle} & \\
\mathcal{E}_v \langle \iota \rangle = \iota & \mathcal{E}_v \langle [\Gamma, x : \tau] \rangle = \mathcal{E}_v \langle [\Gamma] \rangle, x : \mathcal{E}_v \langle \tau \rangle \\
\mathcal{E}_v \langle \tau_1 \rightarrow \tau_2 \rangle = \mathcal{E}_v \langle \tau_1 \rangle \rightarrow \mathcal{E}_v \langle \tau_2 \rangle &
\end{array}$$

Figure 8: Call-by-value encoding in Λ_{ml}

Property 1 For $\cdot \vdash_{ml} e : \tilde{\iota}$,

$$e \mapsto_{ml.D}^* [v] \quad \text{iff} \quad e \mapsto_{ml.C}^* [v]$$

Proof: One method relies on what can be thought of as a generalized version of Plotkin’s colon translation [32] which unrolls reductions in the continuation-passing style pattern until a reduction corresponding to a direct-style reduction is exposed. ■

A (partial) meaning function $eval_{ml}$ for Λ_{ml} programs $\cdot \vdash_{ml} e : \tilde{\iota}$ can be defined as follows.

$$eval_{ml}(e) = v \quad \text{iff} \quad e \mapsto_{ml.D}^* [v] \quad \text{iff} \quad e \mapsto_{ml.C}^* [v]$$

It is straightforward to show that the calculus R_{ml} is sound for reasoning about Λ_{ml} programs.

2.3 Encoding evaluation orders of Λ in Λ_{ml}

Figures 7 and 8 present the Λ_{ml} encodings of the standard call-by-name and call-by-value strategies. In these particular transformations, the double brackets $\langle \cdot \rangle$ are used when building computations and computation types. Single brackets $\langle \cdot \rangle$ are used when building values and value types. Elsewhere, where a distinction between computations and values is unimportant to the structure of the transformation, we use $\langle \cdot \rangle$ by default.

The encodings \mathcal{E}_n and \mathcal{E}_v preserve typing, as captured in the following property.

Property 2

- If $\Gamma \vdash e : \tau$ then $\mathcal{E}_n \langle [\Gamma] \rangle \vdash_{ml} \mathcal{E}_n \langle [e] \rangle : \mathcal{E}_n \langle \tau \rangle$.
- If $\Gamma \vdash e : \tau$ then $\mathcal{E}_v \langle [\Gamma] \rangle \vdash_{ml} \mathcal{E}_v \langle [e] \rangle : \mathcal{E}_v \langle \tau \rangle$.

The two encodings differ in that call-by-name functions receive computations as arguments (hence the typing $\mathcal{E}_n \langle \tau_1 \rangle \rightarrow \mathcal{E}_n \langle \tau_2 \rangle$) while call-by-value functions receive values as arguments (hence the typing $\mathcal{E}_v \langle \tau_1 \rangle \rightarrow \mathcal{E}_v \langle \tau_2 \rangle$). The encodings also capture the distinction between identifiers as computations for call-by-name and identifiers as values for call-by-value, as pointed out in Section 2.1.2. Correctness is captured as follows.

Property 3 For all programs $\cdot \vdash e : \iota$,

- $eval_n(e) = v$ iff $eval_{ml}(\mathcal{E}_n \langle [e] \rangle) = v$
- $eval_v(e) = v$ iff $eval_{ml}(\mathcal{E}_v \langle [e] \rangle) = v$

Proof: The proof takes advantage of the fact that $\mapsto_{ml.D}$ reductions describe direct-style evaluation. For example, for call-by-name, the proof relies on the fact that $e \mapsto_n e'$ implies $\mathcal{E}_n \langle [e] \rangle \mapsto_{ml.D}^* \mathcal{E}_n \langle [e'] \rangle$. ■

2.4 Conclusion

As advocated by Moggi and as illustrated here with call-by-name and call-by-value, Λ_{ml} offers a framework for encoding the computational properties of Λ terms. Section 4 presents other practical evaluation orders. The following section formally connects Λ_{ml} terms and CPS terms.

3 Computations as Continuation-Passing Terms

3.1 Introducing continuations

Figure 9 presents the translation \mathcal{C} from Λ_{ml} to continuation-passing terms of Λ . \mathcal{C} relies on a term representation of the *monad of continuations* [26, page 58]. We use the monad of continuations because it naturally accounts for passing continuations. We use a term representation because we are aiming for a program transformation. The following property captures the fact that \mathcal{C} maps well-typed Λ_{ml} terms to well-typed CPS terms.

Property 4 If $\Gamma \vdash_{ml} e : \tau$, then $\mathcal{C} \langle [\Gamma] \rangle \vdash \mathcal{C} \langle [e] \rangle : \mathcal{C} \langle \tau \rangle$.

The translation on computation types $\tilde{\tau}$ shows that computations correspond to continuation-passing terms. We use the notation $\neg\tau$ to abbreviate $\tau \rightarrow ans$ where ans is a distinguished type of answers. Thus

$$\mathcal{C} \langle \tilde{\tau} \rangle = \neg\mathcal{C} \langle \tau \rangle = (\mathcal{C} \langle \tau \rangle \rightarrow ans) \rightarrow ans.$$

The translation on terms shows that the monadic constructors $[\cdot]$ and let correspond to the basic components of continuation-passing terms:

- $[e]$ abstracts the application of a continuation to the result $\mathcal{C} \langle [e] \rangle$, and
- $\text{let } x \Leftarrow e_1 \text{ in } e_2$ abstracts the composition of computations (continuation-passing terms) by forming the continuation $\lambda x . \mathcal{C} \langle [e_2] \rangle k$ and passing it to $\mathcal{C} \langle [e_1] \rangle$.

In the following sections, we consider an optimized transformation \mathcal{C}' producing terms without redexes of the form $(\lambda k . e) k$. Section 5 discusses administrative reductions in general.

A fundamental property of \mathcal{C} is that all λ -terms in its image are evaluation-order independent. Furthermore, \mathcal{C} preserves Λ_{ml} equational properties and operational semantics of Λ_{ml} .

$$\begin{aligned}
\mathcal{C}\langle c \rangle &= c \\
\mathcal{C}\langle x \rangle &= x \\
\mathcal{C}\langle \lambda x . e \rangle &= \lambda x . \mathcal{C}\langle e \rangle \\
\mathcal{C}\langle \text{rec } f(x) . e \rangle &= \text{rec } f(x) . \mathcal{C}\langle e \rangle \\
\mathcal{C}\langle e_0^{\tau_1 \rightarrow \tilde{\tau}_2} e_1 \rangle &= \lambda k^{\neg \mathcal{C}\langle \tau_2 \rangle} . (\mathcal{C}\langle e_0 \rangle \mathcal{C}\langle e_1 \rangle) k \\
\mathcal{C}\langle [e] \tilde{\tau} \rangle &= \lambda k^{\neg \mathcal{C}\langle \tau \rangle} . k \mathcal{C}\langle e \rangle \\
\mathcal{C}\langle \text{let } x \Leftarrow e_1^{\tilde{\tau}_1} \text{ in } e_2^{\tilde{\tau}_2} \rangle &= \lambda k^{\neg \mathcal{C}\langle \tau_2 \rangle} . \mathcal{C}\langle e_1 \rangle (\lambda x^{\mathcal{C}\langle \tau_1 \rangle} . \mathcal{C}\langle e_2 \rangle) k \\
\\
\mathcal{C}\langle \iota \rangle &= \iota \\
\mathcal{C}\langle \tau_1 \rightarrow \tilde{\tau}_2 \rangle &= \mathcal{C}\langle \tau_1 \rangle \rightarrow \neg \mathcal{C}\langle \tau_2 \rangle & \mathcal{C}\langle \Gamma, x : \tau \rangle &= \mathcal{C}\langle \Gamma \rangle, x : \mathcal{C}\langle \tau \rangle \\
\mathcal{C}\langle \tilde{\tau} \rangle &= \neg \mathcal{C}\langle \tau \rangle
\end{aligned}$$

Figure 9: Continuation introduction — translation from Λ_{ml} into CPS

To formalize these properties we establish an equational correspondence between the R_{ml} calculus of Λ_{ml} and CPS terms under the R_i calculi. This first requires defining a translation \mathcal{C}^{-1} from CPS terms to Λ_{ml} .

Figure 10 presents the language $\mathcal{C}\langle \Lambda_{ml} \rangle$ of CPS λ -terms closed under R_i reduction.⁸ Note that $\mathcal{C}\langle \Lambda_{ml} \rangle$ is a sublanguage of Λ . The judgement \vdash_{val} enforces the property that terms in the image of \mathcal{C} are values (this property is discussed in detail in Section 3.4). The judgements \vdash_{ans} and \vdash_{cont} rely on type assumptions that include a distinguished identifier $k \notin \Gamma$.

Figure 11 presents all possible R_n reductions on $\mathcal{C}\langle \Lambda_{ml} \rangle$ terms. It is easy to show that each reduction is also a R_v reduction. Also, reductions on CPS terms preserve syntactic categories, *e.g.*, reducing an expression satisfying the judgement \vdash_{val} yields an expression that still satisfies the judgement \vdash_{val} .

3.2 Eliminating continuations

Figure 12 presents the translation \mathcal{C}^{-1} from the language of CPS terms $\mathcal{C}\langle \Lambda_{ml} \rangle$ back to Λ_{ml} . A key component of \mathcal{C}^{-1} is the transformation of continuations to what we call *reduction contexts*.⁹ Reduction contexts $\Gamma \vdash_{ml} \varphi : \tilde{\tau}_2[\tilde{\tau}_1]$ of type $\tilde{\tau}_2$ with holes of type $\tilde{\tau}_1$ are described by the following syntax rules.

$$\begin{aligned}
\Gamma \vdash_{ml} [\cdot] : \tilde{\tau}[\tilde{\tau}] & \quad (\text{trivial contexts}) \\
\frac{\Gamma, x : \tau_1 \vdash_{ml} e : \tilde{\tau}_2}{\Gamma \vdash_{ml} \text{let } x \Leftarrow [\cdot] \text{ in } e : \tilde{\tau}_2[\tilde{\tau}_1]} & \quad (\text{let contexts})
\end{aligned}$$

The following property captures the fact that the transformation preserves well-typed terms.

Property 5

- If $\Gamma \vdash_{val} w : \tau$ then $\mathcal{C}^{-1}\langle \Gamma \rangle \vdash_{ml} \mathcal{C}_{val}^{-1}\langle w \rangle : \mathcal{C}^{-1}\langle \tau \rangle$.

⁸A formal statement of correctness is omitted for lack of space.

⁹Felleisen and Friedman first pointed out that continuations in CPS correspond to *evaluation contexts* in direct-style terms (*e.g.*, terms from the language Λ) [11]. When considering Λ_{ml} terms, continuations correspond to *reduction contexts*. Reduction contexts represent an intermediate step between evaluation contexts and continuations where, among other things, the term in the “hole” of a non-trivial evaluation context is given a name.

- If $\Gamma \vdash_{exp} w : \tau$ then $\mathcal{C}^{-1}\langle \Gamma \rangle \vdash_{ml} \mathcal{C}_{exp}^{-1}\langle w \rangle : \mathcal{C}^{-1}\langle \tau \rangle$.
- If $\langle \Gamma ; k : \neg \tau \rangle \vdash_{ans} \alpha : ans$ then $\mathcal{C}^{-1}\langle \Gamma \rangle \vdash_{ml} \mathcal{C}_{ans}^{-1}\langle \alpha \rangle : \mathcal{C}^{-1}\langle \tau \rangle$.
- If $\langle \Gamma ; k : \neg \tau_0 \rangle \vdash_{cont} \kappa : \neg \tau_1$ then $\mathcal{C}^{-1}\langle \Gamma \rangle \vdash_{ml} \mathcal{C}_{cont}^{-1}\langle \kappa \rangle : \mathcal{C}^{-1}\langle \tau_0 \rangle[\mathcal{C}^{-1}\langle \tau_1 \rangle]$.

3.3 Relating operational semantics and equational theories

We can now state an adequacy property for the translations \mathcal{C} and \mathcal{C}^{-1} . The following theorem recasts Plotkin’s Simulation and Indifference theorems for call-by-name and call-by-value CPS [32, Section 6] in terms of the generic introduction of continuations by \mathcal{C} .

Theorem 1 (Simulation and Indifference)

If $\cdot \vdash_{ml} e : \tilde{\iota}$ then

$$eval_{ml}(e) = v \quad \text{iff} \quad eval_h(\mathcal{C}\langle e \rangle (\lambda x^t . x)) = v$$

Proof: The proof takes advantage of the fact that $\mapsto_{ml.C}$ reductions describe “continuation-passing style” evaluation. ■

The corresponding property for \mathcal{C}^{-1} follows.

Theorem 2 If $\cdot \vdash_{val} w : \neg \iota$ then

$$eval_{ml}(\mathcal{C}_{val}^{-1}\langle w \rangle) = v \quad \text{iff} \quad eval_h(w (\lambda x^t . x)) = v.$$

Proof: For $eval_n$,

$$\begin{aligned}
eval_{ml}(\mathcal{C}_{val}^{-1}\langle w \rangle) &= eval_n((\mathcal{C} \circ \mathcal{C}^{-1})\langle w \rangle (\lambda x^t . x)) \\
&\dots \text{Theorem 1} \\
&= eval_n(w (\lambda x^t . x)) \\
&\dots \text{Theorem 3 \& soundness of } R_n
\end{aligned}$$

Similarly for $eval_v$. ■

Plotkin’s Translation theorems show how his call-by-name and call-by-value CPS transformations relate equational theories over direct-style terms and theories over CPS terms [32]. We relate the equational theories of the metalanguage and CPS terms by showing an *equational correspondence* between Λ_{ml} terms under the R_{ml} calculus and $\mathcal{C}\langle \Lambda_{ml} \rangle$ terms under the R_{cps} (*i.e.*, R_i) calculus. In essence, this means that the equivalence classes of each theory are in a one-to-one correspondence.

<i>Values</i>	$\frac{\Gamma \vdash_{exp} w : \tau}{\Gamma \vdash_{val} w : \tau}$ where w is a value	
<i>Expressions</i>	$\Gamma \vdash_{exp} c : \iota \quad \Gamma \vdash_{exp} x : \Gamma(x) \quad \frac{\Gamma, x : \tau_1 \vdash_{val} w : \neg\neg\tau_2}{\Gamma \vdash_{exp} \lambda x. w : \tau_1 \rightarrow \neg\neg\tau_2} \quad \frac{\Gamma, f : \tau_1 \rightarrow \neg\neg\tau_2, x : \tau_1 \vdash_{val} w : \neg\neg\tau_2}{\Gamma \vdash_{exp} rec f(x). w : \tau_1 \rightarrow \neg\neg\tau_2}$ $\frac{\Gamma \vdash_{val} w_0 : \tau_1 \rightarrow \neg\neg\tau_2 \quad \Gamma \vdash_{val} w_1 : \tau_1}{\Gamma \vdash_{exp} w_0 w_1 : \neg\neg\tau_2} \quad \frac{\langle \Gamma ; k : \neg\tau \rangle \vdash_{ans} \alpha : ans}{\Gamma \vdash_{exp} \lambda k. \alpha : \neg\neg\tau}$	
<i>Continuations</i>	$\langle \Gamma ; k : \neg\tau \rangle \vdash_{cont} k : \neg\tau \quad \frac{\langle \Gamma, x : \tau_1 ; k : \neg\tau_0 \rangle \vdash_{ans} \alpha : ans}{\langle \Gamma ; k : \neg\tau_0 \rangle \vdash_{cont} \lambda x. \alpha : \neg\tau_1}$	
<i>Answers</i>	$\frac{\Gamma \vdash_{exp} w : \neg\neg\tau_0 \quad \langle \Gamma ; k : \neg\tau_1 \rangle \vdash_{cont} \kappa : \neg\tau_0}{\langle \Gamma ; k : \neg\tau_1 \rangle \vdash_{ans} w \kappa : ans}$ $\frac{\langle \Gamma ; k : \neg\tau_0 \rangle \vdash_{cont} \kappa : \neg\tau_1 \quad \Gamma \vdash_{val} w : \tau_1}{\langle \Gamma ; k : \neg\tau_0 \rangle \vdash_{ans} \kappa w : ans}$	
<i>Types and Assumptions</i>	$\tau ::= \iota \mid \tau_1 \rightarrow \neg\neg\tau_2 \mid \neg\neg\tau$ $\Gamma ::= \cdot \mid \Gamma, x : \tau$	

Figure 10: Abstract syntax of $\mathcal{C}\langle\Lambda_{ml}\rangle$, the language of CPS terms

$\begin{aligned} (\lambda x. w_0) w_1 &\longrightarrow_{\beta_{exp}} w_0[x := w_1] \\ (rec f(x). w_0) w_1 &\longrightarrow_{rec_{exp}} w_0[f := rec f(x). w_0, x := w_1] \\ (\lambda x. \alpha) w &\longrightarrow_{\beta_{ans.1}} \alpha[x := w] \\ (\lambda k. \alpha) \kappa &\longrightarrow_{\beta_{ans.2}} \alpha[k := \kappa] \end{aligned}$ $\begin{aligned} \lambda k. w k &\longrightarrow_{\eta_{exp}} w \quad k \notin FV(w) \text{ where } w \text{ is a value} \\ \lambda x. \kappa x &\longrightarrow_{\eta_{cont}} \kappa \quad x \notin FV(\kappa) \text{ where } \kappa \text{ is a value} \end{aligned}$
Figure 11: The set of reductions R_{cps} for $\mathcal{C}\langle\Lambda_{ml}\rangle$

$\begin{aligned} \mathcal{C}_{val}^{-1}\langle w \rangle &= \mathcal{C}_{exp}^{-1}\langle w \rangle \\ \mathcal{C}_{ans}^{-1}\langle w \kappa \rangle &= \mathcal{C}_{cont}^{-1}\langle \kappa \rangle[\mathcal{C}_{exp}^{-1}\langle w \rangle] \\ \mathcal{C}_{ans}^{-1}\langle \kappa w \rangle &= \mathcal{C}_{cont}^{-1}\langle \kappa \rangle[\mathcal{C}_{val}^{-1}\langle w \rangle] \\ \mathcal{C}_{cont}^{-1}\langle k \rangle &= [\cdot] \\ \mathcal{C}_{cont}^{-1}\langle \lambda x. \alpha \rangle &= let x \Leftarrow [\cdot] in \mathcal{C}_{ans}^{-1}\langle \alpha \rangle \end{aligned}$	$\begin{aligned} \mathcal{C}_{exp}^{-1}\langle c \rangle &= c \\ \mathcal{C}_{exp}^{-1}\langle x \rangle &= x \\ \mathcal{C}_{exp}^{-1}\langle \lambda x. w \rangle &= \lambda x. \mathcal{C}_{val}^{-1}\langle w \rangle \\ \mathcal{C}_{exp}^{-1}\langle rec f(x). w \rangle &= rec f(x). \mathcal{C}_{val}^{-1}\langle w \rangle \\ \mathcal{C}_{exp}^{-1}\langle w_0 w_1 \rangle &= \mathcal{C}_{val}^{-1}\langle w_0 \rangle \mathcal{C}_{val}^{-1}\langle w_1 \rangle \\ \mathcal{C}_{exp}^{-1}\langle \lambda k. \alpha \rangle &= \mathcal{C}_{ans}^{-1}\langle \alpha \rangle \end{aligned}$
$\begin{aligned} \mathcal{C}^{-1}\langle \iota \rangle &= \iota \\ \mathcal{C}^{-1}\langle \tau_1 \rightarrow \tau_2 \rangle &= \mathcal{C}^{-1}\langle \tau_1 \rangle \rightarrow \mathcal{C}^{-1}\langle \tau_2 \rangle \\ \mathcal{C}^{-1}\langle \neg\neg\tau \rangle &= \mathcal{C}^{-1}\langle \tau \rangle \end{aligned}$	$\mathcal{C}^{-1}\langle \Gamma, x : \tau \rangle = \mathcal{C}^{-1}\langle \Gamma \rangle, x : \mathcal{C}^{-1}\langle \tau \rangle$
Figure 12: Continuation elimination — translation from CPS back to Λ_{ml}	

Theorem 3 (Equational Correspondence)

1. $e =_{R_{ml}} \mathcal{C}_{val}^{-1} \circ \mathcal{C}(\llbracket e \rrbracket)$
2. $w =_{R_{cps}} \mathcal{C} \circ \mathcal{C}_{val}^{-1}(\llbracket w \rrbracket)$
3. $e_1 =_{R_{ml}} e_2$ iff $\mathcal{C}(\llbracket e_1 \rrbracket) =_{R_{cps}} \mathcal{C}(\llbracket e_2 \rrbracket)$
4. $w_1 =_{R_{cps}} w_2$ iff $\mathcal{C}_{val}^{-1}(\llbracket w_1 \rrbracket) =_{R_{ml}} \mathcal{C}_{val}^{-1}(\llbracket w_2 \rrbracket)$

Proof: Follows the outline of [36, Theorem 16]. However, the proof here is simpler because our framework is typed and \mathcal{C} does not perform administrative reductions “on the fly”. (see also [2, Theorem 7.3.10] and [17, Theorem 4]). ■

3.4 Assessment

Evaluation-order independence for all terms in the image of \mathcal{C} holds because all $\mathcal{C}(\llbracket \Lambda_{ml} \rrbracket)$ function arguments are values. Specifically,

- if a Λ_{ml} argument e has a value type, then $\mathcal{C}(\llbracket e \rrbracket)$ is a value; and,
- if a Λ_{ml} argument e has a computation type, then $\mathcal{C}(\llbracket e \rrbracket)$ takes the form $\lambda k \dots$ (*i.e.*, a value).

Obtaining evaluation-order independence requires slightly more than simply instantiating the monadic constructs $[\]$ and let with the continuation monad (witness the η -redex in $\mathcal{C}(\llbracket e_0 e_1 \rrbracket)$ of Figure 9). Such η -redexes are important since they suspend call-by-value evaluation when terms corresponding to computations occur as function arguments, *e.g.*, in CPS terms encoding call-by-name. Let \mathcal{C}'' be a translation that only instantiates $[\]$ and let . Following this strategy gives $\mathcal{C}''(\llbracket e_0 e_1 \rrbracket) = \mathcal{C}''(\llbracket e_0 \rrbracket) \mathcal{C}''(\llbracket e_1 \rrbracket)$. Now, if $e_0 \equiv \lambda z. [c_0]$ and $e_1 \equiv (rec\ f(y). f\ y)\ c_1$, then

$$\mathcal{C}''(\llbracket e_0 e_1 \rrbracket) (\lambda x.x) = ((\lambda z.\lambda k.k\ c_0) ((rec\ f(y). f\ y)\ c_1)) (\lambda x.x)$$

which diverges under call-by-value but terminates under call-by-name, and thus is not evaluation-order independent.

The above example also illustrates why η is not sound for reasoning about $\mathcal{C}(\llbracket \Lambda_{ml} \rrbracket)$ terms under call-by-value evaluation. For example, $\mathcal{C}(\llbracket e_0 e_1 \rrbracket) (\lambda x.x)$ terminates under call-by-value but η -reduces to $\mathcal{C}''(\llbracket e_0 e_1 \rrbracket) (\lambda x.x)$ which has just been shown to diverge under call-by-value.¹⁰ In reality, problems are encountered only when one attempts to generalize η_{exp} redexes to η — all η redexes of the form given by η_{cont} are also η_v redexes.

3.5 Generalizing the notion of value

Suppose the types of Λ_{ml} are extended as follows.

$$\tau ::= \iota \mid \tau_1 \rightarrow \tau_2 \mid \tilde{\tau}$$

This typing generalizes the notion of value to include applications of functions that always terminate when applied. Such functions do not need to be passed continuations to achieve evaluation-order independence [9]. Theorem 1 and 2 hold for a language with this generalized type system. Λ_{ml} reductions in the generalized system induce a set of reductions R'_{cps} on CPS terms that are sound under call-by-name and call-by-value evaluation. However, the generalized calculus R'_{cps} no longer equationally corresponds to R_n or R_v

¹⁰Similar examples of η being unsound exist for “traditional” untyped call-by-name CPS terms under call-by-value evaluation.

due to the generalized notion of value, but it is a conservative extension of R_n and R_v .

Section 4.5 gives an application of this generalized notion of value. The reader is referred to [16] for a detailed discussion.

4 CPS Transformations from Encodings of Computational Properties

Previous applications of Λ_{ml} focus exclusively on call-by-value or call-by-name [26, 42]. In contrast, the present framework allows the description of many other useful CPS transformations. Further, the correctness of the corresponding CPS transformation, the characterization of administrative reductions (see Section 5), and a correct mapping from CPS back to Λ_{ml} (see Section 6) follow as corollaries from simply identifying the appropriate computational properties. We give several examples below.

For each evaluation order, the corresponding CPS transformation is constructed by composing the encoding \mathcal{E} (of the evaluation order into Λ_{ml}) with the continuation introduction \mathcal{C} (or preferably, the slightly optimized introduction \mathcal{C}' of Section 3 that produces terms without redexes of the form $(\lambda k . e) k$). In general, the correctness of the constructed CPS transformations follows from the correctness of an encoding \mathcal{E} and the correctness of \mathcal{C} (Theorem 1).

4.1 Call-by-name and call-by-value CPS transformations

We describe in detail the construction of correct call-by-name and call-by-value CPS transformations. As outlined above, CPS transformations are obtained by composing the encodings of specific evaluation orders with the generic continuation introduction \mathcal{C}' .

Definition 1 (Construction)

$$\begin{aligned} \mathcal{K}_n &\stackrel{\text{def}}{=} \mathcal{C}' \circ \mathcal{E}_n \\ \mathcal{K}_v &\stackrel{\text{def}}{=} \mathcal{C}' \circ \mathcal{E}_v \end{aligned}$$

The fact that the transformations preserve well-typed terms follow as corollaries.

Property 6 (Type correctness)

- If $\Gamma \vdash e : \tau$ then $\mathcal{K}_n(\llbracket \Gamma \rrbracket) \vdash \mathcal{K}_n(\llbracket e \rrbracket) : \mathcal{K}_n(\llbracket \tau \rrbracket)$.
- If $\Gamma \vdash e : \tau$ then $\mathcal{K}_v(\llbracket \Gamma \rrbracket) \vdash \mathcal{K}_v(\llbracket e \rrbracket) : \mathcal{K}_v(\llbracket \tau \rrbracket)$.

Proof: Follows from the type correctness of the encodings \mathcal{E}_n , \mathcal{E}_v (Property 2) and the type correctness of \mathcal{C} (Property 4). ■

The correctness of the transformations follows as a corollary.

Property 7 (Simulation and Indifference)

If $\cdot \vdash e : \iota$ then

$$\begin{aligned} eval_n(e) = v &\quad \text{iff} \quad eval_i(\mathcal{K}_n(\llbracket e \rrbracket) (\lambda x^t . x)) = v \\ eval_v(e) = v &\quad \text{iff} \quad eval_i(\mathcal{K}_v(\llbracket e \rrbracket) (\lambda x^t . x)) = v \end{aligned}$$

Proof: Follows from the correctness of the encodings \mathcal{E}_n , \mathcal{E}_v (Property 3) and the generic Simulation and Indifference theorem for \mathcal{C} (Theorem 1). ■

The transformations \mathcal{K}_n and \mathcal{K}_v constructed above are actually Plotkin’s CPS transformations. Let \mathcal{P}_n and \mathcal{P}_v respectively denote the typed version of Plotkin’s call-by-name and call-by-value CPS transformations [8, 15, 22].

Property 8 For $\Gamma \vdash e : \tau$,

$$\begin{aligned}\mathcal{P}_n\langle e \rangle &\equiv \mathcal{K}_n\langle e \rangle \\ \mathcal{P}_v\langle e \rangle &\equiv \mathcal{K}_v\langle e \rangle\end{aligned}$$

Proof: by structural induction over e . ■

Thus, the construction and correctness (specifically, the Simulation and Indifference theorems, and type correctness) of the typed versions of Plotkin’s CPS transformation follow from the correctness of the encodings. Relationships between equational theories for direct-style and CPS terms (similar to those established by Plotkin’s Translation theorems for \mathcal{P}_n and \mathcal{P}_v) follow by connecting equational theories over Λ with the theory R_{ml} of Λ_{ml} .

4.2 Reynolds’s call-by-value CPS transformation

We obtain a typed version of Reynolds’s call-by-value CPS transformation [34] by keeping the same encoding of variables and applications in Figure 7 (the call-by-name encoding into Λ_{ml}) but by replacing the encoding of abstractions with the following definition.

$$\mathcal{E}_R\langle \lambda x^\tau . e \rangle = \lambda t^{\mathcal{E}_R\langle \tau \rangle} . \text{let } x_1^{\mathcal{E}_R\langle \tau \rangle} \Leftarrow t \text{ in } (\lambda x^{\mathcal{E}_R\langle \tau \rangle} . \mathcal{E}_R\langle e \rangle) [x_1]$$

Instead of arguments being evaluated in the application (as in Figure 8, the call-by-value encoding into Λ_{ml}), they are passed as computations (essentially as *thunks* [17, 18]), evaluated immediately after the function is applied, and the resulting values are wrapped up again as thunks. Thus, \mathcal{E}_R captures the computational properties of call-by-value, but in a different style than \mathcal{E}_v . This corresponds to the definition of call-by-value in the Algol 60 report [30].

Turning to the CPS transformation,

$$\mathcal{R}_v \stackrel{\text{def}}{=} \mathcal{C}' \circ \mathcal{E}_R$$

Griffin, for example [15, Footnote 3], pointed out that the typing of the function space in \mathcal{R}_v matches the one of \mathcal{P}_n *i.e.*,

$$\mathcal{R}_v\langle \tau_1 \rightarrow \tau_2 \rangle = \mathcal{R}_v\langle \tau_1 \rangle \rightarrow \mathcal{R}_v\langle \tau_2 \rangle$$

This typing coincidence already holds here, before introducing continuations:

$$\mathcal{E}_R\langle \tau_1 \rightarrow \tau_2 \rangle = \mathcal{E}_R\langle \tau_1 \rangle \rightarrow \mathcal{E}_R\langle \tau_2 \rangle$$

and thus it is independent of continuations as such. In any case, this coincidence illustrates that the transformation over types does not always determine the transformation over terms.

4.3 Variation on Reynolds’s call-by-value CPS transformation

One may choose to pass arguments unevaluated and to force them after the function is applied, but not to wrap them into computations again. This is achieved by replacing the encoding of abstractions and applications in Figure 8 (the call-by-value encoding into Λ_{ml}) with the following definitions.

$$\begin{aligned}\mathcal{E}_R\langle \lambda x^\tau . e \rangle &= \lambda t^{\mathcal{E}_R\langle \tau \rangle} . \text{let } x^{\mathcal{E}_R\langle \tau \rangle} \Leftarrow t \text{ in } \mathcal{E}_R\langle e \rangle \\ \mathcal{E}_R\langle [e_0 e_1] \rangle &= \text{let } x_0 \Leftarrow \mathcal{E}_R\langle [e_0] \rangle \text{ in } x_0 \mathcal{E}_R\langle [e_1] \rangle\end{aligned}$$

Thus \mathcal{E}_R captures the computational properties of call-by-value, but in a different style than \mathcal{E}_v and \mathcal{E}_n . This corresponds to the style of capturing computational properties of call-by-value in denotational semantics [38]: (1) either using a strictness check in the applicative structure (corresponding to \mathcal{E}_v); (2) or forming strict functions to create a strict function space (corresponding to Reynolds’s transformation and its variant).

The typing of the function space in \mathcal{E}_R still matches the one of \mathcal{E}_n (see Figure 7) *i.e.*,

$$\mathcal{E}_R\langle \tau_1 \rightarrow \tau_2 \rangle = \mathcal{E}_R\langle \tau_1 \rangle \rightarrow \mathcal{E}_R\langle \tau_2 \rangle.$$

However, the transformation on type assumptions Γ is the same as for \mathcal{E}_v (Figure 8).

4.4 Mixed evaluation strategies based on strictness information

Compile-time analyses of computation properties (such as strictness analysis) indicate where it is safe to mix evaluation strategies [29]. Earlier works show how to derive the corresponding CPS transformation encoding the mixed evaluation strategy into CPS terms [3, 8, 28, 31]. Such transformations can be correctly constructed by an encoding that contains both call-by-value-like (capturing strictness) and call-by-name-like (capturing non-strictness) applications/functions/identifiers. The types of such an encoding are structured as follows.

$$\sigma ::= \iota \mid \sigma_1 \rightarrow \tilde{\sigma}_2 \mid \tilde{\sigma}_1 \rightarrow \tilde{\sigma}_2$$

These types illustrate that arguments to non-strict functions are computations while arguments to strict functions can be safely reduced to values before application. This encoding is based on combining the styles of \mathcal{E}_v and \mathcal{E}_n . However, a correct encoding based on strictness information can also be obtained by combining the styles of \mathcal{E}_R and \mathcal{E}_n or of \mathcal{E}_R and \mathcal{E}_v . Again, as long as the computational properties are correctly identified, the correct CPS transformation and accompanying tools follow.

4.5 Mixed evaluation strategies based on totality information

Similarly, totality information determines when computations are guaranteed to reduce to values. An encoding of terms with totality information obeys the following type structure (capturing the possibility of partiality/totality in the domain and codomain of function spaces).

$$\sigma ::= \iota \mid \sigma_1 \rightarrow \sigma_2 \mid \sigma_1 \rightarrow \tilde{\sigma}_2 \mid \tilde{\sigma}_1 \rightarrow \sigma_2 \mid \tilde{\sigma}_1 \rightarrow \tilde{\sigma}_2$$

Note that this encoding utilizes the generalized type structure of Λ_{ml} discussed in Section 3.5. Such an encoding yields a CPS transformation where Reynolds’s notion of trivial and serious expressions is generalized to functions — functions that always produce values do not need to be passed continuations to achieve evaluation-order independence [9, 33].

Elsewhere [16], we use Moggi’s *existence predicate* [26, Section 2.2] within the meta-language itself to derive such optimizations.

4.6 Other sequencing orders

Since Λ_{ml} makes control flow explicit, one can construct CPS transformations with different sequencing orders for

sub-expression evaluation. For example, replacing the encoding of application in Figure 8 with the one below gives a call-by-value CPS transformation where the argument is evaluated before the function in an application.

$$\mathcal{E}_v\langle e_0 e_1 \rangle = \text{let } x_1 \Leftarrow \mathcal{E}_v\langle e_1 \rangle \text{ in let } x_0 \Leftarrow \mathcal{E}_v\langle e_0 \rangle \text{ in } x_0 x_1$$

4.7 Conclusion

We have shown that a wide variety of evaluation strategies can be described by simple encodings in Λ_{ml} . The corresponding CPS transformations and correctness proofs follow. The next two sections describe how administrative reductions and the corresponding direct-style transformations follow as well.

5 A Generic Account of Administrative Reductions

Practical use of CPS transformations requires one to characterize “administrative reductions” *i.e.*, the reduction of the extraneous abstractions introduced by the transformation to obtain continuation-passing [7, 32, 36]. In fact, administrative reductions are characterized generically by the *monadic reductions* R_{mon} on Λ_{ml} . In particular, let, β and let.assoc correspond to the administrative reductions identified by Plotkin. It is straightforward to show that the reductions R_{mon} are Church-Rosser and strongly normalizing. Let \mathcal{N} be a function mapping every Λ_{ml} term to its R_{mon} normal form and let \mathcal{Z}_v denote a version of Plotkin’s call-by-value CPS transformation that carries out administrative reductions on the fly [1, 7, 43].

Property 9 For $\Gamma \vdash e : \tau$, $\mathcal{Z}_v\langle e \rangle \equiv (\mathcal{C}' \circ \mathcal{N} \circ \mathcal{E}_v)\langle e \rangle$.

A similar property also holds for the corresponding one-pass call-by-name CPS transformation, and for the corresponding CPS transformations after static analyses [8, 9, 31]. This staging and the account of administrative reductions prior to introducing continuations have been recently noted [5, 6, 13, 21, 36]. Typically, CPS transformations are factored into three distinct steps:

1. naming intermediate values (captured by \mathcal{E});
2. flattening nested *let*’s (captured by \mathcal{N}); and
3. introducing continuations (captured by \mathcal{C}).

The last step is provably reversible, and Lawall automated that proof for another meta-language than Λ_{ml} [20].

Recently, Sabry and Felleisen have identified an additional optimization made possible by administrative reductions on call-by-value CPS terms [36]. The optimization corresponds to relocating evaluation contexts (reduction contexts, continuations) inside abstractions in β -redexes. The following R_{ml} equivalence characterizes this optimization:¹¹

$$\varphi[(\lambda x.e_0) e_1] =_{R_{ml}} (\lambda x.\varphi[e_0]) e_1$$

where $\varphi \not\equiv [\cdot]$ and $x \notin FV(\varphi)$. Let let.ctx be the reduction induced by reading this equivalence from left to right.¹²

¹¹ Even with this optimization, Sabry and Felleisen’s call-by-value CPS transformation will produce slightly more compact terms. Having continuations as first arguments to functions makes it possible for all continuations to be relocated inside the abstractions of all β -redexes. Here, trivial continuations (*i.e.*, identifiers k) are not relocated. In any case, this optimization is independent of continuations in general, and in particular of passing them first or last to CPS functions.

¹² A similar optimization also exists for $\text{rec } f(x). e$.

The Church-Rosser and strong normalization property extends to $R_{mon} \cup \{\text{let.ctx}\}$ reductions, characterizing administrative reductions (including the above optimization) generically.

We have characterized administrative reductions abstractly in terms of normalization of Λ_{ml} terms. In practice, one would define an optimized translation that performs administrative reductions “on the fly” [1, 7, 43]. This can either be achieved using brute force [27, 36] or with a two-level specification [7, 8, 31].

6 DS Transformations from Encodings of Computational Properties

Direct-style transformations mapping CPS terms back to direct-style Λ terms are potentially useful in their own right. Our transformation \mathcal{C}^{-1} forms the core of a generic DS transformation, thus generalizing previous work in the absence of computational effects other than non-termination [6, 35]. Direct-style transformations \mathcal{D} are obtained by composing “inverse” encodings \mathcal{E}^{-1} (mapping Λ_{ml} terms to direct-style Λ terms) with the transformation \mathcal{C}^{-1} .

$$\mathcal{D} \stackrel{\text{def}}{=} \mathcal{E}^{-1} \circ \mathcal{C}^{-1}$$

The transformation \mathcal{E}^{-1} may be defined in several ways. A simple technique is to “unfold” *let* constructs and remove $[\cdot]$ constructs — thus collapsing values and computations, under some side-conditions ensuring that the resulting terms remain evaluated in the same order. This is the technique used by *e.g.*, Lawall and Danvy [6, 10, 19, 21]. Alternatively, one may adapt the techniques of Sabry and Felleisen [36] and map reduction contexts to evaluation contexts in Λ .

In general, transformations \mathcal{E}^{-1} defined as above are meaning-preserving only when defined on the language of Λ_{ml} terms in the image of a corresponding encoding \mathcal{E} (or such a language closed under R_{ml} reductions). Considering a more general domain for \mathcal{E}^{-1} usually requires additional constructs in Λ which explicitly direct computation (*e.g.*, strict *let*’s, *thunks*) without resorting to full continuation-passing style.

Formal properties and details of methods to obtain DS transformations for specific evaluation orders are described elsewhere [16].

7 Products and Co-products

This section outlines how products and co-products are incorporated into the generic framework. A detailed discussion and proofs can be found elsewhere [16]. All of the results of the previous sections scale up to the extended language.

Figure 13 extends the syntax of Λ_{ml} to include products and co-products.¹³ The set of *value types* is extended to include types $\tau_1 \times \tau_2$ and $\tau_1 + \tau_2$. The reductions for products and co-products are as follows.¹⁴

$$\begin{aligned} \pi_i(e_1, e_2) &\longrightarrow_{\times, \beta_i} e_i \\ \text{case}(in_i e) \text{ of } (x_1.e_1) \mid (x_2.e_2) &\longrightarrow_{+, \beta_i} e_i[x_i := e] \end{aligned}$$

As with function spaces, the structure of Λ_{ml} types and terms provides a description of constructors with differing

¹³ The presentation of products follows Moggi [26, Section 3.1].

¹⁴ A presentation including the usual \times, η and $+, \eta$ rules for products and co-products can be found elsewhere [16].

$$\begin{array}{c}
\frac{\Gamma \vdash_{ml} e : \tau_1 \times \tau_2}{\Gamma \vdash_{ml} \pi_i e : \tau_i} \quad (i = 1, 2) \\
\frac{\Gamma \vdash_{ml} e_1 : \tau_1 \quad \Gamma \vdash_{ml} e_2 : \tau_2}{\Gamma \vdash_{ml} (e_1, e_2) : \tau_1 \times \tau_2} \\
\frac{\Gamma \vdash_{ml} e : \tau_i}{\Gamma \vdash_{ml} in_i^{\tau_1 + \tau_2} e : \tau_1 + \tau_2} \quad (i = 1, 2) \\
\frac{\Gamma \vdash_{ml} e : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash_{ml} e_1 : \tilde{\tau} \quad \Gamma, x_2 : \tau_2 \vdash_{ml} e_2 : \tilde{\tau}}{\Gamma \vdash_{ml} case\ e\ of\ (x_1.e_1) \mid (x_2.e_2) : \tilde{\tau}} \\
\tau ::= \dots \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2
\end{array}$$

Figure 13: Abstract syntax of products and coproducts for the computational meta-language Λ_{ml}

$$\begin{array}{ll}
\mathcal{C}\langle \pi_i e \rangle = \pi_i \mathcal{C}\langle e \rangle & \mathcal{C}\langle in_i e \rangle = in_i \mathcal{C}\langle e \rangle \\
\mathcal{C}\langle (e_1, e_2) \rangle = (\mathcal{C}\langle e_1 \rangle, \mathcal{C}\langle e_2 \rangle) & \mathcal{C}\langle case\ e\ of\ (x_1.e_1) \mid (x_2.e_2) \rangle = case\ \mathcal{C}\langle e \rangle\ of\ (x_1.\mathcal{C}\langle e_1 \rangle) \mid (x_2.\mathcal{C}\langle e_2 \rangle) \\
\mathcal{C}\langle \tau_1 \times \tau_2 \rangle = \mathcal{C}\langle \tau_1 \rangle \times \mathcal{C}\langle \tau_2 \rangle & \mathcal{C}\langle \tau_1 + \tau_2 \rangle = \mathcal{C}\langle \tau_1 \rangle + \mathcal{C}\langle \tau_2 \rangle
\end{array}$$

Figure 14: Continuation introduction for products and co-products

computational properties (*e.g.*, eager or lazy). For example, eager (*i.e.*, call-by-value) pairing can be expressed via products of values.

$$\begin{array}{l}
\mathcal{E}_v\langle \tau_1 \times \tau_2 \rangle = \mathcal{E}_v\langle \tau_1 \rangle \times \mathcal{E}_v\langle \tau_2 \rangle \\
\mathcal{E}_v\langle (e_1, e_2) \rangle = let\ x_1 \leftarrow \mathcal{E}_v\langle e_1 \rangle \\
\quad in\ let\ x_2 \leftarrow \mathcal{E}_v\langle e_2 \rangle \\
\quad in\ [(x_1, x_2)] \\
\mathcal{E}_v\langle \pi_i e \rangle = let\ x \leftarrow \mathcal{E}_v\langle e \rangle\ in\ [\pi_i x]
\end{array}$$

Lazy (*i.e.*, call-by-name) injections can be expressed via co-products of computations.

$$\begin{array}{l}
\mathcal{E}_n\langle \tau_1 + \tau_2 \rangle = \mathcal{E}_n\langle \tau_1 \rangle + \mathcal{E}_n\langle \tau_2 \rangle \\
\mathcal{E}_n\langle in_i e \rangle = [in_i \mathcal{E}_n\langle e \rangle] \\
\mathcal{E}_n\langle case\ e\ of\ (x_1.e_1) \mid (x_2.e_2) \rangle = let\ x \leftarrow \mathcal{E}_n\langle e \rangle \\
\quad in\ case\ x\ of\ (x_1.\mathcal{E}_n\langle e_1 \rangle) \\
\quad \quad \quad \mid (x_2.\mathcal{E}_n\langle e_2 \rangle)
\end{array}$$

Moreover, the framework naturally describes non-standard forms of products and co-products (*e.g.*, one lazy component, one eager component) such as might occur in a program after strictness and/or termination analysis.

Figure 14 extends \mathcal{C} to products and co-products. As before, correct CPS transformations for the extended language are obtained by composing the encodings \mathcal{E} with \mathcal{C} . The correctness hinges on the fact that all CPS terms will have only values as constructor arguments (*i.e.*, either terms corresponding to meta-language values, or abstractions $\lambda k \dots$ corresponding to meta-language computations). This generalizes our earlier work [8], where we presented a CPS transformation after strictness analysis, handling both strict and non-strict products.

Note that the definition of \mathcal{C} in Figure 14 relies on the monadic constructs to structure continuation-passing properly. However, the definition below gives an alternate structure commonly used when transforming conditional expressions.

$$\begin{array}{l}
\mathcal{C}\langle case\ e\ of\ (x_1.e_1) \mid (x_2.e_2) \rangle \\
= \lambda k. case\ \mathcal{C}\langle e \rangle\ of\ (x_1.\mathcal{C}\langle e_1 \rangle k) \mid (x_2.\mathcal{C}\langle e_2 \rangle k)
\end{array}$$

The latter definition allows reduction contexts (in the form of continuations) to be relocated inside *case* constructs — which duplicates the contexts.¹⁵ Indeed, this definition requires adding the following reduction to the set of Λ_{ml} reductions to obtain an equational correspondence with CPS terms:¹⁶

$$\begin{array}{l}
\varphi\langle case\ e\ of\ (x_1.e_1) \mid (x_2.e_2) \rangle \\
\longrightarrow_{+.ctxt} case\ e\ of\ (x_1.\varphi\langle e_1 \rangle) \mid (x_2.\varphi\langle e_2 \rangle)
\end{array}$$

where $\varphi \not\equiv [\]$ and $x_1, x_2 \notin FV(\varphi)$. This reduction is sound with respect to the operational semantics of Λ_{ml} extended with products and co-products.

8 Compiling with Monadic Normal Forms

In situations where explicit continuations are not needed (*e.g.*, for compiling programs without jumps), Λ_{ml} stands as an alternative language to CPS — very close to CPS but without continuations. A language with similar properties (“A-normal forms”) has been proposed by Flanagan *et al.* [13] and studied for untyped, call-by-value λ -terms. In particular, “A-reductions” provide the following standard compiler optimizations [13, page 243]:

1. code segments are merged across declarations and conditions;
2. reductions are lifted out of evaluation contexts and intermediate results are named.

These properties occur naturally in Λ_{ml} . The Λ_{ml} reductions *let.assoc* and *+.ctxt* merge code segments across declarations (*i.e.*, *let*) and conditionals (more generally, *case* statements). Encodings \mathcal{E} into Λ_{ml} name intermediate results and the reduction *let.assoc* lifts reductions out of reduction contexts.

¹⁵This duplication can be avoided inserting a β -redex when introducing continuations [7, 36].

¹⁶Sabry and Felleisen [36] give a similar reduction for conditional expressions.

Thus, Moggi’s meta-language is not only a flexible formal tool, but also an attractive intermediate language for compiling. In particular, a sub-language of Λ_{ml} that we call the language of *monadic normal forms* gives the properties discussed above for any evaluation order that can be encoded into Λ_{ml} . (The word “monadic” is slightly abused here since $+.ctxt$ is not a monadic reduction.) The strong normalization and confluence properties of R_{mon} reductions extend to $R_{mon} \cup \{+.ctxt\}$. Let \mathcal{N}' be a function taking a Λ_{ml} term to its $R_{mon} \cup \{+.ctxt\}$ normal form. The function $\mathcal{N}' \circ \mathcal{E}$ maps Λ terms to monadic normal forms for an arbitrary encoding \mathcal{E} , thereby performing administrative reductions independently of continuations (see Section 5).

Recent trends indicate that types are important for intermediate languages. For example, Burn and Le Métayer point out that types on CPS transformations give a useful characterization of boxed and unboxed values [3]. This observation applies here as well — computation types correspond to boxed values, and value types correspond to unboxed values.

9 Related Work

The framework presented here relies on a formal connection between Moggi’s computational meta-language and CPS terms and types. Moggi proposes the meta-language as a means of abstractly capturing the basic computational structure of programs. Semantic definitions of programs are obtained by a categorical interpretation parameterized with different *monads* capturing various *notions of computation*. Moggi gives a *continuation monad* in the category **Set** as particular example of a notion of computation — establishing a correspondence between the meta-language and set-theoretic continuation-passing functions [26, page 58].

Wadler illustrated the usefulness of Moggi’s ideas when applied to functional programming [42]. In essence, he showed how programs written in the style of the meta-language (*i.e.*, *monadic* style) could be parameterized with term representations of monads — thus abstractly capturing various computational effects such as side-effects on a global state, *etc.* In particular, he showed how call-by-value and call-by-name CPS interpreters can be obtained by instantiating call-by-value and call-by-name monadic-style interpreters with a term representation of the CPS monad — thereby informally relating the encodings \mathcal{E}_v and \mathcal{E}_n with call-by-value and call-by-name CPS terms.

In contrast, we formalize the relationship between the complete meta-language Λ_{ml} and CPS terms. Based on this formulation, we generically capture many different aspects associated with CPS (construction of CPS transformations, correctness of transformations with regard to computational adequacy and preservation of equational theories, administrative reductions, construction of DS transformations, typing of transformations, *etc.*) which were previously handled individually for each evaluation order. We emphasize that Λ_{ml} is powerful enough to describe not only the standard call-by-value and call-by-name strategies but many other useful strategies appearing in the literature. One only needs to identify computational properties with Λ_{ml} and all the aspects mentioned above follow as corollaries in the framework. To the best of our knowledge, this is the first attempt of such a global investigation of CPS.

Sabry and Felleisen, in their recent work [36], hint at the relationship between Moggi’s computational framework and CPS terms. They derive a calculus for untyped call-

by-value DS terms which equationally corresponds to call-by-value CPS terms under the $\beta\eta$ calculus. They note that the resulting calculus equationally corresponds to an untyped variant of Moggi’s computational λ -calculus λ_c [23] — a calculus for call-by-value terms capturing equivalences that hold for any *notion of computation*.

However, this correspondence seems to stem more from the emphasis on naming intermediate values present in both calculi rather than from any deliberate structural connection with *e.g.*, the CPS monad. For example, the terms produced by Sabry and Felleisen’s CPS transformation (a curried version of Fischer’s transformation [12], where continuations occur first in functions) do not have the fundamental computational structure dictated by Moggi’s framework. This is most easily seen by observing the mismatch between the typing of function spaces in Fischer’s transformation and in the transformations generated by the CPS monad (curried and with continuations occurring last). In contrast, our framework is deliberately based on the structural (and equational) correspondence between Λ_{ml} and generic CPS terms.

Using techniques analogous to those of Sabry and Felleisen [36], Sabry and Field have investigated “state-passing style” (uncurried and with state occurring last), deriving calculi for an untyped language with state. In contrast, one can take the *state monad* [26] and translate from the meta-language Λ_{ml} to various state-passing styles (curried and with state occurring last) in the same way as we have used the continuation monad to generate a variety of continuation-passing styles: one then obtains a state-passing transformation for any evaluation order, generic administrative reductions, and the corresponding “direct-style” transformations. As for CPS, these tools are obtained by showing an equational correspondence between the meta-language and a term representation of the state monad.

Thus Moggi’s framework seems to provide a solid basis for studying both the relation between implicit and explicit representations of control and the relation between implicit and explicit representations of state, in a typed setting. In particular, we are currently investigating how the continuation/state monad (obtained *e.g.*, by applying the state-monad constructor to the continuation monad [25]) offers a generic relation between implicit and explicit representations of both control and state. We are also considering to add computational effects on control (first-class continuations) and on the state (side-effects).

10 Conclusion and Issues

We have characterized CPS transformations of typed λ -terms for any evaluation order, their administrative reductions, and the corresponding DS transformations, in one generic framework based on Moggi’s computational meta-language and using a term representation of the CPS monad. Plotkin’s Indifference, Simulation, and Translation theorems are generalized for the continuation introduction \mathcal{C} . Characterizations of administrative reductions (including Sabry and Felleisen’s optimization) are scaled up in a typed framework for any evaluation order. Moggi’s computational meta-language appears as a generic typed intermediate language for compiling, alternatively to CPS and with an equivalent expressive power, in the absence of first-class continuations. We are currently considering other monads for applying the methodology developed here. Preliminary investigation for the state monad suggests that the same benefits hold: state-

passing style for any evaluation order, uniform administrative reductions, and the corresponding “direct-style” transformations. Ditto for the continuation/state monad.

This investigation should make it possible to come back to Griffin’s connection between double-negation translation and CPS transformation [14] and to Murthy’s intrepid display of continuation-passing styles, from a logical standpoint [28, Chapters 9 & 10]. Griffin identified Plotkin’s call-by-value CPS transformation as a logical embedding. Murthy identified it as a variant of the Kuroda negative translation, and Plotkin’s call-by-name CPS transformation as the Kolmogorov translation. In fact, looking back at Murthy’s PhD thesis, it is striking that his “slightly modified” Kuroda translation providing for mixed call-by-value and call-by-name evaluation [28, page 159] corresponds to the mixed CPS transformation of Section 4.4, and that his “pervasive Kolmogorov translation” [28, pages 164-167] corresponds to Reynolds’s CPS transformation in Section 4.2. The equational correspondence between Λ_{ml} and CPS strongly suggests that the encoding of any evaluation order into Λ_{ml} could be formalized as a logical embedding. We leave this point for a future work.

Acknowledgements

We are grateful to Andrzej Filinski and Bob Harper for fundamental observations and encouragements at an early stage of this work. Thanks are also due to Matthias Felleisen, Sergey Kotov, Julia Lawall, Peter Lee, Karoline Malmkjær, Chet Murthy, Frank Pfenning, Amr Sabry, Dave Schmidt, and the referees for comments.

References

- [1] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.
- [2] Henk Barendregt. *The Lambda Calculus — Its Syntax and Semantics*. North-Holland, 1984.
- [3] Geoffrey Burn and Daniel Le Métayer. Proving the correctness of compiler optimisations based on a global program analysis. Technical report Doc 92/20, Department of Computing, Imperial College of Science, Technology and Medicine, London, England, 1992.
- [4] William Clinger, editor. *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, San Francisco, California, June 1992. ACM Press.
- [5] Olivier Danvy. Three steps for the CPS transformation. Technical Report CIS-92-2, Kansas State University, Manhattan, Kansas, December 1991.
- [6] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 1993. Special issue on ESOP’92, the Fourth European Symposium on Programming, Rennes, February 26-28, 1992. To appear.
- [7] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. In Wand [44], pages 361–391.
- [8] Olivier Danvy and John Hatcliff. CPS transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, 1(3):195–212, 1993.
- [9] 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*, Lecture Notes in Computer Science, New Orleans, Louisiana, April 1993. To appear.
- [10] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In Clinger [4], pages 299–310.
- [11] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD machine, and the λ -calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.
- [12] Michael J. Fischer. Lambda-calculus schemata. In Talcott [39]. An earlier version appeared in an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices, Vol. 7, No. 1, January 1972.
- [13] 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.
- [14] Timothy G. Griffin. A formulae-as-types notion of control. In Paul Hudak, editor, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, California, January 1990. ACM Press.
- [15] Bob Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. In Talcott [39].
- [16] John Hatcliff. PhD thesis, Department of Computing and Information Sciences, Kansas State University, Manhattan, Kansas, USA, March 1994. Forthcoming.
- [17] John Hatcliff and Olivier Danvy. Thunks and the λ -calculus. Technical Report CIS-93-15, Kansas State University, Manhattan, Kansas, September 1993.
- [18] Peter Z. Ingerman. Thunks, a way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM*, 4(1):55–58, 1961.
- [19] Julia L. Lawall. PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, USA, 1993. Forthcoming.
- [20] Julia L. Lawall. Proofs by structural induction using partial evaluation. In David A. Schmidt, editor, *Proceedings of the Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 155–166, Copenhagen, Denmark, June 1993. ACM Press.
- [21] Julia L. Lawall and Olivier Danvy. Separating stages in the continuation-passing style transformation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 124–136, Charleston, South Carolina, January 1993. ACM Press.

- [22] 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.
- [23] Eugenio Moggi. Computational lambda-calculus and monads. Report ECS-LFCS-88-66, University of Edinburgh, Edinburgh, Scotland, October 1988.
- [24] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, June 1989. IEEE Computer Society Press.
- [25] Eugenio Moggi. An abstract view of programming languages. Course notes ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, Edinburgh, Scotland, April 1990.
- [26] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.
- [27] Luc Moreau and Daniel Ribbens. Sound rules for parallel evaluation of a functional language with `callcc`. In Arvind, editor, *Proceedings of the Sixth ACM Conference on Functional Programming and Computer Architecture*, pages 125–135, Copenhagen, Denmark, June 1993. ACM Press.
- [28] Chetan R. Murthy. *Extracting Constructive Content from Classical Proofs*. PhD thesis, Department of Computer Science, Cornell University, 1990.
- [29] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In Bernard Robinet, editor, *Proceedings of the Fourth International Symposium on Programming*, number 83 in Lecture Notes in Computer Science, pages 269–281, Paris, France, April 1980.
- [30] Peter Naur (editor). Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6(1):1–17, 1962.
- [31] Chris Okasaki, Peter Lee, and David Tarditi. Call-by-need and continuation-passing style. In Talcott [39].
- [32] Gordon D. Plotkin. Call-by-name, call-by-value and the λ -calculus. *Theoretical Computer Science*, 1:125–159, 1975.
- [33] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, 1972.
- [34] John C. Reynolds. On the relation between direct and continuation semantics. In Jacques Loeckx, editor, *2nd Colloquium on Automata, Languages and Programming*, number 14 in Lecture Notes in Computer Science, pages 141–156, Saarbrücken, West Germany, July 1974.
- [35] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In Clinger [4], pages 288–298.
- [36] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In Talcott [39].
- [37] Amr Sabry and John Field. Reasoning about explicit and implicit representation of state. In Paul Hudak, editor, *Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages*, pages 17–30, Copenhagen, Denmark, June 1993.
- [38] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.
- [39] Carolyn L. Talcott, editor. *Special issue on continuations*, LISP and Symbolic Computation, Vol. 6, Nos. 3/4. Kluwer Academic Publishers, 1993.
- [40] Robert D. Tennent. *Semantics of Programming Languages*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1991.
- [41] Philip Wadler. Comprehending monads. In Wand [44], pages 461–493.
- [42] Philip Wadler. The essence of functional programming (tutorial). In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.
- [43] Mitchell Wand. Correctness of procedure representations in higher-order assembly language. In Stephen Brookes, Michael Main, Austin Melton, Michael Mislove, and David Schmidt, editors, *Mathematical Foundations of Programming Semantics*, volume 598 of *Lecture Notes in Computer Science*, pages 294–311, Pittsburgh, Pennsylvania, March 1991. 7th International Conference.
- [44] Mitchell Wand, editor. *Special issue on the 1990 ACM Conference on Lisp and Functional Programming*, Mathematical Structures in Computer Science, Vol. 2, No. 4. Cambridge University Press, December 1992.