The Structure of Continuation-Passing Styles

by

John Mark Hatcliff

B.A., Mount Vernon Nazarene College, 1988
M.Sc., Queen's University, 1991

A DISSERTATION

submitted in partial fulfillment of the

requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1994

DRAFT: July 25, 1994

Abstract

Continuation-passing style (CPS) is a method of representing program evaluation order in a purely functional manner. Many applications of CPS rely on *CPS transformations* which explicitly encode evaluation strategies (*e.g.*, call-by-name, call-by-value, *etc.*) into the structure of programs. Existing CPS transformations are based almost entirely on the call-by-name and call-by-value CPS transformations defined by Plotkin, Reynolds, and Fischer over twenty years ago. These transformations introduce the same pattern of evaluation (either call-by-name or call-by-value) throughout the entire program. Such rigidness ignores additional computational properties such as strictness and termination that are used to optimize program evaluation.

We take a more general approach to CPS — we direct CPS transformations by computational properties instead of by fixed evaluation strategies. For example, a CPS transformation directed by *strictness* properties subsumes both the call-by-name transformation (in case all functions are non-strict) and the call-by-value transformation (in case all functions are strict). Moreover, it can transform programs with both strict and non-strict functions such as a call-by-name program after strictness analysis. Considering *termination* properties allows CPS transformations to avoid introducing unnecessary continuations. Based on the property of *value*, we show how CPS transformations (*e.g.*, the strictness-directed transformation) can be factored into the call-by-value transformation and a "thunk"-based transformation. This simplifies reasoning about the factored transformations.

To formalize the approach, extensions of traditional λ-calculi are proposed to capture strictness and termination properties. Our general CPS transformations translate the extended calculi into the standard λ and λ_v calculi. The transformations are shown to preserve operational semantics and equational properties of the extended languages. As a result, we obtain a variety of CPS transformations which have applications to compilation and partial evaluation.

Standard CPS transformations (as well as the new ones presented here) have striking structural similarities. However, these similarities have never been exploited since previous work has always dealt with each transformation separately. To remedy this, we propose a generic framework for studying CPS transformations based on Moggi's computational meta-language. The framework captures the essence of the structure of continuation-passing styles and allows generic formalizations of CPS transformations, proofs of preservation of operational semantics and program calculi, optimizations known as "administrative reductions", typing properties, and inverse transformations. Results for specific evaluation orders follow as corollaries of the generic results. Because the extended type system of the meta-language directly captures computational properties, we can describe all aspects of the standard call-by-value and call-by-name CPS transformations as well as the variations presented in the first part of the dissertation. Thus, previous work on CPS and CPS transformations is unified in a single framework.

Acknowledgements

Olivier Danvy deserves my warmest gratitude for supervising this work. His concern and attention to my progress were extraordinary — he continues to be a mentor and true friend. Olivier and I were co-authors of several papers which form the core of this dissertation and his ideas were the genesis of many of the results presented here.

Dave Schmidt served as my co-supervisor and his wisdom was invaluable. He and Allen Stoughton provided important technical advice at various stages of this work. I thank Olivier, Dave, and Allen for their financial support of my studies.

The good fellowship of the programming language group at Kansas State University has made my stay in Kansas a very enjoyable experience. In particular, I wish to thank my office partner Sergey Kotov for his careful proofreading, keen criticism, his encouraging suggestions.

Andrzej Filinski and Julia Lawall provided valuable feedback and advice at various stages of this work. Their work on continuations also provided inspiration.

Finally, I thank my wife Lorie for her long-suffering patience, warm encouragement, and delicious apple pies.

To Lorie Jill

Contents

List of Figures

Chapter 1

Introduction

Representing *control* is an important aspect of formally describing program evaluation. A control representation provides the basis for defining an *evaluation strategy*, *i.e.*, a strategy for scheduling program components for evaluation.

To be suitable for a mathematical study of program evaluation, a control representation should be simple yet expressive. It should be simple enough so that one may reason about the course of evaluation using a limited number of concepts. Yet it must be expressive enough to encode all manifestations of control transfer in the programming language under consideration. This may include non-local transfer of control such as exits, breaks, or gotos.

Control can be represented in a purely functional manner using *continuations*. Intuitively, a continuation is a function that represents "the rest of the program". Associating a continuation with a program component specifies the computational steps that take place after that component is evaluated. Thus, a particular evaluation strategy or course of evaluation can be determined by associating a continuation with each program component.

In functional programming languages, continuations can be encoded directly into program structure. Each program component (*e.g.*, each function) is passed a continuation which determines how evaluation will continue after the component is evaluated. This results in a style of program construction called *continuation-passing style* (CPS). A functional program can be turned into CPS automatically using program transformations known as *CPS transformations*.

Because continuations can represent control using the simple concepts of function abstraction and application, and because they can express manifestations of control transfer that occur in conventional programming languages, continuations appear in many areas of programming language theory. Interestingly, continuations and CPS possess properties that make them very attractive for implementation-oriented applications as well.

Strachey and Wadsworth, who pioneered the use of continuations in denotational semantics during the early 1970's, testify to their simplicity:

> *Those of us who have worked with continuations for some time have soon learned to think of them as natural and in fact often simpler than the earlier methods.*

However, this view hardly seems representative of the programming language community at large. Indeed, others have noted that continuation-passing style is often perceived as "enigmatic" [20], "deep", "subtle" [34], "complicated", "unfriendly to read" [19, p. 3], "horribly opaque" [66, p. 8].

These disparate views seem to stem from the fact while continuations provide *mathematical* simplicity, the structure of programs in continuation-passing style is far-removed from what programmers naturally create. Continuation-passing style is unnatural because it overloads the program structure with many control details that programmers usually resolve intuitively.

Perhaps as a result of its "unnaturalness", many issues regarding continuation-passing structure have not been fully explored.

Rigid structure: Existing CPS transformations are based almost entirely on the transformations defined by Plotkin [76], Reynolds [80], and Fischer [31] over twenty years ago. These transformations introduce the same continuation-passing structure throughout an entire program. Such rigidness ignores additional computational properties that are commonly used to optimize program evaluation.

Unexploited structural similarities: Many continuation-passing styles having striking structural similarities, but these have not been exploited to present a unifying framework. As a consequence, many meta-theory activities such as correctness proofs of transformations, *etc.*, must be performed repeatedly even though the objects under consideration only vary in minor ways.

Lack of structural abstractions: Even though the structure of continuation-passing styles appears complex, continuations actually are manipulated in only a few very regular patterns. This suggests that the view of continuation-passing styles might be simplified by characterizing them using a set of abstract instructions (corresponding to the patterns of continuation-passing).

In this dissertation, we explore these issues in the context of purely functional programming languages. Below we present a gentle introduction to basic concepts related to our work. We conclude this introduction with a statement of our thesis and an overview of the dissertation contents.

1.1 Overview of concepts

This section gives a brief informal overview of evaluation strategies, continuations, and continuation-passing style. For detailed introduction, the reader is referred to the textbook of Friedman, Wand, and Haynes [34].

1.1.1 Evaluating functional programs

An evaluation strategy specifies the order of evaluation of program components. More precisely, an evaluation strategy is a method for determining which program component to evaluate next. The two most common evaluation strategies for functional programs are *call-by-name* and *call-by-value*.

Call-by-name

The call-by-name strategy is based on the "copy rule". A function application is evaluated by replacing the application with a new version of the applied function where the argument (actual parameter) is substituted (*i.e.*, "copied") for the bound identifier (formal parameter).[1] In the evaluation below, the argument $5 + 5$ is substituted for the identifier x in the body of the function $\lambda x \,.\, x + x$.

$$\begin{aligned}(\lambda x \,.\, x + x)\,(5 + 5) &\longmapsto_n (5 + 5) + (5 + 5) \\ &\longmapsto_n 10 + (5 + 5) \\ &\longmapsto_n 10 + 10 \\ &\longmapsto_n 20\end{aligned} \tag{1.1}$$

Call-by-value

The call-by-value strategy dictates that arguments be evaluated to values before being passed to functions. A function application is evaluated by first reducing the argument to a value and then substituting this *value* for the bound identifier in the body of the function. In the evaluation below, the argument $5 + 5$ is first reduced to the value 10; this value is then substituted for the identifier x in the body of the function $\lambda x \,.\, x + x$.

$$\begin{aligned}(\lambda x \,.\, x + x)\,(5 + 5) &\longmapsto_v (\lambda x \,.\, x + x)\,10 \\ &\longmapsto_v 10 + 10 \\ &\longmapsto_v 20\end{aligned} \tag{1.2}$$

Note that the argument $5 + 5$ was evaluated twice under call-by-name, but only once under call-by-value.

[1] The technical issues associated with the possible "capture of free identifiers" are addressed in Chapter 2. We will ignore such matters in this exposition since they are dealt with in a straightforward manner [6].

Assessment

The call-by-name and call-by-value evaluation strategies can be contrasted with respect to their relative simplicity, efficiency, and meaning.

- *Simplicity:* Call-by-name *evaluation* is conceptually simpler since processing function applications amounts to blindly copying arguments into function bodies. This has implementation benefits. For example, it simplifies in-lining of operations in compiling [3].

  Reasoning about behavior of programs under call-by-name evaluation can be more complicated than under call-by-value if the language being considered includes computational effects such as state updates, and input/output (I/O). In essence, it is hard to predict the number of times arguments will be evaluated (and thus the number of effects) under call-by-name whereas, under call-by-value, arguments will be evaluated exactly once (and thus each effect represented in the argument will occur exactly once).

- *Efficiency:* Because arguments are evaluated exactly once under call-by-value, it is generally more efficient than call-by-name. Under call-by-name, arguments may be evaluated any number of times — roughly depending on the number of occurrences of the formal parameter in the body of the function. In the illustration above, the argument $5 + 5$ was evaluated once under call-by-value and twice under call-by-name.

  Call-by-value evaluation is often described as *eager* evaluation since function arguments are evaluated as soon as possible. Eager evaluation may lead to an argument being evaluated unnecessarily if the argument is never used by the function. Call-by-name evaluation is often described as *lazy* evaluation since argument evaluation is delayed is long is possible. Lazy evaluation never unnecessarily evaluates an argument, but must often pay the price by evaluating arguments more than once.[2]

- *Meaning:* This difference in number of times arguments are evaluated gives rise to a difference in meaning in the presence of computational effects such as non-termination, state, and I/O. Let Ω represent some expression which fails to terminate under both call-by-name and call-by-value. In the following example, call-by-name evaluation yields the value 5 whereas call-by-value evaluation *diverges* (*i.e.*, fails to terminate).

  Call-by-name:

$$\begin{array}{lll} (\lambda x\,.\,(\lambda y\,.\,x)\,\Omega)\,5 & \longmapsto_n & (\lambda y\,.\,5)\,\Omega \\ & \longmapsto_n & 5 \end{array} \tag{1.3}$$

  Call-by-value:

$$\begin{array}{lll} (\lambda x\,.\,(\lambda y\,.\,x)\,\Omega)\,5 & \longmapsto_v & (\lambda y\,.\,5)\,\Omega \\ & \longmapsto_v & (\lambda y\,.\,5)\,\Omega' \\ & \longmapsto_v & (\lambda y\,.\,5)\,\Omega'' \\ & \longmapsto_v & \ldots \end{array} \tag{1.4}$$

Given the absence of the formal parameter y in the body of the function $\lambda y\,.\,5$, call-by-name discards the non-terminating argument Ω whereas call-by-value demands its evaluation. In the following example where y occurs in the body of the function $\lambda y\,.\,y$, Ω is not discarded, and call-by-name and call-by-value meaning coincide.

Call-by-name:

$$(\lambda x\,.\,(\lambda y\,.\,y)\,\Omega)\,5 \quad \longmapsto_n \quad (\lambda y\,.\,y)\,\Omega \tag{1.5}$$

[2] A compromise between call-by-name and call-by-value is called *call-by-need*. Call-by-need is similar to call-by-name in that argument evaluation is delayed until the argument is actually used. However, multiple evaluations of the same argument are avoided (as in call-by-value) by saving the result of the first evaluation. See [8] for an intuitive presentation of call-by-need implementation, [49] for an operational semantics, and [73] for relationships between call-by-need and CPS.

$$\begin{aligned} &\longmapsto_n \quad \Omega \\ &\longmapsto_n \quad \Omega' \\ &\longmapsto_n \quad \Omega'' \\ &\longmapsto_n \quad \ldots \end{aligned}$$

Call-by-value:

$$\begin{aligned} (\lambda x\,.\,(\lambda y\,.\,y)\,\Omega)\,5 \quad &\longmapsto_v \quad (\lambda y\,.\,y)\,\Omega \\ &\longmapsto_v \quad (\lambda y\,.\,y)\,\Omega' \\ &\longmapsto_v \quad (\lambda y\,.\,y)\,\Omega'' \\ &\longmapsto_v \quad \ldots \end{aligned} \tag{1.6}$$

In cases such as above where the call-by-name and call-by-value meaning of a term coincide, the term is said to be *evaluation-order independent.*[3]

The call-by-name function $\lambda y\,.\,y$ of the preceding example is an instance of a *strict* function. A function is strict if it diverges when applied to a diverging argument. The function $\lambda y\,.\,5$ used above is *non-strict* under call-by-name since its application to Ω terminates. Strictness is only of interest when considering call-by-name evaluation since all functions are (by definition) strict under call-by-value.

The strictness of a call-by-name function represents a potential for optimization. If a call-by-name function is strict, it is *safe* (*i.e.*, meaning is preserved) to evaluate arguments of the function eagerly. Consider the following example (where $\longmapsto_{n.e}$ denotes the eager call-by-name evaluation of a strict function application).

$$\begin{aligned} (\lambda x\,.\,(\lambda y\,.\,y)\,\Omega)\,5 \quad &\longmapsto_n \quad (\lambda y\,.\,y)\,\Omega \\ &\longmapsto_{n.e} \quad (\lambda y\,.\,y)\,\Omega' \\ &\longmapsto_{n.e} \quad (\lambda y\,.\,y)\,\Omega'' \\ &\longmapsto_{n.e} \quad \ldots \end{aligned}$$

Even though the argument Ω is evaluated eagerly, the result is still the same as that given by the original call-by-name evaluation at line (1.7). Eager evaluation is an optimization since it guarantees that an argument will be evaluated once instead of possibly many times. Optimizing CPS transformations based on strictness properties is the subject of Chapter 4.

It is also safe to evaluate an application eagerly under call-by-name if it is known that the evaluation of the argument terminates. For example, the argument $5 + 5$ in the application $(\lambda x\,.\,x + x)\,(5 + 5)$ always terminates and thus can be evaluated eagerly.

$$\begin{aligned} (\lambda x\,.\,x + x)\,(5 + 5) \quad &\longmapsto_{n.e} \quad (\lambda x\,.\,x + x)\,10 \\ &\longmapsto_{n.e} \quad 10 + 10 \\ &\longmapsto_n \quad 20 \end{aligned}$$

This optimizes the conventional call-by-name evaluation given at line (1.1). Optimizing CPS transformations based on termination properties is the subject of Chapter 5. Note that optimizations based on strictness rely on properties of functions whereas optimizations based on termination rely on properties of arguments.

1.1.2 Representing control using continuations

Evaluation contexts

In the examples above, control information was implicit — the course of evaluation was expressed only by the informal description of the evaluation strategies. Aspects of control become more perspicuous when we view each evaluation step $e \longmapsto e'$ as a three-phase process:

1. factoring e into a context $E[\cdot]$ (a term with a "hole" $[\cdot]$) and the next component r to be reduced,

[3] To be precise, the *meaning* of the term is evaluation-order independent.

2. reducing r to obtain a new component r', and

3. replacing r by r' within the context $E[\cdot]$ to obtain e'.

For example, taking e to be $(\lambda x\,.\,x\ +\ x)\,((\lambda y\,.\,5)\,10)\ +\ 20$, the call-by-value evaluation step

$$(\lambda x\,.\,x\ +\ x)\,((\lambda y\,.\,5)\,10)\ +\ 20 \quad \longmapsto_v \quad (\lambda x\,.\,x\ +\ x)\,5\ +\ 20$$

can be viewed as

$$E_v[(\lambda y\,.\,5)\,10] \quad \longmapsto_v \quad E_v[5] \tag{1.7}$$

where $E_v[\cdot] \equiv (\lambda x\,.\,x\ +\ x)\,[\cdot]\ +\ 20$. An evaluation strategy can now be understood as a strategy for factoring a program e into a context $E[\cdot]$ and the next component r to be reduced. Contexts $E[\cdot]$ that result from such factorings are called *evaluation contexts* [27] since they form a distinguished subset of contexts in which evaluation occurs.

As expected, call-by-name gives an alternative factoring of the example expression e. The call-by-name evaluation step

$$(\lambda x\,.\,x\ +\ x)\,((\lambda y\,.\,5)\,10)\ +\ 20 \quad \longmapsto_n \quad ((\lambda y\,.\,5)\,10\ +\ (\lambda y\,.\,5)\,10)\ +\ 20$$

can be viewed as

$$E_n[(\lambda x\,.\,x\ +\ x)\,((\lambda y\,.\,5)\,10)] \quad \longmapsto_n \quad E_n[(\lambda y\,.\,5)\,10\ +\ (\lambda y\,.\,5)\,10] \tag{1.8}$$

where $E_n[\cdot] \equiv [\cdot]\ +\ 20$.

From evaluation contexts to continuations

Evaluation contexts represent control information in that they specify what is to be evaluated after the hole component r is reduced to a value. This control information can be encoded explicitly into the program structure by turning a context $E[\cdot]$ into a function $\lambda v\,.\,E[v]$ and passing it as a parameter to the hole component r. For example, the call-by-value factoring $E_v[(\lambda y\,.\,5)\,10]$ discussed above can be encoded as $((\lambda y\,.\,\lambda k\,.\,k\,5)\,10)\,(\lambda v_1\,.\,E_v[v_1])$. The evaluation steps below show that the encoding is correct, *i.e.*, it achieves the same effect as the original evaluation step at line (1.7).

$$\begin{aligned} ((\lambda y\,.\,\lambda k\,.\,k\,5)\,10)\,(\lambda v_1\,.\,E_v[v_1]) \quad &\longmapsto_v \quad (\lambda k\,.\,k\,5)\,(\lambda v_1\,.\,E_v[v_1]) \\ &\longmapsto_v \quad (\lambda v_1\,.\,E_v[v_1])\,5 \\ &\longmapsto_v \quad E_v[5] \end{aligned} \tag{1.9}$$

However, because the factoring (which captures control information) is now explicitly encoded in the structure of the term, call-by-name evaluation also gives the same effect.

$$\begin{aligned} ((\lambda y\,.\,\lambda k\,.\,k\,5)\,10)\,(\lambda v_1\,.\,E_v[v_1]) \quad &\longmapsto_n \quad (\lambda k\,.\,k\,5)\,(\lambda v_1\,.\,E_v[v_1]) \\ &\longmapsto_n \quad (\lambda v_1\,.\,E_v[v_1])\,5 \\ &\longmapsto_n \quad E_v[5] \end{aligned} \tag{1.10}$$

This should be contrasted with line (1.8) where the call-by-name evaluation of the original expression e gave a *different* effect from call-by-value.

The function $\lambda v_1\,.\,E_v[v_1]$ is called a *continuation* since it tells how the course of evaluation is to continue. The style of coding above is called *continuation-passing style*[4] because control information is represented by passing continuations to program components. An expression that does not employ continuation-passing is said to be in *direct style*.

[4] The term *continuation-passing style* is due to Steele [91].

Transforming into continuation-passing style

Above we have only transformed a single component of e into CPS. To transform the entire expression, we must turn each "potential" evaluation context into a continuation — thus associating all reducible program components with a continuation.

To clarify matters, the full call-by-value evaluation of e is given below. The boxes $\boxed{\cdot}$ are used to mark the hole of the evaluation context in each step.

$$\begin{aligned} e &\equiv (\lambda x\,.\,x\ +\ x)\,\boxed{(\lambda y\,.\,5)\,10}\ +\ 20 \\ &\xrightarrow{(1)}_v \boxed{(\lambda x\,.\,x\ +\ x)\,5}\ +\ 20 \\ &\xrightarrow{(2)}_v \boxed{(5\ +\ 5)}\ +\ 20 \\ &\xrightarrow{(3)}_v \boxed{10\ +\ 20} \\ &\xrightarrow{(4)}_v 30 \end{aligned} \tag{1.11}$$

To turn e into CPS, we locate the components of e that will appear as hole components and pass each a continuation. This is done in four steps, corresponding to the four evaluation steps above.

Step 1: The transformation associated with step 1 was discussed in detail above. In summary,

$$e \equiv (\lambda x\,.\,x\ +\ x)\,\boxed{((\lambda y\,.\,5)\,10)}\ +\ 20$$

is transformed to

$$e_1 \equiv ((\lambda y\,.\,\lambda k\,.\,k\,5)\,10)\,(\lambda v_1\,.\,(\lambda x\,.\,x\ +\ x)\,\boxed{v_1}\ +\ 20).$$

Step 2: The component $(\lambda x\,.\,x\ +\ x)\,v_1$ of e_1 corresponds to the term in the hole of the evaluation context $E_{v_2}[\cdot] \equiv [\cdot]\ +\ 20$ in step 2. Encoding E_{v_2} as the continuation $\lambda v_2\,.\,E_{v_2}[v_2]$,

$$e_1 \equiv ((\lambda y\,.\,\lambda k\,.\,k\,5)\,10)\,(\lambda v_1\,.\,\boxed{(\lambda x\,.\,x\ +\ x)\,v_1}\ +\ 20)$$

is transformed to

$$e_2 \equiv ((\lambda y\,.\,\lambda k\,.\,k\,5)\,10)\,(\lambda v_1\,.\,((\lambda x\,.\,\lambda k\,.\,k\,(x\ +\ x))\,v_1)\,(\lambda v_2\,.\,\boxed{v_2}\ +\ 20)).$$

Note that at this point all the functions in e_2 are in continuation-passing style even though steps 3 and 4 have not been encoded using continuations. One option is to view the transformation to CPS as complete since only occurrences of the *primitive operator* $+$ (which are reduced in steps 3 and 4) are not passed continuations. Another option is to introduce a continuation-passing version of $+$ which we write as $+^k$.[5] The evaluation rule for $+^k$ is as follows

$$n_1\ +^k\ n_2 \longmapsto_n \lambda k\,.\,k\,m$$

where m equals the sum of n_1 and n_2. Given $+^k$, we can encode the remaining two steps using continuation-passing.

Step 3: The component $x\ +\ x$ of e_2 corresponds to the term in the hole of the evaluation context for step 3. Since the identifier k already represents a continuation (during evaluation, k will be replaced by the continuation $\lambda v_2\,.\,v_2\ +\ 20$ which corresponds to the evaluation context $E_{v_3}[\cdot] \equiv [\cdot]\ +\ 20$ of step 3),

$$e_2 \equiv ((\lambda y\,.\,\lambda k\,.\,k\,5)\,10)\,(\lambda v_1\,.\,((\lambda x\,.\,\lambda k\,.\,k\,\boxed{(x\ +\ x)})\,v_1)\,(\lambda v_2\,.\,v_2\ +\ 20))$$

is transformed to

$$e_3 \equiv ((\lambda y\,.\,\lambda k\,.\,k\,5)\,10)\,(\lambda v_1\,.\,((\lambda x\,.\,\lambda k\,.\,(x\ +^k\ x)\,k)\,v_1)\,(\lambda v_2\,.\,v_2\ +\ 20)).$$

[5] The implications of each option are discussed in Chapter 2.

by simply passing k as the continuation.

Step 4: The component $v_2 + 20$ of e_3 corresponds to the term in the hole of the evaluation context for step 4. The transformation for this step requires that we pass a continuation to the operation $v_2 +^k 20$. But what is the continuation here? First, note that if e had appeared in some initial evaluation context E_i, then step 4 would appear as

$$E_i[10 + 20] \quad \stackrel{(4)}{\longmapsto_v} \quad E_i[30]$$

and the continuation-passing encoding of this step would be

$$\begin{array}{lll} (10 +^k 20)\,(\lambda v_4 \,.\, E_i[v_4]) & \longmapsto_v & (\lambda k \,.\, k\,30)\,(\lambda v_4 \,.\, E_i[v_4]) \\ & \longmapsto_v & (\lambda v_4 \,.\, E_i[v_4])\,30 \\ & \longmapsto_v & E_i[30] \end{array}$$

To allow for the fact that e may occur in an arbitrary evaluation context (and thus its continuation version may be passed an arbitrary continuation), we take e_3

$$e_3 \;\equiv\; ((\lambda y \,.\, \lambda k \,.\, k\,5)\,10)\,(\lambda v_1 \,.\, ((\lambda x \,.\, \lambda k \,.\, (x +^k x)\,k)\,v_1)\,(\lambda v_2 \,.\, v_2 + 20))$$

and parameterize it by a continuation to obtain

$$e_3' \;\equiv\; \lambda k \,.\, ((\lambda y \,.\, \lambda k \,.\, k\,5)\,10)\,(\lambda v_1 \,.\, ((\lambda x \,.\, \lambda k \,.\, (x +^k x)\,k)\,v_1)\,(\lambda v_2 \,.\, k\,\boxed{(v_2 + 20)}))$$

which is then transformed to

$$e_v \;\equiv\; \lambda k \,.\, ((\lambda y \,.\, \lambda k \,.\, k\,5)\,10)\,(\lambda v_1 \,.\, ((\lambda x \,.\, \lambda k \,.\, (x +^k x)\,k)\,v_1)\,(\lambda v_2 \,.\, (v_2 +^k 20)\,k))$$

In the previous section (at lines (1.9) and (1.10)), we showed how a continuation-passing encoding simulated a single call-by-value evaluation step. Having transformed the entire expression e into CPS, we now show how the complete call-by-value evaluation of e at line (1.11) is simulated by the evaluation of e_v.

Evaluating e_v requires that we supply an initial continuation — corresponding to the evaluation context in which e occurs. At line (1.11), we evaluated e in the empty context $[\cdot]$. Following the previous strategy of turning a context into a continuation gives $\lambda v_0 \,.\, [v_0]$ (which we can simply write as $\lambda v_0 \,.\, v_0$) as the initial continuation. The evaluation proceeds as follows (the labelled steps correspond to the labelled steps in the call-by-value evaluation of e at line (1.11)).

$$\begin{array}{lll} (\lambda k \,.\, ((\lambda y \,.\, \lambda k \,.\, k\,5)\,10)\,(\lambda v_1 \,.\, ((\lambda x \,.\, \lambda k \,.\, (x +^k x)\,k)\,v_1)\,(\lambda v_2 \,.\, (v_2 +^k 20)\,k)))\,(\lambda v_0 \,.\, v_0) & & (1.12) \\ \quad \longmapsto_{n,v} & ((\lambda y \,.\, \lambda k \,.\, k\,5)\,10)\,(\lambda v_1 \,.\, ((\lambda x \,.\, \lambda k \,.\, (x +^k x)\,k)\,v_1)\,(\lambda v_2 \,.\, (v_2 +^k 20)\,(\lambda v_0 \,.\, v_0))) & \\ \quad \stackrel{(1)}{\longmapsto_{n,v}} & (\lambda k \,.\, k\,5)\,(\lambda v_1 \,.\, ((\lambda x \,.\, \lambda k \,.\, (x +^k x)\,k)\,v_1)\,(\lambda v_2 \,.\, (v_2 +^k 20)\,(\lambda v_0 \,.\, v_0))) & \\ \quad \longmapsto_{n,v} & (\lambda v_1 \,.\, ((\lambda x \,.\, \lambda k \,.\, (x +^k x)\,k)\,v_1)\,(\lambda v_2 \,.\, (v_2 +^k 20)\,(\lambda v_0 \,.\, v_0)))\,5 & \\ \quad \longmapsto_{n,v} & ((\lambda x \,.\, \lambda k \,.\, (x +^k x)\,k)\,5)\,(\lambda v_2 \,.\, (v_2 +^k 20)\,(\lambda v_0 \,.\, v_0)) & \\ \quad \stackrel{(2)}{\longmapsto_{n,v}} & (\lambda k \,.\, (5 +^k 5)\,k)\,(\lambda v_2 \,.\, (v_2 +^k 20)\,(\lambda v_0 \,.\, v_0)) & \\ \quad \longmapsto_{n,v} & (5 +^k 5)\,(\lambda v_2 \,.\, (v_2 +^k 20)\,(\lambda v_0 \,.\, v_0)) & \\ \quad \stackrel{(3)}{\longmapsto_{n,v}} & (\lambda k \,.\, k\,10)\,(\lambda v_2 \,.\, (v_2 +^k 20)\,(\lambda v_0 \,.\, v_0)) & \\ \quad \longmapsto_{n,v} & (\lambda v_2 \,.\, (v_2 +^k 20)\,(\lambda v_0 \,.\, v_0))\,10 & \\ \quad \longmapsto_{n,v} & (10 +^k 20)\,(\lambda v_0 \,.\, v_0) & \\ \quad \stackrel{(4)}{\longmapsto_{n,v}} & (\lambda k \,.\, k\,30)\,(\lambda v_0 \,.\, v_0) & \\ \quad \longmapsto_{n,v} & (\lambda v_0 \,.\, v_0)\,30 & \\ \quad \longmapsto_{n,v} & 30 & \end{array}$$

Since continuations are used to explicitly encode the call-by-value strategy, both call-by-name and call-by-value evaluations proceed in lock-step fashion — in essence, the explicit encoding of control leaves no ambiguity regarding which program component to evaluate next. This form of evaluation-order independence is a general property of CPS terms.

Finally, in this example we have encoded the control pattern associated with the call-by-value strategy, but (omitting the details) we can encode the control pattern associated with call-by-name just as well:

$$e_n \equiv \lambda k\,.\,((\lambda x\,.\,\lambda k\,.\,x\,(\lambda v_1\,.\,x\,(\lambda v_2\,.\,(v_1\ +^k\ v_2)\,k)))\,(\lambda k\,.\,((\lambda y\,.\,\lambda k\,.\,k\,5)\,(\lambda k\,.\,k\,5))\,k))\,(\lambda v_3\,.\,(v_3\ +^k\ 20)\,k).$$

Since call-by-name factorings are different than call-by-value factorings (as discussed in the previous section), the structure of continuation-passing in e_n and e_v is also different. For this reason, a distinction is often made between *call-by-name continuation-passing style* (as exemplified by e_n) and *call-by-value continuation-passing style* (as exemplified by e_v). In fact, we will see that there are many different styles of continuation-passing.

CPS transformations

In one of the classic works on CPS, Plotkin [76] showed how programs could be rearranged into CPS automatically using program transformations called *CPS transformations*. Plotkin gave two transformations: a call-by-value CPS transformation $\mathcal{C}_v$ which encoded the call-by-value strategy into the structure of terms, and a call-by-name CPS transformation $\mathcal{C}_n$ which encoded the call-by-name strategy into the structure of terms. Plotkin proved that $\mathcal{C}_v$ allows call-by-name to simulate call-by-value, and that $\mathcal{C}_n$ allows call-by-value to simulate call-by-name. He also proved that the continuation-passing terms produced by $\mathcal{C}_v$ and $\mathcal{C}_n$ are evaluation-order independent. We will adopt these two properties of evaluation-order independence and simulation as the primary correctness criteria for CPS transformations.

1.2 Overview of the dissertation

The broad goal of our work is to address the issues of rigid structure, unexploited structural similarities, and lack of structural abstractions of continuation-passing styles stated at the outset.

1.2.1 Thesis statement

Our thesis is three-fold.

- Continuation-passing styles can be generalized so that their structure is no longer rigid but is instead tuned to take advantage of computational properties such as strictness and termination.
- Structural similarities can be exploited to present a unified framework for reasoning about continuation-passing styles.
- The complex structure of continuation-passing styles can be simplified by suitable abstractions of continuation-passing patterns.

1.2.2 Outline and contributions

Chapter 2 provides necessary technical background. The first part of the chapter defines the syntax, operational semantics, and program calculi for the languages used throughout the dissertation. The latter part gives definitions and properties associated with CPS.

Chapter 3 illustrates how an analysis of the structure of continuation-passing styles leads to simplifications. We begin the chapter by showing that the continuation-passing structure of Plotkin's call-by-name CPS transformation $\mathcal{C}_n$ is not necessary for achieving evaluation-order independence or simulation properties. Instead, these properties can be achieved using a simpler *thunk*-based transformation $\mathcal{T}$. This confirms the folk-theorem that thunks provide a simulation of call-by-name under call-by-value. In addition, we show that the thunk-based simulation $\mathcal{T}$ preserves and reflects the convertibility relation of the call-by-name calculus. The conclusion is that all the formal properties Plotkin gave for his continuation-based simulation $\mathcal{C}_n$ can be achieved using the simpler thunk-based transformation $\mathcal{T}$.

In the second part of the chapter, we show that the continuation-passing structure of $\mathcal{C}_n$ can actually be obtained by composing Plotkin's call-by-value CPS transformation $\mathcal{C}_v$ with the thunk transformation $\mathcal{T}$. As a result, most of the formal properties of $\mathcal{C}_n$ follow as corollaries of properties of $\mathcal{C}_v$ and $\mathcal{T}$. This often reduces reasoning about call-by-name and call-by-value CPS to reasoning about call-by-value CPS and thunks. We conclude the chapter by giving variations on thunks and applications of the factorization of $\mathcal{C}_n$ by $\mathcal{C}_v$ and $\mathcal{T}$.

Chapters 4 and 5 illustrate how CPS transformations can be optimized and generalized by taking into account computational properties such as strictness and termination.

In Chapter 4, we present a CPS transformation $\mathcal{C}_s$ directed by strictness information. This allows CPS encoding of call-by-name programs to be optimized in much the same way as the compilation of call-by-name programs is optimized using strictness information. The new transformation $\mathcal{C}_s$ is derived using the factorization of $\mathcal{C}_n$ through $\mathcal{C}_v$ and $\mathcal{T}$ given in Chapter 3. The correctness of $\mathcal{C}_s$ is a corollary of the correctness of the $\mathcal{C}_v$ and $\mathcal{T}$. $\mathcal{C}_s$ can be viewed as a generalization of $\mathcal{C}_n$ and $\mathcal{C}_v$: if all functions input are non-strict, the output of $\mathcal{C}_s$ corresponds to the output of $\mathcal{C}_n$; if all functions input are strict, the output of $\mathcal{C}_s$ corresponds to the output of $\mathcal{C}_v$.

$\mathcal{C}_s$ enables a new strategy for compiling call-by-name programs combining the traditional advantages of CPS (tail-recursive code, evaluation-order independence) and the usual benefits of strictness analysis (elimination of unnecessary thunks). Finally, we express $\mathcal{C}_s$ in Nielson and Nielson's two-level λ-calculus, enabling a simple and efficient implementation in a functional programming language.

In Chapter 5, we present a CPS transformation $\mathcal{C}_t$ directed by termination information. We use this transformation to automatically derive a realistic continuation-style denotational semantics specification from a direct-style denotational semantics specification for a simple imperative language. Similar attempts using existing CPS transformations yield unnatural continuation-style specifications — essentially because they introduce too many continuations. By taking termination properties into account, $\mathcal{C}_t$ introduces fewer continuations and thus simplifies the required continuation-passing structure. $\mathcal{C}_t$ can be viewed as a generalization of $\mathcal{C}_n$ and the identity transformation: if no term input can be guaranteed to terminate, the output of $\mathcal{C}_t$ corresponds to the output of $\mathcal{C}_n$; if all terms input can be guaranteed to terminate, the output of $\mathcal{C}_t$ corresponds to the output of the identity transformation. The chapter concludes with a discussion of how $\mathcal{C}_t$ can be applied to semantics-directed compilation and partial evaluation.

Chapters 6 and 7 unify previous work on CPS by exploiting structural similarities of continuation-passing styles.

In Chapter 6, we propose a generic framework for studying continuation-passing styles based on Moggi's computational meta-language. The framework allows generic formalizations of CPS transformations, proofs related to computational adequacy and program calculi, optimizations known as "administrative reductions", typing properties, and inverse transformations. Results for specific evaluation strategies follow as corollaries of the generic results. Because the extended type system of the meta-language directly captures computational properties, we can describe all aspects of the standard call-by-value and call-by-name CPS transformations as well as the variations proposed in the first part of the dissertation. Thus, previous work on CPS and CPS transformations is unified in a single framework.

In Chapter 7, we illustrate in detail how the generic framework can be applied to obtain standard CPS transformations as well as the new ones given in preceding chapters.

In Chapter 8, we assess the results presented, discuss related literature, and suggest directions for future work.

Chapter 2

Background

This chapter defines the syntax and semantics of languages used throughout the dissertation. We will consider both untyped and typed λ-term based languages. Our strategy is to designate an untyped core language Λ and a typed core language Λ^σ and study these in detail. We also define an extended typed language Λ^{σ^+} to be used for applications in Chapters 4 and 5. Since the core languages will be studied in depth, we feel free to abbreviate presentations for the extended language.

In Section 2.1 we introduce notation used to define each of these languages as well as the annotated languages presented in Chapters 4, 5, and 6. Section 2.2 gives syntax, operational semantics, and program calculi for the core language Λ of untyped λ-terms. Section 2.3 presents type assignment rules and associated properties for the typed core language Λ^σ. Section 2.4 presents the extended typed language Λ^{σ^+}. Section 2.5 reviews fundamental concepts and definitions associated with continuation-passing style. We assume familiarity with λ-calculi [6, 41].

2.1 Language notation and conventions

2.1.1 Terms

We use the term *language* in an informal sense to refer collectively to the terms, types, programs, *etc.* associated with a particular programming setting. For a given language l, *Terms*[l] denotes the set of terms associated with l. To simplify substitution we work with the quotient of l-terms under α-equivalence (*i.e.*, we identify terms which differ only in the names of bound variables) and follow Barendregt's variable convention: in terms occurring in definitions and proofs *etc.*, all bound variables are chosen to be different from free variables [6, p. 26]. We write $e_1 \equiv e_2$ when e_1 and e_2 are α-equivalent.

The notation $FV(e)$ denotes the set of free variables in e and $e[x_1 := e_1]$ denotes the result of the capture-free substitution of all free occurrences of x in e_1 by e_2. We will sometimes represent simultaneous substitutions $e[x_1 := e_1\ ,\ \ldots\ ,\ x_n := e_n]$ as the application of a substitution function $\phi : \mathit{Terms}[l] \rightarrow \mathit{Terms}[l]$ to the term e, *i.e.*, $\phi(e)$. A substitution ϕ is said to close a term e if $\phi(e)$ is a closed term.

A *context* C is a term with a *hole*, $[\cdot]$, in place of one subterm. The operation of *filling* the context C with a term e yields the term $C[e]$, possibly capturing some free variables of e in the process. The notation *Contexts*[l] denotes the set of contexts from some language l.

Programs

Programs of l, denoted *Progs*[l], are those l-terms that are suitable for evaluation using the operational semantics of l. The definition of *program* varies across the languages we consider, but in general only closed terms (terms with no free variables) are suitable for evaluation.

Values

Values of l, denoted *Values*[l], are terms that are *canonical* in that they are irreducible according to operational semantics of l. As such, the output of an evaluator for l will be canonical programs (*i.e.*, closed values).

$$
\begin{array}{lcl}
e & \in & \mathit{Terms}[\Lambda] \\
e & ::= & c \mid x \mid \lambda x\,.\,e \mid e_0\,e_1 \\
 & & \\
c & \in & \mathit{Constants} \\
x & \in & \mathit{Identifiers}
\end{array}
$$

Figure 2.1: The untyped core language Λ

2.1.2 Types

We use Curry-style type assignment rules to assign types to the untyped terms of a language l [7]. $\mathit{Types}[l]$ denotes the set of types associated with language l. The typing rules rely on *type assumptions* — finite sets $\Gamma = \{x_1 : \sigma_1, ..., x_n : \sigma_n\}$ of pairs associating identifiers x_i with types σ_i. We assume that the identifiers of Γ are distinct. This allows Γ to be viewed as a partial function from identifiers to types. $\Gamma, x : \sigma$ abbreviates $\Gamma \cup \{x : \sigma\}$. $\mathit{Assums}[l]$ denotes the set of type assumptions associated with language l. $\Gamma \vdash e : \sigma$ means e can be assigned type σ under assumptions Γ using the type assignment rules for language l.

In general, the type assignment systems we use can be viewed as an inductive definition of the type correct terms of l. $\mathit{TypedTerms}[l]$ denotes the set of type-correct terms inductively generated by the typing rules of l. When no confusion results, we simply write l for either $\mathit{Terms}[l]$ or $\mathit{TypedTerms}[l]$.

2.2 The untyped core language Λ

2.2.1 Syntax

Figure 2.1 presents the syntax of the untyped core language Λ of λ-terms. The language includes base (*i.e.*, non-functional) constants, identifiers (variables), λ-abstractions (functions), and function applications.

Constants and identifiers

Λ is parameterized by the denumerable sets $\mathit{Constants}$ and $\mathit{Identifiers}$. The meta-variable c ranges over constants while meta-variables x, y, and z range over $\mathit{Identifiers}$. Since we intend the same set of constants and identifiers to parameterize each of the languages we consider, we omit further references to these sets in language definitions.

Values

The sets $\mathit{Values}_n[\Lambda]$ and $\mathit{Values}_v[\Lambda]$ below represent the set of values from the language Λ under call-by-name and call-by-value semantics respectively.

$$
\begin{array}{lcll}
v & \in & \mathit{Values}_n[\Lambda] & \\
v & ::= & c \mid \lambda x\,.\,e & \ldots \mathit{where}\ e \in \mathit{Terms}[\Lambda].
\end{array}
$$

$$
\begin{array}{lcll}
v & \in & \mathit{Values}_v[\Lambda] & \\
v & ::= & c \mid x \mid \lambda x\,.\,e & \ldots \mathit{where}\ e \in \mathit{Terms}[\Lambda].
\end{array}
$$

Identifiers are included in $\mathit{Values}_v[\Lambda]$ since only values will be substituted for identifiers under call-by-value evaluation. We use v as a meta-variable for values and where no ambiguity results we will ignore the distinction between call-by-name and call-by-value values.

2.2.2 Operational semantics

This section defines call-by-name and call-by-value evaluators for Λ programs. We give three different notions of *evaluation equivalence*. Each of these will be used to characterize the correctness of program transformations.

Call-by-name:

$$(\lambda x\,.\,e_1)\,e_2 \longmapsto_n e_1[x := e_2] \qquad \frac{e_1 \longmapsto_n e_1'}{e_1\,e_2 \longmapsto_n e_1'\,e_2}$$

Call-by-value:

$$(\lambda x\,.\,e)\,v \longmapsto_v e[x := v] \qquad \frac{e_1 \longmapsto_v e_1'}{e_1\,e_2 \longmapsto_v e_1'\,e_2} \qquad \frac{e_2 \longmapsto_v e_2'}{(\lambda x\,.\,e_1)\,e_2 \longmapsto_v (\lambda x\,.\,e_1)\,e_2'}$$

Figure 2.2: Call-by-name and call-by-value single-step evaluation rules for Λ

We then derive a notion of term meaning (with respect to a particular evaluation strategy) based on how terms behave under different experiments an evaluator.

Program Evaluation

Figure 2.2 presents single-step evaluation rules that capture the call-by-name and call-by-value evaluation strategies on Λ programs. The evaluators $eval_n$ and $eval_v$ are defined in terms of the reflexive, transitive closure (denoted $\longmapsto^*$) of single-step evaluation.

Definition 2.1 (Program evaluation) *For all* $e \in Progs[\Lambda]$,

$$eval_n(e) = v \quad \textit{iff} \quad e \longmapsto_n^* v$$
$$eval_v(e) = v \quad \textit{iff} \quad e \longmapsto_v^* v$$

In the evaluators used in this dissertation, at most one evaluation step is applicable for any given program. Thus, the $eval_n$ and $eval_v$ define (partial) functions.

We write $eval(e)\!\downarrow$ (pronounced "the evaluation of e is defined", or "the evaluation of e is terminates") if there exists a value v such that $e \longmapsto^* v$. We write $eval(e)\!\uparrow$ (pronounced "the evaluation of e is undefined") if there does not exist a value v such that $e \longmapsto^* v$. We write $e\!\downarrow_n$ for a possibly open term e if for all substitutions ϕ that close e, $eval_n(\phi(e))\!\downarrow$. Similarly for $e\!\downarrow_v$.

The evaluation functions may be partial (*i.e.*, $eval(e)$ may be undefined for a term e) for two reasons:

1. e is the origin of an infinite evaluation sequence, *i.e.*, $e \longmapsto e_0 \longmapsto e_1 \longmapsto \ldots$,
2. e is the origin of an evaluation sequence which ends in a *stuck* term — a non-value which cannot be further reduced (*e.g.*, the application of a constant to some argument).

The following definition gives programs that are stuck under call-by-name and call-by-value evaluation.

$$\begin{array}{lcll} s & \in & Stuck_n[\Lambda] & \\ s & ::= & c\,e \mid s\,e & \ldots \textit{where } e \in Terms[\Lambda]. \end{array}$$

$$\begin{array}{lcll} s & \in & Stuck_v[\Lambda] & \\ s & ::= & c\,e \mid s\,e \mid (\lambda x\,.\,e)\,s & \ldots \textit{where } e \in Terms[\Lambda]. \end{array}$$

The correctness of the definition is a consequence of the following property.

Property 2.1 (Single-step evaluation characteristics for Λ) *For all* $e \in Progs[\Lambda]$,

Call-by-name: *exactly one of the following holds:* $e \longmapsto_n e'$ *or* $e \in Values_n[\Lambda]$ *or* $e \in Stuck_n[\Lambda]$

Call-by-value: *exactly one of the following holds:* $e \longmapsto_v e'$ *or* $e \in Values_v[\Lambda]$ *or* $e \in Stuck_v[\Lambda]$

Proof: by induction over the structure of e. (See [76, p. 151]) ∎

The properties of program evaluation outlined above can now be formalized as follows.

Property 2.2 (Program evaluation characteristics for Λ) *For all $e \in Progs[\Lambda]$,*

Call-by-name: *exactly one of the following holds: either $e \longmapsto_n^* v \in Values_n[\Lambda]$, or $e \longmapsto_n^* s \in Stuck_n[\Lambda]$, or for all e_m such that $e \longmapsto_n^* e_m$, there exists an e_{m+1} such that $e_m \longmapsto_n e_{m+1}$ (i.e., there is an infinite evaluation sequence beginning with e).*

Call-by-value: *exactly one of the following holds: either $e \longmapsto_v^* v \in Values_v[\Lambda]$, or $e \longmapsto_n^* s \in Stuck_v[\Lambda]$, or for all e_m such that $e \longmapsto_v^* e_m$, there exists an e_{m+1} such that $e_m \longmapsto_v e_{m+1}$ (i.e., there is an infinite evaluation sequence beginning with e).*

Evaluation equivalence

Having defined an evaluator for Λ programs, a semantics for arbitrary Λ-terms can be obtained by associating term *meaning* with *behavior* under evaluation. With this view, a term's meaning is revealed by subjecting it to evaluation experiments, *i.e.*, by supplying it to an evaluator in different program contexts. This requires that we consider what it means for two evaluations to be equivalent. In fact, there are several reasonable notions — depending on the degree of distinction we wish to make in the output of our evaluators.

Since our evaluators yield canonical programs (which are terms) as output, term equivalence is an acceptable basis for evaluation equivalence. Thus far, the only term equivalence discussed has been α-equivalence. In the following section we introduce program calculi. Therefore, another view is to consider two terms to be equivalent if they are convertible in a certain calculus (*e.g.*, if they are β-convertible).

Both of these views are rather intensional since they relate the structure of terms. In fact, most existing language implementations do not allow the term structure of all program results to be distinguished. Instead, distinguishable results are limited to basic data objects *e.g.*, numerals. In our case, this corresponds to limiting distinguishable results to terms $c \in Constants$. For other types of results (*i.e.*, abstractions), one can only observe that evaluation has terminated. We express this by defining a set of observations

$$Observations \stackrel{\text{def}}{=} Constants \cup \{\top\}$$

where $\top$ denotes the observation that evaluation has terminated with a non-constant result. The following function collapses arbitrary results to observable results.

$$\begin{aligned} Obs &: Values[\Lambda] \rightarrow Observations \\ Obs(c) &= c \\ Obs(\lambda x\,.\,e) &= \top \end{aligned}$$

With this view, the meaning of a program that yields a constant is immediately clear. However, the full meaning of a program e that results in observation $\top$ can only be revealed by additional experiments that place e in different program contexts. This gives an extensional view of equivalence based on term *behavior* as opposed to the intensional view based on term *structure* given above.

Up to this point, we have only considered the equivalence of evaluations that terminate. We adopt a notion of *evaluation approximation* to express relationships between evaluations that may be undefined. Intuitively, $eval(e_1)$ approximates $eval(e_2)$ if $eval(e_1)\downarrow$ implies (1) $eval(e_2)\downarrow$, and (2) the result given by $eval(e_1)$ is equivalent to the result given by $eval(e_2)$. In such a case, $eval(e_1)$ is generally less informative than $eval(e_2)$ since $eval(e_1)$ may be undefined when $eval(e_2)$ is defined. Yet, when $eval(e_1)$ (and thus $eval(e_2)$) is defined, the results produced are equivalent.

We formalize three notions of evaluation approximation based on the forms of term equivalence discussed above.

Definition 2.2 *Given meta-language expressions E_1 and E_2 where one or both may be undefined, $E_1 \preceq E_2$ iff $E_1\downarrow$ implies $E_2\downarrow$ and $E_1 \equiv E_2$.*

Here, E_1 and E_2 are meta-language expressions of evaluation (*e.g.*, $eval_n(e)$). Therefore, when we write $E_1 \equiv E_2$, we mean the terms denoted by E_1 and E_2 are α-equivalent.

Next, we give a definition based on r-convertibility where where r is a notion of reduction defining a program calculus (as presented in the following section).

Definition 2.3 *Given meta-language expressions E_1 and E_2 where one or both may be undefined, $E_1 \preceq_r E_2$ iff $E_1\downarrow$ implies $E_2\downarrow$ and $E_1 =_r E_2$.*

Finally, we give a definition based on equivalence of observable results.

Definition 2.4 *Given meta-language expressions E_1 and E_2 where one or both may be undefined, $E_1 \preceq_{obs} E_2$ iff $E_1\downarrow$ implies $E_2\downarrow$ and $Obs(E_1) = Obs(E_2)$.*

Each of the forms of evaluation approximation gives rise to a notion of evaluation equivalence.

- Evaluations E_1 and E_2 are equivalent up to α-equivalence (denoted $E_1 \simeq E_2$) if $E_1 \preceq E_2$ and $E_2 \preceq E_1$.
- Evaluations E_1 and E_2 are equivalent up to r-convertibility (denoted $E_1 \simeq_r E_2$) if $E_1 \preceq_r E_2$ and $E_2 \preceq_r E_1$.
- Evaluations E_1 and E_2 are equivalent up to observations (denoted $E_1 \simeq_{obs} E_2$) if $E_1 \preceq_{obs} E_2$ and $E_2 \preceq_{obs} E_1$.

Term meaning

A notion of term meaning can be defined based on how terms behave with respect all possible evaluation experiments.

Definition 2.5 (Call-by-name observational approximation for Λ) *For $e_1, e_2 \in \Lambda$, $e_1 \sqsubseteq_n e_2$ iff for all contexts $C \in Contexts[\Lambda]$ such that $C[e_1]$ and $C[e_2]$ are programs, $eval_n(C[e_1]) \preceq_{obs} eval_n(C[e_2])$.*

Definition 2.6 (Call-by-value observational approximation for Λ) *For $e_1, e_2 \in \Lambda$, $e_1 \sqsubseteq_v e_2$ iff for all contexts $C \in Contexts[\Lambda]$ such that $C[e_1]$ and $C[e_2]$ are programs, $eval_v(C[e_1]) \preceq_{obs} eval_v(C[e_2])$.*

Terms e_1 and e_2 are said to be *observationally equivalent with respect to call-by-name evaluation* (denoted $e_1 \approx_n e_2$), if both $e_1 \sqsubseteq_n e_2$ and $e_2 \sqsubseteq_n e_1$. Similarly for $e_1 \approx_v e_2$.

Our definitions of observational equivalence are equivalent to the definitions given by Plotkin [76, pp. 144–147]. For example, Plotkin's definition of call-by-name observation equivalence is as follows.

> For $e_1, e_2 \in \Lambda$, $e_1 \approx_n e_2$ *iff* for any context $C \in Contexts[\Lambda]$ such that $C[e_1]$ and $C[e_2]$ are programs, $eval_n(C[e_1])$ and $eval_n(C[e_2])$ are either both undefined, or else both defined and one is a given base constant c *iff* the other is.

2.2.3 Program calculi

Reduction relations

We present program calculi for λ-terms *via* reduction relations. A *notion of reduction* r (written $\rightsquigarrow_r$ in infix form) for language l is a binary relation on $Terms[l]$. For Λ we consider the following standard notions of reduction [6, 76].

$$\begin{array}{rcll} (\lambda x\,.\,e_1)\,e_2 & \rightsquigarrow_\beta & e_1[x := e_2] & (\beta) \\ (\lambda x\,.\,e)\,v & \rightsquigarrow_{\beta_v} & e[x := v] \quad v \in Values_v[\Lambda] & (\beta_v) \\ \lambda x\,.\,e\,x & \rightsquigarrow_\eta & e \quad x \notin FV(e) & (\eta) \end{array}$$

If $\rightsquigarrow_r$ and $\rightsquigarrow_s$ are notions of reduction on language l then the relation $\rightsquigarrow_r \cup \rightsquigarrow_s$ (denoted $\rightsquigarrow_{rs}$) is also a notion of reduction on l.

For a given notion of reduction $\rightsquigarrow_r$ on language l, the compatible closure of $\rightsquigarrow_r$ (denoted $\longrightarrow_r$), the reflexive, transitive closure of $\longrightarrow_r$ (denoted $\longrightarrow\!\!\!\!\!\rightarrow_r$), and the equivalence relation generated by $\longrightarrow\!\!\!\!\!\rightarrow_r$ (denoted $=_r$), are defined inductively as follows where $C \in Contexts[l]$ [6, p. 51].

$$e_0 \rightsquigarrow_r e_1 \quad \Rightarrow \quad C[e_0] \longrightarrow_r C[e_1]$$

$$\begin{array}{rcl} & & e \longrightarrow\!\!\!\!\!\rightarrow_r e \\ e_0 \longrightarrow_r e_1 & \Rightarrow & e_0 \longrightarrow\!\!\!\!\!\rightarrow_r e_1 \\ e_0 \longrightarrow\!\!\!\!\!\rightarrow_r e_1, e_1 \longrightarrow_r e_2 & \Rightarrow & e_0 \longrightarrow\!\!\!\!\!\rightarrow_r e_2 \end{array}$$

$$\begin{array}{rcl} e_0 \longrightarrow\!\!\!\!\!\rightarrow_r e_1 & \Rightarrow & e_0 =_r e_1 \\ e_0 =_r e_1 & \Rightarrow & e_1 =_r e_0 \\ e_0 =_r e_1, e_1 =_r e_2 & \Rightarrow & e_0 =_r e_2 \end{array}$$

The relations are pronounced as follows.

$e_1 \longrightarrow_r e_2$: e_1 r-reduces to e_2 in one step
$e_1 \longrightarrow\!\!\!\!\!\rightarrow_r e_2$: e_1 r-reduces to e_2
$e_1 =_r e_2$: e_1 and e_2 are r-convertible, or alternatively, r-equivalent

If $e \rightsquigarrow_r e'$ then e is an *r-redex* and e' is an *r-contractum*. A term e is an *r-normal form* if e does not contain an r-redex as a subterm. For a given notion of reduction r, we refer to the relation $=_r$ as the *calculus generated by r*, or simply, the *r-calculus*, or the theory λr.

Definition 2.7 (Church-Rosser) *A notion of reduction r on language l is* Church-Rosser *if for all $e \in Terms[l]$, $e_1 =_r e_2$ implies the existence of some $e_3 \in Terms[l]$ such that $e_1 \longrightarrow\!\!\!\!\!\rightarrow_r e_3$ and $e_2 \longrightarrow\!\!\!\!\!\rightarrow_r e_3$.*

Barendregt [6, pp. 59–67] gives detailed proofs of Church-Rosser properties for β and η. Plotkin [76, p. 135] states the Church-Rosser property for β_v.

Definition 2.8 (Compatibility) *A binary relation r on language l is* compatible *(with the constructs of language l) if for all $e_1, e_2 \in Terms[l]$ and all contexts $C \in Contexts[l]$,*

$$e_1 \; r \; e_2 \quad \Rightarrow \quad C[e_1] \; r \; C[e_2]$$

Property 2.3 *For any notion of reduction r on language l, the relations $\longrightarrow_r$, $\longrightarrow\!\!\!\!\!\rightarrow_r$, and $=_r$ generated by r are compatible with respect to l [6, p. 52].*

Definition 2.9 (Substitutivity) *A binary relation r on language l is* substitutive *if for all $e_1, e_2, e_3 \in Terms[l]$ and all variables x,*

$$e_1 \; r \; e_2 \quad \Rightarrow \quad e_1[x := e_3] \; r \; e_2[x := e_3]$$

Unlike compatibility (which follows directly from the definitions of $\longrightarrow_r$, $\longrightarrow\!\!\!\!\!\rightarrow_r$, and $=_r$), substitutivity is sensitive to the side conditions on reduction definitions (*e.g.*, the conditions on values in β_v).

Property 2.4 *For reductions β, β_v, η,*

- *relations (e.g., $\longrightarrow$, $\longrightarrow\!\!\!\!\!\rightarrow$, and $=$) generated from β and η are substitutive.*
- *relations (e.g., $\longrightarrow$, $\longrightarrow\!\!\!\!\!\rightarrow$, and $=$) generated from β_v are* value-*substitutive (i.e., in the definition for substitutivity, e_3 is only allowed to range over $Values_v[\Lambda]$).*

Proof: The proof for relations generated from β can be found in [6, p. 55]. The proofs for relations generated from η and β_v can be obtained in a similar manner. ∎

We follow conventional notation and write $\lambda r \vdash e_1 = e_2$ when $e_1 =_r e_2$ and similarly for $\longrightarrow_r$ and $\longrightarrow\!\!\!\!\!\rightarrow_r$. The subscripted notation (*e.g.*, $=_r$, $\longrightarrow_r$, $\longrightarrow\!\!\!\!\!\rightarrow_r$) will generally be reserved for calculational style proofs etc.

We typically use the β-calculus when reasoning about call-by-name evaluation and the β_v-calculus when reasoning about call-by-value evaluation. However, in some cases the terms under consideration will be evaluation-order independent and it will be convenient to state that a property P holds for β_a when P holds for both β and β_v calculi. Similarly, we state that a property P holds for $eval_a$ if P holds for both $eval_n$ and $eval_v$.

Reasoning about programs

The calculi of the preceding section can be used to reason about term behavior. To establish the observational equivalence two terms, it is sufficient to show that the terms are convertible in an appropriate calculus.

Theorem 2.1 (Soundness of calculi for Λ) *For $e_1, e_2 \in \Lambda$,*

$$\lambda\beta \vdash e_1 = e_2 \quad \Rightarrow \quad e_1 \approx_n e_2$$
$$\lambda\beta_v \vdash e_1 = e_2 \quad \Rightarrow \quad e_1 \approx_v e_2$$

Proof: See [76, pp. 144,147] ∎

In many presentations of λ-calculi where meaning is based on *normal-forms* or *head-normal-forms* [6, p. 25–43], η is used to express extensionality. However, the operational semantics presented here captures the behavior of most practical applications which evaluate to *weak-head-normal-form*. η-reduction is unsound in these settings because it does not preserve the property of having a weak-head-normal-form. Operationally speaking, η does not preserve termination properties. For example, the reduction

$$\lambda x\,.\,\Omega\,x \longrightarrow_\eta \Omega$$

(where $\Omega \equiv (\lambda x\,.\,x\,x)\,(\lambda x\,.\,x\,x)$) contracts a weak-head-normal-form to term which has no weak-head-normal-form. Operationally, $eval_n(\lambda x\,.\,\Omega\,x)\!\downarrow$ whereas $eval_n(\Omega)\!\uparrow$. η is unsound for reasoning about terms under call-by-value evaluation for similar reasons.

A termination-predicated form of η-reduction which does maintain the property of having a weak-head-normal-form is suggested by Abramsky [2, p. 71]. Adapting this idea to our setting gives the following reductions.

$$\lambda x\,.\,e\,x \quad \rightsquigarrow_{\eta\downarrow n} \quad e \qquad x \notin FV(e) \ \wedge\ e\!\downarrow_n \qquad (\eta\!\downarrow\! n)$$
$$\lambda x\,.\,e\,x \quad \rightsquigarrow_{\eta\downarrow v} \quad e \qquad x \notin FV(e) \ \wedge\ e\!\downarrow_v \qquad (\eta\!\downarrow\! v)$$

A more common approach is to require the η-contractum to be a value.

$$\lambda x\,.\,v\,x \quad \rightsquigarrow_{\eta_n} \quad v \qquad x \notin FV(v) \ \wedge\ v \in Values_n[\Lambda] \qquad (\eta_n)$$
$$\lambda x\,.\,v\,x \quad \rightsquigarrow_{\eta_v} \quad v \qquad x \notin FV(v) \ \wedge\ v \in Values_v[\Lambda] \qquad (\eta_v)$$

Although we will show both the termination-predicated and value-predicated forms of η-reduction to be sound in type settings, they are unsound for the untyped language Λ due to "improper" uses of constants.[1] For example,

$$\lambda x\,.\,c\,x \longrightarrow_{\eta_v} c$$

but $eval_v(\lambda x\,.\,c\,x)$ yields observation $\top$ whereas $eval_v(c)$ yields observation c. The same counterexample applies to η_n and to $\eta_{\downarrow n}$ and $\eta_{\downarrow v}$ since η_n, η_v are instances of $\eta_{\downarrow n}$,$\eta_{\downarrow v}$ respectively. Given these problems, we are forced to consider soundness of η-reduction in untyped settings on a case-by-case basis.

Finally, we note that β is unsound for reasoning about Λ programs under call-by-value evaluation since termination properties are not preserved. For example,

$$(\lambda x\,.\,c)\,\Omega \longrightarrow_\beta c$$

but $(\lambda x\,.\,c)\,\Omega \not\approx_v c$ since $eval_v((\lambda x\,.\,c)\,\Omega)\!\uparrow$ whereas $eval_v(c) = c$. However, β_v is sound for reasoning about Λ programs under call-by-name evaluation since every β_v-redex is also a β-redex.

2.3 The typed core language Λ^σ

2.3.1 Syntax

The syntax of the typed core language Λ^σ (Figure 2.3) is obtained by adding recursive abstractions $rec\,f(x).\,e$ to the syntax of Λ (Figure 2.1). We introduce a new countable set of identifiers $f \in RecIdentifiers$ disjoint from *Identifiers*. The distinction between kinds of identifiers is important under call-by-name evaluation; recursive identifiers only bind to recursive abstractions (values) whereas non-recursive identifiers may bind to non-values. As a result, we will see that recursive and non-recursive identifiers must be treated differently in the call-by-name CPS transformation.

[1] Abramsky [2, p. 71] states that $\eta_{\downarrow n}$ is sound for the untyped language $e ::= x \mid \lambda x\,.\,e \mid e_0\,e_1$ which excludes constants.

$$
\begin{array}{lcl}
e & \in & \mathit{Terms}[\Lambda^\sigma] \\
e & ::= & \ldots \mid f \mid \mathit{rec}\, f(x).\, e \\
 & & \\
f & \in & \mathit{RecIdentifiers}
\end{array}
$$

Figure 2.3: The typed core language Λ^σ

Values

The call-by-name and call-by-value value sets are extended as follows.

$$
\begin{array}{lcll}
v & \in & \mathit{Values}_n[\Lambda^\sigma] & \\
v & ::= & \ldots \mid f \mid \mathit{rec}\, f(x).\, e & \ldots \mathit{where}\ e \in \mathit{Terms}[\Lambda^\sigma].
\end{array}
$$

$$
\begin{array}{lcll}
v & \in & \mathit{Values}_v[\Lambda^\sigma] & \\
v & ::= & \ldots \mid f \mid \mathit{rec}\, f(x).\, e & \ldots \mathit{where}\ e \in \mathit{Terms}[\Lambda^\sigma].
\end{array}
$$

Typing rules for Λ^σ

Figure 2.4 presents Curry-style type assignment rules for Λ^σ. $\mathit{TypedTerms}[\Lambda^\sigma]$ denotes the set of type-correct terms inductively generated by the rules of Figure 2.4. When we refer to a Λ^σ-term e, we assume that e is type-correct (*i.e.*, $e \in \mathit{TypedTerms}[\Lambda^\sigma]$) unless stated otherwise. $\mathit{TypingDerivations}[\Lambda^\sigma]$ denotes the set of typing derivations inductively generated by the rules of Figure 2.4. When a typing derivation $d \in \mathit{TypingDerivations}[\Lambda^\sigma]$ ends in a judgement $\Gamma \vdash e : \sigma$, we write $d\{\Gamma \vdash e : \sigma\}$. This will be shortened to $d\{e : \sigma\}$ if we have no interest in the particular set of assumptions Γ.

We use a linear notation for typing derivations to simplify type-setting in proofs and definitions. The linear version of a typing derivation has the form

$$\mathit{rule\text{-}name} :: \mathit{antecedent\ derivations} :: \mathit{succedent}.$$

For example, a derivation consisting of only *(const)* rule of Figure 2.4 is written

$$\mathit{(const)} :: \Gamma \vdash c : \iota$$

while a derivation ending with the *(app)* rule of Figure 2.4 is written

$$\mathit{(app)} :: d_0\{\Gamma \vdash e_0 : \sigma_1 \rightarrow \sigma_2\}, d_1\{\Gamma \vdash e_1 : \sigma_1\} :: \Gamma \vdash e_0\, e_1 : \sigma_2.$$

In cases when the antecedents are clear from the context, the derivation is abbreviated as

$$\mathit{(app)} :: \Gamma \vdash e_0\, e_1 : \sigma_2.$$

2.3.2 Operational semantics

The single-step evaluation rules for Λ^σ are obtained by adding the rules for recursive abstractions given in Figure 2.5 to the rules for Λ given in Figure 2.2. As before, evaluation functions eval_n and eval_v are defined in terms of the reflexive, transitive closure (denoted $\longmapsto^*$) of single-step evaluation.

In describing the evaluation of untyped terms, we noted that eval_n and eval_v may be undefined if a *stuck* term e (*e.g.*, $e \equiv c\, c$) is encountered in an evaluation sequence. The following properties state that the typing rules for Λ^σ disallow such terms.

Property 2.5 (Single-step evaluation characteristics for Λ^σ) *For all* $\cdot \vdash e : \sigma$,

$$\textit{(const)} \qquad \Gamma \vdash c : \iota$$

$$\textit{(var)} \qquad \Gamma \vdash x : \Gamma(x)$$

$$\textit{(recvar)} \qquad \Gamma \vdash f : \Gamma(f)$$

$$\textit{(abs)} \qquad \frac{\Gamma, x:\sigma_1 \vdash e : \sigma_2}{\Gamma \vdash \lambda x\,.\,e : \sigma_1 \rightarrow \sigma_2}$$

$$\textit{(rec)} \qquad \frac{\Gamma, x:\sigma_1, f:\sigma_1 \rightarrow \sigma_2 \vdash e : \sigma_2}{\Gamma \vdash \mathit{rec}\, f(x).\, e : \sigma_1 \rightarrow \sigma_2}$$

$$\textit{(app)} \qquad \frac{\Gamma \vdash e_0 : \sigma_1 \rightarrow \sigma_2 \qquad \Gamma \vdash e_1 : \sigma_1}{\Gamma \vdash e_0\, e_1 : \sigma_2}$$

$$\begin{array}{lcl} \sigma & \in & \mathit{Types}[\Lambda^\sigma] \\ \sigma & ::= & \iota \mid \sigma_1 \rightarrow \sigma_2 \end{array}$$

$$\begin{array}{lcl} \Gamma & \in & \mathit{Assums}[\Lambda^\sigma] \\ \Gamma & ::= & \cdot \mid \Gamma, x:\sigma \mid \Gamma, f:\sigma \end{array}$$

Figure 2.4: Typing rules for Λ^σ

Call-by-name:

$$(\mathit{rec}\, f(x).\, e_1)\, e_2 \longmapsto_n e_1[f := \mathit{rec}\, f(x).\, e_1\,,\, x := e_2]$$

Call-by-value:

$$\frac{e_2 \longmapsto_v e_2'}{(\mathit{rec}\, f(x).\, e_1)\, e_2 \longmapsto_v (\mathit{rec}\, f(x).\, e_1)\, e_2'} \qquad (\mathit{rec}\, f(x).\, e_1)\, v_2 \longmapsto_v e_1[f := \mathit{rec}\, f(x).\, e_1\,,\, x := v_2]$$

Figure 2.5: Call-by-name and call-by-value single-step evaluation rules for Λ^σ

Call-by-name: *exactly one of the following holds:* $e \longmapsto_n e'$ *or* $e \in \mathit{Values}_n[\Lambda^\sigma]$

Call-by-value: *exactly one of the following holds:* $e \longmapsto_v e'$ *or* $e \in \mathit{Values}_v[\Lambda^\sigma]$

Proof: by induction on the structure of e. The proof for call-by-name is given below.

case $e \equiv c, \lambda x\,.\,e, \mathit{rec}\, f(x).\,e$: immediate, since $e \in \mathit{Values}_n[\Lambda^\sigma]$:

case $e \equiv x, f$: disallowed since the property only applies to closed terms.

case $e \equiv e_0\, e_1$: by ind. hyp., $e_0 \in \mathit{Values}_n[\Lambda^\sigma]$ or $e_0 \longmapsto_n e_0'$.

case $e_0 \in \mathit{Values}_n[\Lambda^\sigma]$:

case $e_0 \equiv \lambda x\,.\,e_a$: $(\lambda x\,.\,e_a)\, e_1 \longmapsto_n e_a[x := e_1]$

case $e_0 \equiv \mathit{rec}\, f(x).\,e_a$: $(\mathit{rec}\, f(x).\,e_a)\, e_1 \longmapsto_n e_a[f := \mathit{rec}\, f(x).\,e_a\,,\, x := e_1]$

case $e_0 \equiv f$: disallowed since the property only applies to closed terms

case $e_0 \longmapsto_n e_0'$: then $e_0\, e_1 \longmapsto_n e_0'\, e_1$

∎

Therefore, for typed programs e, $\mathit{eval}(e)$ is undefined only if e is the origin of an infinite evaluation sequence.

Property 2.6 (Program evaluation characteristics for Λ^σ) *For all* $\cdot \vdash e : \sigma$,

Call-by-name: *exactly one of the following holds:* $e \longmapsto_n^* v$, *or for all* e_m *such that* $e \longmapsto_n^* e_m$, *there exists an* e_{m+1} *such that* $e_m \longmapsto_n e_{m+1}$ *(i.e., there is an infinite evaluation sequence beginning with* e*).*

Call-by-value: *exactly one of the following holds:* $e \longmapsto_v^* v$, *or for all* e_m *such that* $e \longmapsto_v^* e_m$, *there exists an* e_{m+1} *such that* $e_m \longmapsto_v e_{m+1}$ *(i.e., there is an infinite evaluation sequence beginning with* e*).*

The definitions for evaluation equivalence carry over in the expected way to Λ^σ. As was the case with abstractions, we only allow observations of termination for recursive abstractions, *i.e.*,

$$\begin{array}{rcl} \mathit{Obs} & : & \mathit{Values}[\Lambda^\sigma] \to \mathit{Observations} \\ & \cdots & \\ \mathit{Obs}(\mathit{rec}\, f(x).\,e) & = & \top. \end{array}$$

Definitions of observational approximation and equivalence carry over as expected.

2.3.3 Program calculi

Reduction relations

For reasoning about Λ^σ terms we introduce the following notions of reduction for recursive abstractions.

$$\begin{array}{lll} (\mathit{rec}\, f(x).\,e_0)\, e_1 \;\leadsto_{\mathrm{rec}}\; e_0[f := \mathit{rec}\, f(x).\,e_0\,,\, x := e_1] & & (\mathrm{rec}) \\ (\mathit{rec}\, f(x).\,e_0)\, v_1 \;\leadsto_{\mathrm{rec}_v}\; e_0[f := \mathit{rec}\, f(x).\,e_0\,,\, x := v_1] & & (\mathrm{rec}_v) \end{array}$$

Reduction relations and calculi are obtained as in Section 2.2.3. The Compatibility, Substitutivity, Church-Rosser properties extend in the expected way to the above reductions. All the reductions considered thus far obey a subject-reduction property, *i.e.*, Λ^σ typings are preserved by the reductions.

Reasoning about programs

The following theorem specifies calculi for reasoning about Λ^σ programs.

Theorem 2.2 (Soundness of calculi for Λ^σ) *For* $e_1, e_2 \in \Lambda^\sigma$,

$$\begin{array}{rcl} \lambda\beta\mathrm{rec}\,\eta_{\downarrow n} \vdash e_1 = e_2 & \Rightarrow & e_1 \approx_n e_2 \\ \lambda\beta_v\mathrm{rec}_v\,\eta_{\downarrow v} \vdash e_1 = e_2 & \Rightarrow & e_1 \approx_v e_2 \end{array}$$

$$
\begin{array}{lcl}
e & \in & \mathit{Terms}[\Lambda^{\sigma^+}] \\
e & ::= & \ldots \mid \mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \mid \mathit{let}\ x \Leftarrow e_1\ \mathit{in}\ e_2 \mid \\
 & & \mathit{pair}\ e_1\ e_2 \mid \mathit{proj}_i\ e \mid \mathit{inj}_i\ e \mid \mathit{case}\ e_0\ \mathit{of}\ (x_1.e_1) \mid (x_2.e_2)
\end{array}
$$

Figure 2.6: The extended typed language Λ^{σ^+}

Proof: (This requires a long series of proofs which ends in a result similar to the *Context Lemma* for PCF (see [37]). This result is then used to establish the soundness. These proofs remain to be typed in.) ∎

It follows that η_n and η_v are sound for reasoning about Λ^σ programs under call-by-name and call-by-value evaluation, respectively.

2.4 The extended typed language Λ^{σ^+}

For several applications, we will want to consider languages richer than Λ^σ. This section shows how Λ^σ can be extended with conditional expressions, eager binding constructs, products (pairs), and co-products (sums).

2.4.1 Syntax

Terms of Λ^{σ^+} are obtained by adding the constructs of Figure 2.6 to the terms of Λ^σ (Figure 2.3). The types, assumptions, and typing rules for Λ^{σ^+} are obtained by adding the contents of Figure 2.7 to the types, assumptions, and typing rules of Λ^σ (Figure 2.4). Note that a type index is required to properly type the injection $\mathit{inj}_i^{\sigma_1+\sigma_2}\ e$. However, the indexing will be dropped when no ambiguity results.

Values

The call-by-name and call-by-value value sets are extended as follows.

$$
\begin{array}{lcll}
v & \in & \mathit{Values}_n[\Lambda^{\sigma^+}] & \\
v & ::= & \ldots \mid \mathit{pair}\ e_1\ e_2 \mid \mathit{inj}_i\ e & \ldots \mathit{where}\ e \in \mathit{Terms}[\Lambda^{\sigma^+}].
\end{array}
$$

$$
\begin{array}{lcll}
v & \in & \mathit{Values}_v[\Lambda^{\sigma^+}] & \\
v & ::= & \ldots \mid \mathit{pair}\ v_1\ v_2 \mid \mathit{inj}_i\ v & \ldots \mathit{where}\ e \in \mathit{Terms}[\Lambda^{\sigma^+}]\ \mathit{and}\ v, v_1, v_2 \in \mathit{Values}_v[\Lambda^{\sigma^+}].
\end{array}
$$

The definition of values for the pairing and injection constructors reveals that these will be interpreted *lazily* under call-by-name evaluation and *eagerly* under call-by-value.

2.4.2 Operational semantics

The single-step evaluation rules for Λ^{σ^+} are obtained by adding the rules of Figure 2.8 to the rules for Λ^σ (Figure 2.5). For conditional evaluation, we assume a distinguished element $\mathsf{c}_t \in \mathit{Constants}$. The first argument of the eager binding construct *let* is evaluated eagerly under both call-by-name and call-by-value. Pairing and injection constructs are evaluated lazily under call-by-name and eagerly under call-by-value. The definition of evaluators eval_n and eval_v, evaluation characteristics, and definitions of observational equivalence extend to the expanded language in the obvious ways. In keeping with our convention of only allowing full observation of constants, we extend the definition of *Obs* as follows.

$$
\begin{array}{rcl}
\mathit{Obs} & : & \mathit{Values}[\Lambda^{\sigma^+}] \rightarrow \mathit{Observations} \\
 & \ldots & \\
\mathit{Obs}(\mathit{inj}_i\ e) & = & \top
\end{array}
$$

$$(cnd) \qquad \frac{\Gamma \vdash e_0 : \iota \qquad \Gamma \vdash e_1 : \sigma \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash \mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2 : \sigma}$$

$$(let) \qquad \frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x{:}\sigma_1 \vdash e_2 : \sigma}{\Gamma \vdash \mathit{let}\ x \Leftarrow e_1\ \mathit{in}\ e_2 : \sigma}$$

$$(proj) \qquad \frac{\Gamma \vdash e : \sigma_1 \times \sigma_2}{\Gamma \vdash \mathit{proj}_i\, e : \sigma_i} \quad (i = 1, 2)$$

$$(inj) \qquad \frac{\Gamma \vdash e : \sigma_i}{\Gamma \vdash \mathit{inj}_i^{\sigma_1+\sigma_2}\ e : \sigma_1 + \sigma_2} \quad (i = 1, 2)$$

$$(pair) \qquad \frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash \mathit{pair}\, e_1\, e_2 : \sigma_1 \times \sigma_2}$$

$$(case) \qquad \frac{\Gamma \vdash e : \sigma_1 + \sigma_2 \qquad \Gamma, x_1{:}\sigma_1 \vdash e_1 : \sigma \qquad \Gamma, x_2{:}\sigma_2 \vdash e_2 : \sigma}{\Gamma \vdash \mathit{case}\ e\ \mathit{of}\ (x_1.e_1) \mid (x_2.e_2) : \sigma}$$

$$\begin{array}{lcl} \sigma & \in & \mathit{Types}[\Lambda^{\sigma^+}] \\ \sigma & ::= & \ldots \mid \sigma_1 \times \sigma_2 \mid \sigma_1 + \sigma_2 \end{array}$$

$$\begin{array}{lcl} \Gamma & \in & \mathit{Assums}[\Lambda^{\sigma^+}] \\ \Gamma & ::= & \cdot \mid \Gamma, x{:}\sigma \mid \Gamma, f{:}\sigma \end{array}$$

Figure 2.7: Typing rules for the extended typed language Λ^{σ^+}

Call-by-name:

$$\frac{e_0 \longmapsto_n e_0'}{\mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \longmapsto_n \mathit{if}\ e_0'\ \mathit{then}\ e_1\ \mathit{else}\ e_2}$$

$$\mathit{if}\ c_t\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \longmapsto_n e_1 \qquad \mathit{if}\ c\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \longmapsto_n e_2 \quad \ldots\text{where } c \not\equiv \mathtt{c}_t$$

$$\mathit{let}\ x \Leftarrow v_1\ \mathit{in}\ e_2 \longmapsto_n e_2[x := v_1] \qquad \frac{e_1 \longmapsto_n e_1'}{\mathit{let}\ x \Leftarrow e_1\ \mathit{in}\ e_2 \longmapsto_n \mathit{let}\ x \Leftarrow e_1\ \mathit{in}\ e_2}$$

$$\frac{e \longmapsto_n e'}{\mathit{proj}_i\ e \longmapsto_n \mathit{proj}_i\ e'} \qquad \mathit{proj}_i\ (\mathit{pair}\ e_1\ e_2) \longmapsto_n e_i$$

$$\mathit{case}\ \mathit{inj}_i\ e\ \mathit{of}\ (x_1.e_1) \mid (x_2.e_2) \longmapsto_n e_i[x_i := e] \qquad \frac{e_0 \longmapsto_n e_0'}{\mathit{case}\ e_0\ \mathit{of}\ (x_1.e_1) \mid (x_2.e_2) \longmapsto_n \mathit{case}\ e_0'\ \mathit{of}\ (x_1.e_1) \mid (x_2.e_2)}$$

Call-by-value:

$$\frac{e_0 \longmapsto_v e_0'}{\mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \longmapsto_v \mathit{if}\ e_0'\ \mathit{then}\ e_1\ \mathit{else}\ e_2}$$

$$\mathit{if}\ c_t\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \longmapsto_v e_1 \qquad \mathit{if}\ c\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \longmapsto_v e_2 \quad \ldots\text{where } c \not\equiv \mathtt{c}_t$$

$$\mathit{let}\ x \Leftarrow v_1\ \mathit{in}\ e_2 \longmapsto_v e_2[x := v_1] \qquad \frac{e_1 \longmapsto_v e_1'}{\mathit{let}\ x \Leftarrow e_1\ \mathit{in}\ e_2 \longmapsto_v \mathit{let}\ x \Leftarrow e_1\ \mathit{in}\ e_2}$$

$$\frac{e_1 \longmapsto_v e_1'}{\mathit{pair}\ e_1\ e_2 \longmapsto_v \mathit{pair}\ e_1'\ e_2} \qquad \frac{e_2 \longmapsto_v e_2'}{\mathit{pair}\ v_1\ e_2 \longmapsto_v \mathit{pair}\ v_1\ e_2'}$$

$$\frac{e \longmapsto_v e'}{\mathit{proj}_i\ e \longmapsto_v \mathit{proj}_i\ e'} \qquad \mathit{proj}_i\ (\mathit{pair}\ v_1\ v_2) \longmapsto_v v_i \qquad \frac{e \longmapsto_v e'}{\mathit{inj}_i\ e \longmapsto_v \mathit{inj}_i\ e'}$$

$$\frac{e_0 \longmapsto_v e_0'}{\mathit{case}\ e_0\ \mathit{of}\ (x_1.e_1) \mid (x_2.e_2) \longmapsto_v \mathit{case}\ e_0'\ \mathit{of}\ (x_1.e_1) \mid (x_2.e_2)} \qquad \mathit{case}\ \mathit{inj}_i\ v\ \mathit{of}\ (x_1.e_1) \mid (x_2.e_2) \longmapsto_v e_i[x_i := v]$$

Figure 2.8: Call-by-name and call-by-value single-step evaluation rules for $\Lambda^{\sigma+}$

$$Obs(pair\, e_1\, e_2) \quad = \quad \top$$

Definitions of observational approximation and equivalence carry over as expected.

2.4.3 Program calculi

The notions of reductions for Λ^{σ^+} are presented below. The reduction rules for products and co-products reflect the laziness and eagerness associated with call-by-name and call-by-value respectively.

Call-by-name:

$$\begin{array}{rlll}
if\ \mathsf{c}_t\ then\ e_1\ else\ e_2 & \leadsto_{\mathrm{cnd.t}} & e_1 & & (\mathrm{cnd.t}) \\
if\ c\ then\ e_1\ else\ e_2 & \leadsto_{\mathrm{cnd.f}} & e_2 & c \not\equiv \mathsf{c}_t & (\mathrm{cnd.f}) \\
let\ x \Leftarrow v\ in\ e & \leadsto_{\mathrm{let}_n} & e[x := v] & v \in Values_n[\Lambda^{\sigma^+}] & (\mathrm{let}_n) \\
proj_i\,(pair\, e_1\, e_2) & \leadsto_{\times_i.\beta} & e_i & & (\times_i.\beta) \\
case\,(inj_i\, e)\ of\ (x_1.e_1)\,|\,(x_2.e_2) & \leadsto_{+_i.\beta} & e_i[x_i := e] & & (+_i.\beta)
\end{array}$$

Call-by-value:

$$\begin{array}{rlll}
if\ \mathsf{c}_t\ then\ e_1\ else\ e_2 & \leadsto_{\mathrm{cnd.t}} & e_1 & & (\mathrm{cnd.t}) \\
if\ c\ then\ e_1\ else\ e_2 & \leadsto_{\mathrm{cnd.f}} & e_2 & c \not\equiv \mathsf{c}_t & (\mathrm{cnd.f}) \\
let\ x \Leftarrow v\ in\ e & \leadsto_{\mathrm{let}_v} & e[x := v] & v \in Values_v[\Lambda^{\sigma^+}] & (\mathrm{let}_v) \\
proj_i\,(pair\, v_1\, v_2) & \leadsto_{\times_i.\beta_v} & e_i & v_1, v_2 \in Values_v[\Lambda^{\sigma^+}] & (\times_i.\beta_v) \\
case\,(inj_i\, v)\ of\ (x_1.e_1)\,|\,(x_2.e_2) & \leadsto_{+_i.\beta_v} & e_i[x_i := v] & v \in Values_v[\Lambda^{\sigma^+}] & (+_i.\beta_v)
\end{array}$$

The following abbreviations for reduction collections will be convenient.

$$\begin{array}{rcl}
R_n^{\sigma^+} & \stackrel{\mathrm{def}}{=} & \beta \cup \mathrm{rec} \cup \mathrm{cnd.t} \cup \mathrm{cnd.f} \cup \mathrm{let}_n \cup \times_i.\beta \cup +_i.\beta \\
R_v^{\sigma^+} & \stackrel{\mathrm{def}}{=} & \beta_v \cup \mathrm{rec}_v \cup \mathrm{cnd.t} \cup \mathrm{cnd.f} \cup \mathrm{let}_v \cup \times_i.\beta_v \cup +_i.\beta_v
\end{array}$$

Reductions relations and calculi are obtained as in Section 2.2.3. The Compatibility, Substitutivity, Church-Rosser properties, and soundness of calculi extend in the expected way to the above reductions.

2.5 CPS transformations

This section gives definitions and properties associated CPS transformations for the untyped core language, the typed core language, and extensions. The section concludes with a discussion of administrative reductions.

2.5.1 CPS transformations for the untyped core language Λ

Call-by-name continuation-passing style

Figure 2.9 gives Plotkin's call-by-name CPS transformation $\mathcal{C}_n$ where the k's and the y's are fresh variables. The transformation is defined using two functions: $\mathcal{C}_n\langle\!|\cdot|\!\rangle$ is the general transformation function for terms of Λ; and $\mathcal{C}_n\langle\cdot\rangle$ is the transformation function for call-by-name values. The following theorem captures formal properties of the transformation.

Theorem 2.3 (Plotkin 1975) *For $e \in Progs[\Lambda]$ and $e_1, e_2 \in \Lambda$,*

1. **Indifference:** $eval_v(\mathcal{C}_n\langle\!|e|\!\rangle\,(\lambda y\,.\,y)) \simeq eval_n(\mathcal{C}_n\langle\!|e|\!\rangle\,(\lambda y\,.\,y))$
2. **Simulation:** $\mathcal{C}_n\langle eval_n(e)\rangle \simeq eval_v(\mathcal{C}_n\langle\!|e|\!\rangle\,(\lambda y\,.\,y))$
3. **Translation:** $\lambda\beta \vdash e_1 = e_2$ *iff* $\lambda\beta_v \vdash \mathcal{C}_v\langle\!|e_1|\!\rangle = \mathcal{C}_v\langle\!|e_2|\!\rangle$ *iff* $\lambda\beta \vdash \mathcal{C}_n\langle\!|e_1|\!\rangle = \mathcal{C}_n\langle\!|e_2|\!\rangle$

$$
\begin{array}{rcl}
\mathcal{C}_n\langle\!|\cdot|\!\rangle & : & \mathit{Terms}[\Lambda] \rightarrow \mathit{Values}_v[\Lambda] \\
\mathcal{C}_n\langle\!|v|\!\rangle & = & \lambda k\,.\,k\,\mathcal{C}_n\langle v\rangle \\
\mathcal{C}_n\langle\!|x|\!\rangle & = & x \\
\mathcal{C}_n\langle\!|e_1\,e_2|\!\rangle & = & \lambda k\,.\,\mathcal{C}_n\langle\!|e_1|\!\rangle\,(\lambda y_1\,.\,(y_1\,\mathcal{C}_n\langle\!|e_2|\!\rangle)\,k)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}_n\langle\cdot\rangle & : & \mathit{Values}_n[\Lambda] \rightarrow \mathit{Values}_v[\Lambda] \\
\mathcal{C}_n\langle c\rangle & = & c \\
\mathcal{C}_n\langle\lambda x\,.\,e\rangle & = & \lambda x\,.\,\mathcal{C}_n\langle\!|e|\!\rangle
\end{array}
$$

Figure 2.9: Call-by-name CPS transformation for Λ

$$
\begin{array}{rcl}
\mathcal{C}_v\langle\!|\cdot|\!\rangle & : & \mathit{Terms}[\Lambda] \rightarrow \mathit{Values}_v[\Lambda] \\
\mathcal{C}_v\langle\!|v|\!\rangle & = & \lambda k\,.\,k\,\mathcal{C}_v\langle v\rangle \\
\mathcal{C}_v\langle\!|e_1\,e_2|\!\rangle & = & \lambda k\,.\,\mathcal{C}_v\langle\!|e_1|\!\rangle\,(\lambda y_1\,.\,\mathcal{C}_v\langle\!|e_2|\!\rangle\,(\lambda y_2\,.\,(y_1\,y_2)\,k))
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}_v\langle\cdot\rangle & : & \mathit{Values}_v[\Lambda] \rightarrow \mathit{Values}_v[\Lambda] \\
\mathcal{C}_v\langle c\rangle & = & c \\
\mathcal{C}_v\langle x\rangle & = & x \\
\mathcal{C}_v\langle\lambda x\,.\,e\rangle & = & \lambda x\,.\,\mathcal{C}_v\langle\!|e|\!\rangle
\end{array}
$$

Figure 2.10: Call-by-value CPS transformation for Λ

The **Indifference** property states that, given an initial continuation (the identity function), the result of evaluating a CPS term using call-by-value evaluation is the same as the result of using call-by-name evaluation, *i.e.*, terms in the image of the transformation are evaluation-order independent. This follows from the fact that all function arguments are values in the image of the transformation.

The **Simulation** property states that, given an initial continuation (the identity function), evaluating a CPS term using call-by-value simulates the evaluation of the original λ-term using call-by-name.

The **Translation** property states that equivalence classes of β-convertible terms are preserved and reflected by $\mathcal{C}_n$.

Call-by-value continuation-passing style

Figure 2.10 gives Plotkin's call-by-value CPS transformation $\mathcal{C}_v$ where the k's and the y's are fresh variables. The transformation is defined using two functions: $\mathcal{C}_v\langle\!|\cdot|\!\rangle$ is the general transformation function for terms of Λ; and $\mathcal{C}_v\langle\cdot\rangle$ is the transformation function for call-by-value values. The following theorem captures formal properties of the transformation.

Theorem 2.4 (Plotkin 1975) *For $e \in \mathit{Progs}[\Lambda]$ and $e_1, e_2 \in \Lambda$,*

1. **Indifference:** $\mathit{eval}_n(\mathcal{C}_v\langle\!|e|\!\rangle\,(\lambda y\,.\,y)) \simeq \mathit{eval}_v(\mathcal{C}_v\langle\!|e|\!\rangle\,(\lambda y\,.\,y))$
2. **Simulation:** $\mathcal{C}_v\langle\mathit{eval}_v(e)\rangle \simeq \mathit{eval}_n(\mathcal{C}_v\langle\!|e|\!\rangle\,(\lambda y\,.\,y))$

3. **Translation:**
If $\lambda\beta_v \vdash e_1 = e_2$ *then* $\lambda\beta_v \vdash \mathcal{C}_v\langle\!\langle e_1 \rangle\!\rangle = \mathcal{C}_v\langle\!\langle e_2 \rangle\!\rangle$
Also $\lambda\beta_v \vdash \mathcal{C}_v\langle\!\langle e_1 \rangle\!\rangle = \mathcal{C}_v\langle\!\langle e_2 \rangle\!\rangle$ *iff* $\lambda\beta \vdash \mathcal{C}_v\langle\!\langle e_1 \rangle\!\rangle = \mathcal{C}_v\langle\!\langle e_2 \rangle\!\rangle$

The **Indifference** property states that terms in the image of the transformation are evaluation-order independent. As with $\mathcal{C}_n$, this follows from the fact that all function arguments are values in the image of the transformation.

The **Simulation** property states that, given an initial continuation (the identity function), evaluating a CPS term using call-by-name simulates the evaluation of the original λ-term using call-by-value.

The **Translation** property states that β_v-convertible terms are also convertible in the image of $\mathcal{C}_v$. In contrast to the theory $\lambda\beta$ appearing in the **Translation** property for $\mathcal{C}_n$ (Theorem 2.3), the theory $\lambda\beta_v$ is *incomplete* in the sense that it cannot establish the convertibility of some pairs of terms in the image of the CPS transformation [86].[2]

Assessment

The two flavors of the CPS transformation (*i.e.*, $\mathcal{C}_n$ and $\mathcal{C}_v$) are alike in that they both yield evaluation-order independent, continuation-passing λ-terms. The fact that each function is not only applied to its regular argument, but also to a continuation — a representation of how succeeding evaluation will use the value produced by the function — leads to the tail-call property of CPS terms.

$\mathcal{C}_n$ and $\mathcal{C}_v$ are different due to the different treatment of functions arguments. Call-by-value functions must be applied to values only and thus $\mathcal{C}_v$ explicitly encodes the evaluation of all function arguments before the function is applied. Under call-by-name, there is no distinction between arbitrary argument expressions and values — the evaluation of all arguments is suspended. Thus $\mathcal{C}_n$ explicitly encodes each argument as a suspended computation which needs a continuation before evaluation can proceed.

2.5.2 CPS transformations for the typed core language Λ^σ

Plotkin's CPS transformations were originally stated for untyped Λ-terms. These transformations have been shown to preserve well-typedness of terms [36, 38, 55, 66]. Based on these results, we extend Plotkin's CPS transformations to Λ^σ.

Figure 2.11 presents the call-by-name CPS transformation for typed core language Λ^σ. We use the notation $\neg\sigma$ to abbreviate $\sigma \to ans$ where $ans \in Types[\Lambda^\sigma]$ is a distinguished type of answers [55]. Thus

$$\neg\neg\mathcal{C}_n\langle\!\langle\sigma\rangle\!\rangle \;=\; (\mathcal{C}_n\langle\sigma\rangle \to ans) \to ans.$$

The definition of $\mathcal{C}_n$ on function types and on type assumptions reflects the fact that source functions are translated to functions whose arguments are terms needing a continuation.

The following property [36, 38, 66] states that $\mathcal{C}_n$ preserves well-typedness of terms.

Property 2.7 *For* $\Gamma \vdash e : \sigma$ *and* $\Gamma \vdash v : \sigma$ *where* $v \in Values_n[\Lambda^\sigma]$,

$$\mathcal{C}_n\langle\!\langle\Gamma\rangle\!\rangle \vdash \mathcal{C}_n\langle\!\langle e\rangle\!\rangle : \mathcal{C}_n\langle\!\langle\sigma\rangle\!\rangle$$
$$\mathcal{C}_n\langle\!\langle\Gamma\rangle\!\rangle \vdash \mathcal{C}_n\langle v\rangle : \mathcal{C}_n\langle\sigma\rangle$$

Figure 2.12 presents the call-by-value CPS transformation for the typed core language Λ^σ. The definition of $\mathcal{C}_v$ on function types and on type assumptions reflects the fact that source functions are translated to functions whose arguments are values.

The following property [55] states that $\mathcal{C}_v$ preserves well-typedness of terms.

Property 2.8 *For* $\Gamma \vdash e : \sigma$ *and* $\Gamma \vdash v : \sigma$ *where* $v \in Values_n[\Lambda^\sigma]$,

$$\mathcal{C}_v\langle\!\langle\Gamma\rangle\!\rangle \vdash \mathcal{C}_v\langle\!\langle e\rangle\!\rangle : \mathcal{C}_v\langle\!\langle\sigma\rangle\!\rangle$$
$$\mathcal{C}_v\langle\!\langle\Gamma\rangle\!\rangle \vdash \mathcal{C}_v\langle v\rangle : \mathcal{C}_v\langle\sigma\rangle$$

[2] Plotkin gives the following example of the incompleteness [76, p. 153]. Let $e_1 \equiv ((\lambda x\,.\,x\,x)\,(\lambda x\,.\,x\,x))\,y$ and $e_2 \equiv (\lambda x\,.\,x\,y)\,((\lambda x\,.\,x\,x)\,(\lambda x\,.\,x\,x))$. Then $\lambda\beta_v \vdash \mathcal{C}_v\langle\!\langle e_1\rangle\!\rangle = \mathcal{C}_v\langle\!\langle e_2\rangle\!\rangle$ and $\lambda\beta \vdash \mathcal{C}_v\langle\!\langle e_1\rangle\!\rangle = \mathcal{C}_v\langle\!\langle e_2\rangle\!\rangle$ but $\lambda\beta_v \not\vdash e_1 = e_2$. Sabry and Felleisen [86] give an equational theory λA (where A is a set of axioms including $\beta_v\eta_v$) and show it complete in the sense that $\lambda A \vdash e_1 = e_2$ *iff* $\lambda\beta\eta \vdash \mathcal{C}'_v\langle\!\langle e_1\rangle\!\rangle = \mathcal{C}'_v\langle\!\langle e_2\rangle\!\rangle$. $\mathcal{C}'_v$ is Fischer's call-by-value CPS transformation [31] where continuations are the first arguments to functions (instead of the second arguments as in Plotkin's $\mathcal{C}_v$).

$$
\begin{array}{rcl}
\mathcal{C}_n\langle\cdot\rangle & : & \mathit{Values}_n[\Lambda^\sigma] \rightarrow \mathit{Values}_v[\Lambda^\sigma] \\
& \cdots & \\
\mathcal{C}_n\langle f\rangle & = & f \\
\mathcal{C}_n\langle \mathit{rec}\, f(x).\, e\rangle & = & \mathit{rec}\, f(x).\, \mathcal{C}_n\langle\!\langle e\rangle\!\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}_n\langle\!\langle\cdot\rangle\!\rangle & : & \mathit{Types}[\Lambda^\sigma] \rightarrow \mathit{Types}[\Lambda^\sigma] \\
\mathcal{C}_n\langle\!\langle\sigma\rangle\!\rangle & = & \neg\neg\mathcal{C}_n\langle\sigma\rangle \\
\mathcal{C}_n\langle\iota\rangle & = & \iota \\
\mathcal{C}_n\langle\sigma_1 \rightarrow \sigma_2\rangle & = & \mathcal{C}_n\langle\!\langle\sigma_1\rangle\!\rangle \rightarrow \mathcal{C}_n\langle\!\langle\sigma_2\rangle\!\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}_n\langle\!\langle\cdot\rangle\!\rangle & : & \mathit{Assums}[\Lambda^\sigma] \rightarrow \mathit{Assums}[\Lambda^\sigma] \\
\mathcal{C}_n\langle\!\langle\Gamma, x:\sigma\rangle\!\rangle & = & \mathcal{C}_n\langle\!\langle\Gamma\rangle\!\rangle, x:\mathcal{C}_n\langle\!\langle\sigma\rangle\!\rangle \\
\mathcal{C}_n\langle\!\langle\Gamma, f:\sigma\rangle\!\rangle & = & \mathcal{C}_n\langle\!\langle\Gamma\rangle\!\rangle, f:\mathcal{C}_n\langle\sigma\rangle
\end{array}
$$

Figure 2.11: Call-by-name CPS transformation for Λ^σ

$$
\begin{array}{rcl}
\mathcal{C}_v\langle\cdot\rangle & : & \mathit{Values}_v[\Lambda^\sigma] \rightarrow \mathit{Values}_v[\Lambda^\sigma] \\
& \cdots & \\
\mathcal{C}_v\langle f\rangle & = & f \\
\mathcal{C}_v\langle \mathit{rec}\, f(x).\, e\rangle & = & \mathit{rec}\, f(x).\, \mathcal{C}_v\langle\!\langle e\rangle\!\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}_v\langle\!\langle\cdot\rangle\!\rangle & : & \mathit{Types}[\Lambda^\sigma] \rightarrow \mathit{Types}[\Lambda^\sigma] \\
\mathcal{C}_v\langle\!\langle\sigma\rangle\!\rangle & = & \neg\neg\mathcal{C}_v\langle\sigma\rangle \\
\mathcal{C}_v\langle\iota\rangle & = & \iota \\
\mathcal{C}_v\langle\sigma_1 \rightarrow \sigma_2\rangle & = & \mathcal{C}_v\langle\sigma_1\rangle \rightarrow \mathcal{C}_v\langle\!\langle\sigma_2\rangle\!\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}_v\langle\!\langle\cdot\rangle\!\rangle & : & \mathit{Assums}[\Lambda^\sigma] \rightarrow \mathit{Assums}[\Lambda^\sigma] \\
\mathcal{C}_v\langle\!\langle\Gamma, x:\sigma\rangle\!\rangle & = & \mathcal{C}_v\langle\!\langle\Gamma\rangle\!\rangle, x:\mathcal{C}_v\langle\sigma\rangle \\
\mathcal{C}_v\langle\!\langle\Gamma, f:\sigma\rangle\!\rangle & = & \mathcal{C}_v\langle\!\langle\Gamma\rangle\!\rangle, f:\mathcal{C}_v\langle\sigma\rangle
\end{array}
$$

Figure 2.12: Call-by-value CPS transformation for Λ^σ

$$
\begin{array}{rcl}
\mathcal{C}_n\langle\!|\cdot|\!\rangle & : & \mathit{Terms}[\Lambda^{\sigma^+}] \to \mathit{Values}_v[\Lambda^{\sigma^+}] \\
 & \cdots & \\
\mathcal{C}_n\langle\!|\mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2|\!\rangle & = & \lambda k\,.\,\mathcal{C}_n\langle\!|e_0|\!\rangle\,(\lambda y\,.\,(\mathit{if}\ y\ \mathit{then}\ \mathcal{C}_n\langle\!|e_1|\!\rangle\ \mathit{else}\ \mathcal{C}_n\langle\!|e_2|\!\rangle)\,k) \\
\mathcal{C}_n\langle\!|\mathit{let}\ x \Leftarrow e_1\ \mathit{in}\ e_2|\!\rangle & = & \lambda k\,.\,\mathcal{C}_n\langle\!|e_1|\!\rangle\,(\lambda x\,.\,\mathcal{C}_n\langle\!|e_2|\!\rangle\,k) \\
\mathcal{C}_n\langle\!|\mathit{proj}_i\,e|\!\rangle & = & \lambda k\,.\,\mathcal{C}_n\langle\!|e|\!\rangle\,(\lambda y\,.\,(\mathit{proj}_i\,y)\,k) \\
\mathcal{C}_n\langle\!|\mathit{case}\ e\ \mathit{of}\ (x_1.e_1)\,|\,(x_2.e_2)|\!\rangle & = & \lambda k\,.\,\mathcal{C}_n\langle\!|e|\!\rangle\,(\lambda y\,.\,(\mathit{case}\ y\ \mathit{of}\ (x_1.\mathcal{C}_n\langle\!|e_1|\!\rangle)\,|\,(x_2.\mathcal{C}_n\langle\!|e_2|\!\rangle))\,k)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}_n\langle\cdot\rangle & : & \mathit{Values}_n[\Lambda^{\sigma^+}] \to \mathit{Values}_v[\Lambda^{\sigma^+}] \\
 & \cdots & \\
\mathcal{C}_n\langle\mathit{pair}\ e_1\,e_2\rangle & = & \mathit{pair}\ \mathcal{C}_n\langle\!|e_1|\!\rangle\ \mathcal{C}_n\langle\!|e_2|\!\rangle \\
\mathcal{C}_n\langle\mathit{inj}_i\,e\rangle & = & \mathit{inj}_i\ \mathcal{C}_n\langle\!|e|\!\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}_n\langle\cdot\rangle & : & \mathit{Types}[\Lambda^{\sigma^+}] \to \mathit{Types}[\Lambda^{\sigma^+}] \\
 & \cdots & \\
\mathcal{C}_n\langle\sigma_1 \times \sigma_2\rangle & = & \mathcal{C}_n\langle\!|\sigma_1|\!\rangle \times \mathcal{C}_n\langle\!|\sigma_2|\!\rangle \\
\mathcal{C}_n\langle\sigma_1 + \sigma_2\rangle & = & \mathcal{C}_n\langle\!|\sigma_1|\!\rangle + \mathcal{C}_n\langle\!|\sigma_2|\!\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}_n\langle\!|\cdot|\!\rangle & : & \mathit{Assums}[\Lambda^{\sigma^+}] \to \mathit{Assums}[\Lambda^{\sigma^+}] \\
\mathcal{C}_n\langle\!|\Gamma, x:\sigma|\!\rangle & = & \mathcal{C}_n\langle\!|\Gamma|\!\rangle, x:\mathcal{C}_n\langle\!|\sigma|\!\rangle \\
\mathcal{C}_n\langle\!|\Gamma, f:\sigma|\!\rangle & = & \mathcal{C}_n\langle\!|\Gamma|\!\rangle, f:\mathcal{C}_n\langle\sigma\rangle
\end{array}
$$

Figure 2.13: Call-by-name CPS transformation for Λ^{σ^+}

Assessment

As pointed out in the previous section, $\mathcal{C}_n$ and $\mathcal{C}_v$ are alike in that they both introduce continuation-passing terms. This is reflected by the similarity in the definitions $\mathcal{C}_n\langle\!|\sigma|\!\rangle = \neg\neg\mathcal{C}_n\langle\sigma\rangle$ and $\mathcal{C}_v\langle\!|\sigma|\!\rangle = \neg\neg\mathcal{C}_v\langle\sigma\rangle$. $\mathcal{C}_n$ and $\mathcal{C}_v$ differ in how arguments are treated. This is reflected by the difference in the definitions $\mathcal{C}_n\langle\sigma_1 \to \sigma_2\rangle = \mathcal{C}_n\langle\!|\sigma_1|\!\rangle \to \mathcal{C}_n\langle\!|\sigma_2|\!\rangle$ and $\mathcal{C}_v\langle\sigma_1 \to \sigma_2\rangle = \mathcal{C}_v\langle\sigma_1\rangle \to \mathcal{C}_v\langle\!|\sigma_2|\!\rangle$.

2.5.3 CPS transformations for the extended typed language Λ^{σ^+}

Call-by-name continuation-passing style

Figure 2.13 gives the call-by-name CPS transformation for Λ^{σ^+}. The transformation encodes the laziness of products and co-products in the same way that function arguments are delayed, *i.e.*, abstractions $\lambda k\,.\,\ldots$ have the effect of suspending the evaluation of constructor arguments. The laziness of products and co-products is also reflected in the transformation of types given in Figure 2.13; the components of products and co-products are always terms deprived of a continuation (*i.e.*, they have types of the form $\neg\neg\sigma \equiv (\sigma \to \mathit{ans}) \to \mathit{ans}$).

Call-by-value continuation-passing style

Figure 2.14 gives the call-by-value CPS transformation for Λ^{σ^+}. The transformation encodes the eagerness of products and co-products by forcing the evaluation of arguments before applying constructors. This is reflected in the transformation of types given in Figure 2.14; the components of products and co-products are always value types (*i.e.*, the types are given by $\mathcal{C}_v\langle\cdot\rangle$ instead of $\mathcal{C}_v\langle\!|\cdot|\!\rangle$).

Assessment

Plotkin's **Indifference** and **Simulation** theorems for $\mathcal{C}_n$ and $\mathcal{C}_v$ on Λ and Λ^σ hold up to α-equivalence of terms. This correspondence is rather strong and fails for many reasonable CPS transformations. For example,

$$
\begin{array}{rcl}
\mathcal{C}_v\langle\!|[\cdot]|\!\rangle & : & \mathit{Terms}[\Lambda^{\sigma^+}] \rightarrow \mathit{Values}_v[\Lambda^{\sigma^+}] \\
 & \cdots & \\
\mathcal{C}_v\langle\!|[\mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2]|\!\rangle & = & \lambda k\,.\,\mathcal{C}_v\langle\!|[e_0]|\!\rangle\,(\lambda y\,.\,(\mathit{if}\ y\ \mathit{then}\ \mathcal{C}_v\langle\!|[e_1]|\!\rangle\ \mathit{else}\ \mathcal{C}_v\langle\!|[e_2]|\!\rangle)\,k) \\
\mathcal{C}_v\langle\!|[\mathit{let}\ x \Leftarrow e_1\ \mathit{in}\ e_2]|\!\rangle & = & \lambda k\,.\,\mathcal{C}_v\langle\!|[e_1]|\!\rangle\,(\lambda x\,.\,\mathcal{C}_v\langle\!|[e_2]|\!\rangle\,k) \\
\mathcal{C}_v\langle\!|[\mathit{pair}\ e_1\ e_2]|\!\rangle & = & \lambda k\,.\,\mathcal{C}_v\langle\!|[e_1]|\!\rangle\,(\lambda y_1\,.\,\mathcal{C}_v\langle\!|[e_2]|\!\rangle\,(\lambda y_2\ .\,k\,(\mathit{pair}\ y_1\ y_2))) \\
\mathcal{C}_v\langle\!|[\mathit{proj}_i\ e]|\!\rangle & = & \lambda k\,.\,\mathcal{C}_v\langle\!|[e]|\!\rangle\,(\lambda y\,.\,k\,(\mathit{proj}_i\ y)) \\
\mathcal{C}_v\langle\!|[\mathit{inj}_i\ e]|\!\rangle & = & \lambda k\,.\,\mathcal{C}_v\langle\!|[e]|\!\rangle\,(\lambda y\,.\,k\,(\mathit{inj}_i\ y)) \\
\mathcal{C}_v\langle\!|[\mathit{case}\ e\ \mathit{of}\ (x_1.e_1)\,|\,(x_2.e_2)]|\!\rangle & = & \lambda k\,.\,\mathcal{C}_v\langle\!|[e]|\!\rangle\,(\lambda y\,.\,(\mathit{case}\ y\ \mathit{of}\ (x_1.\mathcal{C}_v\langle\!|[e_1]|\!\rangle)\,|\,(x_2.\mathcal{C}_v\langle\!|[e_2]|\!\rangle))\,k)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}_v\langle\cdot\rangle & : & \mathit{Values}_v[\Lambda^{\sigma^+}] \rightarrow \mathit{Values}_v[\Lambda^{\sigma^+}] \\
 & \cdots & \\
\mathcal{C}_v\langle \mathit{pair}\ v_1\ v_2\rangle & = & \mathit{pair}\ \mathcal{C}_v\langle v_1\rangle\ \mathcal{C}_v\langle v_2\rangle \\
\mathcal{C}_v\langle \mathit{inj}_i\ v\rangle & = & \mathit{inj}_i\ \mathcal{C}_v\langle v\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}_v\langle\cdot\rangle & : & \mathit{Types}[\Lambda^{\sigma^+}] \rightarrow \mathit{Types}[\Lambda^{\sigma^+}] \\
 & \cdots & \\
\mathcal{C}_v\langle\sigma_1 \times \sigma_2\rangle & = & \mathcal{C}_v\langle\sigma_1\rangle \times \mathcal{C}_v\langle\sigma_2\rangle \\
\mathcal{C}_v\langle\sigma_1 + \sigma_2\rangle & = & \mathcal{C}_v\langle\sigma_1\rangle + \mathcal{C}_v\langle\sigma_2\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}_v\langle\!|[\cdot]|\!\rangle & : & \mathit{Assums}[\Lambda^{\sigma^+}] \rightarrow \mathit{Assums}[\Lambda^{\sigma^+}] \\
\mathcal{C}_v\langle\!|[\Gamma, x:\sigma]|\!\rangle & = & \mathcal{C}_v\langle\!|[\Gamma]|\!\rangle, x:\mathcal{C}_v\langle\sigma\rangle \\
\mathcal{C}_v\langle\!|[\Gamma, f:\sigma]|\!\rangle & = & \mathcal{C}_v\langle\!|[\Gamma]|\!\rangle, f:\mathcal{C}_v\langle\sigma\rangle
\end{array}
$$

Figure 2.14: Call-by-value CPS transformation for Λ^{σ^+}

it fails for $\mathcal{C}_v$ on Λ^{σ^+} since we have not used a continuation-passing version of *proj*. We illustrate the failure of **Indifference** below. At the steps marked (†), call-by-name and call-by-value no longer coincide.

Call-by-name:

$$\begin{array}{ll}
& \mathcal{C}_v\,\langle\!\langle (\lambda x\,.\,\lambda y\,.\,x)\,(proj_1\ pair\ c_1\ c_2)\rangle\!\rangle\,(\lambda z\,.\,z) \\
= & (\lambda k\,.\,\mathcal{C}_v\,\langle\!\langle \lambda x\,.\,\lambda y\,.\,x\rangle\!\rangle\,(\lambda y_1\,.\,\mathcal{C}_v\,\langle\!\langle proj_1\ pair\ c_1\ c_2\rangle\!\rangle\,(\lambda y_2\,.\,(y_1\,y_2)\,k)))\,(\lambda z\,.\,z) \\
\longmapsto_n & \mathcal{C}_v\,\langle\!\langle \lambda x\,.\,\lambda y\,.\,x\rangle\!\rangle\,(\lambda y_1\,.\,\mathcal{C}_v\,\langle\!\langle proj_1\ pair\ c_1\ c_2\rangle\!\rangle\,(\lambda y_2\,.\,(y_1\,y_2)\,(\lambda z\,.\,z))) \\
\longmapsto_n^* & (\lambda y_2\,.\,((\lambda x\,.\,\lambda k\,.\,k\,(\lambda y\,.\,\lambda k\,.\,k\,x))\,y_2)\,(\lambda z\,.\,z))\,(proj_1\ pair\ c_1\ c_2) \\
\longmapsto_n & ((\lambda x\,.\,\lambda k\,.\,k\,(\lambda y\,.\,\lambda k\,.\,k\,x))\,(proj_1\ pair\ c_1\ c_2))\,(\lambda z\,.\,z) \qquad (\dagger) \\
\longmapsto_n & (\lambda k\,.\,k\,(\lambda y\,.\,\lambda k\,.\,k\,(proj_1\ pair\ c_1\ c_2)))\,(\lambda z\,.\,z) \\
\longmapsto_n^* & \lambda y\,.\,\lambda k\,.\,k\,(proj_1\ pair\ c_1\ c_2)
\end{array}$$

Call-by-value:

$$\begin{array}{ll}
& \mathcal{C}_v\,\langle\!\langle (\lambda x\,.\,\lambda y\,.\,x)\,(proj_1\ pair\ c_1\ c_2)\rangle\!\rangle\,(\lambda z\,.\,z) \\
= & (\lambda k\,.\,\mathcal{C}_v\,\langle\!\langle \lambda x\,.\,\lambda y\,.\,x\rangle\!\rangle\,(\lambda y_1\,.\,\mathcal{C}_v\,\langle\!\langle proj_1\ pair\ c_1\ c_2\rangle\!\rangle\,(\lambda y_2\,.\,(y_1\,y_2)\,k)))\,(\lambda z\,.\,z) \\
\longmapsto_v & \mathcal{C}_v\,\langle\!\langle \lambda x\,.\,\lambda y\,.\,x\rangle\!\rangle\,(\lambda y_1\,.\,\mathcal{C}_v\,\langle\!\langle proj_1\ pair\ c_1\ c_2\rangle\!\rangle\,(\lambda y_2\,.\,(y_1\,y_2)\,(\lambda z\,.\,z))) \\
\longmapsto_v^* & (\lambda y_2\,.\,((\lambda x\,.\,\lambda k\,.\,k\,(\lambda y\,.\,\lambda k\,.\,k\,x))\,y_2)\,(\lambda z\,.\,z))\,(proj_1\ pair\ c_1\ c_2) \\
\longmapsto_v & (\lambda y_2\,.\,((\lambda x\,.\,\lambda k\,.\,k\,(\lambda y\,.\,\lambda k\,.\,k\,x))\,y_2)\,(\lambda z\,.\,z))\,c_1 \qquad (\dagger) \\
\longmapsto_v & ((\lambda x\,.\,\lambda k\,.\,k\,(\lambda y\,.\,\lambda k\,.\,k\,x))\,c_1)\,(\lambda z\,.\,z) \\
\longmapsto_v & (\lambda k\,.\,k\,(\lambda y\,.\,\lambda k\,.\,k\,c_1))\,(\lambda z\,.\,z) \\
\longmapsto_v^* & \lambda y\,.\,\lambda k\,.\,k\,c_1
\end{array}$$

Simulation fails to hold up to α-equivalence since

$$\begin{array}{ll}
& (\lambda x\,.\,\lambda y\,.\,x)\,(proj_1\ pair\ c_1\ c_2) \\
\longmapsto_v & (\lambda x\,.\,\lambda y\,.\,x)\,c_1 \\
\longmapsto_v & \lambda y\,.\,c_1
\end{array}$$

but $\mathcal{C}_v\,\langle\lambda y\,.\,c_1\rangle\ =\ \lambda y\,.\,\lambda k\,.\,k\,c_1\ \not\equiv\ \lambda y\,.\,\lambda k\,.\,k\,(proj_1\ pair\ c_1\ c_2)$.

These results can be made to hold up to α-equivalence if we introduce a continuation-passing projection $proj^k$ that reduces under both call-by-name and call-by-value as follows.

$$(proj_i^k\ pair\ v_1\ v_2)\,\kappa\ \longmapsto_a\ \kappa\,v_i$$

The definition of $\mathcal{C}_v$ is revised accordingly.

$$\mathcal{C}_v\,\langle\!\langle proj_i\,e\rangle\!\rangle\quad =\quad \lambda k\,.\,\mathcal{C}_v\,\langle\!\langle e\rangle\!\rangle\,(\lambda y\,.\,(proj_i^k\ y)\,k)$$

The evaluation below illustrates that **Simulation** and **Indifference** properties for $\mathcal{C}_v$ now hold up to α-equivalence for the term under consideration.

$$\begin{array}{ll}
& \mathcal{C}_v\,\langle\!\langle (\lambda x\,.\,\lambda y\,.\,x)\,(proj_1\ pair\ c_1\ c_2)\rangle\!\rangle\,(\lambda z\,.\,z) \\
= & (\lambda k\,.\,\mathcal{C}_v\,\langle\!\langle \lambda x\,.\,\lambda y\,.\,x\rangle\!\rangle\,(\lambda y_1\,.\,\mathcal{C}_v\,\langle\!\langle proj_1\ pair\ c_1\ c_2\rangle\!\rangle\,(\lambda y_2\,.\,(y_1\,y_2)\,k)))\,(\lambda z\,.\,z) \\
\longmapsto_a & \mathcal{C}_v\,\langle\!\langle \lambda x\,.\,\lambda y\,.\,x\rangle\!\rangle\,(\lambda y_1\,.\,\mathcal{C}_v\,\langle\!\langle proj_1\ pair\ c_1\ c_2\rangle\!\rangle\,(\lambda y_2\,.\,(y_1\,y_2)\,(\lambda z\,.\,z))) \\
\longmapsto_a^* & (proj_1^k\ pair\ c_1\ c_2)\,(\lambda y_2\,.\,((\lambda x\,.\,\lambda k\,.\,k\,(\lambda y\,.\,\lambda k\,.\,k\,x))\,y_2)\,(\lambda z\,.\,z)) \\
\longmapsto_a & (\lambda y_2\,.\,((\lambda x\,.\,\lambda k\,.\,k\,(\lambda y\,.\,\lambda k\,.\,k\,x))\,y_2)\,(\lambda z\,.\,z))\,c_1 \\
\longmapsto_a & ((\lambda x\,.\,\lambda k\,.\,k\,(\lambda y\,.\,\lambda k\,.\,k\,x))\,c_1)\,(\lambda z\,.\,z) \\
\longmapsto_a & (\lambda k\,.\,k\,(\lambda y\,.\,\lambda k\,.\,k\,c_1))\,(\lambda z\,.\,z) \\
\longmapsto_a^* & \lambda y\,.\,\lambda k\,.\,k\,c_1 \\
\equiv & \mathcal{C}_v\,\langle\lambda y\,.\,c_1\rangle
\end{array}$$

The problem with *proj* described here is an instance of a general problem with transforming strict primitive operations. Without using continuation-passing primitives, we cannot maintain the property that each function

$$
\begin{array}{rcl}
\mathcal{C}_v\langle\!|\cdot|\!\rangle & : & \mathit{Terms}[\Lambda] \rightarrow \mathit{Values}_v[\Lambda] \\
\mathcal{C}_v\langle\!|v|\!\rangle & = & \overline{\lambda}\, k\, .\, k\, \mathcal{C}_v\langle v\rangle \\
\mathcal{C}_v\langle\!|e_1\, e_2|\!\rangle & = & \overline{\lambda}\, k\, .\, \mathcal{C}_v\langle\!|e_1|\!\rangle\, (\overline{\lambda}\, y_1\, .\, \mathcal{C}_v\langle\!|e_2|\!\rangle\, (\overline{\lambda}\, y_2\, .\, (y_1\, y_2)\, k))
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}_v\langle\cdot\rangle & : & \mathit{Values}_v[\Lambda] \rightarrow \mathit{Values}_v[\Lambda] \\
\mathcal{C}_v\langle c\rangle & = & c \\
\mathcal{C}_v\langle x\rangle & = & x \\
\mathcal{C}_v\langle \lambda x\, .\, e\rangle & = & \lambda x\, .\, \mathcal{C}_v\langle\!|e|\!\rangle
\end{array}
$$

Figure 2.15: Call-by-value CPS transformation for Λ (annotated)

is a value. Instead, only a weaker property is maintained — all arguments are guaranteed to terminate.[3] In settings where the only computational effect is non-termination, the weaker property is sufficient if one is willing to give up the "lock-step" behavior of CPS terms. However, in settings which include computational effects such as state or exceptions, CPS primitives are necessary.

Finally, we note that the call-by-name transformation of *proj* using a continuation-passing version is a moot point. The call-by-name continuation-passing version would reduce as follows.

$$(\mathit{proj}_i^k\ \mathit{pair}\, e_1\, e_2)\, \kappa \longmapsto_a e_i\, \kappa$$

But this is precisely the reduction that is captured by the current version of $\mathcal{C}_n$.

Since the stated notions of observations only allow observation of termination of non-constant results, it is quite reasonable to relax the requirements on **Indifference** and **Simulation** for Λ^{σ^+} so that they hold up to observations.

Theorem 2.5 *For all* $e \in \mathit{Progs}[\Lambda^{\sigma^+}]$,

1. **Indifference:** $\mathit{eval}_v(\mathcal{C}_n\langle\!|e|\!\rangle\, (\lambda y\, .\, y)) \simeq_{obs} \mathit{eval}_n(\mathcal{C}_n\langle\!|e|\!\rangle\, (\lambda y\, .\, y))$
2. **Simulation:** $\mathit{eval}_n(e) \simeq_{obs} \mathit{eval}_v(\mathcal{C}_n\langle\!|e|\!\rangle\, (\lambda y\, .\, y))$

Theorem 2.6 *For all* $e \in \mathit{Progs}[\Lambda^{\sigma^+}]$,

1. **Indifference:** $\mathit{eval}_n(\mathcal{C}_v\langle\!|e|\!\rangle\, (\lambda y\, .\, y)) \simeq_{obs} \mathit{eval}_v(\mathcal{C}_v\langle\!|e|\!\rangle\, (\lambda y\, .\, y))$
2. **Simulation:** $\mathit{eval}_v(e) \simeq_{obs} \mathit{eval}_n(\mathcal{C}_v\langle\!|e|\!\rangle\, (\lambda y\, .\, y))$

2.5.4 Administrative reductions

Practical applications of CPS often use optimized versions of $\mathcal{C}_n$ and $\mathcal{C}_v$. These optimized versions produce terms that are more compact and thus require fewer steps to evaluate. The compactness is achieved by removing *administrative redexes* — redexes introduced by CPS transformations to implement continuation passing.

Figure 2.15 presents an annotated version of the call-by-value CPS transformation $\mathcal{C}_v$ presented in Figure 2.10. The abstractions in the image of the transformation are separated into two groups:

1. abstractions introduced by the transformation (these are overlined), and
2. abstractions present in the source term (these are unannotated).

[3] Of course, this is assuming that our primitive operations cannot introduce non-termination.

Administrative redexes are those redexes involving overlined abstractions. In general we will only consider administrative redexes that take the form of β-redexes (call these β_{adm}-redexes). However, some limited forms of administrative η-redexes are also sound.[4] β_{adm}-redexes are strongly normalizing [86] and thus an optimized transformation $\mathcal{C}_v^{opt}$ can be defined as

$$\mathcal{C}_v^{opt} \stackrel{\text{def}}{=} \mathcal{N}_{(adm)} \circ \mathcal{C}_v$$

where $\mathcal{N}_{(adm)}$ is a function taking each term to its unique β_{adm}-normal form.

The output of the other CPS transformations presented in this dissertation can be separated into administrative and source redexes in a similar manner. In the sequel, we omit explicit annotations as in Figure 2.15 but continue to distinguish administrative redexes using notation corresponding to β_{adm}.

We have defined $\mathcal{C}_v^{opt}$ abstractly as a two phase process (*i.e.*, (1) transforming *via* $\mathcal{C}_v$, (2) normalizing *via* $\mathcal{N}_{(adm)}$). For practical applications, it is better to implement $\mathcal{C}_v^{opt}$ in a single phase by eliminating administrative redexes while transforming to CPS. There are several approaches to such an implementation [34, 86, 20]. In Chapter 4, we apply a method introduced by Danvy and Filinski [20]. It is straightforward to apply this method to the other CPS transformations we present.

[4] These will be discussed in Chapter 6.

Chapter 3

Thunks and the λ-calculus

3.1 Introduction

3.1.1 Motivation

In his seminal paper, *Call-by-name, call-by-value and the λ-calculus* [76], Plotkin formalizes both call-by-name and call-by-value procedure calling mechanisms for λ-calculi. Call-by-name evaluation is described with a standardization theorem for the β-calculus. Call-by-value evaluation is described with a standardization theorem for a new calculus (the β_v-calculus). Plotkin then shows that call-by-name can be simulated by call-by-value and vice versa. The simulations also give interpretations of each calculus in terms of the other.

Both of Plotkin's simulations rely on *continuations* However, programming wisdom has it that *thunks*[1] can be used to obtain a simpler simulation of call-by-name by call-by-value. Plotkin acknowledges that thunks provide some simulation properties but states that "...these 'protecting by a λ' techniques do not seem to be extendable to a complete simulation and it is fortunate that the technique of continuations is available." [76, p. 147]. By "protecting by a λ", Plotkin refers to a representation of thunks as λ-abstractions with a dummy parameter.

In this chapter, we investigate thunks and their simulation properties. We begin by defining a thunk-introducing transformation $\mathcal{T}$ and prove that thunks are actually sufficient for a complete simulation of call-by-name by call-by-value. We do this by establishing an analogue of Plotkin's **Simulation**, **Indifference**, and **Translation** properties for thunks.

The transformation $\mathcal{T}$ represents thunks using abstract operators *delay* and *force*. Based on $\mathcal{T}$, we obtain a transformation $\mathcal{T}_{\mathcal{I}}$ that implements the technique of "protecting by a λ" and enjoys the same simulation properties as $\mathcal{T}$. Thus, we are able to show that all the properties Plotkin proved about his call-by-name continuation-passing simulation $\mathcal{C}_n$ can be established without continuations — which is nice, because thunks are simpler than continuations.[2]

Specifically, we show that $\mathcal{T}_{\mathcal{I}}$ simulates call-by-name evaluation under call-by-value evaluation.

$$(\textbf{Simulation}): \quad \mathcal{T}_{\mathcal{I}}(\![eval_n(e)]\!) \simeq_{\beta_a} eval_v(\mathcal{T}_{\mathcal{I}}(\![e]\!))$$

Next we show that the meaning of the program $\mathcal{T}_{\mathcal{I}}(\![e]\!)$ is indifferent as to whether it is evaluated by call-by-name or call-by-value.

$$(\textbf{Indifference}): \quad eval_n(\mathcal{T}_{\mathcal{I}}(\![e]\!)) \simeq eval_v(\mathcal{T}_{\mathcal{I}}(\![e]\!))$$

In terms of equational theories, thunks can be used to obtain an equational correspondence between $\lambda\beta$ and $\lambda\beta_v$ with the following corollary.

$$(\textbf{Translation}): \quad \lambda\beta \vdash e_1 = e_2 \text{ iff } \lambda\beta_v \vdash \mathcal{T}_{\mathcal{I}}(\![e_1]\!) = \mathcal{T}_{\mathcal{I}}(\![e_2]\!) \text{ iff } \lambda\beta \vdash \mathcal{T}_{\mathcal{I}}(\![e_1]\!) = \mathcal{T}_{\mathcal{I}}(\![e_2]\!)$$

[1] The term "thunk" was coined to describe the compiled representation of delayed expressions in implementations of Algol 60 [43]. The terminology has been carried over and applied to various methods of delaying the evaluation of expressions [79].

[2] When we presented these results to him, Gordon Plotkin told us that he had also found recently the "protecting by a λ" technique to be sufficient for a complete simulation [77].

The **Simulation**, **Indifference**, and **Translation** properties correspond to the properties Plotkin proved for his call-by-name simulation $\mathcal{C}_n$.

Given these results one may question what role continuations actually play in $\mathcal{C}_n$ since they are unnecessary for achieving simulation. Are they used to achieve any additional properties not provided by thunks? How are they related to the continuations in Plotkin's call-by-value simulation $\mathcal{C}_v$? We show that $\mathcal{C}_n$ can actually be obtained by extending $\mathcal{C}_v$ to process the abstract representation of thunks and composing the resulting call-by-value simulation $\mathcal{C}_v$ with our call-by-name simulation $\mathcal{T}$, *i.e.*,

$$\lambda\beta_a\eta_v \vdash \mathcal{C}_n\langle\!|e|\!\rangle = (\mathcal{C}_v \circ \mathcal{T})\langle\!|e|\!\rangle.$$

This establishes a previously unrecognized relationship between Plotkin's two continuation-passing simulations. In Section 3.3, we show that Plotkin's **Simulation** property, **Indifference** property, and (most of the) **Translation** property for $\mathcal{C}_n$ follow from the properties of $\mathcal{C}_v$ and $\mathcal{T}$. So as a byproduct, when reasoning about Plotkin's $\mathcal{C}_n$, it is often sufficient to reason about $\mathcal{C}_v$ and the simpler simulation $\mathcal{T}$.

From another perspective, this result provides an additional proof of the folk theorem stating that thunks give the effect of call-by-name under call-by-value. Note that $\mathcal{C}_v$ preserves call-by-value meaning and $\mathcal{C}_n$ preserves call-by-name meaning. The result above states that the call-by-value meaning of $\mathcal{T}\langle\!|e|\!\rangle$ as encoded by $(\mathcal{C}_v \circ \mathcal{T})\langle\!|e|\!\rangle$ corresponds to the call-by-name meaning of e as encoded by $\mathcal{C}_n\langle\!|e|\!\rangle$ — which formalizes the effect of thunks in the folk theorem.

3.1.2 An example

Consider the expression $(\lambda x\,.\,(\lambda y\,.\,x)\,\Omega)\,c$ where Ω represents some term whose evaluation diverges under both call-by-name and call-by-value and where c represents some base constant. Call-by-name evaluation dictates that argument expressions be passed unevaluated to functions. Thus, call-by-name evaluation of the example expression proceeds as follows:

$$\begin{aligned}(\lambda x\,.\,(\lambda y\,.\,x)\,\Omega)\,c &\longmapsto_n (\lambda y\,.\,c)\,\Omega\\ &\longmapsto_n c\end{aligned}$$

Call-by-value evaluation dictates that arguments be simplified to values (*i.e.*, constants or abstractions) before being passed to functions. Call-by-value evaluation of the example expression proceeds as follows:

$$\begin{aligned}(\lambda x\,.\,(\lambda y\,.\,x)\,\Omega)\,c &\longmapsto_v (\lambda y\,.\,c)\,\Omega\\ &\longmapsto_v (\lambda y\,.\,c)\,\Omega'\\ &\longmapsto_v (\lambda y\,.\,c)\,\Omega''\\ &\longmapsto_v \ldots\end{aligned}$$

Since the term Ω never reduces to a value, $\lambda y\,.\,c$ cannot be applied — and the evaluation does not terminate.

The difference between call-by-name and call-by-value evaluation lies in how arguments are treated. To simulate call-by-name with call-by-value evaluation, a mechanism for turning arbitrary argument expressions into values is needed to suspend the evaluation of arguments. This can be accomplished using a suspension constructor *delay*.[3] *delay* e turns the expression e into a value and thus suspends its evaluation. The suspension destructor *force* triggers the evaluation of an expression suspended by *delay*. Therefore, the reduction property

$$force\,(delay\,e) \longmapsto_v e$$

holds for any e. Introducing *delay* and *force* in the example expression provides a simulation of call-by-name under call-by-value evaluation as follows:

$$\begin{aligned}(\lambda x\,.\,(\lambda y\,.\,force\,x)\,(delay\,\Omega))\,(delay\,c) &\longmapsto_v (\lambda y\,.\,force\,(delay\,c))\,(delay\,\Omega)\\ &\longmapsto_v force\,(delay\,c)\\ &\longmapsto_v c\end{aligned}$$

[3] Note that *delay* and *force* technically extend the language beyond the one Plotkin considered. We use these abstract operators to avoid committing ourselves to a particular representation of thunks. However, we point out in Section 3.6 that they can by implemented via λ-abstraction and application. This gives the simulation properties of *delay* and *force* using Plotkin's language.

Applying Plotkin's call-by-name continuation-passing transformation to the example expression also gives a simulation of call-by-name under call-by-value evaluation [76]:

$$(\lambda k\,.\,(\lambda k\,.\,k\,(\lambda x\,.\,\lambda k\,.\,(\lambda k\,.\,k\,(\lambda y\,.\,\lambda k\,.\,x\,k))\,(\lambda z\,.\,(z\,\mathcal{C}_n(\!|\Omega|\!))\,k)))\,(\lambda w\,.\,(w\,(\lambda k\,.\,k\,c))\,k))\,(\lambda x\,.\,x)$$

A tedious but straightforward rewriting shows that the call-by-value evaluation of the term above yields c — the result of the original expression when evaluated under call-by-name. Even after optimizing the above expression by performing "administrative reductions" [20, 76, 86], the evaluation is still more involved than for the thunked term. Performing the reductions by hand gives an appreciation for the simplicity of thunks as a simulation and re-enforces the motivation for systematically exploring their simulation properties.

3.1.3 Summary of contributions and overview

The concept associated with the **Simulation** property for thunks has long been a part of folklore. We focus on the connection with Plotkin's fundamental work on continuation-passing style. As such, we recast the **Simulation** property in terms of Plotkin's original definition. Furthermore, we also establish the **Indifference** property as well as an equational correspondence between calculi, leading to the **Translation** property.

Our factorization of $\mathcal{C}_n$ in terms of $\mathcal{C}_v$ and $\mathcal{T}$ sheds new light on Plotkin's continuation-based simulations. In addition to providing a means of reasoning about $\mathcal{C}_n$ in terms of $\mathcal{C}_v$ and $\mathcal{T}$, the factorization has proved useful for deriving optimized continuation-based simulations for call-by-name and call-by-need languages. We discuss these applications in Section 3.6.

Plotkin's continuation-based simulations were originally stated for untyped λ-terms. Following the presentation of CPS transformations for the typed languages in the preceding chapter, we extend our thunk-based simulation to a typed setting. The factorization of $\mathcal{C}_n$ in terms of $\mathcal{C}_v$ and $\mathcal{T}$ extends to a typed setting as well.

Thunks are often implemented using the "protecting by a λ technique", *i.e.*, by wrapping expressions in parameterless procedures, *e.g.* $\lambda()\,.\,e$, or procedures with unused dummy arguments, *e.g.* $\lambda z\,.\,e$ where z does not occur free in e. We avoid committing ourselves to any particular representation of thunks by using abstract operators *delay* and *force*. In Section 3.6 we state the formal properties of some specific representations. We also discuss various optimizations of thunk-based simulations.

The rest of the chapter is organized as follows. Section 3.2 presents our thunk-based simulation and shows that it satisfies Plotkin's **Indifference**, **Simulation**, and **Translation** properties. Section 3.3 formally connects $\mathcal{C}_n$ and $\mathcal{C}_v$. Section 3.4 extends these results to the typed language Λ^σ. Section 3.5 shows how the call-by-name version of Λ^{σ^+} can be simulated using thunks and the call-by-value version of Λ^{σ^+}. Section 3.6 surveys a variety of thunk-based simulations and discusses formal and practical properties of each variant. Section 3.7 gives a discussion of related work and issues. Section 3.8 gives detailed proofs.

3.2 Thunks

3.2.1 Thunk introduction

As mentioned in Section 3.1, we consider properties of thunks using suspension operators *delay* and *force*. Thus, the language Λ is extended to the language Λ_τ that includes suspension operators.

$$\begin{array}{lcl} e & \in & \mathit{Terms}[\Lambda_\tau] \\ e & ::= & \ldots \mid \mathit{delay}\,e \mid \mathit{force}\,e \end{array}$$

Recall that one of the fundamental properties of thunks is that *delay* suspends the computation of an expression — thereby coercing an arbitrary expression to a value. Therefore, the values of Λ are extended to Λ_τ as follows.

$$\begin{array}{lcll} v & \in & \mathit{Values}_n[\Lambda_\tau] & \\ v & ::= & \ldots \mid \mathit{delay}\,e & \ldots \mathit{where}\ e \in \mathit{Terms}[\Lambda_\tau]. \end{array}$$

$$\begin{array}{lcll} v & \in & \mathit{Values}_v[\Lambda_\tau] & \\ v & \in & \ldots \mid \mathit{delay}\,e & \ldots \mathit{where}\ e \in \mathit{Terms}[\Lambda_\tau]. \end{array}$$

$$
\begin{array}{rcl}
\mathcal{T} & : & \mathit{Terms}[\Lambda] \rightarrow \mathit{Terms}[\Lambda_\tau] \\
\mathcal{T}\langle\!|c|\!\rangle & = & c \\
\mathcal{T}\langle\!|x|\!\rangle & = & \mathit{force}\ x \\
\mathcal{T}\langle\!|\lambda x\,.\,e|\!\rangle & = & \lambda x\,.\,\mathcal{T}\langle\!|e|\!\rangle \\
\mathcal{T}\langle\!|e_0\ e_1|\!\rangle & = & \mathcal{T}\langle\!|e_0|\!\rangle\ (\mathit{delay}\ \mathcal{T}\langle\!|e_1|\!\rangle)
\end{array}
$$

Figure 3.1: Thunk introduction

Figure 3.1 presents the definition of our thunk-based simulation $\mathcal{T}$. $\mathcal{T}$ wraps function arguments with *delay* and identifiers with *force*. In the output of $\mathcal{T}$,

- all function arguments are now suspensions (and therefore values) and thus all identifiers denote suspensions;
- these suspensions are explicitly forced whenever an identifier is encountered.

The following grammar defines the language $\mathcal{T}\langle\!|\Lambda|\!\rangle$ — the language of terms in the image of $\mathcal{T}$.

$$
\begin{array}{rcl}
t & \in & \mathit{Terms}[\mathcal{T}\langle\!|\Lambda|\!\rangle] \\
t & ::= & c \mid \mathit{force}\ x \mid \lambda x\,.\,t \mid t_0\ (\mathit{delay}\ t_1)
\end{array}
$$

Since the translation $\mathcal{T}$ is compositional, it is simple to prove the correctness of the grammar (by structural induction over terms).

3.2.2 Reduction of thunked terms

τ-reduction

The operator *force* triggers the evaluation of a suspension created by *delay*. This is formalized by introducing the following notion of reduction for Λ_τ

Definition 3.1 (τ-reduction)

$$\mathit{force}\ (\mathit{delay}\ e) \rightsquigarrow_\tau e$$

It is easy to show that τ has the Church-Rosser property. The notion of reduction τ generates the τ-calculus as outlined in Section 2.2.3. Combining reductions β and τ generates the $\beta\tau$-calculus. Similarly, β_v and τ give $\beta_v\tau$-calculus. The Church-Rosser property for $\beta\tau$ and $\beta_v\tau$ follows since τ, β, and β_v are Church-Rosser and τ is *orthogonal* to β and β_v [48].

The call-by-name and the call-by-value operational semantics are extended to the language Λ_τ by adding the following rules to the single-step evaluation rules for Λ given in Figure 2.2.

Call-by-name:

$$\frac{e \longmapsto_n e'}{\mathit{force}\ e \longmapsto_n \mathit{force}\ e'} \qquad \mathit{force}\ (\mathit{delay}\ e) \longmapsto_n e$$

Call-by-name:

$$\frac{e \longmapsto_v e'}{\mathit{force}\ e \longmapsto_v \mathit{force}\ e'} \qquad \mathit{force}\ (\mathit{delay}\ e) \longmapsto_v e$$

The stuck terms of Λ_τ are as follows.

$$\begin{array}{lcl} s & \in & \mathit{Stuck}_n[\Lambda_\tau] \\ s & ::= & \ldots \mid \mathit{force}\, c \mid \mathit{force}\,(\lambda x\,.\,e) \mid \mathit{force}\, s \mid (\mathit{delay}\, e_0)\, e_1 \qquad \ldots \mathit{where}\ e, e_0, e_1 \in \mathit{Terms}[\Lambda_\tau]. \end{array}$$

$$\begin{array}{lcl} s & \in & \mathit{Stuck}_v[\Lambda_\tau] \\ s & ::= & \ldots \mid \mathit{force}\, c \mid \mathit{force}\,(\lambda x\,.\,e) \mid \mathit{force}\, s \mid (\mathit{delay}\, e_0)\, e_1 \qquad \ldots \mathit{where}\ e, e_0, e_1 \in \mathit{Terms}[\Lambda_\tau]. \end{array}$$

The single-step and program evaluation characteristics of untyped Λ given in Properties 2.1 and 2.2 extend to Λ_τ in the obvious way.

A language closed under reductions

To determine the formal properties of thunks, we consider the set of terms T which are reachable from $\mathcal{T}(\![\Lambda]\!)$ via β_a and τ reductions.

$$T \stackrel{\text{def}}{=} \{t_2 \in \Lambda_\tau \mid \exists t_1 \in \mathcal{T}(\![\Lambda]\!).\lambda\beta_a\tau \vdash t_1 \longrightarrow\!\!\!\!\!\rightarrow\ t_2\}$$

Note that since all function arguments are values in the image of $\mathcal{T}$, every β-redex is a β_v-redex (and of course, *vice versa*).

The set of terms T can be described with the following grammar.

$$\begin{array}{lcl} t & \in & \mathit{Terms}[\mathcal{T}(\![\Lambda]\!)^*] \\ t & ::= & c \mid \mathit{force}\, x \mid \mathit{force}\,(\mathit{delay}\, t) \mid \lambda x\,.\,t \mid t_0\,(\mathit{delay}\, t_1) \end{array}$$

It is straightforward to show that the language $\mathcal{T}(\![\Lambda]\!)^* = T$ (see Section 3.8.2 for the proof). Note that $\mathcal{T}(\![\Lambda]\!) \subset \mathcal{T}(\![\Lambda]\!)^* \subset \Lambda_\tau$.

3.2.3 A thunk-based simulation

We want to show that thunks are sufficient for establishing a simulation satisfying all of the formal properties of Plotkin's call-by-name continuation-passing simulation. Specifically, we prove the following theorem which recasts Plotkin's theorem for $\mathcal{C}_n$ (Theorem 2.3) in terms of our transformation $\mathcal{T}$.

Theorem 3.1 *For all $e \in \mathit{Progs}[\Lambda]$ and $e_1, e_2 \in \mathit{Terms}[\Lambda]$,*

1. **Indifference:** $\mathit{eval}_v(\mathcal{T}(\![e]\!)) \simeq \mathit{eval}_n(\mathcal{T}(\![e]\!))$
2. **Simulation:** $\mathcal{T}(\![\mathit{eval}_n(e)]\!) \simeq_\tau \mathit{eval}_v(\mathcal{T}(\![e]\!))$
3. **Translation:** $\lambda\beta \vdash e_1 = e_2$ *iff* $\lambda\beta_v\tau \vdash \mathcal{T}(\![e_1]\!) = \mathcal{T}(\![e_2]\!)$ *iff* $\lambda\beta\tau \vdash \mathcal{T}(\![e_1]\!) = \mathcal{T}(\![e_2]\!)$

Note that this theorem mirrors Theorem 2.3 except that the **Simulation** property above holds up to τ-equivalence instead of up to α-equivalence.

Indifference

Plotkin's **Indifference** properties (see Theorem 2.3 and 2.4) state that the final outcome of evaluating a CPS term is independent of whether it is evaluated under call-by-name or call-by-value. In fact, both CPS terms and thunked terms of $\mathcal{T}(\![\Lambda]\!)^*$ enjoy a stronger property — call-by-name and call-by-value coincide not only at the final outcome but also at each single-step evaluation [76, p. 150]. The following property expresses the coincidence of evaluation steps on thunked terms.

Property 3.1 *For all $t \in \mathit{Progs}[\mathcal{T}(\![\Lambda]\!)^*]$,*

$$t \longmapsto_n t' \text{ iff } t \longmapsto_v t'.$$

Proof: by induction over the structure of $\longmapsto$ rules (see Section 3.8.3). ∎

Intuitively, Property 3.1 holds because all arguments of functions are values. Now **Indifference** for thunked terms follows from Property 3.1 and the definitions of eval_n and eval_v (see Section 3.8.3 for details).

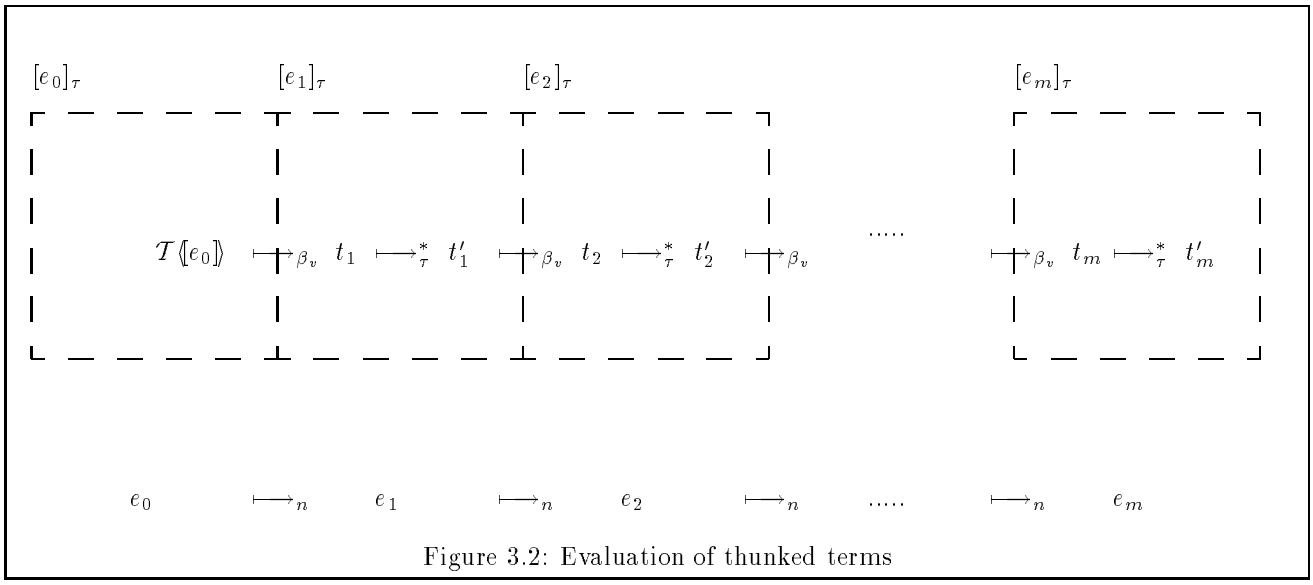


Figure 3.2: Evaluation of thunked terms

Simulation

As noted above, the **Simulation** property for thunks holds up to τ-equivalence instead of up to α-equivalence. Referring to the statement of **Simulation** in Theorem 3.1, the example below illustrates this by taking $e \equiv (\lambda x\,.\,\lambda y\,.\,x)\,c$ and noting that $eval_v(\mathcal{T}\langle\!\langle e \rangle\!\rangle)$ may contain τ-redexes inside the body of an abstraction whereas $\mathcal{T}\langle\!\langle eval_n(e) \rangle\!\rangle$ is in τ-normal-form.

$$eval_v(\mathcal{T}\langle\!\langle e \rangle\!\rangle): \qquad \mathcal{T}\langle\!\langle (\lambda x\,.\,\lambda y\,.\,x)\,c \rangle\!\rangle \;=\; (\lambda x\,.\,\lambda y\,.\,force\,x)\,(delay\,c) \longmapsto_v \lambda y\,.\,force\,(delay\,c)$$

$$\mathcal{T}\langle\!\langle eval_n(e) \rangle\!\rangle: \qquad (\lambda x\,.\,\lambda y\,.\,x)\,c \longmapsto_n \lambda y\,.\,c \text{ and } \mathcal{T}\langle\!\langle \lambda y\,.\,c \rangle\!\rangle \;=\; \lambda y\,.\,c$$

In general, the steps involved in $eval_n(e)$ and $eval_v(\mathcal{T}\langle\!\langle e \rangle\!\rangle)$ can be pictured as in Figure 3.2. In the figure, the notation $[e]_\tau$ denotes the equivalence class of terms which are τ-equivalent to $\mathcal{T}\langle\!\langle e \rangle\!\rangle$, *i.e.*,

$$[e]_\tau \;\stackrel{\text{def}}{=}\; \{t \in Terms[\mathcal{T}\langle\!\langle \Lambda \rangle\!\rangle^*] \mid \lambda\tau \vdash t = \mathcal{T}\langle\!\langle e \rangle\!\rangle\}.$$

Also, $\longmapsto_\tau$ denotes a call-by-value evaluation step $\longmapsto_v$ which correspond to a τ-reduction. Similarly, $\longmapsto_{\beta_v}$ denotes a call-by-value evaluation step $\longmapsto_v$ which correspond to a β_v-reduction.

Proving **Simulation** for thunks amounts to formally establishing the properties of Figure 3.2. Since the τ-reductions alter the structure of terms in a very regular way it is possible to describe equivalence classes of τ-equivalent terms using the following grammars.

$$\begin{array}{rcl}
[c]_\tau & ::= & c \mid force\,(delay\,[c]_\tau) \\
[x]_\tau & ::= & force\,x \mid force\,(delay\,[x]_\tau) \\
[\lambda x\,.\,e]_\tau & ::= & \lambda x\,.\,[e]_\tau \mid force\,(delay\,[\lambda x\,.\,e]_\tau) \\
[e_0\,e_1]_\tau & ::= & [e_0]_\tau\,(delay\,[e_1]_\tau) \mid force\,(delay\,[e_0\,e_1]_\tau)
\end{array}$$

The correctness of the grammars follows from the fact that τ-equality is a congruence over the structure of terms.

The following property states that each $\longmapsto_n$ evaluation step on a source term implies corresponding $\longmapsto_v$ steps on a thunked term. Properties 3.3 and 3.4 state that each $\longmapsto_v$ evaluation step on a thunked term corresponds to zero or one $\longmapsto_n$ steps on a source term.

Property 3.2 *For all* $e_0, e_1 \in Terms[\Lambda]$,

$$e_0 \longmapsto_n e_1 \;\Rightarrow\; \forall t_0 \in [e_0]_\tau\,.\,\exists t_2 \in [e_0]_\tau\,.\,\exists t_1 \in [e_1]_\tau\,.\; t_0 \longmapsto_\tau^* t_2 \longmapsto_{\beta_v} t_1$$

$$
\begin{array}{rcl}
\mathcal{T}^{-1} & : & \mathit{Terms}[\Lambda_\tau] \rightarrow \mathit{Terms}[\Lambda] \\
\mathcal{T}^{-1}(\!|c|\!) & = & c \\
\mathcal{T}^{-1}(\!|x|\!) & = & x \\
\mathcal{T}^{-1}(\!|\lambda x\,.\,e|\!) & = & \lambda x\,.\,\mathcal{T}^{-1}(\!|e|\!) \\
\mathcal{T}^{-1}(\!|e_0\ e_1|\!) & = & \mathcal{T}^{-1}(\!|e_0|\!)\ \mathcal{T}^{-1}(\!|e_1|\!) \\
\mathcal{T}^{-1}(\!|\mathit{delay}\ e|\!) & = & \mathcal{T}^{-1}(\!|e|\!) \\
\mathcal{T}^{-1}(\!|\mathit{force}\ e|\!) & = & \mathcal{T}^{-1}(\!|e|\!)
\end{array}
$$

Figure 3.3: Thunk elimination

Proof: by induction over the structure of the $\longmapsto_n$ rules (see Section 3.8.4). ∎

Property 3.3 *For all $e \in \mathit{Terms}[\Lambda]$, and $t \in [e]_\tau$,*

$$t \longmapsto_\tau t' \ \Rightarrow\ t' \in [e]_\tau$$

Proof: immediate, since $[e]_\tau$ is closed under τ-reductions. ∎

Property 3.4 *For all $e_0, e_1 \in \mathit{Terms}[\Lambda]$, and $t_0 \in [e_0]_\tau$ and $t_1 \in [e_1]_\tau$,*

$$t_0 \longmapsto_{\beta_v} t_1 \ \Rightarrow\ e_0 \longmapsto_n e_1$$

Proof: by induction over the structure of the $\longmapsto_v$ rules (see Section 3.8.4). ∎

As detailed in Section 3.8.4, it is also the case that every terminating evaluation sequence over source terms corresponds to a terminating evaluation sequence over thunked terms (and vice-versa). Thus, the **Simulation** for thunks holds.

Translation

To prove **Translation** for thunks, we establish an equational correspondence between the language Λ under theory $\lambda\beta$ and language $\mathcal{T}(\!|\Lambda|\!)^*$ under theory $\lambda\beta_a\tau$. Basically, equational correspondence holds when a bijective correspondence exists between equivalence classes of the two theories.

The thunk introduction $\mathcal{T}$ of Figure 3.1 establishes a mapping from Λ to $\mathcal{T}(\!|\Lambda|\!)^*$. For the reverse direction, the thunk elimination $\mathcal{T}^{-1}$ of Figure 3.3 establishes a mapping from $\mathcal{T}(\!|\Lambda|\!)^*$ back to Λ.

The formal relationship between source terms and thunked terms can now be stated as follows.

Theorem 3.2 (Equational Correspondence) *For all $e, e_1, e_2 \in \mathit{Terms}[\Lambda]$ and $t, t_1, t_2 \in \mathit{Terms}[\mathcal{T}(\!|\Lambda|\!)^*]$.*

1. $\lambda\beta \vdash e = (\mathcal{T}^{-1} \circ \mathcal{T})(\!|e|\!)$
2. $\lambda\beta_a\tau \vdash t = (\mathcal{T} \circ \mathcal{T}^{-1})(\!|t|\!)$
3. $\lambda\beta \vdash e_1 = e_2$ *iff* $\lambda\beta_a\tau \vdash \mathcal{T}(\!|e_1|\!) = \mathcal{T}(\!|e_2|\!)$
4. $\lambda\beta_a\tau \vdash t_1 = t_2$ *iff* $\lambda\beta \vdash \mathcal{T}^{-1}(\!|t_1|\!) = \mathcal{T}^{-1}(\!|t_2|\!)$

Note that component 3 of Theorem 3.2 corresponds to the thunk **Translation** property (component 3 of Theorem 3.1). In proving the theorem above, we first characterize the interaction of $\mathcal{T}$ and $\mathcal{T}^{-1}$ (components 1 and 2 of Theorem 3.2). Then, we examine the relation between reductions in the theories $\lambda\beta$ and $\lambda\beta_a\tau$ (components 3 and 4 of Theorem 3.2).

The following property states that $\mathcal{T}^{-1} \circ \mathcal{T}$ is the identity function over Λ.

Property 3.5 *For all* $e \in Terms[\Lambda]$, $e = (\mathcal{T}^{-1} \circ \mathcal{T})(\!|e|\!)$.

Proof: by induction of the structure of e. ∎

Intuitively, this follows from the fact that $\mathcal{T}^{-1}$ simply removes all suspension operators. However, removing suspension operators has the effect of collapsing τ-redexes. This leads to a slightly weaker condition for the opposite direction which is captured in the following property.

Property 3.6 *For all* $t \in Terms[\mathcal{T}(\!|\Lambda|\!)^*]$, $\lambda\tau \vdash t = (\mathcal{T} \circ \mathcal{T}^{-1})(\!|t|\!)$.

Proof: by induction on the structure of t. ∎

In other words, $\mathcal{T} \circ \mathcal{T}^{-1}$ is not the identity function, but maintains τ-equivalence. For example,

$$(\mathcal{T} \circ \mathcal{T}^{-1})(\!|(\lambda x\,.\,force\,(delay\,c))\,(delay\,c)|\!) \;=\; \mathcal{T}(\!|(\lambda x\,.\,c)\,c|\!) \;=\; (\lambda x\,.\,c)\,(delay\,c).$$

Components 1 and 2 of Theorem 3.2 follow immediately from Properties 3.5 and 3.6.

We now establish components 3 and 4 of Theorem 3.2. The following property shows that any reduction step in Λ corresponds to *one or more* reduction steps in $\mathcal{T}(\!|\Lambda|\!)^*$.

Property 3.7 *For all* $e_1, e_2 \in Terms[\Lambda]$, $\lambda\beta \vdash e_1 \longrightarrow e_2 \;\Rightarrow\; \lambda\beta_a\tau \vdash \mathcal{T}(\!|e_1|\!) \longrightarrow\!\!\!\!\!\rightarrow\; \mathcal{T}(\!|e_2|\!)$.

Proof: See Section 3.8.5 ∎

For example, the β-reduction

$$\lambda\beta \vdash e_1 \equiv (\lambda x\,.\,x\,c)\,(\lambda y\,.\,y) \longrightarrow (\lambda y\,.\,y)\,c \equiv e_2$$

corresponds to the β_a-reduction

$$\begin{aligned}\lambda\beta_a\tau \vdash \mathcal{T}(\!|e_1|\!) &\equiv (\lambda x\,.\,(force\,x)\,(delay\,c))\,(delay\,(\lambda y\,.\,force\,y))\\ &\longrightarrow (force\,(delay\,(\lambda y\,.\,force\,y)))\,(delay\,c)\end{aligned}$$

However, an additional τ-reduction step (and in general multiple τ-reduction steps) is needed to reach $\mathcal{T}(\!|e_2|\!)$, *i.e.*,

$$\lambda\beta_a\tau \vdash (force\,(delay\,(\lambda y\,.\,force\,y)))\,(delay\,c) \longrightarrow (\lambda y\,.\,force\,y)\,(delay\,c) \equiv \mathcal{T}(\!|e_2|\!).$$

For the other direction, the following property states that any reduction step in $\mathcal{T}(\!|\Lambda|\!)^*$ corresponds to *zero or one* reduction steps in Λ.

Property 3.8 *For all* $t_1, t_2 \in Terms[\mathcal{T}(\!|\Lambda|\!)^*]$, $\lambda\beta_a\tau \vdash t_1 \longrightarrow t_2 \;\Rightarrow\; \lambda\beta \vdash \mathcal{T}^{-1}(\!|t_1|\!) \longrightarrow\!\!\!\!\!\rightarrow\; \mathcal{T}^{-1}(\!|t_2|\!)$

Proof: See Section 3.8.5 ∎

Specifically, a τ-reduction in $\mathcal{T}(\!|\Lambda|\!)^*$ implies no reduction steps in Λ. This is because $\mathcal{T}^{-1}$ collapses τ-redexes. For example,

$$\lambda\beta_a\tau \vdash t_1 \equiv force\,(delay\,c) \longrightarrow c \equiv t_2,$$

but $\mathcal{T}^{-1}(\!|t_1|\!) = c = \mathcal{T}^{-1}(\!|t_2|\!)$ so no reductions occur.

A β_a-reduction in $\mathcal{T}(\!|\Lambda|\!)^*$ implies one β-reduction in Λ. For example, the β_a-reduction

$$\begin{aligned}\lambda\beta_a\tau \vdash t_1 &\equiv (\lambda x\,.\,(force\,x)\,(delay\,c))\,(delay\,(\lambda y\,.\,force\,y))\\ &\longrightarrow (force\,(delay\,\lambda y\,.\,force\,y))\,(delay\,c) \equiv t_2\end{aligned}$$

corresponds to the β-reduction

$$\lambda\beta \vdash \mathcal{T}^{-1}(\!|t_1|\!) \equiv (\lambda x\,.\,x\,c)\,(\lambda y\,.\,y) \longrightarrow (\lambda y\,.\,y)\,c \equiv \mathcal{T}^{-1}(\!|t_2|\!).$$

Components 3 and 4 of Theorem 3.2 are proved as follows:

(1) $\lambda\beta \vdash e_1 \longrightarrow\!\!\!\!\!\rightarrow e_2 \;\Rightarrow\; \lambda\beta_a\tau \vdash \mathcal{T}(\!|e_1|\!) \longrightarrow\!\!\!\!\!\rightarrow \mathcal{T}(\!|e_2|\!)$
...by induction on number of reductions.

(2) $\lambda\beta \vdash e_1 = e_2 \;\Rightarrow\; \lambda\beta_a\tau \vdash \mathcal{T}(\!|e_1|\!) = \mathcal{T}(\!|e_2|\!)$ *...by Church-Rosser and* (1).

(3) $\lambda\beta_a\tau \vdash t_1 \longrightarrow\!\!\!\!\!\rightarrow t_2 \;\Rightarrow\; \lambda\beta \vdash \mathcal{T}^{-1}(\!|t_1|\!) \longrightarrow\!\!\!\!\!\rightarrow \mathcal{T}^{-1}(\!|t_2|\!)$
...by induction on number of reductions.

(4) $\lambda\beta_a\tau \vdash t_1 = t_2 \;\Rightarrow\; \lambda\beta \vdash \mathcal{T}^{-1}(\!|t_1|\!) = \mathcal{T}^{-1}(\!|t_2|\!)$ *...by Church-Rosser and* (3).

(5) $\lambda\beta_a\tau \vdash \mathcal{T}(\!|e_1|\!) = \mathcal{T}(\!|t_2|\!) \;\Rightarrow\; \lambda\beta \vdash (\mathcal{T}^{-1} \circ \mathcal{T})(\!|e_1|\!) = (\mathcal{T}^{-1} \circ \mathcal{T})(\!|e_2|\!)$ *...by* (4).

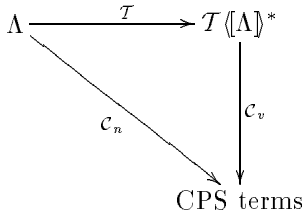
(6) $\Rightarrow\; \lambda\beta \vdash e_1 = e_2$ *...by Property 3.5.*

(7) $\lambda\beta \vdash \mathcal{T}^{-1}(\!|t_1|\!) = \mathcal{T}^{-1}(\!|t_2|\!) \;\Rightarrow\; \lambda\beta_a\tau \vdash (\mathcal{T} \circ \mathcal{T}^{-1})(\!|t_1|\!) = (\mathcal{T} \circ \mathcal{T}^{-1})(\!|t_2|\!)$ *...by* (2).

(8) $\Rightarrow\; \lambda\beta_a\tau \vdash t_1 = t_2$ *...by Property 3.6.*

3.3 Connecting the thunk-based and the continuation-based simulations

We have established that call-by-name can be simulated under call-by-value using thunked terms. We now extend Plotkin's $\mathcal{C}_v$ to a call-by-value CPS transformation $\mathcal{C}_v$ over thunked terms. Clearly $\mathcal{C}_v$ should preserve call-by-value meaning, but in this case call-by-value evaluation of thunked terms gives call-by-name meaning. Therefore, one would expect $\mathcal{C}_v \circ \mathcal{T}$ to produce continuation-passing terms that encode call-by-name meaning. In fact, we show that for any $e \in \Lambda$, $(\mathcal{C}_v \circ \mathcal{T})(\!|e|\!)$ is $\beta_a\eta_v$-equivalent to $\mathcal{C}_n(\!|e|\!)$. As a byproduct, $\mathcal{C}_n$ can be factored as $\mathcal{C}_v \circ \mathcal{T}$ as captured by the following diagram.



Section 3.6 discusses applications of this factorization.

3.3.1 CPS transformation of thunk constructs

We begin by extending $\mathcal{C}_v$ to transform the suspension constructs *delay* and *force*. The definitions follow directly from the two fundamental properties of thunks: (1) $delay\, e$ is a value; and (2) $force\,(delay\, e) \longmapsto_v e$.

First, since $delay\, e \in Values_v[\Lambda_\tau]$, $\mathcal{C}_v(\!|delay\, e|\!) = \lambda k\,.\,k\,(\mathcal{C}_v\langle delay\, e\rangle)$. Notice that in the definition of $\mathcal{C}_v$ (see Figure 2.10) all expressions $\mathcal{C}_v(\!|e|\!)$ require a continuation for evaluation. Therefore, an expression is delayed by simply not passing it a continuation, *i.e.*, $\mathcal{C}_v\langle delay\, e\rangle = \mathcal{C}_v(\!|e|\!)$. As required, $\mathcal{C}_v(\!|e|\!)$ is a value. This effectively implements *delay* by "protecting by a λ". However, the "protecting λ" is not associated with a dummy parameter but with the continuation parameter in $\mathcal{C}_v\langle delay\, e\rangle = \mathcal{C}_v(\!|e|\!)$.

Since the suspension of an expression is achieved by *depriving* it of a continuation, a suspension is naturally forced by *supplying* it with a continuation. This leads to the following definition.

$$\mathcal{C}_v(\!|force\, e|\!) \;=\; \lambda k\,.\,\mathcal{C}_v(\!|e|\!)\,(\lambda v\,.\,v\,k)$$

$$
\begin{array}{rcl}
\mathcal{C}_v\langle\!|\cdot|\!\rangle & : & \mathit{Terms}[\Lambda] \to \mathit{Values}_v[\Lambda] \\
& \cdots & \\
\mathcal{C}_v\langle\!|\mathit{force}\, e|\!\rangle & = & \lambda k\,.\,\mathcal{C}_v\langle\!|e|\!\rangle\,(\lambda y\,.\,y\,k) \\
\\
\mathcal{C}_v\langle\cdot\rangle & : & \mathit{Values}_v[\Lambda] \to \mathit{Values}_v[\Lambda] \\
& \cdots & \\
\mathcal{C}_v\langle\mathit{delay}\, e\rangle & = & \mathcal{C}_v\langle\!|e|\!\rangle
\end{array}
$$

Figure 3.4: Call-by-value CPS transformation (extended to Λ_τ)

The correctness of these definitions follows from the fact that they preserve the property of τ-reduction, *i.e.*, $\mathcal{C}_v\langle\!|\mathit{force}\,(\mathit{delay}\, e)|\!\rangle\, k \longmapsto_a^* \mathcal{C}_v\langle\!|e|\!\rangle\, k$ for an arbitrary continuation k.

Property 3.9 *For all $e \in \mathit{Terms}[\Lambda_\tau]$ and arbitrary continuation k,*

$$\mathcal{C}_v\langle\!|\mathit{force}\,(\mathit{delay}\, e)|\!\rangle\, k \longmapsto_a^* \mathcal{C}_v\langle\!|e|\!\rangle\, k$$

Proof:

$$
\begin{array}{rcl}
\mathcal{C}_v\langle\!|\mathit{force}\,(\mathit{delay}\, e)|\!\rangle\, k & = & (\lambda k\,.\,\mathcal{C}_v\langle\!|\mathit{delay}\, e|\!\rangle\,(\lambda v\,.\,v\,k))\,k \\
& \longmapsto_a & \mathcal{C}_v\langle\!|\mathit{delay}\, e|\!\rangle\,(\lambda v\,.\,v\,k) \\
& = & (\lambda k\,.\,k\,(\mathcal{C}_v\langle\!|e|\!\rangle))\,(\lambda v\,.\,v\,k) \\
& \longmapsto_a & (\lambda v\,.\,v\,k)\,\mathcal{C}_v\langle\!|e|\!\rangle \\
& \longmapsto_a & \mathcal{C}_v\langle\!|e|\!\rangle\, k
\end{array}
$$

∎

Property 3.10 *For all $e \in \mathit{Terms}[\Lambda_\tau]$,*

$$\lambda\beta_a\eta_v \vdash \mathcal{C}_v\langle\!|\mathit{force}\,(\mathit{delay}\, e)|\!\rangle = \mathcal{C}_v\langle\!|e|\!\rangle$$

Proof: Straightforward. ∎

The clauses of Figure 3.4 extend the definition of $\mathcal{C}_v$ on Λ in Figure 2.10 to Λ_τ. The properties of $\mathcal{C}_v$ as stated in Theorem 2.4 can be extended to the language of thunks closed under reduction $\mathcal{T}\langle\!|\Lambda|\!\rangle^*$.

Theorem 3.3 *For all $t \in \mathit{Progs}[\mathcal{T}\langle\!|\Lambda|\!\rangle^*]$ and $t_1, t_2 \in \mathit{Terms}[\mathcal{T}\langle\!|\Lambda|\!\rangle^*]$,*

1. **Indifference:** $\mathit{eval}_n(\mathcal{C}_v\langle\!|t|\!\rangle\,(\lambda y\,.\,y)) \simeq \mathit{eval}_v(\mathcal{C}_v\langle\!|t|\!\rangle\,(\lambda y\,.\,y))$
2. **Simulation:** $\mathcal{C}_v\langle\mathit{eval}_v(t)\rangle \simeq \mathit{eval}_n(\mathcal{C}_v\langle\!|t|\!\rangle\,(\lambda y\,.\,y))$
3. **Translation:** *If* $\lambda\beta_v\tau \vdash t_1 = t_2$ *then* $\lambda\beta_v\eta_v \vdash \mathcal{C}_v\langle\!|t_1|\!\rangle = \mathcal{C}_v\langle\!|t_2|\!\rangle$.
   Also $\lambda\beta_v\eta_v \vdash \mathcal{C}_v\langle\!|t_1|\!\rangle = \mathcal{C}_v\langle\!|t_2|\!\rangle$ *iff* $\lambda\beta\eta_v \vdash \mathcal{C}_v\langle\!|t_1|\!\rangle = \mathcal{C}_v\langle\!|t_2|\!\rangle$.

Proof: Follows in a straightforward manner from Theorem 2.4 and Properties 3.9 and 3.10. ∎

One might expect the theorem to also hold for Λ_τ terms and programs. Unfortunately, the **Simulation** property fails for Λ_τ because some stuck Λ_τ programs do not stick when translated to CPS. For example, $\mathit{eval}_v(\mathit{force}\,(\lambda x\,.\,x))$ sticks but $\mathit{eval}_n(\mathcal{C}_v\langle\!|\mathit{force}\,(\lambda x\,.\,x)|\!\rangle\,(\lambda y\,.\,y)) = \lambda k\,.\,k\,(\lambda y\,.\,y)$. This mismatch on sticking is due to "improper" uses of *delay* and *force*. The proof of Theorem 3.3 goes through since the syntax of $\mathcal{T}\langle\!|\Lambda|\!\rangle^*$ only allows "proper" uses of *delay* and *force*. Furthermore, an analogue of Theorem 3.3 *does hold* for the typed language with thunks Λ_τ^σ (this is presented in Section 3.4, Theorem 3.5) since well-typedness eliminates the possibility of stuck terms.

3.3.2 The connection between the thunk-based and continuation-based simulations

We now show the connection between Plotkin's continuation-based simulations and our thunk-based simulation. First, we show that the call-by-name CPS transformation $\mathcal{C}_n$ can be factored into two conceptually distinct steps:

- the suspension of argument evaluation (captured in $\mathcal{T}$);
- the sequentialization of function application to give the usual tail-calls of CPS terms (captured in $\mathcal{C}_v$).

This is formalized by Theorem 3.4 below but first we state following simple property used in the proof.

Property 3.11 *For all* $e_1 \in Terms[\Lambda_\tau]$ *and* $e_2 \in Terms[\Lambda]$,

$$\lambda\beta_a \vdash \mathcal{C}_v\langle\!|delay\, e_1|\!\rangle\,(\lambda y\,.\,e_2) \longrightarrow (\lambda y\,.\,e_2)\,\mathcal{C}_v\langle\!|e_1|\!\rangle$$

Theorem 3.4 *For all* $e \in Terms[\Lambda]$,

$$\lambda\beta_a\eta_v \vdash (\mathcal{C}_v \circ \mathcal{T})\langle\!|e|\!\rangle = \mathcal{C}_n\langle\!|e|\!\rangle$$

Proof: by structural induction over e:

case $e \equiv c$:

$$\begin{array}{rcl}(\mathcal{C}_v \circ \mathcal{T})\langle\!|c|\!\rangle & = & \mathcal{C}_v\langle\!|c|\!\rangle \\ & = & \lambda k\,.\,k\,c \\ & = & \mathcal{C}_n\langle\!|c|\!\rangle\end{array}$$

case $e \equiv x$:

$$\begin{array}{rcl}(\mathcal{C}_v \circ \mathcal{T})\langle\!|x|\!\rangle & = & \mathcal{C}_v\langle\!|force\, x|\!\rangle \\ & = & \lambda k\,.\,(\lambda k\,.\,k\,x)\,(\lambda y\,.\,y\,k) \\ & \longrightarrow_{\beta_a} & \lambda k\,.\,(\lambda y\,.\,y\,k\,x) \\ & \longrightarrow_{\beta_a} & \lambda k\,.\,x\,k \\ & \longrightarrow_{\eta_v} & x \\ & = & \mathcal{C}_n\langle\!|x|\!\rangle\end{array}$$

case $e \equiv \lambda x\,.\,e'$:

$$\begin{array}{rcl}(\mathcal{C}_v \circ \mathcal{T})\langle\!|\lambda x\,.\,e'|\!\rangle & = & \lambda k\,.\,k\,(\lambda x\,.\,(\mathcal{C}_v \circ \mathcal{T})\langle\!|e'|\!\rangle) \\ & =_{\beta_a\eta_v} & \lambda k\,.\,k\,(\lambda x\,.\,\mathcal{C}_n\langle\!|e'|\!\rangle) \quad \textit{...by the ind. hyp.} \\ & = & \mathcal{C}_n\langle\!|\lambda x\,.\,e'|\!\rangle\end{array}$$

case $e \equiv e_0\,e_1$:

$$\begin{array}{rcl}(\mathcal{C}_v \circ \mathcal{T})\langle\!|e_0\,e_1|\!\rangle & = & \mathcal{C}_v\langle\!|\mathcal{T}\langle\!|e_0|\!\rangle\,(delay\,\mathcal{T}\langle\!|e_1|\!\rangle)|\!\rangle \\ & = & \lambda k\,.\,(\mathcal{C}_v \circ \mathcal{T})\langle\!|e_0|\!\rangle\,(\lambda y_0\,.\,\mathcal{C}_v\langle\!|delay\,\mathcal{T}\langle\!|e_1|\!\rangle|\!\rangle\,(\lambda y_1\,.\,(y_0\,y_1)\,k)) \\ & \longrightarrow_{\beta_a} & \lambda k\,.\,(\mathcal{C}_v \circ \mathcal{T})\langle\!|e_0|\!\rangle\,(\lambda y_0\,.\,(\lambda y_1\,.\,(y_0\,y_1)\,k)\,(\mathcal{C}_v \circ \mathcal{T})\langle\!|e_1|\!\rangle) \quad \textit{...Property 3.11} \\ & \longrightarrow_{\beta_a} & \lambda k\,.\,(\mathcal{C}_v \circ \mathcal{T})\langle\!|e_0|\!\rangle\,(\lambda y_0\,.\,(y_0\,(\mathcal{C}_v \circ \mathcal{T})\langle\!|e_1|\!\rangle)\,k) \\ & =_{\beta_a\eta_v} & \lambda k\,.\,\mathcal{C}_n\langle\!|e_0|\!\rangle\,(\lambda y_0\,.\,(y_0\,(\mathcal{C}_n\langle\!|e_1|\!\rangle))\,k) \quad \textit{...by the ind. hyp.} \\ & = & \mathcal{C}_n\langle\!|e_0\,e_1|\!\rangle\end{array}$$

∎

3.3.3 Discussion

The two flavors of the CPS transformation (*i.e.*, $\mathcal{C}_v$ and $\mathcal{C}_n$) act alike in that they both yield evaluation-order independent and continuation-passing λ-terms. The fact that each function is not only applied to its regular argument, but also to a continuation — a representation of how succeeding evaluation will use the value produced by the function — leads to the tail-call property of CPS terms.

$\mathcal{C}_v$ and $\mathcal{C}_n$ act differently because of the different treatment of functions arguments in the call-by-name and call-by-value strategies. Call-by-value functions must be applied to values only and thus the call-by-value

CPS transformation explicitly encodes the evaluation of all function arguments before the function is applied. Under call-by-name, there is no distinction between arbitrary argument expressions and values — the evaluation of all arguments is suspended. Thus each argument is explicitly encoded as a suspended computation which simply needs a continuation before evaluation can proceed. The explicit encoding of each strategy leads to the evaluation-strategy independence of CPS terms.

The crux of the matter is that the thunk simulation $\mathcal{T}$ explicitly encodes the call-by-name treatment of arguments as suspensions into terms. Since a suspension (*i.e.*, a thunk) is a value, $\mathcal{C}_v$ has nothing further to do than to add the tail-call property (via continuations).

3.3.4 Implications

When working with CPS, one often needs to establish technical properties for both $\mathcal{C}_v$ and $\mathcal{C}_n$. This requires two sets of proofs involving the complicated structure of CPS terms. However, by applying the factoring result, often only one set of proofs over CPS terms is necessary. The second set of proofs deals with thunked terms which have a much simpler structure. For instance, **Indifference** and **Simulation** for Plotkin's $\mathcal{C}_n$ follow from **Indifference** and **Simulation** for $\mathcal{C}_v$ and $\mathcal{T}$ and Theorem 3.4.

For **Indifference**, let $e, c \in \mathit{Terms}[\Lambda]$ where c is a base constant. Then

$$
\begin{array}{lll}
 & eval_v(\mathcal{C}_n\langle\![e]\!\rangle\,(\lambda y\,.\,y)) = c & \\
\Leftrightarrow & eval_v((\mathcal{C}_v \circ \mathcal{T})\langle\![e]\!\rangle\,(\lambda y\,.\,y)) = c & \textit{...by Theorem 3.4 and soundness of } \beta_v\eta_v \textit{ (}\dagger\textit{)}, \\
\Leftrightarrow & eval_n((\mathcal{C}_v \circ \mathcal{T})\langle\![e]\!\rangle\,(\lambda y\,.\,y)) = c & \textit{...by Theorem 3.3 } \textbf{(Indifference)}, \\
\Leftrightarrow & eval_n(\mathcal{C}_n\langle\![e]\!\rangle\,(\lambda y\,.\,y)) = c & \textit{...by Theorem 3.4 and soundness of } \beta\eta_v \textit{ (}\ddagger\textit{)}.
\end{array}
$$

For **Simulation**, let $e, c \in \Lambda$ where c is a base constant. Then

$$
\begin{array}{lll}
 & eval_n(e) = c & \\
\Leftrightarrow & eval_v(\mathcal{T}\langle\![e]\!\rangle) = c & \textit{...by Theorem 3.1 } \textbf{(Simulation)}, \\
\Leftrightarrow & eval_n((\mathcal{C}_v \circ \mathcal{T})\langle\![e]\!\rangle\,(\lambda y\,.\,y)) = c & \textit{...by Theorem 3.3 } \textbf{(Simulation)}, \\
\Leftrightarrow & eval_v((\mathcal{C}_v \circ \mathcal{T})\langle\![e]\!\rangle\,(\lambda y\,.\,y)) = c & \textit{...by Theorem 3.3 } \textbf{(Indifference)}, \\
\Leftrightarrow & eval_v(\mathcal{C}_n\langle\![e]\!\rangle\,(\lambda y\,.\,y)) = c & \textit{...by Theorem 3.4 and soundness of } \beta_v\eta_v \textit{ (}\dagger\textit{)}.
\end{array}
$$

Note that in the above results, the evaluation gives a constant c. If evaluation gives an abstraction, the result holds up to $\beta_a\eta_v$-equality instead of up to syntactic identity.

For **Translation**, it is not possible to establish Theorem 2.3 (**Translation** for $\mathcal{C}_n$) in the manner above since Theorem 2.4 (**Translation** for $\mathcal{C}_v$) is weaker in comparison. However, the following weaker version can be derived.

$$
\begin{array}{lll}
 & \lambda\beta \vdash e_1 = e_2 & \\
\Leftrightarrow & \lambda\beta_v\tau \vdash \mathcal{T}\langle\![e_1]\!\rangle = \mathcal{T}\langle\![e_2]\!\rangle & \textit{...by Theorem 3.1 } \textbf{(Translation)}. \\
\Rightarrow & \lambda\beta_v\eta_v \vdash (\mathcal{C}_v \circ \mathcal{T})\langle\![e_1]\!\rangle = (\mathcal{C}_v \circ \mathcal{T})\langle\![e_2]\!\rangle & \textit{...by Theorem 3.3 } \textbf{(Translation)}. \\
\Leftrightarrow & \lambda\beta_v\eta_v \vdash \mathcal{C}_n\langle\![e_1]\!\rangle = \mathcal{C}_n\langle\![e_2]\!\rangle & \textit{...by Theorem 3.4.} \\
\Leftrightarrow & \lambda\beta\eta_v \vdash \mathcal{C}_n\langle\![e_1]\!\rangle = \mathcal{C}_n\langle\![e_2]\!\rangle & \textit{...by Theorem 3.4.}
\end{array}
$$

This can be summarized as follows.

If $\lambda\beta \vdash e_1 = e_2$ then $\lambda\beta_v\eta_v \vdash \mathcal{C}_n\langle\![e_1]\!\rangle = \mathcal{C}_n\langle\![e_2]\!\rangle$.
Also $\lambda\beta_v\eta_v \vdash \mathcal{C}_n\langle\![e_1]\!\rangle = \mathcal{C}_n\langle\![e_2]\!\rangle$ *iff* $\lambda\beta\eta_v \vdash \mathcal{C}_n\langle\![e_1]\!\rangle = \mathcal{C}_n\langle\![e_2]\!\rangle$.

3.4 Thunks for the typed core language Λ^σ

In this section we extend the thunk transformation $\mathcal{T}$ to the typed language Λ^σ and show that it preserves well-typedness of terms. In addition, we show that **Indifference**, **Simulation**, and equational correspondence for $\mathcal{T}$ as well as the relationship between $\mathcal{C}_v \circ \mathcal{T}$ and $\mathcal{C}_n$ extends to Λ^σ.

3.4.1 Thunk introduction for Λ^σ

$$(const) \qquad \Gamma \vdash c : \iota$$

$$(var) \qquad \Gamma \vdash x : \Gamma(x)$$

$$(recvar) \qquad \Gamma \vdash f : \Gamma(f)$$

$$(abs) \qquad \frac{\Gamma, x:\sigma_1 \vdash e : \sigma_2}{\Gamma \vdash \lambda x\,.\,e : \sigma_1 \rightarrow \sigma_2}$$

$$(rec) \qquad \frac{\Gamma, x:\sigma_1, f:\sigma_1 \rightarrow \sigma_2 \vdash e : \sigma_2}{\Gamma \vdash rec\, f(x).\, e : \sigma_1 \rightarrow \sigma_2}$$

$$(app) \qquad \frac{\Gamma \vdash e_0 : \sigma_1 \rightarrow \sigma_2 \qquad \Gamma \vdash e_1 : \sigma_1}{\Gamma \vdash e_0\, e_1 : \sigma_2}$$

$$(delay) \qquad \frac{\Gamma \vdash_\tau e : \sigma}{\Gamma \vdash_\tau delay\, e : \widetilde{\sigma}}$$

$$(force) \qquad \frac{\Gamma \vdash_\tau e : \widetilde{\sigma}}{\Gamma \vdash_\tau force\, e : \sigma}$$

$$\begin{array}{lcl} \sigma & \in & Types[\Lambda^\sigma_\tau] \\ \sigma & ::= & \iota \mid \sigma_1 \rightarrow \sigma_2 \mid \widetilde{\sigma} \end{array}$$

$$\begin{array}{lcl} \Gamma & \in & Assums[\Lambda^\sigma_\tau] \\ \Gamma & ::= & \cdot \mid \Gamma, x:\sigma \mid \Gamma, f:\sigma \end{array}$$

Figure 3.5: Typing rules for Λ^σ_τ

$$
\begin{array}{rcl}
\mathcal{T} & : & \mathit{Terms}[\Lambda^\sigma] \rightarrow \mathit{Terms}[\Lambda^\sigma_\tau] \\
 & \cdots & \\
\mathcal{T}\langle\!\langle [\![f]\!] \rangle\!\rangle & = & f \\
\mathcal{T}\langle\!\langle [\![\mathit{rec}\, f(x).\, e]\!] \rangle\!\rangle & = & \mathit{rec}\, f(x).\, \mathcal{T}\langle\!\langle [\![e]\!] \rangle\!\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{T} & : & \mathit{Types}[\Lambda^\sigma] \rightarrow \mathit{Types}[\Lambda^\sigma_\tau] \\
\mathcal{T}\langle\!\langle [\![\iota]\!] \rangle\!\rangle & = & \iota \\
\mathcal{T}\langle\!\langle [\![\sigma_1 \rightarrow \sigma_2]\!] \rangle\!\rangle & = & \widetilde{\mathcal{T}\langle\!\langle [\![\sigma_1]\!] \rangle\!\rangle} \rightarrow \mathcal{T}\langle\!\langle [\![\sigma_2]\!] \rangle\!\rangle
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{T} & : & \mathit{Assums}[\Lambda^\sigma] \rightarrow \mathit{Assums}[\Lambda^\sigma_\tau] \\
\mathcal{T}\langle\!\langle [\![\Gamma, x:\sigma]\!] \rangle\!\rangle & = & \mathcal{T}\langle\!\langle [\![\Gamma]\!] \rangle\!\rangle, x:\widetilde{\mathcal{T}\langle\!\langle [\![\sigma]\!] \rangle\!\rangle} \\
\mathcal{T}\langle\!\langle [\![\Gamma, f:\sigma]\!] \rangle\!\rangle & = & \mathcal{T}\langle\!\langle [\![\Gamma]\!] \rangle\!\rangle, f:\mathcal{T}\langle\!\langle [\![\sigma]\!] \rangle\!\rangle
\end{array}
$$

Figure 3.6: Thunk introduction on for the typed core language Λ^σ

The syntax of the typed language with thunks Λ^σ_τ is obtained by adding *delay* and *force* to the syntax of Λ^σ.

$$
\begin{array}{rcl}
e & \in & \mathit{Terms}[\Lambda^\sigma_\tau] \\
e & ::= & \ldots \mid \mathit{delay}\, e \mid \mathit{force}\, e
\end{array}
$$

Figure 3.5 presents type assignment rules for the language Λ^σ_τ. An abstract type constructor $\widetilde{\cdot}$ is included in the types corresponding to the abstract suspension constructs *delay* and *force* in the terms. The type $\widetilde{\sigma}$ denotes the type of a suspension (*i.e.*, a thunk) that will yield an expression of type σ when forced. Note that we use the same meta-variables (Γ for type assumptions, σ for types, and e for terms) for both Λ^σ and Λ^σ_τ. Ambiguity is avoided by subscripting the typing judgement symbol $\vdash_\tau$ for the language Λ^σ_τ.

Figure 3.6 extends the thunk introduction $\mathcal{T}$ for Λ (see Figure 3.1) to Λ^σ. The treatment of recursive abstractions follows that of non-recursive abstractions except that identifiers f never bind to thunks and thus are never forced. The definition of $\mathcal{T}$ on function types and on type assumptions reflects the fact that all arguments to functions in the image of $\mathcal{T}$ are thunks.

The following property states that $\mathcal{T}$ preserves well-typedness of terms.

Property 3.12 *If* $\Gamma \vdash e : \sigma$ *then* $\mathcal{T}\langle\!\langle [\![\Gamma]\!] \rangle\!\rangle \vdash_\tau \mathcal{T}\langle\!\langle [\![e]\!] \rangle\!\rangle : \mathcal{T}\langle\!\langle [\![\sigma]\!] \rangle\!\rangle$.

Proof: by induction over the structure of the typing derivation for $\Gamma \vdash e : \sigma$. ∎

Figure 3.7 extends the thunk elimination $\mathcal{T}^{-1}$ for Λ_τ (Figure 3.3) to Λ^σ_τ. The definition on types reflects the fact that $\mathcal{T}^{-1}$ simply removes suspension constructs.

The following property states that $\mathcal{T}^{-1}$ preserves well-typedness of terms.

Property 3.13 *If* $\Gamma \vdash_\tau e : \sigma$ *then* $\mathcal{T}^{-1}\langle\!\langle [\![\Gamma]\!] \rangle\!\rangle \vdash \mathcal{T}^{-1}\langle\!\langle [\![e]\!] \rangle\!\rangle : \mathcal{T}^{-1}\langle\!\langle [\![\sigma]\!] \rangle\!\rangle$.

Proof: by induction over the structure of the typing derivation for $\Gamma \vdash_\tau e : \sigma$. ∎

The **Indifference**, **Simulation**, and **Translation** properties of $\mathcal{T}$ for the untyped language Λ given in Theorem 3.1 and the equational correspondence of Theorem 3.2 extends to $\mathcal{T}$ on the typed language Λ^σ. It is only necessary to add cases for recursive identifiers and abstractions to the proofs and to add rec and rec_v to the theories under consideration. For details, see Section 3.8.6.

3.4.2 Connecting the thunk-based and the continuation-based simulations

We begin by extending the transformation $\mathcal{C}_v$ on types of Λ^σ (Figure 2.12) to the types of Λ^σ_τ. We need only the following case for suspension types.

$$\mathcal{C}_v\langle\widetilde{\sigma}\rangle \;=\; \mathcal{C}_v\langle\!\langle [\![\sigma]\!] \rangle\!\rangle$$

This reflects the fact that suspensions are translated to terms expecting a continuation. The following property extends the well-typedness of $\mathcal{C}_v$ on Λ^σ to Λ^σ_τ.

$$
\begin{array}{rcl}
\mathcal{T}^{-1} & : & \mathit{Terms}[\Lambda^\sigma_\tau] \rightarrow \mathit{Terms}[\Lambda^\sigma] \\
 & \cdots & \\
\mathcal{T}^{-1}\langle\!|f|\!\rangle & = & f \\
\mathcal{T}^{-1}\langle\!|\mathit{rec}\, f(x).\, e|\!\rangle & = & \mathit{rec}\, f(x).\, \mathcal{T}^{-1}\langle\!|e|\!\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{T}^{-1} & : & \mathit{Types}[\Lambda^\sigma_\tau] \rightarrow \mathit{Types}[\Lambda^\sigma] \\
\mathcal{T}^{-1}\langle\!|\iota|\!\rangle & = & \iota \\
\mathcal{T}^{-1}\langle\!|\sigma_1 \rightarrow \sigma_2|\!\rangle & = & \mathcal{T}^{-1}\langle\!|\sigma_1|\!\rangle \rightarrow \mathcal{T}^{-1}\langle\!|\sigma_2|\!\rangle \\
\mathcal{T}^{-1}\langle\!|\widetilde{\sigma}|\!\rangle & = & \mathcal{T}^{-1}\langle\!|\sigma|\!\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{T}^{-1} & : & \mathit{Assums}[\Lambda^\sigma_\tau] \rightarrow \mathit{Assums}[\Lambda^\sigma] \\
\mathcal{T}^{-1}\langle\!|\Gamma, x:\sigma|\!\rangle & = & \mathcal{T}^{-1}\langle\!|\Gamma|\!\rangle, x : \mathcal{T}^{-1}\langle\!|\sigma|\!\rangle \\
\mathcal{T}^{-1}\langle\!|\Gamma, f:\sigma|\!\rangle & = & \mathcal{T}^{-1}\langle\!|\Gamma|\!\rangle, f : \mathcal{T}^{-1}\langle\!|\sigma|\!\rangle
\end{array}
$$

Figure 3.7: Thunk elimination for the typed language Λ^σ_τ

Property 3.14 *If* $\Gamma \vdash_\tau e : \sigma$ *then* $\mathcal{C}_v\langle\!|\Gamma|\!\rangle \vdash \mathcal{C}_v\langle\!|e|\!\rangle : \mathcal{C}_v\langle\!|\sigma|\!\rangle$.

The following theorem extends the reduction properties of $\mathcal{C}_v$ on Λ^σ to Λ^σ_τ. As noted in Section 3.3, considering typed terms allows a strengthening of Theorem 3.3 to the full language of thunks instead of just the language of the image of $\mathcal{T}$ closed under reductions.

Theorem 3.5 *For all* $\cdot \vdash_\tau e : \sigma$, *and* $\Gamma \vdash_\tau e_1 : \sigma$, $\Gamma \vdash_\tau e_2 : \sigma$,

1. **Indifference:** $\mathit{eval}_n(\mathcal{C}_v\langle\!|e|\!\rangle\,(\lambda y\,.\,y)) \simeq \mathit{eval}_v(\mathcal{C}_v\langle\!|e|\!\rangle\,(\lambda y\,.\,y))$
2. **Simulation:** $\mathcal{C}_v\langle \mathit{eval}_v(e)\rangle \simeq \mathit{eval}_n(\mathcal{C}_v\langle\!|e|\!\rangle\,(\lambda y\,.\,y))$
3. **Translation:** *If* $\lambda\beta_v\mathrm{rec}_v\tau \vdash e_1 = e_2$ *then* $\lambda\beta_v\mathrm{rec}_v\eta_v \vdash \mathcal{C}_v\langle\!|e_1|\!\rangle = \mathcal{C}_v\langle\!|e_2|\!\rangle$.
   Also $\lambda\beta_v\mathrm{rec}_v\eta_v \vdash \mathcal{C}_v\langle\!|e_1|\!\rangle = \mathcal{C}_v\langle\!|e_2|\!\rangle$ *iff* $\lambda\beta\mathrm{rec}\eta_v \vdash \mathcal{C}_v\langle\!|e_1|\!\rangle = \mathcal{C}_v\langle\!|e_2|\!\rangle$.

Proof: Given the extension of $\mathcal{C}_v$ to Λ^σ in Chapter 2, we need only add cases for *delay* and *force*. Proving **Indifference** is trivial since all arguments are values. **Simulation** can be proved by extending Plotkin's "colon translation" to *delay* and *force*. Proving **Translation** is trivial since $\beta\mathrm{rec}$ corresponds to $\beta_v\mathrm{rec}_v$ on CPS terms and τ-reduction is captured by $\beta_a\eta_v$-reduction (Property 3.10) ∎

The factoring result of Theorem 3.4 also extends from Λ to Λ^σ. It is necessary to add only the cases for recursive identifiers f and recursive abstractions $\mathit{rec}\, f(x).\, e$ to the proof. These cases are straightforward since the treatment of f identical to that of c in $\mathcal{C}_v$, $\mathcal{T}$, and $\mathcal{C}_n$, and the treatment of $\mathit{rec}\, f(x).\, e$ is identical to that of $\lambda x\,.\,e$.

Given the $\beta_a\eta_v$ equivalence of the output of $\mathcal{C}_v \circ \mathcal{T}$ and $\mathcal{C}_n$, the following relation between types follows from subject reduction properties of $\beta_a\eta_v$.

Property 3.15 *For all* $\sigma \in \mathit{Types}[\Lambda^\sigma]$ *and* $\Gamma \in \mathit{Assums}[\Lambda^\sigma]$,

1.	$\mathcal{C}_v\langle\!|\mathcal{T}\langle\!|\sigma|\!\rangle|\!\rangle = \mathcal{C}_n\langle\!|\sigma|\!\rangle$	*expression types*
2.	$\mathcal{C}_v\langle\mathcal{T}\langle\!|\sigma|\!\rangle\rangle = \mathcal{C}_n\langle\sigma\rangle$	*value types*
3.	$\mathcal{C}_v\langle\!|\mathcal{T}\langle\!|\Gamma|\!\rangle|\!\rangle = \mathcal{C}_n\langle\!|\Gamma|\!\rangle$	*type assumptions*

Proof: Instead of relying on subject reduction, a direct proof of components 1 and 2 proceeds by induction over the structure of σ. We give the case of function types for values below.

$$
\begin{array}{rcl}
\mathcal{T} & : & \mathit{Terms}[\Lambda^{\sigma^+}] \rightarrow \mathit{Terms}[\Lambda_\tau^{\sigma^+}] \\
 & \cdots & \\
\mathcal{T}\langle\![\mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2]\!\rangle & = & \mathit{if}\ \mathcal{T}\langle\![e_0]\!\rangle\ \mathit{then}\ \mathcal{T}\langle\![e_1]\!\rangle\ \mathit{else}\ \mathcal{T}\langle\![e_2]\!\rangle \\
\mathcal{T}\langle\![\mathit{pair}\ e_1\ e_2]\!\rangle & = & \mathit{pair}\,(\mathit{delay}\ \mathcal{T}\langle\![e_1]\!\rangle)\,(\mathit{delay}\ \mathcal{T}\langle\![e_2]\!\rangle) \\
\mathcal{T}\langle\![\mathit{proj}_i\ e]\!\rangle & = & \mathit{force}\,(\mathit{proj}_i\ \mathcal{T}\langle\![e]\!\rangle) \\
\mathcal{T}\langle\![\mathit{inj}_i\ e]\!\rangle & = & \mathit{inj}_i\,(\mathit{delay}\ \mathcal{T}\langle\![e]\!\rangle) \\
\mathcal{T}\langle\![\mathit{case}\ e\ \mathit{of}\ (x_1.e_1)\,|\,(x_2.e_2)]\!\rangle & = & \mathit{case}\ \mathcal{T}\langle\![e]\!\rangle\ \mathit{of}\ (x_1.\mathcal{T}\langle\![e_1]\!\rangle)\,|\,(x_2.\mathcal{T}\langle\![e_2]\!\rangle)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{T} & : & \mathit{Types}[\Lambda^{\sigma^+}] \rightarrow \mathit{Types}[\Lambda_\tau^{\sigma^+}] \\
 & \cdots & \\
\mathcal{T}\langle\!|\sigma_1 \times \sigma_2|\!\rangle & = & \widetilde{\mathcal{T}\langle\!|\sigma_1|\!\rangle} \times \widetilde{\mathcal{T}\langle\!|\sigma_2|\!\rangle} \\
\mathcal{T}\langle\!|\sigma_1 + \sigma_2|\!\rangle & = & \widetilde{\mathcal{T}\langle\!|\sigma_1|\!\rangle} + \widetilde{\mathcal{T}\langle\!|\sigma_2|\!\rangle}
\end{array}
$$

Figure 3.8: Thunk introduction for the extended typed language Λ^{σ^+}

$$
\begin{array}{rcll}
\mathcal{C}_v\langle\mathcal{T}\langle\!|\sigma_1 \rightarrow \sigma_2|\!\rangle\rangle & = & \mathcal{C}_v\langle\widetilde{\mathcal{T}\langle\!|\sigma_1|\!\rangle} \rightarrow \mathcal{T}\langle\!|\sigma_2|\!\rangle\rangle & \\
 & = & \mathcal{C}_v\langle\widetilde{\mathcal{T}\langle\!|\sigma_1|\!\rangle}\rangle \rightarrow \mathcal{C}_v\langle\!|\mathcal{T}\langle\!|\sigma_2|\!\rangle|\!\rangle & \\
 & = & \mathcal{C}_v\langle\!|\mathcal{T}\langle\!|\sigma_1|\!\rangle|\!\rangle \rightarrow \mathcal{C}_v\langle\!|\mathcal{T}\langle\!|\sigma_2|\!\rangle|\!\rangle & \\
 & = & \mathcal{C}_n\langle\!|\sigma_1|\!\rangle \rightarrow \mathcal{C}_n\langle\!|\sigma_2|\!\rangle & \ldots \textit{by ind. hyp.} \\
 & = & \mathcal{C}_n\langle\sigma_1 \rightarrow \sigma_2\rangle &
\end{array}
$$

The proof of component 3 follows in a straightforward manner from components 1 and 2. ∎

As pointed out in the previous section, $\mathcal{C}_n$ and $\mathcal{C}_v$ are alike in that they both introduce continuation-passing terms. This is reflected by the similarity in the definitions $\mathcal{C}_n\langle\!|\sigma|\!\rangle = \neg\neg\mathcal{C}_n\langle\sigma\rangle$ and $\mathcal{C}_v\langle\!|\sigma|\!\rangle = \neg\neg\mathcal{C}_v\langle\sigma\rangle$. $\mathcal{C}_n$ and $\mathcal{C}_v$ differ in how arguments are treated. This is reflected by the difference in the definitions $\mathcal{C}_n\langle\sigma_1 \rightarrow \sigma_2\rangle = \mathcal{C}_n\langle\!|\sigma_1|\!\rangle \rightarrow \mathcal{C}_n\langle\!|\sigma_2|\!\rangle$ and $\mathcal{C}_v\langle\sigma_1 \rightarrow \sigma_2\rangle = \mathcal{C}_v\langle\sigma_1\rangle \rightarrow \mathcal{C}_v\langle\!|\sigma_2|\!\rangle$. The only effect of $\mathcal{T}$ is to change how arguments are treated. This is reflected by the fact that the only effect of $\mathcal{T}$ on types is the introduction of suspension types for arguments, *i.e.*, $\mathcal{T}\langle\!|\sigma_1 \rightarrow \sigma_2|\!\rangle = \widetilde{\mathcal{T}\langle\!|\sigma_1|\!\rangle} \rightarrow \mathcal{T}\langle\!|\sigma_2|\!\rangle$. Thus, the action by $\mathcal{T}$ is exactly what is needed to move from $\mathcal{C}_v$ to $\mathcal{C}_n$.

3.5 Thunks for the extended typed language Λ^{σ^+}

The syntax of the extended typed language with thunks $\Lambda_\tau^{\sigma^+}$ is obtained by adding *delay* and *force* to the syntax of Λ^{σ^+}.

$$
\begin{array}{rcl}
e & \in & \mathit{Terms}[\Lambda_\tau^{\sigma^+}] \\
e & ::= & \ldots \mid \mathit{delay}\ e \mid \mathit{force}\ e
\end{array}
$$

The types of $\Lambda_\tau^{\sigma^+}$ are obtained by adding the constructor $\widetilde{\sigma}$ to the types of Λ^{σ^+}. We omit the obvious recasting of the types and typing assignment rules.

Figure 3.8 extends the thunk introduction $\mathcal{T}$ for Λ^{σ} (see Figure 3.6) to Λ^{σ^+}. The transformation of conditionals is trivial since the call-by-name and call-by-value semantics coincide.[4] The delaying of arguments for

[4] Since *let* is strict under call-by-name, thunks are not needed to simulate it under call-by-value. However, it cannot be incorporated smoothly into this translation since identifiers of *let* constructs would not bind to suspensions. This is in contrast

$$
\begin{array}{rcl}
\mathcal{T}_{opt} & : & Terms[\Lambda] \rightarrow Terms[\Lambda_\tau] \\
\mathcal{T}_{opt}\langle\!|c|\!\rangle & = & c \\
\mathcal{T}_{opt}\langle\!|x|\!\rangle & = & force\ x \\
\mathcal{T}_{opt}\langle\!|\lambda x\ .\ e|\!\rangle & = & \lambda x\ .\ \mathcal{T}_{opt}\langle\!|e|\!\rangle \\
\mathcal{T}_{opt}\langle\!|e_0\ x|\!\rangle & = & \mathcal{T}_{opt}\langle\!|e_0|\!\rangle\ x \\
\mathcal{T}_{opt}\langle\!|e_0\ e_1|\!\rangle & = & \mathcal{T}_{opt}\langle\!|e_0|\!\rangle\ (delay\ \mathcal{T}_{opt}\langle\!|e_1|\!\rangle) \\
 & & \quad ...where\ e_1\ is\ not\ an\ identifier.
\end{array}
$$

Figure 3.9: Optimized thunk introduction

constructors *pair* and *inj* simulates the laziness associated with call-by-name. The suspended *pair* arguments are forced when projected. Since arguments of *inj* will bind to identifiers in the reduction of *case*, their forcing follows as a consequence of the forcing of identifiers.[5]

Thunk elimination $\mathcal{T}^{-1}$ on $\Lambda_\tau^{\sigma^+}$ extends from Λ_τ^σ in the obvious way. Both $\mathcal{T}$ and $\mathcal{T}^{-1}$ preserve well-typedness of terms. The **Indifference**, **Simulation**, equational correspondence, and $\mathcal{C}_n$ factoring properties extend to Λ^{σ^+}. Section 3.8.7 gives relevant details.

3.6 Variations and related work

Thunks have been used primarily to implement call-by-name and lazy languages [8, 43, 73]. Recently, thunks have been used in more theoretical settings [5, 82]. In this section, we first give some useful variations of our transformation $\mathcal{T}$ and then survey other optimizations and applications of thunks that have recently appeared in the literature.

3.6.1 An optimization in Algol 60

The transformation $\mathcal{T}$ introduces terms of the form

$$e\ (delay\ (force\ x))$$

in essence, suspending the suspension denoted by the identifier x. This double suspension was avoided in the implementation of Algol 60 [78]. Figure 3.9 captures this optimization: the thunk introduction $\mathcal{T}_{opt}$ only delays arguments that are not identifiers. The **Indifference** and **Simulation** properties of Theorem 3.1 hold for this transformation as well. However, for an equational correspondence to hold, the following notion of reduction is needed.

$$delay\ (force\ v) \quad \rightsquigarrow_{\tau_\eta} \quad v \qquad\qquad (\tau_\eta)$$

The following example illustrates the need of τ_η by giving β-convertible terms e_1 and e_2 where $\mathcal{T}_{opt}\langle\!|e_1|\!\rangle$ and $\mathcal{T}_{opt}\langle\!|e_2|\!\rangle$ are not $\beta_a\tau$-convertible.

$$\lambda\beta \vdash\ e_1 \equiv f\ ((\lambda y\ .\ x)\ c) = f\ x \equiv e_2$$

to identifiers declared in abstractions and *case*. This non-uniformity of identifier binding is similar to the situation we face with recursive identifiers (which only bind to non-suspended recursive abstractions). Incorporating *let* is possible, but requires extra machinery. There are at least two possibilities:

- introducing another class of identifiers for *let* (as done for recursive abstractions), or
- parameterizing the translation by an additional argument which keeps track of whether identifiers are declared in *let*'s or abstractions.

Since let is not essential for the application of Λ^{σ^+} in Chapter 4 we omit treating them.

[5] An alternative elimination rule for pairs is $let\ (x_1, x_2) = e'\ in\ e$ where $let\ (x_1, x_2) = (e_1, e_2)\ in\ e \quad \longrightarrow \quad e[x_1 := e_1\ ,\ x_2 := e_2]$. The advantage of this form is that thunks are forced uniformly — only identifiers are forced.

However,

$$\begin{array}{rcl}
\mathcal{T}_{opt}\langle\!\langle [\![e_1]\!] \rangle\!\rangle & = & (force\ f)\,(delay\,(\lambda y\,.\,force\ x)\,(delay\ c)) \\
& \longrightarrow_{\beta_a} & (force\ f)\,(delay\,(force\ x)) \\
& \longrightarrow_{\tau_\eta} & (force\ f)\,x \qquad \ldots\tau_\eta \textit{ is needed for this step.} \\
& = & \mathcal{T}_{opt}\langle\!\langle [\![e_2]\!] \rangle\!\rangle
\end{array}$$

The following property verifies the correctness of τ_η and illustrates that the new reduction corresponds to η_v-reducing the continuation for CPS terms.

Property 3.16 *For all* $v \in Values_v[\Lambda_\tau]$,

$$\lambda\beta_a\eta_v \vdash \mathcal{C}_v\langle\!\langle [\![delay\,(force\ v)]\!] \rangle\!\rangle = \mathcal{C}_v\langle\!\langle [\![v]\!] \rangle\!\rangle$$

Proof:

$$\begin{array}{rcl}
\mathcal{C}_v\langle\!\langle [\![delay\,(force\ v)]\!] \rangle\!\rangle & = & \lambda k\,.\,k\,(\mathcal{C}_v\langle\!\langle [\![force\ v]\!] \rangle\!\rangle) \\
& = & \lambda k\,.\,k\,(\lambda k\,.\,\mathcal{C}_v\langle\!\langle [\![v]\!] \rangle\!\rangle\,(\lambda y\,.\,y\,k)) \\
& = & \lambda k\,.\,k\,(\lambda k\,.\,(\lambda k\,.\,k\,\mathcal{C}_v\langle v\rangle)\,(\lambda y\,.\,y\,k)) \\
& \twoheadrightarrow_{\beta_a} & \lambda k\,.\,k\,(\lambda k\,.\,\mathcal{C}_v\langle v\rangle\ k) \\
& \longrightarrow_{\eta_v} & \lambda k\,.\,k\,(\mathcal{C}_v\langle v\rangle) \\
& = & \mathcal{C}_v\langle\!\langle [\![v]\!] \rangle\!\rangle
\end{array}$$

∎

An equational correspondence holds between Λ terms under β reduction and terms in the image of $\mathcal{T}_{opt}$ closed under $\beta_a\tau\tau_\eta$ reduction. Additionally, the relationship between $\mathcal{C}_v \circ \mathcal{T}$ and $\mathcal{C}_n$ (Theorem 3.4) extends to $\mathcal{T}_{opt}$ as well (for details see Section 3.8.8).

3.6.2 More on Algol 60

In Algol 60, parameter passing follows call-by-name by default, but the user may also specify that some parameters be passed "by value". Nevertheless, Randell and Russell implement parameter passing by delaying all arguments, uniformly [78]. Furthermore, they also treat variables uniformly, by forcing them all. The effect of call-by-value is obtained by creating a "shell procedure" that does nothing but force its argument and pass the resulting value wrapped in a thunk to the transformed version of the original procedure. This corresponds to the definition of call-by-value in the Algol report [69].

Let us express this strategy for a language Λ_s with both non-strict (call-by-name) functions $(\lambda x\,.\,e)$ and strict (call-by-value) functions $(\lambda x_s\,.\,e)$. This language with mixed parameter passing can be simulated simply by extending $\mathcal{T}$ with the following clause.

$$\begin{array}{rcl}
\mathcal{T}\langle\!\langle [\![\lambda x_s\,.\,e]\!] \rangle\!\rangle & = & \lambda t\,.\,let\ y \Leftarrow force\ t \\
& & \qquad in\ (\lambda x\,.\,\mathcal{T}\langle\!\langle [\![e]\!] \rangle\!\rangle)\,(delay\ y)
\end{array}$$

Upon entry into a call-by-value function, the actual parameter is forced (using a strict *let* expression) and the resulting value is packaged into a thunk and bound to a new occurrence of the formal parameter.

This treatment corresponds to Reynolds's CPS transformation $\mathcal{C}_{\mathrm{R}}$ [81, p. 144], displayed in Figure 3.10 (the k's, the t's and the y's are fresh variables). As in Section 3.3.2, we can derive $\mathcal{C}_{\mathrm{R}}$ by composing $\mathcal{C}_v$ and $\mathcal{T}$.

Property 3.17 *For all* $e \in \Lambda_s$,

$$\lambda\beta_a\eta_v \vdash (\mathcal{C}_v \circ \mathcal{T})\langle\!\langle [\![e]\!] \rangle\!\rangle = \mathcal{C}_{\mathrm{R}}\langle\!\langle [\![e]\!] \rangle\!\rangle$$

Proof: The proof is by induction over the structure of e and is identical to the proof of Theorem 3.4 given the addition of the following case.

case $e \equiv \lambda x_s\,.\,e$:

$$
\begin{array}{rcl}
\mathcal{C}_{\mathrm{R}}\langle\!|\cdot|\!\rangle & : & \Lambda \to \Lambda \\
\mathcal{C}_{\mathrm{R}}\langle\!|v|\!\rangle & = & \lambda k\,.\,k\,(\mathcal{C}_{\mathrm{R}}\langle v\rangle) \\
\mathcal{C}_{\mathrm{R}}\langle\!|x|\!\rangle & = & x \\
\mathcal{C}_{\mathrm{R}}\langle\!|e_0\ e_1|\!\rangle & = & \lambda k\,.\,\mathcal{C}_{\mathrm{R}}\langle\!|e_0|\!\rangle\,(\lambda y_0\,.\,(y_0\,\mathcal{C}_{\mathrm{R}}\langle\!|e_1|\!\rangle)\,k)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}_{\mathrm{R}}\langle\cdot\rangle & : & \mathit{Values}_v[\Lambda] \to \Lambda \\
\mathcal{C}_{\mathrm{R}}\langle c\rangle & = & c \\
\mathcal{C}_{\mathrm{R}}\langle\lambda x\,.\,e\rangle & = & \lambda x\,.\,\mathcal{C}_{\mathrm{R}}\langle\!|e|\!\rangle \\
\mathcal{C}_{\mathrm{R}}\langle\lambda x_s\,.\,e\rangle & = & \lambda t\,.\,\lambda k\,.\,t\,(\lambda y\,.\,((\lambda x\,.\,\mathcal{C}_{\mathrm{R}}\langle\!|e|\!\rangle)\,(\lambda k\,.\,k\,y))\,k)
\end{array}
$$

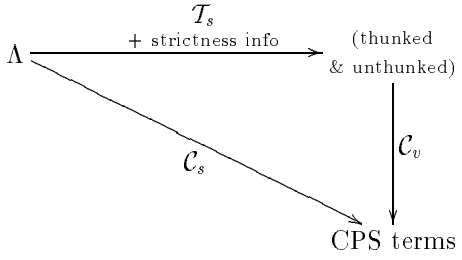
Figure 3.10: Reynolds's CPS transformation

$$
\begin{array}{rl}
& (\mathcal{C}_v \circ \mathcal{T})\langle\!|\lambda x_s\,.\,e|\!\rangle \\
= & \mathcal{C}_v\langle\!|\lambda t\,.\,\mathit{let}\ y \Leftarrow \mathit{force}\ t\ \mathit{in}\ (\lambda x\,.\,\mathcal{T}\langle\!|e|\!\rangle)\,(\mathit{delay}\ y)|\!\rangle \\
= & \lambda k\,.\,k\,(\lambda t\,.\,\lambda k\,.\,\mathcal{C}_v\langle\!|\mathit{force}\ t|\!\rangle\,(\lambda y\,.\,\mathcal{C}_v\langle\!|(\lambda x\,.\,\mathcal{T}\langle\!|e|\!\rangle)\,(\mathit{delay}\ y)|\!\rangle\,k)) \\
= & \lambda k\,.\,k\,(\lambda t\,.\,\lambda k\,.\,(\lambda k\,.\,(\lambda k\,.\,k\,t)\,(\lambda y_0\,.\,y_0\,k))\,(\lambda y\,.\,\mathcal{C}_v\langle\!|(\lambda x\,.\,\mathcal{T}\langle\!|e|\!\rangle)\,(\mathit{delay}\ y)|\!\rangle\,k)) \\
\longrightarrow_{\beta_a} & \lambda k\,.\,k\,(\lambda t\,.\,\lambda k\,.\,(\lambda k\,.\,k\,t)\,(\lambda y_0\,.\,y_0\,(\lambda y\,.\,\mathcal{C}_v\langle\!|(\lambda x\,.\,\mathcal{T}\langle\!|e|\!\rangle)\,(\mathit{delay}\ y)|\!\rangle\,k))) \\
\longrightarrow_{\beta_a} & \lambda k\,.\,k\,(\lambda t\,.\,\lambda k\,.\,(\lambda y_0\,.\,y_0\,(\lambda y\,.\,\mathcal{C}_v\langle\!|(\lambda x\,.\,\mathcal{T}\langle\!|e|\!\rangle)\,(\mathit{delay}\ y)|\!\rangle\,k))\,t) \\
\longrightarrow_{\beta_a} & \lambda k\,.\,k\,(\lambda t\,.\,\lambda k\,.\,t\,(\lambda y\,.\,\mathcal{C}_v\langle\!|(\lambda x\,.\,\mathcal{T}\langle\!|e|\!\rangle)\,(\mathit{delay}\ y)|\!\rangle\,k)) \\
\longrightarrow\!\!\!\!\!\rightarrow_{\beta_a} & \lambda k\,.\,k\,(\lambda t\,.\,\lambda k\,.\,t\,(\lambda y\,.\,((\lambda x\,.\,(\mathcal{C}_v \circ \mathcal{T})\langle\!|e|\!\rangle)\,\mathcal{C}_v\langle\mathit{delay}\ y\rangle)\,k)) \\
= & \lambda k\,.\,k\,(\lambda t\,.\,\lambda k\,.\,t\,(\lambda y\,.\,((\lambda x\,.\,(\mathcal{C}_v \circ \mathcal{T})\langle\!|e|\!\rangle)\,(\lambda k\,.\,k\,y))\,k)) \\
=_{\beta_a \eta_v} & \lambda k\,.\,k\,(\lambda t\,.\,\lambda k\,.\,t\,(\lambda y\,.\,((\lambda x\,.\,\mathcal{C}_{\mathrm{R}}\langle\!|e|\!\rangle)\,(\lambda k\,.\,k\,y))\,k)) \qquad \textit{...by ind. hyp.} \\
= & \mathcal{C}_{\mathrm{R}}\langle\!|\lambda x_s\,.\,e|\!\rangle
\end{array}
$$

∎

3.6.3 Other optimizations

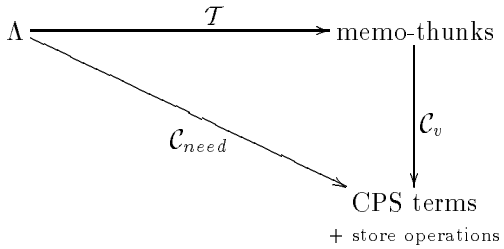
Strictness information indicates arguments that may be safely evaluated eagerly (*i.e.*, without being delayed) — in effect, reducing the number of thunks needed in a program and the overhead associated with creating and evaluating suspensions [8, 22, 73]. In the following chapter, we give a transformation $\mathcal{T}_s$ that optimizes thunk introduction based on strictness information. We then use the factorization of $\mathcal{C}_n$ by $\mathcal{C}_v$ and $\mathcal{T}$ to derive an optimized CPS transformation $\mathcal{C}_s$ for strictness-analyzed call-by-name terms. This situation is summarized by the following diagram.



The resulting transformation $\mathcal{C}_s$ yields both call-by-name-like and call-by-value-like continuation-passing terms. Due to our factorization result, the proof of correctness for the optimized transformation follows as a corollary of the correctness of the strictness analysis, and the correctness of $\mathcal{T}$ and $\mathcal{C}_v$.

Okasaki, Lee, and Tarditi [73] have also applied our factorization to obtain a "call-by-need CPS transformation" $\mathcal{C}_{need}$. The lazy evaluation strategy that characterizes call-by-need is captured by memoizing the thunks

[8]. $\mathcal{C}_{need}$ is obtained by extending $\mathcal{C}_v$ to transform memo-thunks to CPS terms with store operations (which are used to implement the memoization) and composing with the memo-thunk introduction as follows.



Okasaki *et al.* optimize $\mathcal{C}_{need}$ by using strictness information along the lines discussed above. They also use sharing information to detect where memo-thunks can be replaced by ordinary thunks (*i.e.*, if a memo-thunk is never shared, it is more efficient to replace it by an ordinary thunk). In both cases, optimizations are achieved by working with simpler thunked terms as opposed to working directly with CPS terms.

3.6.4 Thunks implemented in Λ

We have chosen to represent thunks via abstract suspension operators *delay* and *force*. However, thunks may also be implemented directly in Λ following the "protecting by a λ" approach noted by Plotkin [76, p. 147]. An expression is delayed by wrapping it in an abstraction with a dummy parameter. A suspension is forced by applying it to a dummy argument. The following transformation formalizes this implementation.

$$\begin{array}{rcll} \mathcal{I} & : & \mathit{Terms}[\Lambda_\tau] \rightarrow \mathit{Terms}[\Lambda] & \\ & \cdots & & \\ \mathcal{I}\langle\![\mathit{delay}\, e]\!\rangle & = & \lambda z\,.\,\mathcal{I}\langle\![e]\!\rangle & \mathit{...where}\ z \notin \mathit{FV}(e). \\ \mathcal{I}\langle\![\mathit{force}\, e]\!\rangle & = & e\ c & \end{array}$$

With this implementation, the τ-reduction of Λ_τ corresponds to a β_a-reduction in Λ, *i.e.*,

$$\mathcal{I}\langle\![\mathit{force}\,(\mathit{delay}\, e)]\!\rangle \;=\; (\lambda z\,.\,\mathcal{I}\langle\![e]\!\rangle)\,c \;\longrightarrow_{\beta_a}\; \mathcal{I}\langle\![e]\!\rangle$$

Now, a thunk-introducing transformation $\mathcal{T}_\mathcal{I}$ that implements thunks directly in Λ is obtained by composing $\mathcal{I}$ with either $\mathcal{T}$ or $\mathcal{T}_{opt}$ (or any other correct encoding into Λ_τ). In either case, the **Simulation** and **Indifference** properties associated with $\mathcal{T}$ (Theorem 3.1) hold for $\mathcal{T}_\mathcal{I}$ as well. If $\mathcal{T}_\mathcal{I} \stackrel{\text{def}}{=} \mathcal{I} \circ \mathcal{T}$, then an equational correspondence exists between Λ under β-reduction and $\mathcal{T}_\mathcal{I}\langle\![\Lambda]\!\rangle^*$ under β_a-reduction (where $\mathcal{T}_\mathcal{I}\langle\![\Lambda]\!\rangle^*$ is the language of terms in the image of $\mathcal{T}_\mathcal{I}$ closed under β_a-reduction). If $\mathcal{T}_\mathcal{I} \stackrel{\text{def}}{=} \mathcal{I} \circ \mathcal{T}_{opt}$, then equational correspondence fails because τ_η-reductions cannot be implemented in the theory $\lambda\beta_a\eta_v$. Specifically, $\mathcal{I}\langle\![\mathit{delay}\,(\mathit{force}\, e)]\!\rangle \;=\; \lambda z\,.\,\mathcal{I}\langle\![e]\!\rangle\, c \;\not\longrightarrow\; \mathcal{I}\langle\![e]\!\rangle$.

The following derivations illustrate that the relationship between between $\mathcal{C}_v \circ \mathcal{T}$ and $\mathcal{C}_n$ does not scale up to $\mathcal{T}_\mathcal{I}$ (*i.e.*, $\lambda\beta_a\eta_v \not\forall\ \mathcal{C}_n\langle\![e]\!\rangle \;=\; (\mathcal{C}_v \circ \mathcal{T}_\mathcal{I})\langle\![e]\!\rangle$).

$$\begin{array}{rcl} (\mathcal{C}_v \circ \mathcal{T}_\mathcal{I})\langle\![x]\!\rangle & = & \mathcal{C}_v\,\langle\![x\ c]\!\rangle \\ & = & \lambda k\,.\,(x\ c)\,k \\ & \neq_{\beta_a\eta_v} & \mathcal{C}_n\,\langle\![x]\!\rangle \end{array}$$

$$\begin{array}{rcl} (\mathcal{C}_v \circ \mathcal{T}_\mathcal{I})\langle\![e_0\ e_1]\!\rangle & = & \mathcal{C}_v\,\langle\![\mathcal{T}_\mathcal{I}\langle\![e_0]\!\rangle\,(\lambda z\,.\,\mathcal{T}_\mathcal{I}\langle\![e_1]\!\rangle)]\!\rangle \\ & = & \lambda k\,.\,(\mathcal{C}_v \circ \mathcal{T}_\mathcal{I})\langle\![e_0]\!\rangle\,(\lambda y\,.\,(y\,(\lambda z\,.\,(\mathcal{C}_v \circ \mathcal{T}_\mathcal{I})\langle\![e_1]\!\rangle))\,k) \\ & \neq_{\beta_a\eta_v} & \mathcal{C}_n\,\langle\![e_0\ e_1]\!\rangle \end{array}$$

However, $\mathcal{C}_v \circ \mathcal{T}_\mathcal{I}$ does give a valid continuation-based simulation of call-by-name under call-by-value evaluation.

3.6.5 Related work

Ingerman [43], in his work on the implementation of Algol 60, gave a general technique for generating machine code implementing procedure parameter passing. The term *thunk* was coined to refer to the compiled representation of a delayed expression as it gets pushed on the control stack [79]. Since then, the term *thunk* has been applied to other higher-level representations of delayed expressions and we have followed this practice.

Bloss, Hudak, and Young [8] study thunks as the basis of implementation of lazy evaluation. Optimizations associated with lazy evaluation (*e.g.*, overwriting a forced expression with its resulting value) are encapsulated in the thunk. They give several representations with differing effects on space and time overhead.

Riecke [82] has used thunks to obtain fully-abstract translations between versions of PCF (an extended language of simply-typed λ-terms) with differing evaluation strategies. In effect, he establishes a fully-abstract version of the **Simulation** property of Theorem 3.1. Fully-abstract translations provide a means of comparing the expressive power of languages. However, the goal of full abstraction requires a thunk-introducing transformation which is much more complicated than our $\mathcal{T}$. In addition, since the translation is based on type-indexed retractions, it does not seem possible to use it in an untyped setting as we require here. Finally, the translation cannot be used to obtain a factoring of $\mathcal{C}_n$ because of problems similar to those that occurred with our $\mathcal{T}_{\mathcal{I}}$.

Asperti and Curien give an interesting formulation of thunks in a categorical setting [5, 16]. Two combinators *freeze* and *unfreeze*, which are analogous to our *delay* and *force* but have slightly different equational properties, are used to implement lazy evaluation in the Categorical Abstract Machine. In addition, *freeze* and *unfreeze* can be elegantly characterized using a co-monad.

In earlier work [21], we presented the factorization of $\mathcal{C}_n$ into $\mathcal{C}_v$ and $\mathcal{T}$, for the untyped lambda-calculus with n-ary functions (*à la* Scheme [12]). This is analogous to the factoring in Section 3.3.2.

3.7 Conclusion

We have shown that all the properties established by Plotkin for his continuation-based simulation $\mathcal{C}_n$ can be obtained via a simpler thunk-based transformation $\mathcal{T}$. As a consequence, simulating call-by-name operational behavior and equational reasoning in a call-by-value setting is simplified. To the best of our knowledge, our treatment is the first formal connection of thunks with Plotkin's seminal work [76].

Furthermore, we have shown that the thunk transformation $\mathcal{T}$ establishes a previously unrecognized connection between Plotkin's simulations $\mathcal{C}_n$ and $\mathcal{C}_v$ — $\mathcal{C}_n$ can be obtained by composing $\mathcal{C}_v$ with $\mathcal{T}$. The benefit of this result is that almost all the formal properties of $\mathcal{C}_n$ follow from the formal properties of $\mathcal{C}_v$ and $\mathcal{T}$. This sheds new light on continuation-passing style in general. Moreover, the factorization has already proved useful in several applications dealing with the implementation of call-by-name and lazy languages [22, 73].

We have connected these results with recent work on the typing of CPS transformations. Both the simulation properties of thunks and the factorization of $\mathcal{C}_n$ by $\mathcal{C}_v$ and $\mathcal{T}$ scale up to a typed setting.

For the most part, thunks have been used informally for implementation purposes. Recent work indicates that they are useful for establishing fundamental theoretical properties as well. This work should give additional insight into both formal and practical applications of thunks.

Acknowledgements

The idea of using an equational correspondence to relate the call-by-name language and the call-by-value language with thunks was inspired by the recent work of Sabry and Felleisen [86]. Their detailed proofs were a great help in presenting the material here.

Andrzej Filinski, Julia Lawall, Sergey Kotov and David Schmidt gave helpful comments on an earlier version of this chapter which appeared as a Kansas State University Computing and Information Sciences technical report [39].

3.8 Proofs

3.8.1 Summary of proof techniques

The following techniques are used in the proofs below.

1. induction over the structure of terms and contexts
2. induction over the structure of single-step evaluation rules
3. induction over the number of steps in an evaluation sequence
4. induction over the number of reductions in a reduction sequence

Relating program calculi often requires proofs of properties of the form

$$\lambda r \vdash e_0 \longrightarrow e_1 \;\Rightarrow\; \lambda s \vdash T\langle\!\langle e_0 \rangle\!\rangle \longrightarrow\!\!\!\!\!\rightarrow T\langle\!\langle e_1 \rangle\!\rangle$$

where r and s are some notions of reduction and T is some compositional translation from language l_a to l_b (*i.e.*, for any context $C \in \mathit{Contexts}[l_a]$, $T\langle\!\langle C \rangle\!\rangle \in \mathit{Contexts}[l_b]$ and $T\langle\!\langle C[e] \rangle\!\rangle \equiv T\langle\!\langle C \rangle\!\rangle[T\langle\!\langle e \rangle\!\rangle]$). Since T is compositional and since the relations $\longrightarrow_r$ and $\longrightarrow\!\!\!\!\!\rightarrow_r$ are compatible for any notion of reduction r, the necessary condition can be simplified so that only proofs involving r-redexes (*i.e.*, the relation $\rightsquigarrow_r$) are required instead of proofs for an arbitrary reduction step (*i.e.*, the relation $\longrightarrow_r$).

Property 3.18

$$\forall e_0, e_1 \in \mathit{Terms}[l_a] \,.\, \lambda r \vdash e_0 \rightsquigarrow e_1 \;\Rightarrow\; \lambda s \vdash T\langle\!\langle e_0 \rangle\!\rangle \longrightarrow\!\!\!\!\!\rightarrow T\langle\!\langle e_1 \rangle\!\rangle$$

implies

$$\forall e_0', e_1' \in \mathit{Terms}[l_a] \,.\, \lambda r \vdash e_0' \longrightarrow e_1' \;\Rightarrow\; \lambda s \vdash T\langle\!\langle e_0' \rangle\!\rangle \longrightarrow\!\!\!\!\!\rightarrow T\langle\!\langle e_1' \rangle\!\rangle$$

Proof: From the definition of $\longrightarrow_r$, $e_0' \longrightarrow_r e_1'$ implies that there exists terms $e_0, e_1 \in \mathit{Terms}[l_a]$ and a context $C \in \mathit{Contexts}[l_a]$ such that $e_0' \equiv C[e_0], e_1' \equiv C[e_1]$ and $e_0 \rightsquigarrow_r e_1$. From the given assumption it follows that $T\langle\!\langle e_0 \rangle\!\rangle \longrightarrow\!\!\!\!\!\rightarrow_s T\langle\!\langle e_1 \rangle\!\rangle$. From compositionality of T and compatibility of $\longrightarrow\!\!\!\!\!\rightarrow_s$, $T\langle\!\langle e_0' \rangle\!\rangle \equiv T\langle\!\langle C \rangle\!\rangle[T\langle\!\langle e_0 \rangle\!\rangle] \longrightarrow\!\!\!\!\!\rightarrow_s T\langle\!\langle C \rangle\!\rangle[T\langle\!\langle e_1 \rangle\!\rangle] \equiv T\langle\!\langle e_1' \rangle\!\rangle$. ∎

3.8.2 $\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*$ — a language closed under reductions

The goal of this section is to show that terms in the language $\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*$ correspond to the set of Λ_τ terms T reachable from $\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle$ (the image of $\mathcal{T}$) via $\beta_a\tau$ reduction. The formal definition of T follows.

$$T \stackrel{\text{def}}{=} \{t_2 \in \Lambda_\tau \mid \exists t_1 \in \mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle . \lambda\beta_a\tau \vdash t_1 \longrightarrow\!\!\!\!\!\rightarrow t_2\}$$

The motivation for introducing $\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*$ is to give an intensional (and inductive) definition of the structure of T terms. This will allow translations on T to be defined by structural recursion and proofs of properties of T to proceed by structural induction.

There are some subtleties involved with this technique: although the linear representation of a term $t \in T$ (which is inductively constructed according to the definition of Λ_τ) may have a counterpart in $\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*$, the tree structure of the two terms is not identical. For example $x\,(\mathit{delay}\,(\mathit{force}\,y)) \in T$ can be factored into $x\,[\cdot] \in \mathit{Contexts}[\Lambda_\tau]$ and $\mathit{delay}\,(\mathit{force}\,y) \in \Lambda_\tau$ whereas $x\,(\mathit{delay}\,(\mathit{force}\,y)) \in \mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*$ cannot be factored in a similar fashion since neither $x\,[\cdot] \in \mathit{Contexts}[\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*]$ nor $\mathit{delay}\,(\mathit{force}\,y) \in \mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*$. This distinction is important since the reduction relations are defined with respect to terms and contexts from a particular language. Using the example above (where $\lambda[l]r$ denotes reductions r being generated with respect to contexts in l),

$$\lambda[\Lambda_\tau]\tau_\eta \vdash x\,(\mathit{delay}\,(\mathit{force}\,y)) \longrightarrow x\,y$$

but

$$\lambda[\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*]\tau_\eta \vdash x\,(\mathit{delay}\,(\mathit{force}\,y)) \not\longrightarrow x\,y.$$

This potential ambiguity can be resolved by introducing a translation $\mathcal{U} : \mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^* \rightarrow \Lambda_\tau$ which "forgets" the extra structure associated with $\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*$. Our intent is that whenever a term $t \in \mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*$ is mentioned, we actually refer to the term $\mathcal{U}\langle\!\langle t \rangle\!\rangle \in \mathcal{U}\langle\!\langle\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*\rangle\!\rangle \subset \Lambda_\tau$ (where in $\mathcal{U}\langle\!\langle\mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^*\rangle\!\rangle$ we intend the obvious extension of $\mathcal{U} : \mathcal{T}\langle\!\langle\Lambda\rangle\!\rangle^* \rightarrow \Lambda_\tau$ to *sets* of terms). However, since we are only interested in $\beta_a\tau$ reduction at this point, the following property establishes that no ambiguity (such as the problem with τ_η redexes above) ever occurs. Thus, we feel free to omit explicit reference to $\mathcal{U}$.

Property 3.19 *For all* $t \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$,

$$\lambda[\mathcal{T}\langle\!|\Lambda|\!\rangle^*]\lambda\beta_a\tau \vdash t \longrightarrow t' \text{ iff } \lambda[\Lambda_\tau]\lambda\beta_a\tau \vdash \mathcal{U}\langle\!|t|\!\rangle \longrightarrow \mathcal{U}\langle\!|t'|\!\rangle$$

Next, we show that $\mathcal{T}\langle\!|\Lambda|\!\rangle^*$ is closed under relevant substitutions (Property 3.20) and under $\beta_a\tau$ reductions (Property 3.21).

Property 3.20 *For all* $t_1, t_2 \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$, $t_1[x := delay\, t_2] \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$.

Proof: by induction over the structure of t_1. The interesting case is...

case $t_1 \equiv force\, x$: $(force\, x)[x := delay\, t_2] = force\,(delay\, t_2) \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$

The other cases are either trivial or follow from the induction hypothesis. ∎

Property 3.21 *For all* $t \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$, $\lambda\beta_a\tau \vdash t \longrightarrow t'$ *implies* $t' \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$.

Proof: by induction over the structure of t.

case $t \equiv c$: no reductions are possible

case $t \equiv force\, x$: no reductions are possible

case $t \equiv force\,(delay\, t_0)$: two cases of possible reductions:

case $force\,(delay\, t_0) \longrightarrow_\tau t_0$: $t_0 \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$
case $force\,(delay\, t_0) \longrightarrow_{\beta_a\tau} force\,(delay\, t_0')$: by ind. hyp., $t_0' \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$ so $force\,(delay\, t_0') \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$.

case $t \equiv \lambda x\,.\,t_0$: if $\lambda x\,.\,t_0 \longrightarrow_{\beta_a\tau} \lambda x\,.\,t_0'$ then $t_0' \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$ by ind. hyp. and so $\lambda x\,.\,t_0' \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$.

case $t \equiv t_0\,(delay\, t_1)$: three cases of possible reductions

case $t_0\,(delay\, t_1) \longrightarrow_{\beta_a\tau} t_0'\,(delay\, t_1)$: follows from ind. hyp. as in case above.
case $t_0\,(delay\, t_1) \longrightarrow_{\beta_a\tau} t_0\,(delay\, t_1')$: follows from ind. hyp. as in case above.
case $(\lambda x\,.\,t_0)\,(delay\, t_1) \longrightarrow_{\beta_a\tau} t_0[x := delay\, t_1]$: follows from the fact that $\mathcal{T}\langle\!|\Lambda|\!\rangle^*$ is closed under substitution (Property 3.20).

∎

Returning to our goal of showing $\mathcal{T}\langle\!|\Lambda|\!\rangle^* = T$, below we show that $T \subseteq \mathcal{T}\langle\!|\Lambda|\!\rangle^*$ (Property 3.22) and $\mathcal{T}\langle\!|\Lambda|\!\rangle^* \subseteq T$ (Property 3.23). First we show $T \subseteq \mathcal{T}\langle\!|\Lambda|\!\rangle^*$, that is, the set T of terms reachable from $t \in \mathcal{T}\langle\!|\Lambda|\!\rangle$ via $\beta_a\tau$-reductions is contained in $\mathcal{T}\langle\!|\Lambda|\!\rangle^*$.

Property 3.22 $T \subseteq \mathcal{T}\langle\!|\Lambda|\!\rangle^*$.

Proof: Let $t \in T$. From the definition of T there exists an $s \in \mathcal{T}\langle\!|\Lambda|\!\rangle$ such that $\lambda\beta_a\tau \vdash s \longrightarrow\!\!\!\!\!\rightarrow t$ in n steps. Now we show $t \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$ by induction on n.

case $n = 0$: then $t \equiv s \in \mathcal{T}\langle\!|\Lambda|\!\rangle \subset \mathcal{T}\langle\!|\Lambda|\!\rangle^*$.
case $n = i + 1$: then $\lambda\beta_a\tau \vdash s \longrightarrow\!\!\!\!\!\rightarrow t \longrightarrow u$. By ind. hyp. $t \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$ and therefore $u \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$ since $\mathcal{T}\langle\!|\Lambda|\!\rangle^*$ is closed under reductions (Property 3.21).

∎

Next, we show that $\mathcal{T}\langle\!|\Lambda|\!\rangle^* \subseteq T$, that is, every term $\mathcal{T}\langle\!|\Lambda|\!\rangle^*$ is reachable from $t \in \mathcal{T}\langle\!|\Lambda|\!\rangle$ via $\beta_a\tau$-reductions.

Property 3.23 $\mathcal{T}\langle\!|\Lambda|\!\rangle^* \subseteq T$. *That is,* $t \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$ *implies* $t \in T$, *i.e., there exists an* $s \in \mathcal{T}\langle\!|\Lambda|\!\rangle$ *such that* $\lambda\beta_a\tau \vdash s \longrightarrow\!\!\!\!\!\rightarrow t$.

Proof: by induction over the structure of t. The interesting case is...

case $t \equiv force\,(delay\, t_0)$:
Since $t_0 \in \mathcal{T}\langle\!|\Lambda|\!\rangle^*$, by the induction hypothesis there exists an $s_0 \in \mathcal{T}\langle\!|\Lambda|\!\rangle$ such that $\lambda\beta_a\tau \vdash s_0 \longrightarrow\!\!\!\!\!\rightarrow t_0$. So take $s \equiv (\lambda x\,.\,force\, x)\,(delay\, s_0)$.

The rest of the cases follow either trivially or by the induction hypothesis. ∎

3.8.3 Indifference for $\mathcal{T}$ on Λ

In this section, we show that the meaning of thunked terms is independent of evaluation order, *i.e.*,

$$eval_v(\mathcal{T}\langle\!|e|\!\rangle) \simeq eval_n(\mathcal{T}\langle\!|e|\!\rangle)$$

First, the following property shows that each call-by-name evaluation step corresponds to a call-by-value evaluation step and vice versa.

Property 3.24 *For all* $t \in Progs[\mathcal{T}\langle\!|\Lambda|\!\rangle^*]$,

$$t \longmapsto_n t' \text{ iff } t \longmapsto_v t'.$$

Proof: ($\Rightarrow$ direction) by induction over the structure of $\longmapsto_n$ rules.

case $(\lambda x\,.\,t_1)\,(delay\,t_2) \longmapsto_n t_1[x := (delay\,t_2)]$:

It is the case that $(\lambda x\,.\,t_1)\,(delay\,t_2) \longmapsto_v t_1[x := (delay\,t_2)]$.

case $force\,(delay\,t) \longmapsto_n t$:

It is the case that $force\,(delay\,t) \longmapsto_v t$.

case $t_1\,(delay\,t_2) \longmapsto_n t_1'\,(delay\,t_2)$ because $t_1 \longmapsto_n t_1'$:

by induction, $t_1 \longmapsto_v t_1'$ and it follows that $t_1\,(delay\,t_2) \longmapsto_v t_1'\,(delay\,t_2)$.

($\Leftarrow$ direction) follows in a similar manner. ∎

Property 3.25 shows that coincidence of evaluation steps extends to evaluation sequences.

Property 3.25 *For all* $t \in Progs[\mathcal{T}\langle\!|\Lambda|\!\rangle^*]$,

$$t_0 \longmapsto_n^* t_n \text{ iff } t_0 \longmapsto_v^* t_n.$$

Proof: ($\Rightarrow$ direction) by induction over the number of steps n in the evaluation sequence.

case $n = 0$: trivial.
case $n = i + 1$: that is, $t_0 \longmapsto_n^* t_i \longmapsto_n t_{i+1}$.
Now $t_0 \longmapsto_v^* t_i$ holds by induction and $t_i \longmapsto_v t_{i+1}$ holds by Property 3.24.

($\Leftarrow$ direction) follows in a similar manner. ∎

Now **Indifference** for thunked terms follows directly from Property 3.25 and the definitions of $eval_n$ and $eval_v$.

3.8.4 Simulation for $\mathcal{T}$ on Λ

This section establishes the **Simulation** property for thunks, *i.e.*,

$$\mathcal{T}\langle\!|eval_n(e)|\!\rangle \simeq_\tau eval_v(\mathcal{T}\langle\!|e|\!\rangle)$$

The strategy for proving **Simulation** proceeds as outlined in the body of the chapter — we formally establish the properties of Figure 3.2. The following property shows how $\mathcal{T}$ interacts with substitution.

Property 3.26 *For all* $e_1, e_2 \in \Lambda$, $\mathcal{T}\langle\!|e_1|\!\rangle[x := delay\,\mathcal{T}\langle\!|e_2|\!\rangle] \longrightarrow\!\!\!\!\rightarrow_\tau \mathcal{T}\langle\!|e_1[x := e_2]|\!\rangle$.

Proof: by induction over the structure of e_1. The interesting case is...

case $e_1 \equiv x$:

$$\begin{array}{lll} \mathcal{T}\langle\!|x|\!\rangle[x := delay\,\mathcal{T}\langle\!|e_2|\!\rangle] & = & (force\,x)[x := delay\,\mathcal{T}\langle\!|e_2|\!\rangle] \\ & = & force\,(delay\,\mathcal{T}\langle\!|e_2|\!\rangle) \\ & \longrightarrow_\tau & \mathcal{T}\langle\!|e_2|\!\rangle \\ & = & \mathcal{T}\langle\!|x[x := e_2]|\!\rangle \end{array}$$

The rest of the cases follow trivially or by the induction hypothesis and compatibility of τ-reduction. ∎

Property 3.27 shows how substitution interacts with τ-equivalence classes.

Property 3.27 *For all $e_0, e_1 \in \Lambda$, and for all $t_0 \in [e_0]_\tau$, $t_1 \in [e_1]_\tau$,*

$$t_0[x := \mathit{delay}\, t_1] \in [e_0[x := e_1]]_\tau$$

Proof: The statement above is equivalent to the following. For all $e_0, e_1 \in \Lambda$, and for all t_0, t_1 such that $\lambda\tau \vdash t_0 = \mathcal{T}(\!|e_0|\!)$ and $\lambda\tau \vdash t_1 = \mathcal{T}(\!|e_1|\!)$, it is the case that $\lambda\tau \vdash t_0[x := \mathit{delay}\, t_1] = \mathcal{T}(\!|e_0[x := e_1]|\!)$. This is shown with the following steps.

$$\begin{array}{lll}
t_0[x := \mathit{delay}\, t_1] & =_\tau \quad \mathcal{T}(\!|e_0|\!)[x := \mathit{delay}\, t_1] & \ldots \textit{substitutivity of } \tau \\
& =_\tau \quad \mathcal{T}(\!|e_0|\!)[x := \mathit{delay}\, \mathcal{T}(\!|e_1|\!)] & \ldots \textit{compatibility of } \tau \\
& =_\tau \quad \mathcal{T}(\!|e_0[x := e_1]|\!) & \ldots \textit{Property 3.26}
\end{array}$$

∎

The following property states that each $\longmapsto_n$ evaluation step on a source term implies corresponding $\longmapsto_v$ evaluation steps on thunked terms.

Property 3.28 *For all $e_0, e_1 \in \Lambda$.*

$$e_0 \longmapsto_n e_1 \;\Rightarrow\; \forall t_0 \in [e_0]_\tau \,.\, \exists t_2 \in [e_0]_\tau \,.\, \exists t_1 \in [e_1]_\tau \,.\, t_0 \longmapsto_\tau^* t_2 \longmapsto_{\beta_v} t_1$$

Proof: by induction(†) over the structure of the $\longmapsto_n$ rules.

case $(\lambda x \,.\, e_a)\, e_b \longmapsto_n e_a[x := e_b]$:
Show that $\forall t_0 \in [(\lambda x \,.\, e_a)\, e_b]_\tau \,.\, \exists t_2 \in [(\lambda x \,.\, e_a)\, e_b]_\tau \,.\, \exists t_1 \in [e_a[x := e_b]]_\tau \,.\, t_0 \longmapsto_\tau^* t_2 \longmapsto_{\beta_v} t_1$:
by induction(‡) over the structure of t_0.
case $t_0 \equiv t_2 \equiv (\lambda x \,.\, t_a)\, (\mathit{delay}\, t_b)$ where $t_a \in [e_a]_\tau$ and $t_b \in [e_b]_\tau$:
$(\lambda x \,.\, t_a)\, (\mathit{delay}\, t_b) \longmapsto_{\beta_v} t_a[x := \mathit{delay}\, t_b] \in [e_a[x := e_b]]_\tau$ *...by Property 3.27*
case $t_0 \equiv \mathit{force}\, (\mathit{delay}\, t_0')$ where $t_0' \in [(\lambda x \,.\, e_a)\, e_b]_\tau$:

$$\begin{array}{lll}
\mathit{force}\, (\mathit{delay}\, t_0') & \longmapsto_\tau \quad t_0' & \\
& \longmapsto_\tau^* \quad t_2 \in [(\lambda x \,.\, e_a)\, e_b]_\tau & \ldots \textit{by ind. hyp.(‡)} \\
& \longmapsto_{\beta_v} \quad t_1 \in [e_a[x := e_b]]_\tau & \ldots \textit{by ind. hyp.(‡)}
\end{array}$$

case $e_a\, e_b \longmapsto_n e_a'\, e_b$ because $e_a \longmapsto_n e_a'$:
Now $\forall t_a \in [e_a]_\tau \,.\, \exists t_a'' \in [e_a]_\tau \,.\, \exists t_a' \in [e_a']_\tau \,.\, t_a \longmapsto_\tau^* t_a'' \longmapsto_{\beta_v} t_a'$ holds by the induction hypothesis(†).
Show that $\forall t_0 \in [e_a\, e_b]_\tau \,.\, \exists t_2 \in [e_a\, e_b]_\tau \,.\, \exists t_1 \in [e_a'\, e_b]_\tau \,.\, t_0 \longmapsto_\tau^* t_2 \longmapsto_{\beta_v} t_1$:
by induction(‡) over the structure of t_0.
case $t_0 \equiv t_a\, (\mathit{delay}\, t_b)$ where $t_a \in [e_a]_\tau$ and $t_b \in [e_b]_\tau$:

$$\begin{array}{lll}
t_a\, (\mathit{delay}\, t_b) & \longmapsto_\tau^* \quad t_a''\, (\mathit{delay}\, t_b) \equiv t_2 \in [e_a\, e_b]_\tau & \\
& \longmapsto_{\beta_v} \quad t_a'\, (\mathit{delay}\, t_b) \equiv t_1 \in [e_a'\, e_b]_\tau & \ldots \textit{by ind. hyp.(†)}
\end{array}$$

case $t_0 \equiv \mathit{force}\, (\mathit{delay}\, t_0')$ where $t_0' \in [e_a\, e_b]_\tau$:

$$\begin{array}{lll}
\mathit{force}\, (\mathit{delay}\, t_0') & \longmapsto_\tau \quad t_0' & \\
& \longmapsto_\tau^* \quad t_2 \in [e_a\, e_b]_\tau & \ldots \textit{by ind. hyp.(‡)} \\
& \longmapsto_{\beta_v} \quad t_1 \in [e_a'\, e_b]_\tau & \ldots \textit{by ind. hyp.(‡)}
\end{array}$$

∎

An evaluation sequence on source terms implies a corresponding evaluation sequence on thunked terms.

Property 3.29 *For all $e_0, e_n \in \Lambda$,*

$$e_0 \longmapsto_n^* e_n \;\Rightarrow\; \forall t_0 \in [e_0]_\tau \,.\, \exists t_n \in [e_n]_\tau \,.\, t_0 \longmapsto_v^* t_1.$$

Proof: by induction on the number of steps in the evaluation sequence — applying Property 3.28 at each step. ∎

To prove that termination of source term evaluation steps implies termination of thunked term evaluation steps, the following property shows a correspondence of source term values and thunked term values. Specifically, if v is a value then any term t which is τ-equivalent to $\mathcal{T}\langle\!|v|\!\rangle$ reduces to a value.

Property 3.30 *For all* $v \in Values_n[\Lambda]$,

$$\forall t \in [v]_\tau \,.\, \exists v_t \in [v]_\tau \,.\; t \longmapsto_\tau^* v_t \;\wedge\; v_t \in Values_v[\Lambda_\tau].$$

Proof: by cases of v:

case $v \equiv c$: by induction over $t \in [c]_\tau$, show $t \longmapsto_\tau^* c \equiv v_t$:
  case $t \equiv c$: *...immediate*
  case $t \equiv force\,(delay\,t')$ where $t' \in [c]_\tau$:
    $force\,(delay\,t') \longmapsto_\tau t' \longmapsto_\tau^* c$ *...by ind. hyp.*

case $e \equiv \lambda x\,.\,e_0$: by induction over $t \in [\lambda x\,.\,e_0]_\tau$, show $t \longmapsto_\tau^* \lambda x\,.\,t_0 \equiv v_t$ for some $t_0 \in [e_0]_\tau$:
  case $t \equiv \lambda x\,.\,t_0$: *...immediate*
  case $t \equiv force\,(delay\,t')$ where $t' \in [\lambda x\,.\,e_0]_\tau$:
    $force\,(delay\,t') \longmapsto_\tau t' \longmapsto_\tau^* \lambda x\,.\,t_0$ *...by ind. hyp.*

∎

The $\Rightarrow$ direction of **Simulation** is established by the following Lemma.

Lemma 3.1 *For all* $e \in Progs[\Lambda]$,

$$eval_n(e) = v_e \quad\Rightarrow\quad eval_v(\mathcal{T}\langle\!|e|\!\rangle) = v_t \;\wedge\; v_t \in [v_e]_\tau$$

Proof: Property 3.29 shows that τ-equivalence is maintained during evaluation steps and Property 3.30 establishes that $eval_v(\mathcal{T}\langle\!|e|\!\rangle)$ is defined whenever $eval_n(e)$ is defined. ∎

Next, the $\Leftarrow$ direction of **Simulation** is proved in a similar way. The following properties state that each $\longmapsto_v$ evaluation step on a thunked term corresponds to zero or one $\longmapsto_n$ evaluation steps on the corresponding source term.

Property 3.31 *For all* $e \in \Lambda$ *and* $t \in [e]_\tau$,

$$t \longmapsto_\tau t' \quad\Rightarrow\quad t' \in [e]_\tau$$

Proof: immediate, since $[e]_\tau$ is closed under τ-reductions. ∎

Property 3.32 *For all* $e_0, e_1 \in \Lambda$ *and* $t_0 \in [e_0]_\tau$ *and* $t_1 \in [e_1]_\tau$,

$$t_0 \longmapsto_{\beta_v} t_1 \quad\Rightarrow\quad e_0 \longmapsto_n e_1$$

Proof: by induction over the structure of the $\longmapsto_v$ rules

case $(\lambda x\,.\,t_a)\,(delay\,t_b) \longmapsto_{\beta_v} t_a[x := delay\,t_b]$:

  Note $t_0 \equiv (\lambda x\,.\,t_a)\,(delay\,t_b) \in (\lambda x\,.\,[e_a]_\tau)\,(delay\,[e_b]_\tau) \subseteq [(\lambda x\,.\,e_a)\,e_b]_\tau$ where $t_a \in [e_a]_\tau$ and $t_b \in [e_b]_\tau$. Now $e_0 \equiv (\lambda x\,.\,e_a)\,e_b \longmapsto_n e_a[x := e_b] \equiv e_1$ and $t_a[x := delay\,t_b] \in [e_a[x := e_b]]_\tau$ by Property 3.27.

case $t_a\,(delay\,t_b) \longmapsto_{\beta_v} t'_a\,(delay\,t_b)$ because $t_a \longmapsto_{\beta_v} t'_a$:

  Note $t_0 \equiv t_a\,(delay\,t_b) \in [e_a]_\tau\,(delay\,[e_b]_\tau) \subseteq [e_a\,e_b]_\tau$ where $t_a \in [e_a]_\tau$ and $t_b \in [e_b]_\tau$. Furthermore, $t_1 \equiv t'_a\,(delay\,t_b) \in [e'_a]_\tau\,(delay\,[e_b]_\tau) \subseteq [e'_a\,e_b]_\tau$ where $t'_a \in [e'_a]_\tau$ and $t_b \in [e_b]_\tau$. Now $t_a \longmapsto_{\beta_v} t'_a \;\Rightarrow$ $e_a \longmapsto_n e'_a$ by the induction hypothesis and so $e_a\,e_b \longmapsto_n e'_a\,e_b$.

An evaluation sequence on thunked terms implies a corresponding evaluation sequence on source terms.

Property 3.33 *For all* $e_0, e_n \in \Lambda$ *and* $t_0 \in [e_0]_\tau$ *and* $t_n \in [e_n]_\tau$,

$$t_0 \longmapsto_v^* t_n \quad \Rightarrow \quad e_0 \longmapsto_n^* e_n$$

Proof: by induction on the number of steps n in the evaluation sequence.

case $n = 0$: trivial.
case $n = i + 1$: that is, $t_0 \longmapsto_v^* t_i \longmapsto_v t_{i+1}$ where $t_0 \in [e_0]_\tau$, $t_i \in [e_i]_\tau$, and $t_{i+1} \in [e_{i+1}]_\tau$.
Now $e_0 \longmapsto_n^* e_i$ holds by induction so it remains to show that $e_i \longmapsto_n e_{i+1}$.
There are two cases:
case $t_i \longmapsto_\tau t_{i+1}$: then $e_i \longmapsto_n e_{i+1}$ by Property 3.31.
case $t_i \longmapsto_{\beta_v} t_{i+1}$: then $e_1 \longmapsto_n e_{i+1}$ by Property 3.32.

∎

The following property establishes that termination of evaluation of thunked terms implies termination of evaluation of the corresponding source terms.

Property 3.34 *For all* $e \in \Lambda$ *and* $t \in [e]_\tau$,

$$t \in \mathit{Values}_v[\Lambda_\tau] \quad \Rightarrow \quad e \in \mathit{Values}_n[\Lambda].$$

Proof: by induction over the structure of e, one sees that t can be a value only when $e \equiv c$ or $e \equiv \lambda x\,.\,e'$. ∎

The $\Leftarrow$ direction of **Simulation** is established by the following Lemma.

Lemma 3.2 *For all* $e \in \Lambda$, $v_t, v_e \in \mathcal{T}(\!|\Lambda|\!)^*$,

$$\mathit{eval}_v(\mathcal{T}(\!|e|\!)) = v_t \quad \Rightarrow \quad \mathit{eval}_n(e) = v_e \quad \text{...where } v_2 \in [v_1]_\tau$$

Proof: Property 3.33 shows that τ-equivalence is maintained during evaluation steps and Property 3.34 establishes that $\mathit{eval}_n(e)$ is defined whenever $\mathit{eval}_v(\mathcal{T}(\!|e|\!))$ is defined. ∎

3.8.5 Translation for $\mathcal{T}$ on Λ

This section establishes the **Translation** property for thunks, *i.e.*,

$$\lambda\beta \vdash e_1 = e_2 \text{ iff } \lambda\beta_a\tau \vdash \mathcal{T}(\!|e_1|\!) = \mathcal{T}(\!|e_2|\!)$$

To prove **Translation**, we show an equational correspondence between the language Λ under theory $\lambda\beta$ and language $\mathcal{T}(\!|\Lambda|\!)^*$ under theory $\lambda\beta_a$ (*i.e.*, $\lambda\beta_v$ as well as $\lambda\beta$). The **Translation** property corresponds to component 3 of the theorem below.

Theorem 3.6 (Equational Correspondence) *For all* $e, e_1, e_2 \in \Lambda$ *and* $t, t_1, t_2 \in \mathcal{T}(\!|\Lambda|\!)^*$,

1. $\lambda\beta \vdash e = (\mathcal{T}^{-1} \circ \mathcal{T})(\!|e|\!)$
2. $\lambda\beta_a\tau \vdash t = (\mathcal{T} \circ \mathcal{T}^{-1})(\!|t|\!)$
3. $\lambda\beta \vdash e_1 = e_2$ *iff* $\lambda\beta_a\tau \vdash \mathcal{T}(\!|e_1|\!) = \mathcal{T}(\!|e_2|\!)$
4. $\lambda\beta_a\tau \vdash t_1 = t_2$ *iff* $\lambda\beta \vdash \mathcal{T}^{-1}(\!|t_1|\!) = \mathcal{T}^{-1}(\!|t_2|\!)$

Component 1 of Theorem 3.6 follows from the property below.

Property 3.35 *For all* $e \in \Lambda$, $e \equiv (\mathcal{T}^{-1} \circ \mathcal{T})(\!|e|\!)$.

Proof: a simple induction on the structure of e. ∎

Component 2 of Theorem 3.6 follows from the property below.

Property 3.36 *For all* $t \in \mathcal{T}(\!|\Lambda|\!)^*$, $\lambda\tau \vdash t = (\mathcal{T} \circ \mathcal{T}^{-1})(\!|t|\!)$.

Proof: by induction on the structure of t.

case $t \equiv c$: $(\mathcal{T} \circ \mathcal{T}^{-1})(\!|c|\!) = c$.
case $t \equiv force\ x$: $(\mathcal{T} \circ \mathcal{T}^{-1})(\!|force\ x|\!) = force\ x$.
case $t \equiv force\ (delay\ t_0)$:

$$\begin{array}{lll} (\mathcal{T} \circ \mathcal{T}^{-1})(\!|force\ (delay\ t_0)|\!) & = & (\mathcal{T} \circ \mathcal{T}^{-1})(\!|t_0|\!) \\ & =_\tau & t_0 \quad \textit{...by ind. hyp.} \\ & \longleftarrow_\tau & force\ (delay\ t_0) \end{array}$$

case $t \equiv \lambda x\,.\,t_0$:

$$\begin{array}{lll} (\mathcal{T} \circ \mathcal{T}^{-1})(\!|\lambda x\,.\,t_0|\!) & = & \lambda x\,.\,(\mathcal{T} \circ \mathcal{T}^{-1})(\!|t_0|\!) \\ & =_\tau & \lambda x\,.\,t_0 \quad \textit{...by ind. hyp. and compatibility of} =_\tau \end{array}$$

case $t \equiv t_0\ (delay\ t_1)$:

$$\begin{array}{lll} (\mathcal{T} \circ \mathcal{T}^{-1})(\!|t_0\ (delay\ t_1)|\!) & = & (\mathcal{T} \circ \mathcal{T}^{-1})(\!|t_0|\!)\ (delay\ (\mathcal{T} \circ \mathcal{T}^{-1})(\!|t_1|\!)) \\ & =_\tau & t_0\ (delay\ t_1) \quad \textit{...by ind. hyp. and compatibility of} =_\tau \end{array}$$

∎

Components 3 and 4 of Theorem 3.6 follow from Properties 3.38 and 3.39 as outlined in Section 3.2 on page 49. But first, the following property shows how $\mathcal{T}^{-1}$ interacts with substitution.

Property 3.37 *For all* $t_1, t_2 \in \Lambda_\tau$, $\mathcal{T}^{-1}(\!|t_1|\!)[x := \mathcal{T}^{-1}(\!|t_2|\!)] = \mathcal{T}^{-1}(\!|t_1[x := delay\ t_2]|\!)$.

Proof: a simple induction over the structure of t_1. ∎

The following property establishes that each reduction on source terms corresponds to one or more reductions on thunked terms.

Property 3.38 *For all* $e_1, e_2 \in \Lambda$, $\lambda\beta \vdash e_1 \longrightarrow e_2 \Rightarrow \lambda\beta_a\tau \vdash \mathcal{T}(\!|e_1|\!) \longrightarrow\!\!\!\!\!\rightarrow \mathcal{T}(\!|e_2|\!)$

Proof: due to Property 3.18, it is sufficient to show $\lambda\beta_a\tau \vdash \mathcal{T}(\!|(\lambda x\,.\,e_1)\,e_2|\!) \longrightarrow\!\!\!\!\!\rightarrow \mathcal{T}(\!|e_1[x := e_2]|\!)$.

$$\begin{array}{lll} \mathcal{T}(\!|(\lambda x\,.\,e_1)\,e_2|\!) & = & (\lambda x\,.\,\mathcal{T}(\!|e_1|\!))\ (delay\ \mathcal{T}(\!|e_2|\!)) \\ & \longrightarrow_{\beta_a} & \mathcal{T}(\!|e_1|\!)[x := delay\ \mathcal{T}(\!|e_2|\!)] \\ & \longrightarrow\!\!\!\!\!\rightarrow_\tau & \mathcal{T}(\!|e_1[x := e_2]|\!) \quad \textit{...by Property 3.26} \end{array}$$

∎

Next it is shown that each reduction on thunked terms corresponds to zero or one reduction on source terms.

Property 3.39 *For all* $t_1, t_2 \in \mathcal{T}(\!|\Lambda|\!)^*$, $\lambda\beta_a\tau \vdash t_1 \longrightarrow t_2 \Rightarrow \lambda\beta \vdash \mathcal{T}^{-1}(\!|t_1|\!) \longrightarrow\!\!\!\!\!\rightarrow \mathcal{T}^{-1}(\!|t_2|\!)$

Proof: due to Property 3.18, it is sufficient to show the following two cases:

- $\lambda\beta \vdash \mathcal{T}^{-1}(\!|force\ (delay\ t)|\!) \longrightarrow\!\!\!\!\!\rightarrow \mathcal{T}^{-1}(\!|t|\!)$.

$$\begin{array}{lll} \mathcal{T}^{-1}(\!|force\ (delay\ t)|\!) & = & \mathcal{T}^{-1}(\!|delay\ t|\!) \\ & = & \mathcal{T}^{-1}(\!|t|\!) \quad \textit{...by definition of} \mathcal{T}^{-1} \end{array}$$

- $\lambda\beta \vdash \mathcal{T}^{-1}(\!|(\lambda x\,.\,t_1)\,(delay\ t_2)|\!) \longrightarrow\!\!\!\!\!\rightarrow \mathcal{T}^{-1}(\!|t_1[x := delay\ t_2]|\!)$.

$$\begin{array}{lll} \mathcal{T}^{-1}(\!|(\lambda x\,.\,t_1)\,(delay\ t_2)|\!) & = & (\lambda x\,.\,\mathcal{T}^{-1}(\!|t_1|\!))\ \mathcal{T}^{-1}(\!|t_2|\!) \\ & \longrightarrow_\beta & \mathcal{T}^{-1}(\!|t_1|\!)[x := \mathcal{T}^{-1}(\!|t_2|\!)] \\ & = & \mathcal{T}^{-1}(\!|t_1[x := delay\ t_2]|\!) \quad \textit{...by Property 3.37.} \end{array}$$

∎

3.8.6 Thunks in the typed core language Λ^σ

The grammar below describes the set of Λ^σ_τ terms in the image of $\mathcal{T}$ on Λ^σ.

$$\begin{array}{lcl} t & \in & Terms[\mathcal{T}(\!|\Lambda^\sigma|\!)] \\ t & ::= & c \mid f \mid force\ x \mid \lambda x\,.\,t \mid rec\, f(x).t \mid t_0\,(delay\, t_1) \end{array}$$

The language of terms in the image of $\mathcal{T}$ on Λ^σ closed under $\beta_a \mathrm{rec}_a \tau$-reductions is as follows.

$$\begin{array}{lcl} t & \in & Terms[\mathcal{T}(\!|\Lambda^\sigma|\!)^*] \\ t & ::= & c \mid f \mid force\ x \mid force\,(delay\, t) \mid \lambda x\,.\,t \mid rec\, f(x).t \mid t_0\,(delay\, t_1) \end{array}$$

The correctness proof for $\mathcal{T}(\!|\Lambda^\sigma|\!)^*$ follows the outline of the proof for $\mathcal{T}(\!|\Lambda|\!)^*$. It is also necessary to add cases for f and $rec\, f(x).e$. These additions require the following property showing how f and $rec\, f(x).e$ interact with substitution.

Property 3.40 *For all* $e_0 \in \Lambda^\sigma$,

$$\mathcal{T}(\!|e_0|\!)[f := rec\, f(x).\mathcal{T}(\!|e_1|\!)] = \mathcal{T}(\!|e_0[f := rec\, f(x).e_1]|\!)$$

Proof: by induction over the structure of e_0. ∎

Turning to the analogue of Theorem 3.1, the proof of **Indifference** is trivial. The proof of **Simulation** requires adding cases for f and $rec\, f(x).e$ to the proofs in Section 3.8.4. Below we prove the equational correspondence between Λ^σ and $\mathcal{T}(\!|\Lambda^\sigma|\!)^*$.

Theorem 3.7 (Equational Correspondence for $\mathcal{T}$ on Λ^σ) *For all* $\Gamma \vdash e, e_1, e_2 : \sigma$ *and* $\Gamma \vdash_\tau t, t_1, t_2 : \sigma'$,

1. $\lambda\beta\mathrm{rec} \vdash e = (\mathcal{T}^{-1} \circ \mathcal{T})(\!|e|\!)$
2. $\lambda\beta_a \mathrm{rec}_a \tau \vdash t = (\mathcal{T} \circ \mathcal{T}^{-1})(\!|t|\!)$
3. $\lambda\beta\mathrm{rec} \vdash e_1 = e_2$ *iff* $\lambda\beta_a \mathrm{rec}_a \tau \vdash \mathcal{T}(\!|e_1|\!) = \mathcal{T}(\!|e_2|\!)$
4. $\lambda\beta_a \mathrm{rec}_a \tau \vdash t_1 = t_2$ *iff* $\lambda\beta\mathrm{rec} \vdash \mathcal{T}^{-1}(\!|t_1|\!) = \mathcal{T}^{-1}(\!|t_2|\!)$

The proofs of components 1 and 2 are simple extensions of the proofs for Λ. To establish components 3 and 4 it is only necessary to prove the following two properties.

Property 3.41 *For all* $\Gamma \vdash e_1, e_2 : \sigma$, $\lambda\beta\mathrm{rec} \vdash e_1 \longrightarrow e_2 \Rightarrow \lambda\beta_a \mathrm{rec}_a \tau \vdash \mathcal{T}(\!|e_1|\!) \longrightarrow\!\!\!\!\!\rightarrow \mathcal{T}(\!|e_2|\!)$

Proof: Since β-reduction is considered in Property 3.38, we need only consider rec. Due to Property 3.18, it is sufficient to show the following.

$$\begin{array}{lcl} \mathcal{T}(\!|(rec\, f(x).e_0)\, e_1|\!) & = & (rec\, f(x).\mathcal{T}(\!|e_0|\!))\, delay\, \mathcal{T}(\!|e_1|\!) \\ & \longrightarrow_{\mathrm{rec}_v} & \mathcal{T}(\!|e_0|\!)[f := rec\, f(x).\mathcal{T}(\!|e_0|\!)\,,\, x := delay\, \mathcal{T}(\!|e_1|\!)] \\ & =_\tau & \mathcal{T}(\!|e_0[f := rec\, f(x).e_0\,,\, x := e_1]|\!) \\ & & \textit{...by Property 3.40 and the extension of Property 3.26 to } \Lambda^\sigma. \end{array}$$

∎

Property 3.42 *For all* $\Gamma \vdash_\tau t_1, t_2 : \sigma'$, $\lambda\beta_a \mathrm{rec}_a \tau \vdash t_1 \longrightarrow t_2 \Rightarrow \lambda\beta\mathrm{rec} \vdash \mathcal{T}^{-1}(\!|t_1|\!) \longrightarrow\!\!\!\!\!\rightarrow \mathcal{T}^{-1}(\!|t_2|\!)$

Proof: Since β_a and τ reductions are considered in Property 3.39, we need only consider rec_a. Due to Property 3.18, it is sufficient to show the following.

$$\begin{array}{lcl} \mathcal{T}^{-1}(\!|(rec\, f(x).t_1)\,(delay\, t_2)|\!) & = & (rec\, f(x).\mathcal{T}^{-1}(\!|t_1|\!))\, \mathcal{T}^{-1}(\!|t_2|\!) \\ & \longrightarrow_{\mathrm{rec}} & \mathcal{T}^{-1}(\!|t_1|\!)[f := rec\, f(x).\mathcal{T}^{-1}(\!|t_1|\!)\,,\, x := \mathcal{T}^{-1}(\!|t_2|\!)] \\ & = & \mathcal{T}^{-1}(\!|t_1[f := rec\, f(x).t_1\,,\, x := delay\, t_2]|\!) \\ & & \textit{...by the extension of Property 3.37 to } \Lambda^\sigma. \end{array}$$

∎

3.8.7 Thunks in the extended typed language Λ^{σ^+}

The grammar below describes the set of Λ^σ_τ terms in the image of $\mathcal{T}$ on Λ^{σ^+}.

$$\begin{array}{lcl} t & \in & Terms[\mathcal{T}(\![\Lambda^{\sigma^+}]\!)] \\ t & ::= & c \mid f \mid force\ x \mid \lambda x.t \mid rec\, f(x).t \mid t_0\,(delay\, t_1) \mid if\ t_0\ then\ t_1\ else\ t_2 \mid \\ & & pair\,(delay\, t_1)\,(delay\, t_2) \mid force\,(proj_i\, t) \mid inj_i\,(delay\, t) \mid case\, t\ of\ (x_1.t_1)\,|\,(x_2.t_2) \end{array}$$

The language of terms in the image of $\mathcal{T}$ on Λ^σ closed under reductions is as follows.

$$\begin{array}{lcl} t & \in & Terms[\mathcal{T}(\![\Lambda^{\sigma^+}]\!)^*] \\ t & ::= & c \mid f \mid force\ x \mid force\,(delay\, t) \mid \lambda x.t \mid rec\, f(x).t \mid t_0\,(delay\, t_1) \mid if\ t_0\ then\ t_1\ else\ t_2 \\ & & pair\,(delay\, t_1)\,(delay\, t_2) \mid force\,(proj_i\, t) \mid inj_i\,(delay\, t) \mid case\, t\ of\ (x_1.t_1)\,|\,(x_2.t_2) \end{array}$$

The correctness proof for $\mathcal{T}(\![\Lambda^{\sigma^+}]\!)^*$ follows the outline of the proof for $\mathcal{T}(\![\Lambda]\!)^*$. The cases for conditional reduction are trivial since no aspects of suspensions are involved directly. The reductions for pairing take the form

$$force\,(proj_i\,(pair\ delay\, t_1\ delay\, t_2)) \longrightarrow_{\times_i.\beta_a} force\,(delay\, t_i)$$

and closedness of $\mathcal{T}(\![\Lambda^{\sigma^+}]\!)^*$ follows since $force\,(delay\, t_i) \in \mathcal{T}(\![\Lambda^{\sigma^+}]\!)^*$. The reductions for *case* take the form

$$case\ inj_i\,(delay\, t)\ of\ (x_1.t_1)\,|\,(x_2.t_2) \longrightarrow_{+_i.\beta_a} t_i[x := delay\, t]$$

and closedness of $\mathcal{T}(\![\Lambda^{\sigma^+}]\!)^*$ follows from the property of being closed under relevant substitutions.

Turning to the analogue of Theorem 3.1, the proof of **Indifference** is trivial. The proof of **Simulation** carries over in the expected way. Below we prove the equational correspondence between Λ^{σ^+} and $\mathcal{T}(\![\Lambda^{\sigma^+}]\!)^*$. But first we give the following abbreviations for reduction sets.

$$\begin{array}{lcl} R^{\sigma^+}_{n-\tau} & \stackrel{\text{def}}{=} & R^{\sigma^+}_n \cup \tau \\ R^{\sigma^+}_{v-\tau} & \stackrel{\text{def}}{=} & R^{\sigma^+}_v \cup \tau \end{array}$$

Theorem 3.8 (Equational Correspondence for $\mathcal{T}$ on Λ^{σ^+}) *For all $\Gamma \vdash e, e_1, e_2 : \sigma$ and $\Gamma \vdash_\tau t, t_1, t_2 : \sigma'$,*

1. $\lambda R^{\sigma^+}_n \vdash e = (\mathcal{T}^{-1} \circ \mathcal{T})(\![e]\!)$
2. $\lambda R^{\sigma^+}_{a-\tau} \vdash t = (\mathcal{T} \circ \mathcal{T}^{-1})(\![t]\!)$
3. $\lambda R^{\sigma^+}_n \vdash e_1 = e_2$ *iff* $\lambda R^{\sigma^+}_{a-\tau} \vdash \mathcal{T}(\![e_1]\!) = \mathcal{T}(\![e_2]\!)$
4. $\lambda R^{\sigma^+}_{a-\tau} \vdash t_1 = t_2$ *iff* $\lambda R^{\sigma^+}_n \vdash \mathcal{T}^{-1}(\![t_1]\!) = \mathcal{T}^{-1}(\![t_2]\!)$

The proofs of components 1 and 2 are simple extensions of the proofs for Λ. To establish components 3 and 4 it is only necessary to prove the following two properties.

Property 3.43 *For all $\Gamma \vdash e_1, e_2 : \sigma$, $\lambda R^{\sigma^+}_n \vdash e_1 \longrightarrow e_2 \Rightarrow \lambda R^{\sigma^+}_{n-\tau} \vdash \mathcal{T}(\![e_1]\!) \longrightarrow\!\!\!\!\!\rightarrow \mathcal{T}(\![e_2]\!)$*

Proof: Since β and rec reduction were considered earlier, we consider only reductions for the extensions. Due to Property 3.18, it is sufficient to show the following.

case $\times_i.\beta$:

$$\begin{array}{lcl} \mathcal{T}(\![proj_i\,(pair\, e_1\, e_2)]\!) & = & force\,(proj_i\,(pair\,(delay\, \mathcal{T}(\![e_1]\!))\,(delay\, \mathcal{T}(\![e_2]\!)))) \\ & \longrightarrow_{\times_i.\beta_a} & force\,(delay\, \mathcal{T}(\![e_i]\!)) \\ & \longrightarrow_\tau & \mathcal{T}(\![e_i]\!) \end{array}$$

case $+_i.\beta$:

$$\begin{array}{lll}
\mathcal{T}\langle\!\langle case\ (inj_i\ e_0)\ of\ (x_1.e_1)\mid(x_2.e_2)\rangle\!\rangle & = & case\ (inj_i\ delay\ \mathcal{T}\langle\!\langle e_0\rangle\!\rangle)\ of\ (x_1.\mathcal{T}\langle\!\langle e_1\rangle\!\rangle)\mid(x_2.\mathcal{T}\langle\!\langle e_2\rangle\!\rangle) \\
 & \longrightarrow_{+_i.\beta_v} & \mathcal{T}\langle\!\langle e_i\rangle\!\rangle[x_i := delay\ \mathcal{T}\langle\!\langle e_0\rangle\!\rangle] \\
 & =_\tau & \mathcal{T}\langle\!\langle e_i[x_i := e_0]\rangle\!\rangle \\
 & & ...by\ extension\ of\ Property\ 3.26\ to\ \Lambda^{\sigma^+}.
\end{array}$$

case cnd.t:

$$\begin{array}{lll}
\mathcal{T}\langle\!\langle if\ \mathsf{c}_t\ then\ e_1\ else\ e_2\rangle\!\rangle & = & if\ \mathsf{c}_t\ then\ \mathcal{T}\langle\!\langle e_1\rangle\!\rangle\ else\ \mathcal{T}\langle\!\langle e_2\rangle\!\rangle \\
 & \longrightarrow_{\text{cnd.t}} & \mathcal{T}\langle\!\langle e_1\rangle\!\rangle
\end{array}$$

The case for cnd.f is similar.

∎

Property 3.44 *For all* $\Gamma \vdash_\tau t_1, t_2 : \sigma'$, $\lambda R^{\sigma^+}_{a-\tau} \vdash t_1 \longrightarrow t_2 \Rightarrow \lambda R^{\sigma^+}_n \vdash \mathcal{T}^{-1}\langle\!\langle t_1\rangle\!\rangle \longrightarrow\!\!\!\!\!\rightarrow \mathcal{T}^{-1}\langle\!\langle t_2\rangle\!\rangle$

Proof: We only consider cases for extensions. Due to Property 3.18, it is sufficient to show the following.

case $force\ (proj_i\ (pair\ (delay\ t_1)\ (delay\ t_2))) \longrightarrow_{\times_i.\beta_a} force\ (delay\ t_i)$:

$$\begin{array}{lll}
\mathcal{T}^{-1}\langle\!\langle force\ (proj_i\ (pair\ (delay\ t_1)\ (delay\ t_2)))\rangle\!\rangle & & \\
\quad = & proj_i\ (pair\ \mathcal{T}^{-1}\langle\!\langle t_1\rangle\!\rangle\ \mathcal{T}^{-1}\langle\!\langle t_2\rangle\!\rangle) & \\
\longrightarrow_{\times_i.\beta} & \mathcal{T}^{-1}\langle\!\langle t_i\rangle\!\rangle & \\
\quad = & \mathcal{T}^{-1}\langle\!\langle force\ (delay\ t_i)\rangle\!\rangle &
\end{array}$$

case $case\ (inj_i\ delay\ t)\ of\ (x_1.t_1)\mid(x_2.t_2) \longrightarrow_{+_i.\beta_a} t_i[x_i := delay\ t]$:

$$\begin{array}{lll}
\mathcal{T}^{-1}\langle\!\langle case\ (inj_i\ delay\ t)\ of\ (x_1.t_1)\mid(x_2.t_2)\rangle\!\rangle & & \\
\quad = & case\ (inj_i\ \mathcal{T}^{-1}\langle\!\langle t\rangle\!\rangle)\ of\ (x_1.\mathcal{T}^{-1}\langle\!\langle t_1\rangle\!\rangle)\mid(x_2.\mathcal{T}^{-1}\langle\!\langle t_2\rangle\!\rangle) & \\
\longrightarrow_{+_i.\beta} & \mathcal{T}^{-1}\langle\!\langle t_i\rangle\!\rangle[x_i := \mathcal{T}^{-1}\langle\!\langle t\rangle\!\rangle] & \\
\quad = & \mathcal{T}^{-1}\langle\!\langle t_i[x_i := delay\ t]\rangle\!\rangle & \\
 & ...by\ extension\ of\ Property\ 3.37\ to\ \Lambda^{\sigma^+}_\tau &
\end{array}$$

The cases for conditionals are straightforward and omitted.

∎

Connecting the thunk-based and the continuation-based simulations for Λ^{σ^+}

The factoring of $\mathcal{C}_n$ by $\mathcal{C}_v$ and $\mathcal{T}$ can be extended to Λ^{σ^+}. Since we do not introduce a continuation-passing version of *proj*, the output of $\mathcal{C}_v \circ \mathcal{T}$ is only $\beta\eta_v$-equivalent to $\mathcal{C}_n$ instead of $\beta_a\eta_v$-equivalent as in the original Theorem 3.4. However, because the arguments of the "offending" β-redexes will always yield values when evaluated, the β-redexes are sound with respect to call-by-value evaluation.

Theorem 3.9 *For all* $e \in Terms[\Lambda^{\sigma^+}]$,

$$\lambda\beta\eta_v \vdash (\mathcal{C}_v \circ \mathcal{T})\langle\!\langle e\rangle\!\rangle = \mathcal{C}_n\langle\!\langle e\rangle\!\rangle$$

Proof: by structural induction over e:

case $e \equiv if\ e_0\ then\ e_1\ else\ e_2$:

$$\begin{array}{lll}
(\mathcal{C}_v \circ \mathcal{T})\langle\!\langle if\ e_0\ then\ e_1\ else\ e_2\rangle\!\rangle & & \\
\quad = & \mathcal{C}_v\langle\!\langle if\ \mathcal{T}\langle\!\langle e_0\rangle\!\rangle\ then\ \mathcal{T}\langle\!\langle e_1\rangle\!\rangle\ else\ \mathcal{T}\langle\!\langle e_2\rangle\!\rangle\rangle\!\rangle & \\
\quad = & \lambda k\,.\,(\mathcal{C}_v \circ \mathcal{T})\langle\!\langle e_0\rangle\!\rangle\ (\lambda y\,.\,(if\ y\ then\ (\mathcal{C}_v \circ \mathcal{T})\langle\!\langle e_1\rangle\!\rangle\ else\ (\mathcal{C}_v \circ \mathcal{T})\langle\!\langle e_2\rangle\!\rangle)\ k) & \\
=_{\beta\eta_v} & \lambda k\,.\,\mathcal{C}_n\langle\!\langle e_0\rangle\!\rangle\ (\lambda y\,.\,(if\ y\ then\ \mathcal{C}_n\langle\!\langle e_1\rangle\!\rangle\ else\ \mathcal{C}_n\langle\!\langle e_2\rangle\!\rangle)\ k) & ...by\ ind.\ hyp. \\
\quad = & \mathcal{C}_n\langle\!\langle if\ e_0\ then\ e_1\ else\ e_2\rangle\!\rangle &
\end{array}$$

case $e \equiv pair\, e_1\, e_2$:

$$
\begin{array}{lll}
(\mathcal{C}_v \circ \mathcal{T})\langle\!|pair\, e_1\, e_2|\!\rangle & & \\
\quad = & \mathcal{C}_v\langle\!|pair\, (delay\, \mathcal{T}\langle\!|e_1|\!\rangle)\, (delay\, \mathcal{T}\langle\!|e_2|\!\rangle)|\!\rangle & \\
\quad = & \lambda k\,.\, k\, (pair\, \mathcal{C}_v\langle delay\, \mathcal{T}\langle\!|e_1|\!\rangle\rangle\, \mathcal{C}_v\langle delay\, \mathcal{T}\langle\!|e_2|\!\rangle\rangle) & \\
\quad = & \lambda k\,.\, k\, (pair\, (\mathcal{C}_v \circ \mathcal{T})\langle\!|e_1|\!\rangle\, (\mathcal{C}_v \circ \mathcal{T})\langle\!|e_2|\!\rangle) & \\
=_{\beta\eta_v} & \lambda k\,.\, k\, (pair\, \mathcal{C}_n\langle\!|e_1|\!\rangle\, \mathcal{C}_n\langle\!|e_2|\!\rangle) & \textit{...by ind. hyp.} \\
\quad = & \mathcal{C}_n\langle\!|pair\, e_1\, e_2|\!\rangle &
\end{array}
$$

case $e \equiv inj_i\, e$:

$$
\begin{array}{lll}
(\mathcal{C}_v \circ \mathcal{T})\langle\!|inj_i\, e|\!\rangle & & \\
\quad = & \mathcal{C}_v\langle\!|inj_i\, (delay\, \mathcal{T}\langle\!|e|\!\rangle)|\!\rangle & \\
\quad = & \lambda k\,.\, k\, (inj_i\, \mathcal{C}_v\langle delay\, \mathcal{T}\langle\!|e|\!\rangle\rangle) & \\
\quad = & \lambda k\,.\, k\, (inj_i\, (\mathcal{C}_v \circ \mathcal{T})\langle\!|e|\!\rangle) & \\
=_{\beta\eta_v} & \lambda k\,.\, k\, (inj_i\, \mathcal{C}_n\langle\!|e|\!\rangle) & \textit{...by ind. hyp.} \\
\quad = & \mathcal{C}_n\langle\!|inj_i\, e|\!\rangle &
\end{array}
$$

case $e \equiv proj_i\, e$:

$$
\begin{array}{lll}
(\mathcal{C}_v \circ \mathcal{T})\langle\!|proj_i\, e|\!\rangle & & \\
\quad = & \mathcal{C}_v\langle\!|force\, (proj_i\, \mathcal{T}\langle\!|e|\!\rangle)|\!\rangle & \\
\quad = & \lambda k\,.\, \mathcal{C}_v\langle\!|proj_i\, \mathcal{T}\langle\!|e|\!\rangle|\!\rangle\, (\lambda y_0\,.\, y_0\, k) & \\
\quad = & \lambda k\,.\, (\lambda k\,.\, (\mathcal{C}_v \circ \mathcal{T})\langle\!|e|\!\rangle\, (\lambda y_1\,.\, k\, (proj_i\, y_1)))\, (\lambda y_0\,.\, y_0\, k) & \\
\longrightarrow_{\beta_a} & \lambda k\,.\, (\mathcal{C}_v \circ \mathcal{T})\langle\!|e|\!\rangle\, (\lambda y_1\,.\, (\lambda y_0\,.\, y_0\, k)\, (proj_i\, y_1)) & \\
\longrightarrow_{\beta} & \lambda k\,.\, (\mathcal{C}_v \circ \mathcal{T})\langle\!|e|\!\rangle\, (\lambda y_1\,.\, (proj_i\, y_1)\, k) & \textit{...this step is not a valid } \beta_v\textit{-reduction} \\
=_{\beta\eta_v} & \lambda k\,.\, \mathcal{C}_n\langle\!|e|\!\rangle\, (\lambda y_1\,.\, (proj_i\, y_1)\, k) & \\
\quad = & \mathcal{C}_n\langle\!|proj_i\, e|\!\rangle &
\end{array}
$$

case $e \equiv case\, e\, of\, (x_1.e_1) \mid (x_2.e_2)$:

$$
\begin{array}{lll}
(\mathcal{C}_v \circ \mathcal{T})\langle\!|case\, e\, of\, (x_1.e_1) \mid (x_2.e_2)|\!\rangle & & \\
\quad = & \mathcal{C}_v\langle\!|case\, \mathcal{T}\langle\!|e|\!\rangle\, of\, (x_1.\mathcal{T}\langle\!|e_1|\!\rangle) \mid (x_2.\mathcal{T}\langle\!|e_2|\!\rangle)|\!\rangle & \\
\quad = & \lambda k\,.\, (\mathcal{C}_v \circ \mathcal{T})\langle\!|e|\!\rangle\, (\lambda y\,.\, (case\, y\, of\, (x_1.(\mathcal{C}_v \circ \mathcal{T})\langle\!|e_1|\!\rangle) \mid (x_2.(\mathcal{C}_v \circ \mathcal{T})\langle\!|e_2|\!\rangle))\, k) & \\
=_{\beta\eta_v} & \lambda k\,.\, \mathcal{C}_n\langle\!|e|\!\rangle\, (\lambda y\,.\, (case\, y\, of\, (x_1.\mathcal{C}_n\langle\!|e_1|\!\rangle) \mid (x_2.\mathcal{C}_n\langle\!|e_2|\!\rangle))\, k) & \\
\quad = & \mathcal{C}_n\langle\!|case\, e\, of\, (x_1.e_1) \mid (x_2.e_2)|\!\rangle &
\end{array}
$$

∎

3.8.8 Optimized thunk introduction $\mathcal{T}_{opt}$

To establish properties associated with $\mathcal{T}_{opt}$, it is convenient to restate the definition of $\mathcal{T}_{opt}$ given in Section 3.6, Figure 3.9. Figure 3.11 states $\mathcal{T}_{opt}$ *via* an alternate transformation $\mathcal{S}_{opt}$ which controls the suspension of function arguments. The crux of the optimization is captured by $\mathcal{S}_{opt}\langle\!|x|\!\rangle$ which avoids suspending (delaying) the suspension that will be bound to x.

The grammar below describes terms in the image of $\mathcal{T}_{opt}$ (and $\mathcal{S}_{opt}\langle\!|\Lambda|\!\rangle$).

$$
\begin{array}{lcl}
t & \in & Terms[\mathcal{T}_{opt}\langle\!|\Lambda|\!\rangle] \\
t & ::= & c \mid force\, x \mid \lambda x\,.\, t \mid t\, s \\
 & & \\
s & \in & Terms[\mathcal{S}_{opt}\langle\!|\Lambda|\!\rangle] \\
s & ::= & delay\, c \mid x \mid delay\, \lambda x\,.\, t \mid delay\, (t\, s)
\end{array}
$$

Note that all terms $t \in Terms[\mathcal{T}_{opt}\langle\!|\Lambda|\!\rangle]$ are in $\tau\tau_\eta$-normal form.

$\mathcal{T}_{opt}$ produces terms identical to $\mathcal{T}$ except $\mathcal{T}\langle\!|e\, x|\!\rangle = \mathcal{T}\langle\!|e|\!\rangle\, (delay\, (force\, x))$ whereas $\mathcal{T}_{opt}\langle\!|e\, x|\!\rangle = \mathcal{T}_{opt}\langle\!|e|\!\rangle\, x$. The following property formalizes the relationship.

$$
\begin{array}{rcl}
\mathcal{T}_{opt} & : & Terms[\Lambda] \rightarrow Terms[\Lambda_\tau] \\
\mathcal{T}_{opt}\langle\![c]\!\rangle & = & c \\
\mathcal{T}_{opt}\langle\![x]\!\rangle & = & force\ x \\
\mathcal{T}_{opt}\langle\![\lambda x\,.\,e]\!\rangle & = & \lambda x\,.\,\mathcal{T}_{opt}\langle\![e]\!\rangle \\
\mathcal{T}_{opt}\langle\![e_0\ e_1]\!\rangle & = & \mathcal{T}_{opt}\langle\![e_0]\!\rangle\ \mathcal{S}_{opt}\langle\![e_1]\!\rangle \\
 & & \\
\mathcal{S}_{opt} & : & Terms[\Lambda] \rightarrow Terms[\Lambda_\tau] \\
\mathcal{S}_{opt}\langle\![c]\!\rangle & = & delay\ \mathcal{T}_{opt}\langle\![c]\!\rangle \\
\mathcal{S}_{opt}\langle\![x]\!\rangle & = & x \\
\mathcal{S}_{opt}\langle\![\lambda x\,.\,e]\!\rangle & = & delay\ \mathcal{T}_{opt}\langle\![\lambda x\,.\,e]\!\rangle \\
\mathcal{S}_{opt}\langle\![e_0\ e_1]\!\rangle & = & delay\ \mathcal{T}_{opt}\langle\![e_0\ e_1]\!\rangle
\end{array}
$$

Figure 3.11: Optimized thunk introduction (restated)

Property 3.45 *For all* $e \in \Lambda$, $\lambda\tau_\eta \vdash \mathcal{T}\langle\![e]\!\rangle \longrightarrow\!\!\!\!\!\rightarrow \mathcal{T}_{opt}\langle\![e]\!\rangle$.

Proof: by induction over the structure of e. ∎

A language closed under reductions

The grammar below describes terms in the image of $\mathcal{T}_{opt}$ (and $\mathcal{S}_{opt}\langle\![\Lambda]\!\rangle$) closed under $\beta_a\tau\tau_\eta$ reduction.

$$
\begin{array}{rcl}
t & \in & Terms[\mathcal{T}_{opt}\langle\![\Lambda]\!\rangle^*] \\
t & ::= & c \mid force\ s \mid \lambda x\,.\,t \mid t\ s \\
 & & \\
s & \in & Terms[\mathcal{S}_{opt}\langle\![\Lambda]\!\rangle^*] \\
s & ::= & x \mid delay\ t
\end{array}
$$

Intuitively, the structure of $\mathcal{T}_{opt}\langle\![\Lambda]\!\rangle^*$ terms follows from $\mathcal{T}_{opt}\langle\![\Lambda]\!\rangle$ terms by noting that

- any suspension s may be substituted for x in $force\ x$; and
- $delay\ ((\lambda x\,.\,t_0)\,s)$ may β_a-reduce to $delay\ t_0[x := s]$.

The formal statement of correctness of $\mathcal{T}_{opt}\langle\![\Lambda]\!\rangle^*$ and proofs are similar to those for $\mathcal{T}\langle\![\Lambda]\!\rangle^*$ given in Section 3.8.2 and the details are omitted. However, one of the required properties is stated below.

Property 3.46 ($\mathcal{T}_{opt}\langle\![\Lambda]\!\rangle^*$,$\mathcal{S}_{opt}\langle\![\Lambda]\!\rangle^*$ closed under reduction)

$$
\begin{array}{llcl}
For\ all\ t \in \mathcal{T}_{opt}\langle\![\Lambda]\!\rangle^* & \lambda\beta_a\tau\tau_\eta \vdash t \longrightarrow t' & \Rightarrow & t' \in \mathcal{T}_{opt}\langle\![\Lambda]\!\rangle^* \\
For\ all\ s \in \mathcal{S}_{opt}\langle\![\Lambda]\!\rangle^* & \lambda\beta_a\tau\tau_\eta \vdash s \longrightarrow s' & \Rightarrow & s' \in \mathcal{S}_{opt}\langle\![\Lambda]\!\rangle^*
\end{array}
$$

Indifference, Simulation, and Translation

The proofs of **Indifference** and **Simulation** for $\mathcal{T}_{opt}$ are straightforward modifications of the proofs for $\mathcal{T}$. One noteworthy point is that the language $\mathcal{T}_{opt}\langle\![\Lambda]\!\rangle^*$ which is closed under arbitrary $\beta_a\tau\tau_\eta$ reductions is more general than the language of terms in the image of $\mathcal{T}_{opt}$ closed under $\longmapsto_v$ evaluation steps. It is the latter we

would be interested in a direct adaptation of the $\mathcal{T}$ **Simulation** proof to $\mathcal{T}_{opt}$. Since the $\longmapsto_v$ rules describe an outermost reduction strategy, no reductions occur in delayed arguments. The language of terms in the image of $\mathcal{T}_{opt}$ closed under $\longmapsto_v$ steps is as follows.

$$\begin{array}{rcl} t & ::= & c \mid \mathit{force}\, s \mid \lambda x\,.\,t \mid t\, s \\ s & ::= & \mathit{delay}\, c \mid x \mid \mathit{delay}\, \lambda x\,.\,t \mid \mathit{delay}\, t\, s \end{array}$$

As with $\mathcal{T}$, the **Translation** property for $\mathcal{T}_{opt}$ follows from an equational correspondence.

Theorem 3.10 (Equational Correspondence for $\mathcal{T}_{opt}$)
For all $e, e_1, e_2 \in \mathit{Terms}[\Lambda]$ and $t, t_1, t_2 \in \mathit{Terms}[\mathcal{T}_{opt}(\![\Lambda]\!)^*]$.

1. $\lambda\beta \vdash e = (\mathcal{T}^{-1} \circ \mathcal{T}_{opt})(\![e]\!)$
2. $\lambda\beta_a\tau\tau_\eta \vdash t = (\mathcal{T}_{opt} \circ \mathcal{T}^{-1})(\![t]\!)$
3. $\lambda\beta \vdash e_1 = e_2$ *iff* $\lambda\beta_a\tau\tau_\eta \vdash \mathcal{T}_{opt}(\![e_1]\!) = \mathcal{T}_{opt}(\![e_2]\!)$
4. $\lambda\beta_a\tau\tau_\eta \vdash t_1 = t_2$ *iff* $\lambda\beta \vdash \mathcal{T}^{-1}(\![t_1]\!) = \mathcal{T}^{-1}(\![t_2]\!)$

We summarize below the steps required for the proof of the theorem. Note that several of the statements are stronger than necessary. The following two properties are sufficient for establishing components 1 and 2 of Theorem 3.10.

Property 3.47 *For all $e \in \Lambda$, $e = (\mathcal{T}^{-1} \circ \mathcal{T}_{opt})(\![e]\!)$*

Proof: by induction over the structure of e. ∎

Property 3.48 *For all $t \in \mathcal{T}_{opt}(\![\Lambda]\!)^*$, $s \in \mathcal{S}_{opt}(\![\Lambda]\!)^*$,*

$$\begin{array}{l} \lambda\tau\tau_\eta \vdash t = (\mathcal{T}_{opt} \circ \mathcal{T}^{-1})(\![t]\!) \\ \lambda\tau\tau_\eta \vdash s = (\mathcal{S}_{opt} \circ \mathcal{T}^{-1})(\![s]\!) \end{array}$$

Proof: by simultaneous induction over the structure of t and s. Case $t \equiv \mathit{force}\,(\mathit{delay}\, t')$ shows $\mathcal{T}^{-1}$ collapses τ-redexes; case $s \equiv \mathit{delay}\,(\mathit{force}\, s)$ shows $\mathcal{T}^{-1}$ collapses τ_η-redexes. ∎

Property 3.49 *For all $e_1, e_2 \in \Lambda$,*

$$\begin{array}{l} \lambda\tau \vdash \mathcal{T}_{opt}(\![e_1]\!)[x := \mathcal{S}_{opt}(\![e_2]\!)] \longrightarrow\!\!\!\!\!\rightarrow \mathcal{T}_{opt}(\![e_1[x := e_2]]\!) \\ \lambda\tau \vdash \mathcal{S}_{opt}(\![e_1]\!)[x := \mathcal{S}_{opt}(\![e_2]\!)] \longrightarrow\!\!\!\!\!\rightarrow \mathcal{S}_{opt}(\![e_1[x := e_2]]\!) \end{array}$$

Proof: by induction over the structure of e_1. Note that τ-equivalence is maintained instead of just $\tau\tau_\eta$-equivalence since the substitution of $\mathcal{S}_{opt}(\![e_2]\!)$ (which is of the form x or $\mathit{delay}\, t$) can only introduce τ-redexes. ∎

Property 3.50 *For all $e_1, e_2 \in \Lambda$, $\lambda\beta \vdash e_1 \longrightarrow e_2 \Rightarrow \lambda\beta_a\tau\tau_\eta \vdash \mathcal{T}_{opt}(\![e_1]\!) \longrightarrow\!\!\!\!\!\rightarrow \mathcal{T}_{opt}(\![e_2]\!)$*

Proof: due to Property 3.18, it is sufficient to show the following.

case $\lambda\beta \vdash (\lambda x\,.\,e_0)\, e_1 \rightsquigarrow e_0[x := e_1]$:

$$\begin{array}{rcl} \mathcal{T}_{opt}(\![(\lambda x\,.\,e_0)\, e_1]\!) & = & \mathcal{T}_{opt}(\![\lambda x\,.\,e_0]\!)\, \mathcal{S}_{opt}(\![e_1]\!) \\ & = & (\lambda x\,.\,\mathcal{T}_{opt}(\![e_0]\!))\, \mathcal{S}_{opt}(\![e_1]\!) \\ & \longrightarrow_{\beta_a} & \mathcal{T}_{opt}(\![e_0]\!)[x := \mathcal{S}_{opt}(\![e_1]\!)] \\ & \longrightarrow\!\!\!\!\!\rightarrow_{\tau} & \mathcal{T}_{opt}(\![e_0[x := e_1]]\!) \end{array}$$

Property 3.51 *For all* $t_1, t_2 \in \mathcal{T}_{opt}\langle\!\langle[\Lambda]\rangle\!\rangle^*$, $\lambda\beta_a\tau\tau_\eta \vdash t_1 \longrightarrow t_2 \Rightarrow \lambda\beta \vdash \mathcal{T}^{-1}\langle\!\langle[t_1]\rangle\!\rangle \longrightarrow \mathcal{T}^{-1}\langle\!\langle[t_2]\rangle\!\rangle$

Proof: due to Property 3.18, it is sufficient to show the following.

case $\lambda\beta_a\tau\tau_\eta \vdash (\lambda x\,.\,t_0)\,s_1 \rightsquigarrow t_0[x := s_1]$:

$$\begin{array}{rcll}
\mathcal{T}^{-1}\langle\!\langle[(\lambda x\,.\,t_0)\,s_1]\rangle\!\rangle & = & \mathcal{T}^{-1}\langle\!\langle[\lambda x\,.\,t_0]\rangle\!\rangle\,\mathcal{T}^{-1}\langle\!\langle[s_1]\rangle\!\rangle & \\
& = & (\lambda x\,.\,\mathcal{T}^{-1}\langle\!\langle[t_0]\rangle\!\rangle)\,\mathcal{T}^{-1}\langle\!\langle[s_1]\rangle\!\rangle & \\
& \longrightarrow_\beta & \mathcal{T}^{-1}\langle\!\langle[t_0]\rangle\!\rangle[x := \mathcal{T}^{-1}\langle\!\langle[s_1]\rangle\!\rangle] & \\
& = & \mathcal{T}^{-1}\langle\!\langle[t_0[x := s_1]]\rangle\!\rangle & \textit{...by Property 3.37}
\end{array}$$

case $\lambda\beta_a\tau\tau_\eta \vdash \mathit{force}\,(\mathit{delay}\,t) \rightsquigarrow t$: immediate, since $\mathcal{T}^{-1}$ removes *force* and *delay*.

case $\lambda\beta_a\tau\tau_\eta \vdash \mathit{delay}\,(\mathit{force}\,s) \rightsquigarrow s$: immediate, since $\mathcal{T}^{-1}$ removes *force* and *delay*.

Connecting the optimized thunk-based and the continuation-based simulations

In this section, we show that the relationship between $\mathcal{C}_v \circ \mathcal{T}$ and $\mathcal{C}_n$ (Theorem 3.4) extends to $\mathcal{T}_{opt}$ as well. That the relationship does hold for $\mathcal{T}_{opt}$ is not immediately obvious. The primary concern is that, in the case of $\mathcal{T}$, *force* was applied to every identifier x and it was application of forced that formed the bridge between the call-by-value and call-by-name treatment of identifiers. However, in the case of $\mathcal{T}_{opt}$, identifiers appearing as arguments are not forced.

Theorem 3.11 *For all* $e \in \mathit{Terms}[\Lambda]$,

$$\begin{array}{lcccl}
\lambda\beta_a\eta_v & \vdash & (\mathcal{C}_v \circ \mathcal{T}_{opt})\langle\!\langle[e]\rangle\!\rangle & = & \mathcal{C}_n\langle\!\langle[e]\rangle\!\rangle \\
\lambda\beta_a\eta_v & \vdash & (\mathcal{C}_v \circ \mathcal{S}_{opt})\langle\!\langle[e]\rangle\!\rangle & = & \lambda k\,.\,k\,\mathcal{C}_n\langle\!\langle[e]\rangle\!\rangle
\end{array}$$

Proof: The proof proceeds by induction over the structure of e. We consider $\mathcal{T}_{opt}$ first.
case $e \equiv c, x, \lambda x\,.\,e'$: identical to correspondings steps in the proof of Theorem 3.4.
case $e \equiv e_0\,e_1$:

$$\begin{array}{rcll}
(\mathcal{C}_v \circ \mathcal{T}_{opt})\langle\!\langle[e_0\,e_1]\rangle\!\rangle & = & \mathcal{C}_v\langle\!\langle[\mathcal{T}_{opt}\langle\!\langle[e_0]\rangle\!\rangle\,\mathcal{S}_{opt}\langle\!\langle[e_1]\rangle\!\rangle]\rangle\!\rangle & \\
& = & \lambda k\,.\,(\mathcal{C}_v \circ \mathcal{T}_{opt})\langle\!\langle[e_0]\rangle\!\rangle\,(\lambda v_0\,.\,(\mathcal{C}_v \circ \mathcal{S}_{opt})\langle\!\langle[e_1]\rangle\!\rangle\,(\lambda v_1\,.\,(v_0\,v_1)\,k)) & \\
& =_{\beta_a\eta_v} & \lambda k\,.\,\mathcal{C}_n\langle\!\langle[e_0]\rangle\!\rangle\,(\lambda v_0\,.\,(\lambda k\,.\,k\,\mathcal{C}_n\langle\!\langle[e_1]\rangle\!\rangle)\,(\lambda v_1\,.\,(v_0\,v_1)\,k)) & \textit{...by ind. hyp.} \\
& \longrightarrow_{\beta_a} & \lambda k\,.\,\mathcal{C}_n\langle\!\langle[e_0]\rangle\!\rangle\,(\lambda v_0\,.\,(\lambda v_1\,.\,(v_0\,v_1)\,k)\,\mathcal{C}_n\langle\!\langle[e_1]\rangle\!\rangle) & \textit{...Property 3.11} \\
& \longrightarrow_{\beta_a} & \lambda k\,.\,\mathcal{C}_n\langle\!\langle[e_0]\rangle\!\rangle\,(\lambda v_0\,.\,(v_0\,\mathcal{C}_n\langle\!\langle[e_1]\rangle\!\rangle)\,k) & \\
& = & \mathcal{C}_n\langle\!\langle[e_0\,e_1]\rangle\!\rangle &
\end{array}$$

Now considering $\mathcal{S}_{opt}$.
case $e \equiv x$:

$$\begin{array}{rl}
(\mathcal{C}_v \circ \mathcal{S}_{opt})\langle\!\langle[x]\rangle\!\rangle = & \mathcal{C}_v\langle\!\langle[x]\rangle\!\rangle \\
= & \lambda k\,.\,k\,x \\
= & \lambda k\,.\,k\,\mathcal{C}_n\langle\!\langle[x]\rangle\!\rangle
\end{array}$$

case $e \not\equiv x$:

$$\begin{array}{rcll}
(\mathcal{C}_v \circ \mathcal{S}_{opt})\langle\!\langle[e]\rangle\!\rangle & = & \mathcal{C}_v\langle\!\langle[\mathit{delay}\,\mathcal{T}_{opt}\langle\!\langle[e]\rangle\!\rangle]\rangle\!\rangle & \\
& = & \lambda k\,.\,k\,(\mathcal{C}_v \circ \mathcal{T}_{opt})\langle\!\langle[e]\rangle\!\rangle & \\
& =_{\beta_a\eta_v} & \lambda k\,.\,k\,\mathcal{C}_n\langle\!\langle[e]\rangle\!\rangle & \textit{...by ind. hyp.}
\end{array}$$

$$
\begin{array}{rcl}
\mathcal{T}_{\mathcal{I}} & : & \mathit{Terms}[\Lambda] \rightarrow \mathit{Terms}[\Lambda] \\
\mathcal{T}_{\mathcal{I}}\langle\!|c|\!\rangle & = & c \\
\mathcal{T}_{\mathcal{I}}\langle\!|x|\!\rangle & = & x\, c \\
\mathcal{T}_{\mathcal{I}}\langle\!|\lambda x\,.\,e|\!\rangle & = & \lambda x\,.\,\mathcal{T}_{\mathcal{I}}\langle\!|e|\!\rangle \\
\mathcal{T}_{\mathcal{I}}\langle\!|e_0\, e_1|\!\rangle & = & \mathcal{T}_{\mathcal{I}}\langle\!|e_0|\!\rangle\,(\lambda z\,.\,\mathcal{T}_{\mathcal{I}}\langle\!|e_1|\!\rangle) \\
 & & \ldots \mathit{where}\ z \notin FV(e_1)
\end{array}
$$

Figure 3.12: Thunk introduction implemented in Λ

$$
\begin{array}{rcl}
\mathcal{T}_{\mathcal{I}}^{-1} & : & \mathit{Terms}[\mathcal{T}_{\mathcal{I}}\langle\!|\Lambda|\!\rangle^*] \rightarrow \mathit{Terms}[\Lambda] \\
\mathcal{T}_{\mathcal{I}}^{-1}\langle\!|c|\!\rangle & = & c \\
\mathcal{T}_{\mathcal{I}}^{-1}\langle\!|x\, c|\!\rangle & = & \mathit{force}\, x \\
\mathcal{T}_{\mathcal{I}}^{-1}\langle\!|(\lambda z\,.\,t)\, c|\!\rangle & = & \mathcal{T}_{\mathcal{I}}^{-1}\langle\!|t|\!\rangle \\
\mathcal{T}_{\mathcal{I}}^{-1}\langle\!|\lambda x\,.\,t|\!\rangle & = & \lambda x\,.\,\mathcal{T}_{\mathcal{I}}\langle\!|t|\!\rangle \\
\mathcal{T}_{\mathcal{I}}^{-1} t_0\,(\lambda z\,.\,t_1) & = & \mathcal{T}_{\mathcal{I}}^{-1}\langle\!|t_0|\!\rangle\, \mathcal{T}_{\mathcal{I}}^{-1}\langle\!|t_1|\!\rangle
\end{array}
$$

Figure 3.13: Thunk elimination for $\mathcal{T}_{\mathcal{I}}$

3.8.9 Thunks implemented in Λ

Figure 3.12 gives the thunk introduction transformation $\mathcal{T}_{\mathcal{I}}$ which implements thunks in the core language Λ. The following grammar describes terms in the image of $\mathcal{T}_{\mathcal{I}}$.

$$
\begin{array}{rcl}
t & \in & \mathit{Terms}[\mathcal{T}_{\mathcal{I}}\langle\!|\Lambda|\!\rangle] \\
t & ::= & c \;\mid\; x\, c \;\mid\; \lambda x\,.\,t \;\mid\; t_0\,(\lambda z\,.\,t_1)
\end{array}
$$

Terms in the image of $\mathcal{T}_{\mathcal{I}}$ closed under β_a reduction are as follows.

$$
\begin{array}{rcl}
t & \in & \mathit{Terms}[\mathcal{T}_{\mathcal{I}}\langle\!|\Lambda|\!\rangle^*] \\
t & ::= & c \;\mid\; x\, c \;\mid\; (\lambda z\,.\,t)\, c \;\mid\; \lambda x\,.\,t \;\mid\; t_0\,(\lambda z\,.\,t_1)
\end{array}
$$

The thunk elimination transformation $\mathcal{T}^{-1}$ of Figure 3.3 is defined on the language Λ_τ. However, the analogue of $\mathcal{T}^{-1}$ for $\mathcal{T}_{\mathcal{I}}$ cannot be defined on Λ (the codomain of $\mathcal{T}_{\mathcal{I}}$), but instead must be restricted to $\mathcal{T}_{\mathcal{I}}\langle\!|\Lambda|\!\rangle^*$ (the image of $\mathcal{T}_{\mathcal{I}}$ closed under reductions) since it is impossible to distinguish suspensions (implemented as abstractions) from conventional abstractions (unless some type of marking scheme is employed). Figure 3.13 defines the thunk elimination $\mathcal{T}_{\mathcal{I}}^{-1}$ corresponding to $\mathcal{T}_{\mathcal{I}}$.

The proof of **Indifference** for $\mathcal{T}_{\mathcal{I}}$ is trivial since all function arguments are values. The proof of **Simulation** can be carried out using the same techniques as for $\mathcal{T}$. The statement of equational correspondence for $\mathcal{T}_{\mathcal{I}}$ is as follows.

Theorem 3.12 (Equational Correspondence for $\mathcal{T}_{\mathcal{I}}$)
For all $e, e_1, e_2 \in \mathit{Terms}[\Lambda]$ *and* $t, t_1, t_2 \in \mathit{Terms}[\mathcal{T}_{\mathcal{I}}\langle\!|\Lambda|\!\rangle^*]$,

1. $\lambda\beta \vdash e = (\mathcal{T}_{\mathcal{I}}^{-1} \circ \mathcal{T}_{\mathcal{I}})\langle\!|e|\!\rangle$
2. $\lambda\beta_a \vdash t = (\mathcal{T}_{\mathcal{I}} \circ \mathcal{T}_{\mathcal{I}}^{-1})\langle\!|t|\!\rangle$

3. $\lambda\beta \vdash e_1 = e_2$ *iff* $\lambda\beta_a \vdash \mathcal{T}_{\mathcal{I}}(\![e_1]\!) = \mathcal{T}_{\mathcal{I}}(\![e_2]\!)$

4. $\lambda\beta_a \vdash t_1 = t_2$ *iff* $\lambda\beta \vdash \mathcal{T}_{\mathcal{I}}^{-1}(\![t_1]\!) = \mathcal{T}_{\mathcal{I}}^{-1}(\![t_2]\!)$

The proofs for the equational correspondence mirror those of the equational correspondence for $\mathcal{T}$.

Chapter 4

CPS Transformation after Strictness Analysis

Strictness analysis is a common component of compilers for call-by-name functional languages; the CPS transformation $\mathcal{C}_v$ is a common component of compilers for call-by-value functional languages. To bridge these two implementation techniques, we present a hybrid CPS transformation $\mathcal{C}_s$ for a language with annotations resulting from strictness analysis. $\mathcal{C}_s$ is derived by symbolically composing two transformations $\mathcal{T}_s$ and $\mathcal{C}_v$ and simplifying, *i.e.*,

$$\mathcal{C}_s \stackrel{\text{def}}{=} \mathcal{C}_v \circ \mathcal{T}_s$$

- $\mathcal{T}_s$ transforms a call-by-name program with strictness annotations into a call-by-value program extended with explicit thunk constructs (*i.e.*, *delay* and *force*).
- $\mathcal{C}_v$ is the traditional call-by-value CPS transformation extended with transformations for the thunk constructs *delay* and *force*.
- $\mathcal{C}_s$ generalizes both the call-by-name and the call-by-value CPS transformations in that restricting it to non-strict constructs gives the call-by-name CPS transformation and restricting it to strict constructs gives the call-by-value CPS transformation.

$\mathcal{C}_s$ enables a new strategy for compiling call-by-name programs combining the traditional advantages of CPS (tail-recursive code, evaluation-order independence) and the usual benefits of strictness analysis (elimination of unnecessary thunks). Finally, we express $\mathcal{C}_s$ in Nielson and Nielson's two-level λ-calculus, enabling a simple and efficient implementation in a functional programming language.

4.1 Introduction

4.1.1 Implementing call-by-name programs

In a call-by-name functional program, parameter passing obeys the copy rule. However, the copy rule is rarely used to implement call-by-name, for efficiency reasons. Instead, since Algol 60 [43], call-by-name parameter passing is implemented by passing thunks using call-by-value. The use of thunks is an improvement over the copy rule, but thunks need to be created for each procedure call and activated for each identifier lookup. This overhead can be reduced using strictness analysis, which tells one where call-by-value binding of thunks can be safely changed to call-by-value binding of values [74].

A call-by-name construct is declared "strict" if a diverging actual parameter implies that the whole construct will diverge. In such a case, changing from call-by-name to call-by-value binding does not change the meaning of the construct. A strictness analyzer establishes properties about the strictness of program constructs. This information is approximate but safe in the sense that if a construct cannot be guaranteed to be strict, it is classified as non-strict. The safety of the analysis ensures that call-by-name binding can be changed to call-by-value binding without altering the meaning of the program.

4.1.2 Implementing call-by-value programs

CPS is commonly used to implement call-by-value functional programs [3, 91]. CPS offers practical benefits [3, pp. 4–6]. For example, the control flow of the evaluation strategy is explicitly encoded and the code is tail-recursive.

However, call-by-name functional programs are not usually implemented with the call-by-name CPS transformation. Also, we do not know of any strictness analysis applied after CPS transformation (a moot point, since CPS terms are evaluation-order independent).

4.1.3 Towards a mixed implementation

A program with strictness annotations specifies both call-by-name and call-by-value parameter passing. In the previous chapter, we showed that the call-by-name CPS transformation $\mathcal{C}_n$ can be factored into two transformations: (1) the introduction of thunks $\mathcal{T}$; and (2) the call-by-value CPS transformation $\mathcal{C}_v$. In this chapter, we use strictness information to decide where it is possible to leave out thunks and we derive a new CPS transformation for strictness-annotated programs.

Specifically, we first encode a call-by-name program with strictness annotations into a call-by-value program with explicit operations over thunks. Then we transform this call-by-value program into CPS. The main result of this chapter is the direct mapping of a program with strictness annotations into a CPS program without strictness annotations. This mapping is obtained by symbolically composing the encoding of strictness properties using thunks with call-by-value CPS transformation, and then simplifying the result of the symbolic composition.

This new CPS transformation enables a new strategy for compiling call-by-name programs, combining the traditional advantages of CPS (tail-recursive code, evaluation-order independence) and the usual benefits of strictness analysis (elimination of unnecessary thunks). This transformation should prove useful in several areas.

- A CPS-based compiler can process the output of any CPS transformation. Yet existing CPS-based compilers, *e.g.* Appel's Standard ML compiler [3] and Steele's Scheme compiler [91] process call-by-value programs. Our new CPS transformation enables one to use an existing CPS-based compiler to compile call-by-name programs, including optimizations enabled by strictness analysis.

- Programs can also be compiled using program-transformation techniques. This approach is used by Kesley and Hudak [47] and by Fradet and Le Métayer [33]. Both include a CPS transformation. Fradet and Le Métayer compile both call-by-name and call-by-value programs by using the call-by-name and the call-by-value CPS transformations. Recently, Burn and Le Métayer have combined this technique with a global program-analysis [10], which is comparable to our goal here.

4.1.4 Overview

Section 4.2 presents the syntax of the strictness-annotated language. Section 4.3 defines a transformation which encodes strictness annotations in a call-by-value language with thunks. Section 4.4 formally derives the CPS transformation for programs with strictness annotations. Section 4.5 presents an optimized CPS transformation that eliminates administrative overhead. Finally, Section 4.6 concludes and puts this work into perspective.

4.2 A strictness annotated language Λ_s

We assume the existence of a strictness analyzer $\mathcal{S} : \mathit{Terms}[\Lambda^{\sigma^+}] \rightarrow \mathit{Terms}[\Lambda_s]$ which maps λ-terms to λ-terms annotated with strictness information. The version of Λ^{σ^+} considered here includes conditionals and pairing. To give clarity to annotations, we define $\mathit{fst} \stackrel{\text{def}}{=} \mathit{proj}_1, \mathit{snd} \stackrel{\text{def}}{=} \mathit{proj}_2$, and write applications as $@\, e_1\, e_2$.

Figure 4.1 presents the syntax of the language Λ_s — the output of $\mathcal{S}$. Strict constructs are annotated with s; non-strict constructs are unannotated.

- Annotations of identifiers indicate that they are declared as formal parameters of strict or of non-strict abstractions.

$$
\begin{array}{lcl}
e & \in & \mathit{Terms}[\Lambda_s] \\
e & ::= & c \mid x_s \mid x \mid f \mid \mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2 \mid \\
 & & \lambda x_s\,.\,e \mid \lambda x\,.\,e \mid \mathit{rec}\, f(x_s).e \mid \mathit{rec}\, f(x).e \mid @_s\, e_0\, e_1 \mid @\, e_0\, e_1 \mid \\
 & & \mathit{pair}_{ss}\, e_0\, e_1 \mid \mathit{pair}_{s\bullet}\, e_0\, e_1 \mid \mathit{pair}_{\bullet s}\, e_0\, e_1 \mid \mathit{pair}_{\bullet\bullet}\, e_0\, e_1 \mid \\
 & & \mathit{fst}_s\, e \mid \mathit{fst}\, e \mid \mathit{snd}_s\, e \mid \mathit{snd}\, e \\
 & & \\
\sigma & \in & \mathit{Types}[\Lambda_s] \\
\sigma & ::= & \iota \mid \sigma_1 \xrightarrow{s} \sigma_2 \mid \sigma_1 \rightarrow \sigma_2 \mid \sigma_1 \overset{ss}{\times} \sigma_2 \mid \sigma_1 \overset{s\bullet}{\times} \sigma_2 \mid \sigma_1 \overset{\bullet s}{\times} \sigma_2 \mid \sigma_1 \overset{\bullet\bullet}{\times} \sigma_2 \\
 & & \\
\Gamma & \in & \mathit{Assums}[\Lambda_s] \\
\Gamma & ::= & \cdot \mid \Gamma, x_s{:}\sigma \mid \Gamma, x{:}\sigma \mid \Gamma, f{:}\sigma
\end{array}
$$

Figure 4.1: The strictness annotated language Λ_s

- Abstractions may be strict or non-strict depending on whether or not the functions they represent are strict or non-strict, respectively.
- Strict (resp. non-strict) applications denote the application of strict (resp. non-strict) abstractions.
- Pairing may be strict or non-strict in either argument. The constructor pair_{ss} is strict in both arguments. $\mathit{pair}_{\bullet\bullet}$ is non-strict in both arguments. $\mathit{pair}_{s\bullet}$ is strict on its first argument and non-strict on its second. $\mathit{pair}_{\bullet s}$ is non-strict in its first argument and strict in its second.
- The projections *fst* and *snd* are strict constructs. Their argument must denote a pair. They are annotated as strict (fst_s , snd_s) or non-strict (*fst* , *snd*) depending on whether the corresponding pair constructor was strict or non-strict in the first or second component.

We assume that the strictness analyzer guarantees the consistency (*i.e.*, the well-formedness) of the annotations. Consistency is specified using the type assignment rules of Figure 4.2.

Property 4.1 (Consistency of strictness analysis $\mathcal{S}$) *If* $\Gamma \vdash e : \sigma$ *then* $\mathcal{S}[\![\Gamma]\!] \vdash_s \mathcal{S}[\![e]\!] : \mathcal{S}[\![\sigma]\!]$.

4.2.1 Values

The set of Λ_s values is defined as follows.

$$
\begin{array}{lcl}
v & \in & \mathit{Values}[\Lambda_s] \\
v & ::= & c \mid x_s \mid f \mid \lambda x_s\,.\,e \mid \lambda x\,.\,e \mid \mathit{rec}\, f(x_s).e \mid \mathit{rec}\, f(x).e \mid \\
 & & \mathit{pair}_{ss}\, v_1\, v_2 \mid \mathit{pair}_{s\bullet}\, v_1\, e_2 \mid \mathit{pair}_{\bullet s}\, e_1\, v_2 \mid \mathit{pair}_{\bullet\bullet}\, e_1\, e_2
\end{array}
$$

$$\ldots \mathit{where}\ e, e_1, e_2 \in \Lambda_s\,.$$

4.2.2 Operational semantics

Figure 4.3 presents single-step evaluation rules which define the operational semantics of Λ_s programs. Intuitively, strict constructs are evaluated eagerly (*i.e.*, using call-by-value) and non-strict constructs are evaluated lazily (*i.e.*, using call-by-name). The (partial) meaning function eval_s is defined in terms of the reflexive, transitive closure (denoted $\longmapsto^*$) of the evaluation rules.

$$\mathit{eval}_s(e) = v \quad \text{iff} \quad e \longmapsto_s^* v$$

$$\Gamma \vdash_s c : \iota \qquad \Gamma \vdash_s x_s : \Gamma(x_s) \qquad \Gamma \vdash_s x : \Gamma(x) \qquad \Gamma \vdash_s f : \Gamma(f)$$

$$\frac{\Gamma, x_s : \sigma_1 \vdash_s e : \sigma_2}{\Gamma \vdash_s \lambda x_s \,.\, e : \sigma_1 \overset{s}{\rightarrow} \sigma_2} \qquad \frac{\Gamma, x : \sigma_1 \vdash_s e : \sigma_2}{\Gamma \vdash_s \lambda x \,.\, e : \sigma_1 \rightarrow \sigma_2}$$

$$\frac{\Gamma, f : \sigma_1 \rightarrow \sigma_2, x_s : \sigma_1 \vdash_s e : \sigma_2}{\Gamma \vdash_s rec\, f(x_s).\, e : \sigma_1 \rightarrow \sigma_2} \qquad \frac{\Gamma, f : \sigma_1 \rightarrow \sigma_2, x : \sigma_1 \vdash_s e : \sigma_2}{\Gamma \vdash_s rec\, f(x).\, e : \sigma_1 \rightarrow \sigma_2}$$

$$\frac{\Gamma \vdash_s e_1 : \iota \qquad \Gamma \vdash_s e_2 : \sigma \qquad \Gamma \vdash_s e_3 : \sigma}{\Gamma \vdash_s \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 : \sigma}$$

$$\frac{\Gamma \vdash_s e_0 : \sigma_1 \overset{s}{\rightarrow} \sigma_2 \qquad \Gamma \vdash_s e_1 : \sigma_1}{\Gamma \vdash_s @_s\, e_0\, e_1 : \sigma_2} \qquad \frac{\Gamma \vdash_s e_0 : \sigma_1 \rightarrow \sigma_2 \qquad \Gamma \vdash_s e_1 : \sigma_1}{\Gamma \vdash_s @\, e_0\, e_1 : \sigma_2}$$

$$\frac{\Gamma \vdash_s e_0 : \sigma_0 \qquad \Gamma \vdash_s e_1 : \sigma_1}{\Gamma \vdash_s \mathit{pair}_{ss}\, e_0\, e_1 : \sigma_0 \overset{ss}{\times} \sigma_1} \qquad \frac{\Gamma \vdash_s e_0 : \sigma_0 \qquad \Gamma \vdash_s e_1 : \sigma_1}{\Gamma \vdash_s \mathit{pair}_{s\bullet}\, e_0\, e_1 : \sigma_0 \overset{s\bullet}{\times} \sigma_1}$$

$$\frac{\Gamma \vdash_s e_0 : \sigma_0 \qquad \Gamma \vdash_s e_1 : \sigma_1}{\Gamma \vdash_s \mathit{pair}_{\bullet s}\, e_0\, e_1 : \sigma_0 \overset{\bullet s}{\times} \sigma_1} \qquad \frac{\Gamma \vdash_s e_0 : \sigma_0 \qquad \Gamma \vdash_s e_1 : \sigma_1}{\Gamma \vdash_s \mathit{pair}_{\bullet\bullet}\, e_0\, e_1 : \sigma_0 \overset{\bullet\bullet}{\times} \sigma_1}$$

$$\frac{\Gamma \vdash_s e : \sigma_0 \overset{ss}{\times} \sigma_1}{\Gamma \vdash_s \mathit{fst}_s\, e : \sigma_0} \qquad \frac{\Gamma \vdash_s e : \sigma_0 \overset{s\bullet}{\times} \sigma_1}{\Gamma \vdash_s \mathit{fst}_s\, e : \sigma_0} \qquad \frac{\Gamma \vdash_s e : \sigma_0 \overset{\bullet s}{\times} \sigma_1}{\Gamma \vdash_s \mathit{fst}\, e : \sigma_0} \qquad \frac{\Gamma \vdash_s e : \sigma_0 \overset{\bullet\bullet}{\times} \sigma_1}{\Gamma \vdash_s \mathit{fst}\, e : \sigma_0}$$

$$\frac{\Gamma \vdash_s e : \sigma_0 \overset{ss}{\times} \sigma_1}{\Gamma \vdash_s \mathit{snd}_s\, e : \sigma_1} \qquad \frac{\Gamma \vdash_s e : \sigma_0 \overset{s\bullet}{\times} \sigma_1}{\Gamma \vdash_s \mathit{snd}\, e : \sigma_1} \qquad \frac{\Gamma \vdash_s e : \sigma_0 \overset{\bullet s}{\times} \sigma_1}{\Gamma \vdash_s \mathit{snd}_s\, e : \sigma_1} \qquad \frac{\Gamma \vdash_s e : \sigma_0 \overset{\bullet\bullet}{\times} \sigma_1}{\Gamma \vdash_s \mathit{snd}\, e : \sigma_1}$$

Figure 4.2: Type assignment rules for the strictness annotated language Λ_s

$$@\,(\lambda x\,.\,e_1)\,e_2 \longmapsto_s e_1[x := e_2] \qquad @\,(rec\,f(x).\,e_1)\,e_2 \longmapsto_s e_1[f := rec\,f(x).\,e_1\,,\,x := e_2]$$

$$@_s\,(\lambda x_s\,.\,e_1)\,v_2 \longmapsto_s e_1[x := v_2] \qquad @_s\,(rec\,f(x_s).\,e_1)\,v_2 \longmapsto_s e_1[f := rec\,f(x_s).\,e_1\,,\,x := v_2]$$

$$\frac{e_1 \longmapsto_s e_1'}{@\,e_1\,e_2 \longmapsto_s @\,e_1'\,e_2} \qquad \frac{e_1 \longmapsto_s e_1'}{@_s\,e_1\,e_2 \longmapsto_s @_s\,e_1'\,e_2}$$

$$\frac{e_2 \longmapsto_s e_2'}{@_s\,(\lambda x_s\,.\,e_1)\,e_2 \longmapsto_s @_s\,(\lambda x_s\,.\,e_1)\,e_2'} \qquad \frac{e_2 \longmapsto_s e_2'}{@_s\,(rec\,f(x_s).\,e_1)\,e_2 \longmapsto_s @_s\,(rec\,f(x_s).\,e_1)\,e_2'}$$

$$\frac{e_0 \longmapsto_s e_0'}{if\ e_0\ then\ e_1\ else\ e_2 \longmapsto_s if\ e_0\ then\ e_1\ else\ e_2}$$

$$if\ \mathsf{c}_t\ then\ e_1\ else\ e_2 \longmapsto_s e_1 \qquad if\ \mathsf{c}_f\ then\ e_1\ else\ e_2 \longmapsto_s e_2$$

$$\frac{e_1 \longmapsto_s e_1'}{pair_{ss}\,e_1\,e_2 \longmapsto_s pair_{ss}\,e_1'\,e_2} \qquad \frac{e_2 \longmapsto_s e_2'}{pair_{ss}\,v_1\,e_2 \longmapsto_s pair_{ss}\,v_1\,e_2'}$$

$$\frac{e_1 \longmapsto_s e_1'}{pair_{s\bullet}\,e_1\,e_2 \longmapsto_s pair_{s\bullet}\,e_1'\,e_2} \qquad \frac{e_2 \longmapsto_s e_2'}{pair_{\bullet s}\,e_1\,e_2 \longmapsto_s pair_{\bullet s}\,e_1\,e_2'}$$

$$\frac{e \longmapsto_s e'}{fst_s\,e \longmapsto_s fst_s\,e'} \qquad \frac{e \longmapsto_s e'}{fst\,e \longmapsto_s fst\,e'}$$

$$fst_s\,(pair_{ss}\,v_1\,v_2) \longmapsto_s v_1 \qquad fst_s\,(pair_{s\bullet}\,v_1\,e_2) \longmapsto_s v_1$$

$$fst\,(pair_{\bullet s}\,e_1\,v_2) \longmapsto_s e_1 \qquad fst\,(pair_{\bullet\bullet}\,e_1\,e_2) \longmapsto_s e_1$$

$$\frac{e \longmapsto_s e'}{snd_s\,e \longmapsto_s snd_s\,e'} \qquad \frac{e \longmapsto_s e'}{snd\,e \longmapsto_s snd\,e'}$$

$$snd_s\,(pair_{ss}\,v_1\,v_2) \longmapsto_s v_2 \qquad snd_s\,(pair_{s\bullet}\,e_1\,v_2) \longmapsto_s v_2$$

$$snd\,(pair_{\bullet s}\,v_1\,e_2) \longmapsto_s e_2 \qquad snd\,(pair_{\bullet\bullet}\,e_1\,e_2) \longmapsto_s e_2$$

Figure 4.3: Single-step evaluation rules for Λ_s

We follow our convention and only observe termination of non-ground-type programs.

We assume the correctness of the strictness analyzer $\mathcal{S}$, *i.e.*, $\mathcal{S}$ informs us of where it is safe to use eager evaluation in call-by-name programs.

Property 4.2 (Correctness of strictness analysis $\mathcal{S}$)
For all programs $\cdot \vdash e : \sigma$,

$$eval_n(e) \simeq_{obs} eval_s(\mathcal{S}\langle\![e]\!\rangle)$$

4.3 Encoding strictness properties with thunks

Based on the ideas in the previous chapter, we translate the language with strictness annotations to a call-by-value language extended with the thunk constructs *delay* and *force*. Figure 4.4 presents the translation $\mathcal{T}_s$. There is nothing surprising in this translation; all strict constructs are essentially copied while non-strict constructs have their arguments suspended by *delay*. Evaluation of expressions yielding thunks is initiated using *force*.

The following property and theorem capture the correctness of $\mathcal{T}_s$.

Property 4.3 (Type correctness of $\mathcal{T}_s$) *If* $\Gamma \vdash_s e : \sigma$ *then* $\mathcal{T}_s\langle\![\Gamma]\!\rangle \vdash_\tau \mathcal{T}_s\langle\![e]\!\rangle : \mathcal{T}_s\langle\![\sigma]\!\rangle$.

Proof: by induction over the structure of the typing derivation for $\Gamma \vdash_s e : \sigma$. ∎

Theorem 4.1 (Correctness of $\mathcal{T}_s$)
For all programs $\cdot \vdash_s e : \sigma$,

$$eval_s(e) \simeq_{obs} eval_v(\mathcal{T}_s\langle\![e]\!\rangle)$$

Proof: The proof is a tedious but straightforward extension of the proof of **Simulation** for thunks (Theorem 3.1) in the previous chapter. ∎

4.4 CPS transformation of a language with strictness annotations

We now derive a new CPS transformation $\mathcal{C}_s$ by composing $\mathcal{C}_v$ with $\mathcal{T}_s$ symbolically and simplifying the result.

4.4.1 Terms

We first derive the transformation on terms. The derivations will apply Property 3.11 which states

$$\lambda\beta_a \vdash \mathcal{C}_v\langle\![delay\, e_1]\!\rangle\,(\lambda y\,.\,e_2) \longrightarrow (\lambda y\,.\,e_2)\,\mathcal{C}_v\langle\![e_1]\!\rangle.$$

The definition proceeds by induction over the structure of e and four of the more interesting constructs are given below.

case $e \equiv x$:

$$\begin{array}{rl}
(\mathcal{C}_v \circ \mathcal{T}_s)\langle\![x]\!\rangle & \\
= & \mathcal{C}_v\langle\![force\ x]\!\rangle \\
= & \lambda k\,.\,\mathcal{C}_v\langle\![x]\!\rangle\,(\lambda y\,.\,y\,k) \\
\longrightarrow\!\!\!\!\!\rightarrow_{\beta_a} & \lambda k\,.\,x\,k \\
\longrightarrow_{\eta_v} & x \\
\stackrel{\text{def}}{=} & \mathcal{C}_s\langle\![x]\!\rangle
\end{array}$$

case $e \equiv @\,e_0\,e_1$:

$$
\begin{array}{rcl}
\mathcal{T}_s & : & Terms[\Lambda_s] \rightarrow Terms[\Lambda_\tau^{\sigma^+}] \\
\mathcal{T}_s\langle\![c]\!\rangle & = & c \\
\mathcal{T}_s\langle\![x_s]\!\rangle & = & x \\
\mathcal{T}_s\langle\![\lambda x_s\,.\,e]\!\rangle & = & \lambda x\,.\,\mathcal{T}_s\langle\![e]\!\rangle \\
\mathcal{T}_s\langle\![rec\,f(x_s).\,e]\!\rangle & = & rec\,f(x).\,\mathcal{T}_s\langle\![e]\!\rangle \\
\mathcal{T}_s\langle\![@_s\,e_0\,e_1]\!\rangle & = & \mathcal{T}_s\langle\![e_0]\!\rangle\,\mathcal{T}_s\langle\![e_1]\!\rangle \\
\mathcal{T}_s\langle\![pair_{ss}\,e_0\,e_1]\!\rangle & = & pair\,\mathcal{T}_s\langle\![e_0]\!\rangle \\
 & & \quad \mathcal{T}_s\langle\![e_1]\!\rangle \\
\mathcal{T}_s\langle\![pair_{s\bullet}\,e_0\,e_1]\!\rangle & = & pair\,\mathcal{T}_s\langle\![e_0]\!\rangle \\
 & & \quad (delay\,\mathcal{T}_s\langle\![e_1]\!\rangle) \\
\mathcal{T}_s\langle\![fst_s\,e]\!\rangle & = & fst\,\mathcal{T}_s\langle\![e]\!\rangle \\
\mathcal{T}_s\langle\![snd_s\,e]\!\rangle & = & snd\,\mathcal{T}_s\langle\![e]\!\rangle
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{T}_s\langle\![f]\!\rangle & = & f \\
\mathcal{T}_s\langle\![x]\!\rangle & = & force\,x \\
\mathcal{T}_s\langle\![\lambda x\,.\,e]\!\rangle & = & \lambda x\,.\,\mathcal{T}_s\langle\![e]\!\rangle \\
\mathcal{T}_s\langle\![rec\,f(x).\,e]\!\rangle & = & rec\,f(x).\,\mathcal{T}_s\langle\![e]\!\rangle \\
\mathcal{T}_s\langle\![@\,e_0\,e_1]\!\rangle & = & \mathcal{T}_s\langle\![e_0]\!\rangle\,(delay\,\mathcal{T}_s\langle\![e_1]\!\rangle) \\
\mathcal{T}_s\langle\![pair_{\bullet\bullet}\,e_0\,e_1]\!\rangle & = & pair\,(delay\,\mathcal{T}_s\langle\![e_0]\!\rangle) \\
 & & \quad (delay\,\mathcal{T}_s\langle\![e_1]\!\rangle) \\
\mathcal{T}_s\langle\![pair_{\bullet s}\,e_0\,e_1]\!\rangle & = & pair\,(delay\,\mathcal{T}_s\langle\![e_0]\!\rangle) \\
 & & \quad \mathcal{T}_s\langle\![e_1]\!\rangle \\
\mathcal{T}_s\langle\![fst\,e]\!\rangle & = & force\,(fst\,\mathcal{T}_s\langle\![e]\!\rangle) \\
\mathcal{T}_s\langle\![snd\,e]\!\rangle & = & force\,(snd\,\mathcal{T}_s\langle\![e]\!\rangle)
\end{array}
$$

$$
\mathcal{T}_s\langle\![if\ e_0\ then\ e_1\ else\ e_2]\!\rangle \quad = \quad if\ \mathcal{T}_s\langle\![e_0]\!\rangle\ then\ \mathcal{T}_s\langle\![e_1]\!\rangle\ else\ \mathcal{T}_s\langle\![e_2]\!\rangle
$$

$$
\begin{array}{rcl}
\mathcal{T}_s & : & Types[\Lambda_s] \rightarrow Types[\Lambda_\tau^{\sigma^+}] \\
\mathcal{T}_s\langle\![\sigma_1 \stackrel{s}{\rightarrow} \sigma_2]\!\rangle & = & \mathcal{T}_s\langle\![\sigma_1]\!\rangle \rightarrow \mathcal{T}_s\langle\![\sigma_2]\!\rangle \\
\mathcal{T}_s\langle\![\sigma_1 \stackrel{ss}{\times} \sigma_2]\!\rangle & = & \mathcal{T}_s\langle\![\sigma_1]\!\rangle \times \mathcal{T}_s\langle\![\sigma_2]\!\rangle \\
\mathcal{T}_s\langle\![\sigma_1 \stackrel{s\bullet}{\times} \sigma_2]\!\rangle & = & \mathcal{T}_s\langle\![\sigma_1]\!\rangle \times \widetilde{\mathcal{T}_s\langle\![\sigma_2]\!\rangle}
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{T}_s\langle\![\iota]\!\rangle & = & \iota \\
\mathcal{T}_s\langle\![\sigma_1 \rightarrow \sigma_2]\!\rangle & = & \widetilde{\mathcal{T}_s\langle\![\sigma_1]\!\rangle} \rightarrow \mathcal{T}_s\langle\![\sigma_2]\!\rangle \\
\mathcal{T}_s\langle\![\sigma_1 \stackrel{\bullet\bullet}{\times} \sigma_2]\!\rangle & = & \widetilde{\mathcal{T}_s\langle\![\sigma_1]\!\rangle} \times \widetilde{\mathcal{T}_s\langle\![\sigma_2]\!\rangle} \\
\mathcal{T}_s\langle\![\sigma_1 \stackrel{\bullet s}{\times} \sigma_2]\!\rangle & = & \widetilde{\mathcal{T}_s\langle\![\sigma_1]\!\rangle} \times \mathcal{T}_s\langle\![\sigma_2]\!\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{T}_s : Assums[\Lambda_s] & \rightarrow & Assums[\Lambda_\tau^{\sigma^+}] \\
\mathcal{T}_s\langle\![\Gamma, x_s : \sigma]\!\rangle & = & \mathcal{T}_s\langle\![\Gamma]\!\rangle, x : \mathcal{T}_s\langle\![\sigma]\!\rangle \\
\mathcal{T}_s\langle\![\Gamma, f : \sigma]\!\rangle & = & \mathcal{T}_s\langle\![\Gamma]\!\rangle, f : \widetilde{\mathcal{T}_s\langle\![\sigma]\!\rangle} \\
\mathcal{T}_s\langle\![\Gamma, x : \sigma]\!\rangle & = & \mathcal{T}_s\langle\![\Gamma]\!\rangle, x : \widetilde{\mathcal{T}_s\langle\![\sigma]\!\rangle}
\end{array}
$$

Figure 4.4: Encoding strictness properties with suspension operators

$$
\begin{array}{ll}
(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|@\, e_0\, e_1|\!\rangle & \\
\quad = & \mathcal{C}_v\langle\!|\mathcal{T}_s\langle\!|e_0|\!\rangle\,(delay\ \mathcal{T}_s\langle\!|e_1|\!\rangle)|\!\rangle \\
\quad = & \lambda k\,.\,(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|e_0|\!\rangle\,(\lambda y_0\,.\,\mathcal{C}_v\langle\!|delay\ \mathcal{T}_s\langle\!|e_1|\!\rangle|\!\rangle\,(\lambda y_1\,.\,(y_0\,y_1)\,k)) \\
\longrightarrow_{\beta_a} & \lambda k\,.\,(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|e_0|\!\rangle\,(\lambda y_0\,.\,(\lambda y_1\,.\,(y_0\,y_1)\,k)\,(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|e_1|\!\rangle) \qquad \textit{...Property 3.11} \\
\longrightarrow_{\beta_a} & \lambda k\,.\,(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|e_0|\!\rangle\,(\lambda y_0\,.\,(y_0\,(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|e_1|\!\rangle)\,k) \\
=_{\beta_a\,\eta_v} & \lambda k\,.\,\mathcal{C}_s\langle\!|e_0|\!\rangle\,(\lambda y_0\,.\,(y_0\,(\mathcal{C}_s\langle\!|e_1|\!\rangle))\,k) \qquad \textit{...by ind. hyp.} \\
\stackrel{\text{def}}{=} & \mathcal{C}_s\langle\!|e_0\, e_1|\!\rangle
\end{array}
$$

case $e \equiv pair_{\bullet\bullet}\, e_0\, e_1$:

$$
\begin{array}{ll}
(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|pair_{\bullet\bullet}\, e_0\, e_1|\!\rangle & \\
\quad = & \mathcal{C}_v\langle\!|pair\,(delay\ \mathcal{T}_s\langle\!|e_0|\!\rangle)\,(delay\ \mathcal{T}_s\langle\!|e_1|\!\rangle)|\!\rangle \\
\quad = & \lambda k\,.\,\mathcal{C}_v\langle\!|delay\ \mathcal{T}_s\langle\!|e_0|\!\rangle|\!\rangle\,(\lambda y_0\,.\,\mathcal{C}_v\langle\!|delay\ \mathcal{T}_s\langle\!|e_1|\!\rangle|\!\rangle\,(\lambda y_1\,.\,k\,(pair\ y_0\ y_1))) \\
\longrightarrow\!\!\!\!\!\rightarrow & \lambda k\,.\,(\lambda y_0\,.\,(\lambda y_1\,.\,k\,(pair\ y_0\ y_1))\,(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|e_1|\!\rangle)\,(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|e_0|\!\rangle \qquad \textit{...Property 3.11 (twice)} \\
\longrightarrow\!\!\!\!\!\rightarrow_{\beta_a} & \lambda k\,.\,k\,(pair\,(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|e_0|\!\rangle\,(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|e_1|\!\rangle) \\
=_{\beta_a\,\eta_v} & \lambda k\,.\,k\,(pair\ \mathcal{C}_s\langle\!|e_0|\!\rangle\ \mathcal{C}_s\langle\!|e_1|\!\rangle) \qquad \textit{...by ind. hyp.} \\
\stackrel{\text{def}}{=} & \mathcal{C}_s\langle\!|pair_{\bullet\bullet}\, e_0\, e_1|\!\rangle
\end{array}
$$

case $e \equiv fst\, e$:

$$
\begin{array}{ll}
(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|fst\, e_0|\!\rangle & \\
\quad = & \mathcal{C}_v\langle\!|force\ (fst\ \mathcal{T}_s\langle\!|e|\!\rangle)|\!\rangle \\
\quad = & \lambda k\,.\,\mathcal{C}_v\langle\!|fst\ \mathcal{T}_s\langle\!|e|\!\rangle|\!\rangle\,(\lambda y\,.\,y\,k) \\
\quad = & \lambda k\,.\,(\lambda k\,.\,(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|e|\!\rangle\,(\lambda y\,.\,k\,(fst\ y)))\,(\lambda y\,.\,y\,k) \\
\longrightarrow_{\beta_a} & \lambda k\,.\,(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|e|\!\rangle\,(\lambda y\,.\,(\lambda y\,.\,y\,k)\,(fst\ y)) \\
\longrightarrow_{\beta_a} & \lambda k\,.\,(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|e|\!\rangle\,(\lambda y\,.\,(fst\ y)\,k) \\
=_{\beta_a\,\eta_v} & \lambda k\,.\,\mathcal{C}_s\langle\!|e|\!\rangle\,(\lambda y\,.\,(fst\ y)\,k) \qquad \textit{...by ind. hyp.} \\
\stackrel{\text{def}}{=} & \mathcal{C}_s\langle\!|fst\, e|\!\rangle
\end{array}
$$

4.4.2 Types

We also obtain $\mathcal{C}_s$ on types by inductively composing $\mathcal{C}_v$ and $\mathcal{T}_s$. The derivations for four constructs are given below.

case $\sigma \equiv \iota$:

$$
\begin{array}{ll}
(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|\iota|\!\rangle & \\
\quad = & \iota \\
\stackrel{\text{def}}{=} & \mathcal{C}_s\langle\iota\rangle
\end{array}
$$

case $\sigma \equiv \sigma_1 \stackrel{s}{\rightarrow} \sigma_2$:

$$
\begin{array}{ll}
(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|\sigma_1 \stackrel{s}{\rightarrow} \sigma_2|\!\rangle & \\
\quad = & \mathcal{C}_v\langle\mathcal{T}_s\langle\!|\sigma_1|\!\rangle \rightarrow \mathcal{T}_s\langle\!|\sigma_2|\!\rangle\rangle \\
\quad = & (\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|\sigma_1|\!\rangle \rightarrow \neg\neg(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|\sigma_2|\!\rangle \\
\quad = & \mathcal{C}_s\langle\sigma_1\rangle \rightarrow \neg\neg\mathcal{C}_s\langle\sigma_2\rangle \qquad \textit{...by ind. hyp.} \\
\stackrel{\text{def}}{=} & \mathcal{C}_s\langle\sigma_1 \stackrel{s}{\rightarrow} \sigma_2\rangle
\end{array}
$$

case $\sigma \equiv \sigma_1 \rightarrow \sigma_2$:

$$
\begin{array}{ll}
(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|\sigma_1 \rightarrow \sigma_2|\!\rangle & \\
\quad = & \mathcal{C}_v\langle\widetilde{\mathcal{T}_s\langle\!|\sigma_1|\!\rangle} \rightarrow \mathcal{T}_s\langle\!|\sigma_2|\!\rangle\rangle \\
\quad = & \neg\neg(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|\sigma_1|\!\rangle \rightarrow \neg\neg(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|\sigma_2|\!\rangle \\
\quad = & \neg\neg\mathcal{C}_s\langle\sigma_1\rangle \rightarrow \neg\neg\mathcal{C}_s\langle\sigma_2\rangle \qquad \textit{...by ind. hyp.} \\
\stackrel{\text{def}}{=} & \mathcal{C}_s\langle\sigma_1 \rightarrow \sigma_2\rangle
\end{array}
$$

case $\sigma \equiv \sigma_1 \overset{s\bullet}{\times} \sigma_2$:

$$
\begin{array}{lll}
(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|\sigma_1 \overset{s\bullet}{\times} \sigma_2|\!\rangle & & \\
\quad = & \mathcal{C}_v\langle \mathcal{T}_s\langle\!|\sigma_1|\!\rangle \times \widetilde{\mathcal{T}_s\langle\!|\sigma_2|\!\rangle}\rangle & \\
\quad = & (\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|\sigma_1|\!\rangle \times \neg\neg(\mathcal{C}_v \circ \mathcal{T}_s)\langle\!|\sigma_2|\!\rangle & \\
\quad = & \mathcal{C}_s\langle\sigma_1\rangle \times \neg\neg\mathcal{C}_s\langle\sigma_2\rangle & \textit{...by ind. hyp.} \\
\quad \overset{\text{def}}{=} & \mathcal{C}_s\langle\sigma_1 \overset{s\bullet}{\times} \sigma_2\rangle &
\end{array}
$$

4.4.3 Conclusion

The full transformation $\mathcal{C}_s$ is given in Figure 4.5. A careful examination of the definition reveals that elements of both $\mathcal{C}_n$ and $\mathcal{C}_v$ are captured in $\mathcal{C}_s$. Essentially, the transformation of a non-strict construct follows the pattern of the call-by-name CPS transformation (*cf.* Figure 2.9), and the transformation of a strict construct follows the pattern of the call-by-value CPS transformation (*cf.* Figure 2.10). The same observation applies for strict pairs and for non-strict pairs.
Let us analyze two extreme cases.

- If the strictness analysis determines that *all* program constructs are strict, then no thunk is introduced and $\mathcal{C}_s$ can be simplified into $\mathcal{C}_v$.
- If the strictness analysis determines that *no* program construct is strict, then thunks are introduced everywhere and $\mathcal{C}_s$ can be simplified into $\mathcal{C}_n$.

Thus, $\mathcal{C}_s$ generalizes both $\mathcal{C}_v$ and $\mathcal{C}_n$.

The second extreme case coincides with the result of the previous chapter: The call-by-name CPS transformation can be factored into (1) introduction of thunks $\mathcal{T}$ and (2) call-by-value CPS transformation.

The following property and theorem capture the correctness of $\mathcal{C}_s$.

Property 4.4 (Type correctness of $\mathcal{C}_s$) *If* $\Gamma \vdash_s e : \sigma$ *then* $\mathcal{C}_s\langle\!|\Gamma|\!\rangle \vdash \mathcal{C}_s\langle\!|e|\!\rangle : \mathcal{C}_s\langle\!|\sigma|\!\rangle$.

Proof: Follows from type correctness of $\mathcal{T}_s$ (Property 4.3) and type correctness of $\mathcal{C}_v$ (Property 2.8, Chapter 2). ∎

Theorem 4.2 (Correctness of $\mathcal{C}_s$)
For all programs $\cdot \vdash_s e : \sigma$,

1. **Indifference:** $eval_v(\mathcal{C}_s\langle\!|e|\!\rangle\ (\lambda y\,.\,y)) \simeq_{obs} eval_n(\mathcal{C}_s\langle\!|e|\!\rangle\ (\lambda y\,.\,y))$
2. **Simulation:** $eval_s(e) \simeq_{obs} eval_v(\mathcal{C}_s\langle\!|e|\!\rangle\ (\lambda y\,.\,y))$

Proof: Follows from the correctness of $\mathcal{T}_s$ (Theorem 4.1), the correctness of $\mathcal{C}_v$ (Theorem 3.5, Chapter 3), and soundness of $\beta_a\eta_v$ for call-by-name and call-by-value evaluation of CPS terms. ∎

4.5 Optimizing $\mathcal{C}_s$ using a well-staged transformation

In practice, the rewriting system of Figure 4.5 is only half of the CPS transformation. The resulting term needs to be simplified to be of practical use. These simplifications are known as "administrative reductions" [76]. In an earlier work, Danvy and Filinski propose a method for staging a CPS transformation, by separating the administrative redexes from the syntax constructors [20]. Applying this method to the translation of Figure 4.5 over terms yields the translation of Figures 4.6, 4.7, and 4.8. This translation yields terms without extraneous redexes, in one pass.

$\mathcal{C}_s^1\langle\!|\cdot|\!\rangle$ prevents thunks from being suspended again. $\mathcal{C}_s^2\langle\!|\cdot|\!\rangle$ avoids extraneous η-redexes. $\mathcal{C}_s^3\langle\!|\cdot|\!\rangle$ is the main transformation function. The y's and a's are fresh variables.

The equations of Figures 4.6, 4.7, and 4.8 can be read as a two-level specification *à la* Nielson and Nielson [72] and thus they can be implemented directly in a functional language. Operationally, the overlined λ's and

$$
\begin{array}{rcl}
\mathcal{C}_s & : & \mathit{Terms}[\Lambda_s] \;\to\; \mathit{Terms}[\Lambda^{\sigma^+}] \\
\mathcal{C}_s\langle\![c]\!\rangle & = & \lambda k\,.\,k\,c \\
\mathcal{C}_s\langle\![f]\!\rangle & = & \lambda k\,.\,k\,f \\
\mathcal{C}_s\langle\![x_s]\!\rangle & = & \lambda k\,.\,k\,x \\
\mathcal{C}_s\langle\![x]\!\rangle & = & x \\
\mathcal{C}_s\langle\![\lambda x_s\,.\,e]\!\rangle & = & \lambda k\,.\,k\,(\lambda x\,.\,\mathcal{C}_s\langle\![e]\!\rangle) \\
\mathcal{C}_s\langle\![\lambda x\,.\,e]\!\rangle & = & \lambda k\,.\,k\,(\lambda x\,.\,\mathcal{C}_s\langle\![e]\!\rangle) \\
\mathcal{C}_s\langle\![\mathit{rec}\,f(x_s).\,e]\!\rangle & = & \lambda k\,.\,k\,(\mathit{rec}\,f(x).\,\mathcal{C}_s\langle\![e]\!\rangle) \\
\mathcal{C}_s\langle\![\mathit{rec}\,f(x).\,e]\!\rangle & = & \lambda k\,.\,k\,(\mathit{rec}\,f(x).\,\mathcal{C}_s\langle\![e]\!\rangle) \\
\mathcal{C}_s\langle\![@_s\;e_0\;e_1]\!\rangle & = & \lambda k\,.\,\mathcal{C}_s\langle\![e_0]\!\rangle\,(\lambda y_0\,.\,\mathcal{C}_s\langle\![e_1]\!\rangle\,(\lambda y_1\,.\,(y_0\;y_1)\,k)) \\
\mathcal{C}_s\langle\![@\;e_0\;e_1]\!\rangle & = & \lambda k\,.\,\mathcal{C}_s\langle\![e_0]\!\rangle\,(\lambda y_0\,.\,(y_0\,\mathcal{C}_s\langle\![e_1]\!\rangle)\,k) \\
\mathcal{C}_s\langle\![\mathit{if}\;e_0\;\mathit{then}\;e_1\;\mathit{else}\;e_2]\!\rangle & = & \lambda k\,.\,\mathcal{C}_s\langle\![e_0]\!\rangle\,(\lambda y_0\,.\,\mathit{let}\;k' = k \\
 & & \qquad\qquad \mathit{in}\;\mathit{if}\;y_0\;\mathit{then}\;\mathcal{C}_s\langle\![e_1]\!\rangle\;k'\;\mathit{else}\;\mathcal{C}_s\langle\![e_2]\!\rangle\;k') \\
\mathcal{C}_s\langle\![\mathit{pair}_{ss}\;e_0\;e_1]\!\rangle & = & \lambda k\,.\,\mathcal{C}_s\langle\![e_0]\!\rangle\,(\lambda y_0\,.\,\mathcal{C}_s\langle\![e_1]\!\rangle\,(\lambda y_1\,.\,k\,(\mathit{pair}\;y_0\;y_1))) \\
\mathcal{C}_s\langle\![\mathit{pair}_{s\bullet}\;e_0\;e_1]\!\rangle & = & \lambda k\,.\,\mathcal{C}_s\langle\![e_0]\!\rangle\,(\lambda y_0\,.\,k\,(\mathit{pair}\;y_0\;\mathcal{C}_s\langle\![e_1]\!\rangle)) \\
\mathcal{C}_s\langle\![\mathit{pair}_{\bullet s}\;e_0\;e_1]\!\rangle & = & \lambda k\,.\,\mathcal{C}_s\langle\![e_1]\!\rangle\,(\lambda y_1\,.\,k\,(\mathit{pair}\;\mathcal{C}_s\langle\![e_0]\!\rangle\;y_1)) \\
\mathcal{C}_s\langle\![\mathit{pair}_{\bullet\bullet}\;e_0\;e_1]\!\rangle & = & \lambda k\,.\,k\,(\mathit{pair}\;\mathcal{C}_s\langle\![e_0]\!\rangle\;\mathcal{C}_s\langle\![e_1]\!\rangle) \\
\mathcal{C}_s\langle\![\mathit{fst}_s\;e]\!\rangle & = & \lambda k\,.\,\mathcal{C}_s\langle\![e]\!\rangle\,(\lambda y\,.\,k\,(\mathit{fst}\;y)) \\
\mathcal{C}_s\langle\![\mathit{fst}\;e]\!\rangle & = & \lambda k\,.\,\mathcal{C}_s\langle\![e]\!\rangle\,(\lambda y\,.\,(\mathit{fst}\;y)\;k) \\
\mathcal{C}_s\langle\![\mathit{snd}_s\;e]\!\rangle & = & \lambda k\,.\,\mathcal{C}_s\langle\![e]\!\rangle\,(\lambda y\,.\,k\,(\mathit{snd}\;y)) \\
\mathcal{C}_s\langle\![\mathit{snd}\;e]\!\rangle & = & \lambda k\,.\,\mathcal{C}_s\langle\![e]\!\rangle\,(\lambda y\,.\,(\mathit{snd}\;y)\;k)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}_s & : & \mathit{Types}[\Lambda_s] \to \mathit{Types}[\Lambda^{\sigma^+}] \\
\mathcal{C}_s\langle\!|\sigma|\!\rangle & = & \neg\neg\mathcal{C}_s\langle\sigma\rangle \\
\mathcal{C}_s\langle\iota\rangle & = & \iota \\
\mathcal{C}_s\langle\sigma_1 \stackrel{s}{\to} \sigma_2\rangle & = & \mathcal{C}_s\langle\sigma_1\rangle \to \mathcal{C}_s\langle\!|\sigma_2|\!\rangle \\
\mathcal{C}_s\langle\sigma_1 \to \sigma_2\rangle & = & \mathcal{C}_s\langle\!|\sigma_1|\!\rangle \to \mathcal{C}_s\langle\!|\sigma_2|\!\rangle \\
\mathcal{C}_s\langle\sigma_1 \stackrel{ss}{\times} \sigma_2\rangle & = & \mathcal{C}_s\langle\sigma_1\rangle \times \mathcal{C}_s\langle\sigma_2\rangle \\
\mathcal{C}_s\langle\sigma_1 \stackrel{s\bullet}{\times} \sigma_2\rangle & = & \mathcal{C}_s\langle\sigma_1\rangle \times \mathcal{C}_s\langle\!|\sigma_2|\!\rangle \\
\mathcal{C}_s\langle\sigma_1 \stackrel{\bullet s}{\times} \sigma_2\rangle & = & \mathcal{C}_s\langle\!|\sigma_1|\!\rangle \times \mathcal{C}_s\langle\sigma_2\rangle \\
\mathcal{C}_s\langle\sigma_1 \stackrel{\bullet\bullet}{\times} \sigma_2\rangle & = & \mathcal{C}_s\langle\!|\sigma_1|\!\rangle \times \mathcal{C}_s\langle\!|\sigma_2|\!\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}_s\langle\!|\cdot|\!\rangle & : & \mathit{Assums}[\Lambda_s] \to \mathit{Assums}[\Lambda^{\sigma^+}] \\
\mathcal{C}_s\langle\!|\Gamma, x_s : \sigma|\!\rangle & = & \mathcal{C}_s\langle\!|\Gamma|\!\rangle, x : \mathcal{C}_s\langle\sigma\rangle \\
\mathcal{C}_s\langle\!|\Gamma, x : \sigma|\!\rangle & = & \mathcal{C}_s\langle\!|\Gamma|\!\rangle, x : \mathcal{C}_s\langle\!|\sigma|\!\rangle \\
\mathcal{C}_s\langle\!|\Gamma, f : \sigma|\!\rangle & = & \mathcal{C}_s\langle\!|\Gamma|\!\rangle, f : \mathcal{C}_s\langle\sigma\rangle
\end{array}
$$

Figure 4.5: CPS transformation for a language with strictness annotations

$$\mathcal{C}^1_s \quad : \quad Terms[\Lambda_s] \rightarrow Terms[\Lambda^{\sigma^+}]$$

$$\begin{array}{rcl}
\mathcal{C}^1_s\langle\!\langle x \rangle\!\rangle & = & x \\
\mathcal{C}^1_s\langle\!\langle e \rangle\!\rangle & = & \underline{\lambda}\, k\, .\, \overline{@}\, \mathcal{C}^2_s\langle\!\langle e \rangle\!\rangle\, k
\end{array}$$

Figure 4.6: Staged CPS transformation for a language with strictness annotations (part 1)

$$\mathcal{C}^2_s \quad : \quad Terms[\Lambda_s] \rightarrow \{k\} \rightarrow Terms[\Lambda^{\sigma^+}]$$

$$\begin{array}{rcl}
\mathcal{C}^2_s\langle\!\langle c \rangle\!\rangle & = & \overline{\lambda}\, k\, .\, \underline{@}\, k\, c \\
\mathcal{C}^2_s\langle\!\langle f \rangle\!\rangle & = & \overline{\lambda}\, k\, .\, \underline{@}\, k\, f \\
\mathcal{C}^2_s\langle\!\langle x_s \rangle\!\rangle & = & \overline{\lambda}\, k\, .\, \underline{@}\, k\, x \\
\mathcal{C}^2_s\langle\!\langle x \rangle\!\rangle & = & \overline{\lambda}\, k\, .\, \underline{@}\, x\, k \\
\mathcal{C}^2_s\langle\!\langle \lambda x_s\, .\, e \rangle\!\rangle & = & \overline{\lambda}\, k\, .\, \underline{@}\, k\, (\underline{\lambda}\, x\, .\, \mathcal{C}^1_s\langle\!\langle e \rangle\!\rangle) \\
\mathcal{C}^2_s\langle\!\langle \lambda x\, .\, e \rangle\!\rangle & = & \overline{\lambda}\, k\, .\, \underline{@}\, k\, (\underline{\lambda}\, x\, .\, \mathcal{C}^1_s\langle\!\langle e \rangle\!\rangle) \\
\mathcal{C}^2_s\langle\!\langle rec\, f(x_s).\, e \rangle\!\rangle & = & \overline{\lambda}\, k\, .\, \underline{@}\, k\, (\underline{rec}\, f(x).\, \mathcal{C}^1_s\langle\!\langle e \rangle\!\rangle) \\
\mathcal{C}^2_s\langle\!\langle rec\, f(x).\, e \rangle\!\rangle & = & \overline{\lambda}\, k\, .\, \underline{@}\, k\, (\underline{rec}\, f(x).\, \mathcal{C}^1_s\langle\!\langle e \rangle\!\rangle) \\
\mathcal{C}^2_s\langle\!\langle @_s\, e_0\, e_1 \rangle\!\rangle & = & \overline{\lambda}\, k\, .\, \overline{@}\, \mathcal{C}^3_s\langle\!\langle e_0 \rangle\!\rangle\, (\overline{\lambda}\, y_0\, .\, \overline{@}\, \mathcal{C}^3_s\langle\!\langle e_1 \rangle\!\rangle\, (\overline{\lambda}\, y_1\, .\, \underline{@}\, (\underline{@}\, y_0\, y_1)\, k)) \\
\mathcal{C}^2_s\langle\!\langle @\, e_0\, e_1 \rangle\!\rangle & = & \overline{\lambda}\, k\, .\, \overline{@}\, \mathcal{C}^3_s\langle\!\langle e_0 \rangle\!\rangle\, (\overline{\lambda}\, y_0\, .\, \underline{@}\, (\underline{@}\, y_0\, \mathcal{C}^1_s\langle\!\langle e_1 \rangle\!\rangle)\, k) \\
\mathcal{C}^2_s\langle\!\langle if\ e_0\ then\ e_1\ else\ e_2 \rangle\!\rangle & = & \overline{\lambda}\, k\, .\, \overline{@}\, \mathcal{C}^3_s\langle\!\langle e_0 \rangle\!\rangle\, (\overline{\lambda}\, y_0\, .\, \underline{if}\ y_0\ \underline{then}\ \overline{@}\, \mathcal{C}^2_s\langle\!\langle e_1 \rangle\!\rangle\, k\ \underline{else}\ \overline{@}\, \mathcal{C}^2_s\langle\!\langle e_2 \rangle\!\rangle\, k) \\
\mathcal{C}^2_s\langle\!\langle pair_{ss}\ e_0\ e_1 \rangle\!\rangle & = & \overline{\lambda}\, k\, .\, \overline{@}\, \mathcal{C}^3_s\langle\!\langle e_0 \rangle\!\rangle\, (\overline{\lambda}\, y_0\, .\, \overline{@}\, \mathcal{C}^3_s\langle\!\langle e_1 \rangle\!\rangle\, (\overline{\lambda}\, y_1\, .\, \underline{@}\, k\, (\underline{pair}\, y_0\, y_1))) \\
\mathcal{C}^2_s\langle\!\langle pair_{s\bullet}\ e_0\ e_1 \rangle\!\rangle & = & \overline{\lambda}\, k\, .\, \overline{@}\, \mathcal{C}^3_s\langle\!\langle e_0 \rangle\!\rangle\, (\overline{\lambda}\, y_0\, .\, \underline{@}\, k\, (\underline{pair}\, y_0\, \mathcal{C}^1_s\langle\!\langle e_1 \rangle\!\rangle)) \\
\mathcal{C}^2_s\langle\!\langle pair_{\bullet s}\ e_0\ e_1 \rangle\!\rangle & = & \overline{\lambda}\, k\, .\, \overline{@}\, \mathcal{C}^3_s\langle\!\langle e_1 \rangle\!\rangle\, (\overline{\lambda}\, y_1\, .\, \underline{@}\, k\, (\underline{pair}\, \mathcal{C}^1_s\langle\!\langle e_0 \rangle\!\rangle\, y_1)) \\
\mathcal{C}^2_s\langle\!\langle pair_{\bullet\bullet}\ e_0\ e_1 \rangle\!\rangle & = & \overline{\lambda}\, k\, .\, \underline{@}\, k\, (\underline{pair}\, \mathcal{C}^1_s\langle\!\langle e_0 \rangle\!\rangle\, \mathcal{C}^1_s\langle\!\langle e_1 \rangle\!\rangle) \\
\mathcal{C}^2_s\langle\!\langle fst_s\ e \rangle\!\rangle & = & \overline{\lambda}\, k\, .\, \overline{@}\, \mathcal{C}^3_s\langle\!\langle e \rangle\!\rangle\, (\overline{\lambda}\, y\, .\, \underline{@}\, k\, (\underline{fst}\, y)) \\
\mathcal{C}^2_s\langle\!\langle fst\ e \rangle\!\rangle & = & \overline{\lambda}\, k\, .\, \overline{@}\, \mathcal{C}^3_s\langle\!\langle e \rangle\!\rangle\, (\overline{\lambda}\, y\, .\, \underline{@}\, (\underline{fst}\, y)\, k) \\
\mathcal{C}^2_s\langle\!\langle snd_s\ e \rangle\!\rangle & = & \overline{\lambda}\, k\, .\, \overline{@}\, \mathcal{C}^3_s\langle\!\langle e \rangle\!\rangle\, (\overline{\lambda}\, y\, .\, \underline{@}\, k\, (\underline{snd}\, y)) \\
\mathcal{C}^2_s\langle\!\langle snd\ e \rangle\!\rangle & = & \overline{\lambda}\, k\, .\, \overline{@}\, \mathcal{C}^3_s\langle\!\langle e \rangle\!\rangle\, (\overline{\lambda}\, y\, .\, \underline{@}\, (\underline{snd}\, y)\, k)
\end{array}$$

Figure 4.7: Staged CPS transformation for a language with strictness annotations (part 2)

$$\mathcal{C}_s^3 \quad : \quad \mathit{Terms}[\Lambda_s] \to [\mathit{Terms}[\Lambda^{\sigma^+}] \to \mathit{Terms}[\Lambda^{\sigma^+}]] \to \mathit{Terms}[\Lambda^{\sigma^+}]$$

$$\begin{array}{rcl}
\mathcal{C}_s^3\langle\!\langle c \rangle\!\rangle & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\kappa\,c \\
\mathcal{C}_s^3\langle\!\langle f \rangle\!\rangle & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\kappa\,f \\
\mathcal{C}_s^3\langle\!\langle x_s \rangle\!\rangle & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\kappa\,x \\
\mathcal{C}_s^3\langle\!\langle x \rangle\!\rangle & = & \overline{\lambda}\,\kappa\,.\,\underline{@}\,x\,(\underline{\lambda}\,a\,.\,\overline{@}\,\kappa\,a) \\
\mathcal{C}_s^3\langle\!\langle \lambda x_s\,.\,e \rangle\!\rangle & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\kappa\,(\underline{\lambda}\,x\,.\,\mathcal{C}_s^1\langle\!\langle e \rangle\!\rangle) \\
\mathcal{C}_s^3\langle\!\langle \lambda x\,.\,e \rangle\!\rangle & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\kappa\,(\underline{\lambda}\,x\,.\,\mathcal{C}_s^1\langle\!\langle e \rangle\!\rangle) \\
\mathcal{C}_s^3\langle\!\langle \mathit{rec}\,f(x_s).\,e \rangle\!\rangle & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\kappa\,(\underline{\mathit{rec}}\,f(x).\,\mathcal{C}_s^1\langle\!\langle e \rangle\!\rangle) \\
\mathcal{C}_s^3\langle\!\langle \mathit{rec}\,f(x).\,e \rangle\!\rangle & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\kappa\,(\underline{\mathit{rec}}\,f(x).\,\mathcal{C}_s^1\langle\!\langle e \rangle\!\rangle) \\
\mathcal{C}_s^3\langle\!\langle @_s\,e_0\,e_1 \rangle\!\rangle & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\langle\!\langle e_0 \rangle\!\rangle\,(\overline{\lambda}\,y_0\,.\,\overline{@}\,\mathcal{C}_s^3\langle\!\langle e_1 \rangle\!\rangle\,(\overline{\lambda}\,y_1\,.\,\underline{@}\,(\underline{@}\,y_0\,y_1)\,(\underline{\lambda}\,a\,.\,\overline{@}\,\kappa\,a))) \\
\mathcal{C}_s^3\langle\!\langle @\,e_0\,e_1 \rangle\!\rangle & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\langle\!\langle e_0 \rangle\!\rangle\,(\overline{\lambda}\,y_0\,.\,\underline{@}\,(\underline{@}\,y_0\,\mathcal{C}_s^1\langle\!\langle e_1 \rangle\!\rangle)\,(\underline{\lambda}\,a\,.\,\overline{@}\,\kappa\,a)) \\
\mathcal{C}_s^3\langle\!\langle \mathit{if}\,e_0\,\mathit{then}\,e_1\,\mathit{else}\,e_2 \rangle\!\rangle & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\langle\!\langle e_0 \rangle\!\rangle\,(\overline{\lambda}\,y_0\,.\,\underline{@}\,(\underline{\lambda}\,k\,.\,\underline{\mathit{if}}\,y_0\,\underline{\mathit{then}}\,\overline{@}\,\mathcal{C}_s^2\langle\!\langle e_1 \rangle\!\rangle\,k\,\underline{\mathit{else}}\,\overline{@}\,\mathcal{C}_s^2\langle\!\langle e_2 \rangle\!\rangle\,k)\,(\underline{\lambda}\,a\,.\,\overline{@}\,\kappa\,a)) \\
\mathcal{C}_s^3\langle\!\langle \mathit{pair}_{ss}\,e_0\,e_1 \rangle\!\rangle & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\langle\!\langle e_0 \rangle\!\rangle\,(\overline{\lambda}\,y_0\,.\,\overline{@}\,\mathcal{C}_s^3\langle\!\langle e_1 \rangle\!\rangle\,(\overline{\lambda}\,y_1\,.\,\overline{@}\,\kappa\,(\underline{\mathit{pair}}\,y_0\,y_1))) \\
\mathcal{C}_s^3\langle\!\langle \mathit{pair}_{s\bullet}\,e_0\,e_1 \rangle\!\rangle & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\langle\!\langle e_0 \rangle\!\rangle\,(\overline{\lambda}\,y_0\,.\,\overline{@}\,\kappa\,(\underline{\mathit{pair}}\,y_0\,\mathcal{C}_s^1\langle\!\langle e_1 \rangle\!\rangle)) \\
\mathcal{C}_s^3\langle\!\langle \mathit{pair}_{\bullet s}\,e_0\,e_1 \rangle\!\rangle & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\langle\!\langle e_1 \rangle\!\rangle\,(\overline{\lambda}\,y_1\,.\,\overline{@}\,\kappa\,(\underline{\mathit{pair}}\,\mathcal{C}_s^1\langle\!\langle e_0 \rangle\!\rangle\,y_1)) \\
\mathcal{C}_s^3\langle\!\langle \mathit{pair}_{\bullet\bullet}\,e_0\,e_1 \rangle\!\rangle & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\kappa\,(\underline{\mathit{pair}}\,\mathcal{C}_s^1\langle\!\langle e_0 \rangle\!\rangle\,\mathcal{C}_s^1\langle\!\langle e_1 \rangle\!\rangle) \\
\mathcal{C}_s^3\langle\!\langle \mathit{fst}_s\,e \rangle\!\rangle & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\langle\!\langle e \rangle\!\rangle\,(\overline{\lambda}\,y\,.\,\overline{@}\,\kappa\,(\underline{\mathit{fst}}\,y)) \\
\mathcal{C}_s^3\langle\!\langle \mathit{fst}\,e \rangle\!\rangle & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\langle\!\langle e \rangle\!\rangle\,(\overline{\lambda}\,y\,.\,\underline{@}\,(\underline{\mathit{fst}}\,y)\,(\underline{\lambda}\,a\,.\,\overline{@}\,\kappa\,a)) \\
\mathcal{C}_s^3\langle\!\langle \mathit{snd}_s\,e \rangle\!\rangle & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\langle\!\langle e \rangle\!\rangle\,(\overline{\lambda}\,y\,.\,\overline{@}\,\kappa\,(\underline{\mathit{snd}}\,y)) \\
\mathcal{C}_s^3\langle\!\langle \mathit{snd}\,e \rangle\!\rangle & = & \overline{\lambda}\,\kappa\,.\,\overline{@}\,\mathcal{C}_s^3\langle\!\langle e \rangle\!\rangle\,(\overline{\lambda}\,y\,.\,\underline{@}\,(\underline{\mathit{snd}}\,y)\,(\underline{\lambda}\,a\,.\,\overline{@}\,\kappa\,a))
\end{array}$$

Figure 4.8: Staged CPS transformation for a language with strictness annotations (part 3)

@'s correspond to functional abstractions and applications in the translation program, while only the underlined occurrences represent abstract-syntax constructors.

The result of transforming a term e into CPS in an empty context (represented by the identity function) is given by

$$\overline{@}\,\mathcal{C}_s^3\langle\!\langle e \rangle\!\rangle\,(\overline{\lambda}\,y\,.\,y)$$

With regard to complexity, the well-staged transformation combines the three linear-time transformations,

1. thunk introduction $\mathcal{T}_s$;
2. call-by-value CPS transformation $\mathcal{C}_v$;
3. and reduction of administrative redexes of the CPS transformation,

into a single linear-time transformation.

4.6 Conclusion and issues

Strictness analysis enables one to transform a program with call-by-name applications only into a program with both call-by-name and call-by-value applications. Thunks allow one to transform the remaining call-by-name applications to call-by-value applications. We have composed the introduction of thunks with the call-by-value CPS transformation, thereby specifying how to transform a λ-term with strictness annotations into continuation-passing style directly. The resulting transformation generalizes both the call-by-name and call-by-value CPS

transformations in that restricting it to non-strict constructs gives the call-by-name CPS transformation and restricting it to strict constructs gives the call-by-value CPS transformation.

This new transformation enables one to implement a call-by-name language by using an existing CPS-based compiler [3, 91] or an existing program-transformation system [33, 47]. This method is extended to call-by-need and expanded on in several ways by Okasaki, Lee, and Tarditi [73].

Acknowledgements

An earlier version of this chapter appeared in *ACM Letters on Programming Languages and Systems* [22]. In addition to an abbreviated exposition of the material presented here, that version includes a discussion of strictness annotations and associated CPS transformation for a fixpoint operator.

Earlier versions of this chapter benefited from the comments of Andrzej Filinski, Julia Lawall, Karoline Malmkjær, and David Schmidt.

Chapter 5

On the Transformation between Direct and Continuation Semantics

5.1 Introduction

Proving the congruence between a denotational semantics in direct style and a denotational semantics in continuation style is somewhat tedious [81, 87, 92]. Yet,

- both direct and continuation semantics can be specified using simply-typed λ-terms as a meta-language: semantic domains are represented by types, and valuation functions by λ-terms [71, 87];
- simply-typed λ-terms can be transformed into continuation-passing style *automatically* using continuation-passing-style (CPS) transformations as introduced in the previous chapters.

We have transformed several direct typed λ-term specifications into continuation-passing-style. Since such specifications are typically processed using *normal-order reduction* [87], we have used the call-by-name CPS transformation $\mathcal{C}_n$. The result is *not* a conventional continuation specification.

It is sufficient to look at types to see where a mismatch occurs. Figure 5.1 gives the types of two valuation functions for a simple imperative language. $\mathcal{C}_d[\![\cdot]\!]$ is a direct-style valuation function and $\mathcal{C}_k[\![\cdot]\!]$ is a continuation-passing-style valuation function. Transforming the types of the direct-style valuation function $\mathcal{C}_d[\![\cdot]\!]$ does not yield the types of the continuation-passing-style valuation function $\mathcal{C}_k[\![\cdot]\!]$. For example, the transformation of the function space $Env \rightarrow Com_d$ yields

$$((Env \rightarrow ans) \rightarrow ans) \rightarrow (\mathcal{C}_n\langle Com_d\rangle \rightarrow ans) \rightarrow ans$$

which does not match the type of the corresponding function space

$$Env \rightarrow Com_k$$

in $\mathcal{C}_k[\![\cdot]\!]$. Essentially, $\mathcal{C}_n$ introduces too many continuations.

This mismatch is significant because it shows that a continuation semantics is not just a direct semantics with continuations introduced naively according to the usual rules for converting to CPS.

There are at least two options for overcoming the mismatch.

- We could establish the relationship between the result of CPS-transforming a direct specification and a realistic continuation specification (*i.e.*, a specification such as an experienced denotational-semanticist would write), and perhaps map one into the other.
- We could devise a new CPS transformation that would transform a direct specification immediately into a realistic continuation specification.

We choose the latter option. Essentially, we identify properties of a direct-style specification (*e.g.*, termination), and we design a new call-by-name CPS transformation $\mathcal{C}_t$ that uses these properties to avoid introducing

$$\mathcal{C}_d[\![\cdot]\!] : Env \to Com_d \quad \text{where} \quad Com_d = Store \to Store$$

$$\mathcal{C}_k[\![\cdot]\!] : Env \to Com_k \quad \text{where} \quad Com_k = Store \to (Store \to ans) \to ans$$

Figure 5.1: Types of valuation functions for a simple imperative language

"unnecessary" continuations. As a result, we can produce a realistic continuation semantics specification, automatically.

It is important to understand the transformation between direct and continuation specifications for several reasons.

1. It is these specifications that get processed in any kind of semantics-based program manipulation (*e.g.*, compiling, compiler generation, and partial evaluation).

2. The properties of the transformation should give insight into proving the congruence between the direct semantics and the continuation semantics.

The tools used in this chapter are interesting in their own right.

1. The new call-by-name CPS transformation $\mathcal{C}_t$ is directed by type annotations capturing termination properties. Our annotation system extends Reynolds's *trivial/serious* termination classification scheme[1] and allows finer distinctions to be made when describing termination properties of terms. For example, unlike Reynolds's original scheme, it allows one to classify some terminating function applications as trivial.

2. We point out that the annotations can be obtained by an automatic control-flow analysis that extends Mycroft's $\flat$ termination analysis to higher-order programs [67, 68]. This tool has applications in other areas such as compiling and partial evaluation.

3. The new transformation $\mathcal{C}_t$ follows Reynolds's method of introducing continuations only in serious terms. Thus, $\mathcal{C}_t$ generalizes the call-by-name CPS transformation $\mathcal{C}_n$ (should all terms be serious) and the identity transformation (should all terms be trivial).

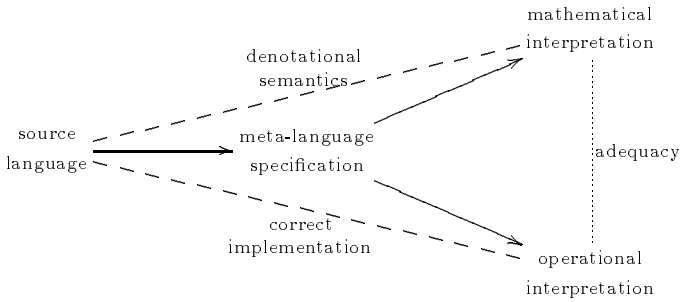
The rest of this chapter is organized as follows. Section 5.2 describes how semantic definitions can be specified using simply-typed λ-terms as a meta-language. Section 5.3 presents an example language and two semantics specifications, one in direct style and one in continuation style. Section 5.4 motivates the generalization of Reynolds's termination classification scheme. Section 5.5 formalizes the new scheme using an annotated type system. Section 5.6 gives the new call-by-name CPS transformation $\mathcal{C}_t$ directed by termination properties captured *via* annotated types. In Section 5.7, we examine the properties of the direct-style specification of Section 5.3 and we annotate the specification as motivated by the previous sections. In Section 5.8, we transform the annotated direct-style specification into continuation style using $\mathcal{C}_t$ and we obtain the expected continuation semantics specification. Finally, Section 5.9 concludes and puts this work in perspective.

5.2 Implementation-oriented denotational specifications

5.2.1 Correct implementations from denotational specifications

Many frameworks exist for deriving correct language implementations from denotational semantics definitions. A common approach is to specify the semantics of the source language in a meta-language that has both a mathematical interpretation (*e.g.*, a fixpoint semantics) and an operational interpretation (*e.g.*, an evaluator stated in terms of an abstract machine).

[1] In Reynolds's scheme, evaluation of a *trivial* term always terminates whereas evaluation of a *serious* term may not terminate. As an approximation, Reynolds identified trivial terms with *values*.



The mathematical interpretation of the *specification* gives a denotational *semantics* for the source language.[2] The operational interpretation of the specification provides a *compile-evaluate* implementation [87, p. 217] of the source language:

- the specification *compiles* the source language program to a (target) meta-language program; and
- the meta-language evaluator *evaluates* the resulting meta-language program.

A *computational adequacy* theorem ensures that the meta-language evaluator adequately computes the mathematical interpretation of the meta-language. This guarantees that the compile-evaluate strategy correctly implements the meaning of the source language as defined by its denotational semantics.

5.2.2 Specifying denotational semantics using simply-typed λ-terms

The language of simply-typed λ-terms (*e.g.*, Λ^{σ^+}) is a common choice for the meta-language in the approach above [87, 70, 72]. In its mathematical interpretation, types represent computational domains (*e.g.*, *cpo*'s), terms specify valuation functions, and primitive operations specify semantic algebra operations. Following the convention of processing semantic meta-languages using normal-order reduction [87, p. 221], $eval_n$ (as presented in preceding chapters) provides a suitable operational interpretation. An adequacy property exists which ensures that the computational notions such non-termination and recursion (represented in the mathematical interpretation by *e.g.*, least fixpoint operations over *cpo*'s) are captured correctly in the evaluation properties of typed terms. (See [37] for a detailed discussion and adequacy proof for PCF — a language similar to Λ^{σ^+}.)

5.2.3 Goals and perspectives

The goal of this chapter is to suggest directions for automatically deriving a realistic continuation *semantics* from a direct *semantics*. Working within the framework outlined above, we attack this problem by appropriately transforming direct-style *specifications*. Given an adequacy property, a correct CPS transformation which preserves operational behavior will preserve the denotational semantics as well.

We feel free to limit our discussion to the operational properties of specifications since,

- techniques for relating operational and denotational meaning are well-documented [37], and
- applications of our work to partial evaluation and compiling will focus on the operational properties of specifications rather than on the mathematical properties of the denotational semantics.

5.3 Example denotational specifications

Figure 5.2 presents the abstract syntax of a simple imperative language with global non-recursive first-order procedures. Figures 5.3 and 5.4 give a direct semantics specification and a continuation semantics specification for the simple imperative language. Following the approach outlined above, the specifications are stated using the extended typed language Λ^{σ^+}. Primitive domains (*e.g.*, *Store*, *Env*, *etc.*) form the base types and the semantic-algebra operations such as `upd` and `fetch` are represented by δ-rules. The definitions of the semantic algebras for stores, environments, and natural numbers are the usual ones and therefore omitted.

[2] Note that we distinguish between the *specification* (the meta-language terms) and the actual *semantics* (the mathematical meaning represented by the terms.)

$$
\begin{array}{rcl@{\qquad\qquad}rcl}
z & \in & \text{Program} & l & \in & \text{Location} \\
c & \in & \text{Command} & m & \in & \text{Ident[num]} \\
e & \in & \text{Expression} & p & \in & \text{Ident[proc]} \\
n & \in & \text{Numeral} & f & \in & \text{Ident[fun]}
\end{array}
$$

$$
\begin{array}{l}
z ::= \text{proc}\, p\,(m) = c\,\text{in}\, z \mid \text{fun}\, f\,(m) = e\,\text{in}\, z \mid c. \\
c ::= \text{skip} \mid c_1\,;\, c_2 \mid l := e \mid \text{if}\, e\,\text{then}\, c_1\,\text{else}\, c_2 \mid \text{while}\, e\,\text{do}\, c \mid \text{call}\, p\,(e) \\
e ::= n \mid m \mid \text{succ}\, e \mid \text{pred}\, e \mid \text{deref}\, l \mid \text{apply}\, f\,(e)
\end{array}
$$

Figure 5.2: Abstract syntax of the simple imperative language

Valuation Functions:

$$
\begin{array}{rcl}
\mathcal{Z}_d[\![\text{Program}]\!] & : & Env \to Com_d \\
\mathcal{C}_d[\![\text{Command}]\!] & : & Env \to Com_d \\
\mathcal{E}_d[\![\text{Expression}]\!] & : & Env \to Exp_d \\
\mathcal{N}_d[\![\text{Numeral}]\!] & : & Nat \\
\mathcal{L}_d[\![\text{Location}]\!] & : & Loc
\end{array}
$$

Semantic Domains:

$$
\begin{array}{rcl}
Com_d & = & Store \to Store \\
Exp_d & = & Store \to Nat \\
Proc_d & = & Nat \to Com_d \\
Fun_d & = & Nat \to Exp_d
\end{array}
$$

Programs:

$$
\begin{array}{rcl}
\mathcal{Z}_d[\![\text{proc}\, p\,(m) = c\,\text{in}\, z]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathcal{Z}_d[\![z]\!]\,(\mathsf{ext}\,\rho\,p\,(\lambda i\,.\,\mathcal{C}_d[\![c]\!]\,(\mathsf{ext}\,\rho\,m\,i)))\,\sigma \\
\mathcal{Z}_d[\![\text{fun}\, f\,(m) = e\,\text{in}\, p]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathcal{Z}_d[\![z]\!]\,(\mathsf{ext}\,\rho\,f\,(\lambda i\,.\,\mathcal{E}_d[\![e]\!]\,(\mathsf{ext}\,\rho\,m\,i)))\,\sigma \\
\mathcal{Z}_d[\![c.]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathcal{C}_d[\![c]\!]\,\rho\,\sigma
\end{array}
$$

Commands:

$$
\begin{array}{rcl}
\mathcal{C}_d[\![\text{skip}]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\sigma \\
\mathcal{C}_d[\![c_1\,;\, c_2]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,let\ \sigma' = \mathcal{C}_d[\![c_1]\!]\,\rho\,\sigma\ in\ \mathcal{C}_d[\![c_2]\!]\,\rho\,\sigma' \\
\mathcal{C}_d[\![l := e]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathsf{upd}\,\sigma\,\mathcal{L}_d[\![l]\!]\,(\mathcal{E}_d[\![e]\!]\,\rho\,\sigma) \\
\mathcal{C}_d[\![\text{if}\, e\,\text{then}\, c_1\,\text{else}\, c_2]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,if\ \mathsf{iszero?}\,(\mathcal{E}_d[\![e]\!]\,\rho\,\sigma)\ then\ (\mathcal{C}_d[\![c_1]\!]\,\rho\,\sigma)\ else\ (\mathcal{C}_d[\![c_2]\!]\,\rho\,\sigma) \\
\mathcal{C}_d[\![\text{while}\, e\,\text{do}\, c]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,(rec\, w(\sigma).\ \ if\ \ \mathsf{iszero?}\,(\mathcal{E}_d[\![e]\!]\,\rho\,\sigma) \\
 & & \qquad\qquad\qquad\qquad then\ let\ \sigma' = \mathcal{C}_d[\![c]\!]\,\rho\,\sigma\ in\ w\,\sigma' \\
 & & \qquad\qquad\qquad\qquad else\ \ \sigma)\,\sigma \\
\mathcal{C}_d[\![\text{call}\, p\,(e)]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,(\mathsf{lookup}\,\rho\,p)\,(\mathcal{E}_d[\![e]\!]\,\rho\,\sigma)\,\sigma
\end{array}
$$

Expressions:

$$
\begin{array}{rcl}
\mathcal{E}_d[\![n]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathcal{N}_d[\![n]\!] \\
\mathcal{E}_d[\![m]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathsf{lookup}\,\rho\,m \\
\mathcal{E}_d[\![\text{succ}\, e]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathsf{succ}\,(\mathcal{E}_d[\![e]\!]\,\rho\,\sigma) \\
\mathcal{E}_d[\![\text{pred}\, e]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathsf{pred}\,(\mathcal{E}_d[\![e]\!]\,\rho\,\sigma) \\
\mathcal{E}_d[\![\text{deref}\, l]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathsf{fetch}\,\sigma\,\mathcal{L}_d[\![l]\!] \\
\mathcal{E}_d[\![\text{apply}\, f\,(e)]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,(\mathsf{lookup}\,\rho\,f)\,(\mathcal{E}_d[\![e]\!]\,\rho\,\sigma)\,\sigma
\end{array}
$$

Figure 5.3: Direct semantics specification for the simple imperative language

Valuation Functions:

$$\begin{array}{rcl}
\mathcal{Z}_k[\![\text{Program}]\!] & : & Env \to Com_k \\
\mathcal{C}_k[\![\text{Command}]\!] & : & Env \to Com_k \\
\mathcal{E}_k[\![\text{Expression}]\!] & : & Env \to Exp_k \\
\mathcal{N}_k[\![\text{Numeral}]\!] & : & Nat \\
\mathcal{L}_k[\![\text{Location}]\!] & : & Loc
\end{array}$$

Semantic Domains:

$$\begin{array}{rcl}
Com_k & = & Store \to \neg\neg Store \\
Exp_k & = & Store \to Nat \\
Proc_k & = & Nat \to Com_k \\
Fun_k & = & Nat \to Exp_k
\end{array}$$

Programs:

$$\begin{array}{rcl}
\mathcal{Z}_k[\![\text{proc } p\ (m) = c \text{ in } z]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda\kappa\,.\,\mathcal{Z}_k[\![z]\!]\,(\mathsf{ext}\,\rho\,p\,(\lambda i\,.\,\mathcal{C}_k[\![c]\!]\,(\mathsf{ext}\,\rho\,m\,i)))\,\sigma\,\kappa \\
\mathcal{Z}_k[\![\text{fun } f\ (m) = e \text{ in } z]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda\kappa\,.\,\mathcal{Z}_k[\![z]\!]\,(\mathsf{ext}\,\rho\,f\,(\lambda i\,.\,\mathcal{E}_k[\![e]\!]\,(\mathsf{ext}\,\rho\,m\,i)))\,\sigma\,\kappa \\
\mathcal{Z}_k[\![c.]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda\kappa\,.\,\mathcal{C}_k[\![c]\!]\,\rho\,\sigma\,\kappa
\end{array}$$

Commands:

$$\begin{array}{rcl}
\mathcal{C}_k[\![\text{skip}]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda\kappa\,.\,\kappa\,\sigma \\
\mathcal{C}_k[\![c_1\ ;\ c_2]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda\kappa\,.\,\mathcal{C}_k[\![c_1]\!]\,\rho\,\sigma\,(\lambda\sigma'\,.\,\mathcal{C}_k[\![c_2]\!]\,\rho\,\sigma'\,\kappa) \\
\mathcal{C}_k[\![l := e]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda\kappa\,.\,\kappa\,(\mathsf{upd}\,\sigma\,\mathcal{L}_k[\![l]\!]\,(\mathcal{E}_k[\![e]\!]\,\rho\,\sigma)) \\
\mathcal{C}_k[\![\text{if } e \text{ then } c_1 \text{ else } c_2]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda\kappa\,.\ \ \mathit{if}\ \mathsf{iszero?}\ (\mathcal{E}_k[\![e]\!]\,\rho\,\sigma)\ \mathit{then}\ (\mathcal{C}_k[\![c_1]\!]\,\rho\,\sigma\,\kappa) \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{else}\ (\mathcal{C}_k[\![c_2]\!]\,\rho\,\sigma\,\kappa) \\
\mathcal{C}_k[\![\text{while } e \text{ do } c]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda\kappa\,.\,(\mathit{rec}\ w(\sigma).\,\lambda\kappa'\,.\ \ \mathit{if}\ \ \mathsf{iszero?}\ (\mathcal{E}_k[\![e]\!]\,\rho\,\sigma) \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{then}\ \mathcal{C}_k[\![c]\!]\,\rho\,\sigma\,(\lambda\sigma'\,.\,w\,\sigma'\,\kappa') \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{else}\ \ \kappa'\,\sigma)\,\sigma\,\kappa \\
\mathcal{C}_k[\![\text{call } p\ (e)]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda\kappa\,.\,(\mathsf{lookup}\,\rho\,p)\,(\mathcal{E}_k[\![e]\!]\,\rho\,\sigma)\,\sigma\,\kappa
\end{array}$$

Expressions:

$$\begin{array}{rcl}
\mathcal{E}_k[\![n]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathcal{N}_k[\![n]\!] \\
\mathcal{E}_k[\![m]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathsf{lookup}\,\rho\,m \\
\mathcal{E}_k[\![\text{succ } e]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathsf{succ}\,(\mathcal{E}_k[\![e]\!]\,\rho\,\sigma) \\
\mathcal{E}_k[\![\text{pred } e]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathsf{pred}\,(\mathcal{E}_k[\![e]\!]\,\rho\,\sigma) \\
\mathcal{E}_k[\![\text{deref } l]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathsf{fetch}\,\sigma\,\mathcal{L}_k[\![l]\!] \\
\mathcal{E}_k[\![\text{apply } f\ (e)]\!] & = & \lambda\rho\,.\,\lambda\sigma\,.\,(\mathsf{lookup}\,\rho\,f)\,(\mathcal{E}_k[\![e]\!]\,\rho\,\sigma)\,\sigma
\end{array}$$

Figure 5.4: Continuation semantics specification for the simple imperative language

Proposition 5.1 *The denotational semantics given by the specifications in Figures 5.3 and 5.4 are congruent (i.e., they give the same meaning to the source language).*

A detailed proof for a similar language is given in [92] and [87, p. 210].

5.4 A re-examination of evaluation-order-independence and CPS

As pointed out in Section 5.1, transforming the direct specification into continuation-passing-style using the call-by-name CPS transformation $\mathcal{C}_n$ does *not* yield a realistic continuation specification. Essentially, the $\mathcal{C}_n$ introduces more continuations than are needed. For example, in Figure 5.4, $\mathcal{E}$ is expressed in direct style even though it is part of a continuation semantics. In this section, we re-examine the motivation for introducing continuations as outlined by Reynolds in his classic paper "Definitional Interpreters for Higher-Order Programming Languages" [80]. We aim to clarify why $\mathcal{E}$ does not need any continuation, and to determine conditions that allow a CPS transformation to avoid introducing continuations in similar situations.

5.4.1 Reynolds's notion of trivial and serious terms

One of Reynolds's goals in [80] was to give an evaluation-order-independent definitional interpreter. He noted that the evaluation-order-independence property fails only in the presence of non-termination. To regain the property in the presence of non-termination, Reynolds took the following steps.

1. First, a classification scheme was proposed to describe termination properties of terms. Terms that are demonstrably terminating were called *trivial*; terms that are possibly non-terminating were called *serious*. As an approximation, Reynolds identified trivial terms with *values*.

2. This classification scheme was used informally to justify the structure of continuation-passing required for evaluation-order independence. In principle, trivial terms require no continuation-passing since they are already evaluation-order independent. On the other hand, serious terms require continuation-passing for evaluation-order independence.

Although quite suitable for his goal, Reynolds's identification of trivial terms with values is too coarse an approximation for some applications. For example, in denotational specifications, valuation functions are usually curried. Most of the time, the result of applying a valuation function to an abstract-syntax tree is a λ-abstraction. In fact, this is the case for $\mathcal{P}$, $\mathcal{C}$, and $\mathcal{E}$ in Figure 5.3. Since a λ-abstraction is a trivial term, applying a valuation function does not yield a serious term.[3] This suggests that it is overly conservative to consider all non-values as serious. In particular, the example above illustrates that the application of some functions can be demonstrated to terminate. Such functions do not need to be passed a continuation to obtain evaluation-order independence.

5.4.2 Generalizing Reynolds's notion of trivial and serious terms

In the remainder of this section, we outline a scheme where the notion of *trivial* term is generalized from *values* to include *demonstrably terminating non-values*. This generalization will justify a new call-by-name CPS transformation $\mathcal{C}_t$ which introduces less continuation-passing than $\mathcal{C}_n$. In particular, $\mathcal{C}_t$ avoids introducing continuations in demonstrably terminating terms.

We first consider λ-abstraction bodies.

- If the body of a λ-abstraction is trivial, then the result of applying that abstraction will be trivial (*i.e.*, the function computed is *total*). We refer to such abstractions as *result-trivial*.

- If the body of a λ-abstraction is serious, then the result of applying that abstraction will be serious (*i.e.*, the function computed may be *partial*). We refer to such abstractions as *result-serious*.

[3] This is probably why Reynolds's definitional interpreters are uncurried [80].

Following the justification of continuation-passing above, the body of a result-serious abstraction should employ continuation-passing whereas the body of of a result-trivial abstraction need not. Therefore, result-serious abstractions should be passed a continuation when applied whereas result-trivial abstractions should not.

In most cases, whether or not an abstraction body is trivial or serious depends on whether or not the argument passed to the abstraction is trivial or serious.[4]

- If an abstraction is applied to only trivial arguments, we refer to it as *argument-trivial.*
- If an abstraction may be applied to a serious argument, we refer to it as *argument-serious.*

Following the justification of continuation-passing above, a argument-serious abstraction must be prepared to evaluate its argument with a continuation whereas a argument-trivial abstraction does not need to supply a continuation when evaluating its argument.

This gives four cases of abstractions:

1. trivial-argument, trivial-result abstractions,
2. trivial-argument, serious-result abstractions,
3. serious-argument, trivial-result abstractions, and
4. serious-argument, serious-result abstractions.

Note that an abstraction may not have a unique classification. For example, $\lambda x\,.\,x$ could be classified as *argument-trivial, result-trivial* as well as *argument-serious, result-serious.*[5]

In the following section we will see that the generalization of the trivial-serious classification scheme extends to constructs other than abstractions.

5.5 An annotated type system capturing termination properties

In this section we formalize the classification scheme outlined above. First, we present a language Λ_t with annotated types capturing desired termination properties. Next, we state the correctness criteria for the annotations. Finally, we discuss how annotations can be assigned automatically and give useful annotation abbreviations.

5.5.1 The termination annotated language Λ_t

The annotated language Λ_t formalizes the classification scheme outlined in the previous section. Figure 5.5 presents terms, types, and assumptions for Λ_t. Note that $\mathit{Terms}[\Lambda_t] = \mathit{Terms}[\Lambda^{\sigma^+}]$ since we intend all necessary information to be captured in the typing system.[6] The four function space constructors in $\mathit{Types}[\Lambda_t]$ correspond to the four cases of abstractions given in the previous section (*e.g.*, an argument-trivial, result-serious abstraction will be assigned a type $\sigma_1 \rightarrow_{ts} \sigma_2$). The tags in the partially ordered set $(\mathit{Tags}[\Lambda_t], \leq)$ (where $\mathit{trivial} \leq \mathit{serious}$) indicate whether a term is trivial (always terminating) or serious (possibly non-terminating). Besides being associated with a type, each (non-recursive) identifier in $\Gamma \in \mathit{Assums}[\Lambda_t]$ is associated with a tag indicating whether it may bind to a trivial or serious term. Tags are unnecessary for recursive identifiers since they only bind to recursive abstractions (which are values).

Figures 5.6 and 5.7 present the annotated type assignment rules. A term is annotated by exhibiting a derivation $d \in \mathit{TypingDerivations}[\Lambda_t]$ of a judgement $\Gamma \vdash_t e : (\sigma\,,\,\theta\,)$. For example, the judgement $\Gamma \vdash_t \lambda x\,.\,e : (\sigma_1 \rightarrow_{st} \sigma_2\,,\,\mathit{trivial})$ means that $\lambda x\,.\,e$ is argument-serious, result-trivial. Furthermore, since $\lambda x\,.\,e$ is a value, it is assigned the tag *trivial.* As a second example, $\Gamma \vdash_t e : (\sigma_1 \rightarrow_{tt} \sigma_2\,,\,\mathit{serious})$ means that the

[4] Recall that since we are considering call-by-name evaluation, both terminating and non-terminating arguments may be passed to functions.

[5] In fact as the scheme is detailed in the following section, $\lambda x\,.\,x$ could also be classified as *argument-trivial, result-serious* since trivial terms can be safely coerced to serious terms.

[6] Although products and co-products are included in Λ^{σ^+}, we omit them here since they are not required in example semantics specifications. Their addition is straightforward.

$$Terms[\Lambda_t] \quad = \quad Terms[\Lambda^{\sigma^+}]$$

$$\begin{array}{lcl} \sigma & \in & Types[\Lambda_t] \\ \sigma & ::= & \iota \mid \sigma_1 \rightarrow_{tt} \sigma_2 \mid \sigma_1 \rightarrow_{ts} \sigma_2 \mid \sigma_1 \rightarrow_{st} \sigma_2 \mid \sigma_1 \rightarrow_{ss} \sigma_2 \end{array}$$

$$\begin{array}{lcl} \theta & \in & Tags[\Lambda_t] \\ \theta & ::= & trivial \mid serious \end{array}$$

$$\begin{array}{lcl} \Gamma & \in & Assums[\Lambda_t] \\ \Gamma & ::= & \cdot \mid \Gamma, x{:}(\sigma\,,\, trivial) \mid \Gamma, x{:}(\sigma\,,\, serious) \mid \Gamma, f{:}\sigma \end{array}$$

Figure 5.5: The termination-annotated language Λ_t

term e is serious. But, if its evaluation terminates, the resulting value will be an argument-trivial, result-trivial abstraction.

Corresponding to the four function space constructors, we have four introduction rules for abstractions *(abs[tt])*, *(abs[ts])*, *(abs[st])*, *(abs[ss])* and four introduction rules for recursive abstractions *(rec[tt])*, *(rec[ts])*, *(rec[st])*, *(rec[ss])*. Tags on abstraction bodies and identifier assumptions dictate the appropriate function space type constructor.

The four elimination rules *(app[tt])*, *(app[ts])*, *(app[st])*, *(app[ss])* ensure that annotated function space types are compatible with argument and application annotations. Note that since application is strict in the function position, applications *(app[tt])*, *(app[st])* may be serious even though the applied function produces trivial results.

The rules for variables and constants are straightforward. A serious term in either argument position will cause the constructor in *(opr)* to be serious since primitive operators are strict. However, both arguments being trivial gives a trivial construct since we assume that primitive operators cannot introduce non-termination.

Since conditionals are strict in their test position, a serious test will cause the entire construct to be serious. Furthermore, since we cannot determine statically which branch of the conditional will be chosen at run-time, a serious term in either branch will cause the construct to be serious. In the eager binding construct, a serious term in the actual paramenter body will cause the construct to be serious. Note that since the actual parameter is evaluated eagerly, the formal parameter (if it binds) will be bound to a value and thus is trivial.

Finally, the generalization rule *(gen)* allows a trivial term to be coerced to a serious term.

5.5.2 Issues: correctness and precision

Having formalized the notation for classifying a term, we now consider what it means for a classification to be correct. Not every derivation $d \in TypingDerivations[\Lambda_t]$ represents a sound classification. For example, $\cdot \vdash_t (rec\, f(x).\, f\, x)\, c : (\iota\,,\, trivial)$ can be derived even though the evaluation of $(rec\, f(x).\, f\, x)\, c$ diverges.

Let $\mathcal{A} : TypedTerms[\Lambda^{\sigma^+}] \rightarrow TypingDerivations[\Lambda_t]$ denote an annotation-assigning function. $\mathcal{A}$ is considered to be correct when it satisfies the following property.

Property 5.1 (Soundness) *An annotation function $\mathcal{A}$ is* sound *iff for all $\cdot \vdash e : \sigma$, $\mathcal{A}$ yielding the derivation $\mathcal{A}[\![e]\!]\{(\sigma\,,\, trivial)\}$ implies $eval_n(e)\downarrow$ — that is, the evaluation of e terminates under call-by-name reduction.*

The *Generalization* rule may lead to more than one correct annotation for a particular term. Indeed, a trivially correct annotation can always be obtained by annotating every construct as serious. It would be useful

Identifiers:

$$(var) \quad \Gamma, x : (\sigma\ ,\ \theta\ ) \vdash_t x : (\sigma\ ,\ \theta\ )$$

$$(recvar) \quad \Gamma, f : \sigma \vdash_t f : (\sigma\ ,\ trivial)$$

Constants and Primitive Operators:

$$(const) \qquad \Gamma \vdash_t c : (\iota\ ,\ trivial)$$

$$(opr) \qquad \frac{\Gamma \vdash_t e_1 : (\iota\ ,\ \theta_1\ ) \qquad \Gamma \vdash_t e_2 : (\iota\ ,\ \theta_2\ )}{\Gamma \vdash_t \mathsf{op}\, e_1\, e_2 : (\iota\ ,\ \theta_1 \sqcup \theta_2\ )}$$

Abstractions:

$$(abs[tt]) \quad \frac{\Gamma, x : (\sigma_1\ ,\ trivial) \vdash_t e : (\sigma_2\ ,\ trivial)}{\Gamma \vdash_t \lambda x\,.\,e : (\sigma_1 \rightarrow_{tt} \sigma_2\ ,\ trivial)}$$

$$(abs[ts]) \quad \frac{\Gamma, x : (\sigma_1\ ,\ trivial) \vdash_t e : (\sigma_2\ ,\ serious\ )}{\Gamma \vdash_t \lambda x\,.\,e : (\sigma_1 \rightarrow_{ts} \sigma_2\ ,\ trivial)}$$

$$(abs[st]) \quad \frac{\Gamma, x : (\sigma_1\ ,\ serious\ ) \vdash_t e : (\sigma_2\ ,\ trivial)}{\Gamma \vdash_t \lambda x\,.\,e : (\sigma_1 \rightarrow_{st} \sigma_2\ ,\ trivial)}$$

$$(abs[ss]) \quad \frac{\Gamma, x : (\sigma_1\ ,\ serious\ ) \vdash_t e : (\sigma_2\ ,\ serious\ )}{\Gamma \vdash_t \lambda x\,.\,e : (\sigma_1 \rightarrow_{ss} \sigma_2\ ,\ trivial)}$$

Recursive Abstractions:

$$(rec[tt]) \quad \frac{\Gamma, f : \sigma_1 \rightarrow_{tt} \sigma_2, x : (\sigma_1\ ,\ trivial) \vdash_t e : (\sigma_2\ ,\ trivial)}{\Gamma \vdash_t rec\, f(x).\, e : (\sigma_1 \rightarrow_{tt} \sigma_2\ ,\ trivial)}$$

$$(rec[ts]) \quad \frac{\Gamma, f : \sigma_1 \rightarrow_{ts} \sigma_2, x : (\sigma_1\ ,\ trivial) \vdash_t e : (\sigma_2\ ,\ serious\ )}{\Gamma \vdash_t rec\, f(x).\, e : (\sigma_1 \rightarrow_{ts} \sigma_2\ ,\ trivial)}$$

$$(rec[st]) \quad \frac{\Gamma, f : \sigma_1 \rightarrow_{st} \sigma_2, x : (\sigma_1\ ,\ serious\ ) \vdash_t e : (\sigma_2\ ,\ trivial)}{\Gamma \vdash_t rec\, f(x).\, e : (\sigma_1 \rightarrow_{st} \sigma_2\ ,\ trivial)}$$

$$(rec[ss]) \quad \frac{\Gamma, f : \sigma_1 \rightarrow_{ss} \sigma_2, x : (\sigma_1\ ,\ serious\ ) \vdash_t e : (\sigma_2\ ,\ serious\ )}{\Gamma \vdash_t rec\, f(x).\, e : (\sigma_1 \rightarrow_{ss} \sigma_2\ ,\ trivial)}$$

Applications:

$$(app[tt]) \quad \frac{\Gamma \vdash_t e_0 : (\sigma_1 \rightarrow_{tt} \sigma_2\ ,\ \theta\ ) \qquad \Gamma \vdash_t e_1 : (\sigma_1\ ,\ trivial)}{\Gamma \vdash_t e_0\, e_1 : (\sigma_2\ ,\ \theta\ )}$$

$$(app[ts]) \quad \frac{\Gamma \vdash_t e_0 : (\sigma_1 \rightarrow_{ts} \sigma_2\ ,\ trivial) \qquad \Gamma \vdash_t e_1 : (\sigma_1\ ,\ trivial)}{\Gamma \vdash_t e_0\, e_1 : (\sigma_2\ ,\ serious\ )}$$

$$(app[st]) \quad \frac{\Gamma \vdash_t e_0 : (\sigma_1 \rightarrow_{st} \sigma_2\ ,\ \theta\ ) \qquad \Gamma \vdash_t e_1 : (\sigma_1\ ,\ serious\ )}{\Gamma \vdash_t e_0\, e_1 : (\sigma_2\ ,\ \theta\ )}$$

$$(app[ss]) \quad \frac{\Gamma \vdash_t e_0 : (\sigma_1 \rightarrow_{ss} \sigma_2\ ,\ trivial) \qquad \Gamma \vdash_t e_1 : (\sigma_1\ ,\ serious\ )}{\Gamma \vdash_t e_0\, e_1 : (\sigma_2\ ,\ serious\ )}$$

Figure 5.6: Totality-annotation assignment rules for simply-typed λ-terms (part 1)

Conditional and Eager Binding:

$$(cnd)\quad \frac{\Gamma \vdash_t e_1 : (\iota\,,\,\theta_1) \qquad \Gamma \vdash_t e_2 : (\sigma\,,\,\theta_2) \qquad \Gamma \vdash_t e_3 : (\sigma\,,\,\theta_3)}{\Gamma \vdash_t \mathit{if}\ e_1\ \mathit{then}\ e_2\ \mathit{else}\ e_3 : (\sigma\,,\,\theta_1 \sqcup \theta_2 \sqcup \theta_3)}$$

$$(let)\quad \frac{\Gamma \vdash_t e_0 : (\sigma_0\,,\,\theta_0) \qquad \Gamma, x : (\sigma_0\,,\,\mathit{trivial}) \vdash_t e_1 : (\sigma_1\,,\,\theta_1)}{\Gamma \vdash_t \mathit{let}\ x \Leftarrow e_0\ \mathit{in}\ e_1 : (\sigma_1\,,\,\theta_0 \sqcup \theta_1)}$$

Generalization:

$$(gen)\quad \frac{\Gamma \vdash_t e : (\sigma\,,\,\mathit{trivial})}{\Gamma \vdash_t e : (\sigma\,,\,\mathit{serious})}$$

Figure 5.7: Totality-annotation assignment rules for simply-typed λ-terms (part 2)

to have a formalized notion of annotation *precision* based on the degree of "seriousness" in an annotation.[7] Note that it is impossible to compute a "completely precise" annotation function $\mathcal{A}$ (*i.e.*, one that never annotates a terminating term as serious) since this would be equivalent to solving the halting problem.

5.5.3 Automatic assignment of correct annotations

Correct annotations can be assigned automatically using well-known *static analysis* techniques. Termination analyses based on the technique of *abstract interpretation* are presented in [1, 67]. However, the technique of *non-standard type inference* seems most appropriate since we have formalized annotations via types. Based on preliminary study, it seems that our rules can be adapted to a framework similar to Jensen's [45] or Solberg's [90] in a straightforward manner. The only non-trivial work likes in properly formalizing the rules for *rec* so as to disallow unsound derivations like the example shown in the previous section.

5.5.4 Abbreviating annotations

We use the following abbreviations for annotations.

- $\lambda_{tt} x.e$ indicates an abstraction introduced by the *(abs[tt])* rule.
- $\lambda_{ts} x.e$ indicates an abstraction introduced by the *(abs[ts])* rule.
- $\lambda_{st} x.e$ indicates an abstraction introduced by the *(abs[st])* rule.
- $\lambda_{ss} x.e$ indicates an abstraction introduced by the *(abs[ss])* rule.

The abbreviations for recursive abstractions are similar.

- $e_0 \bullet_{tt} e_1$ indicates an application introduced by the *(app[tt])* rule.
- $e_0 \bullet_{ts} e_1$ indicates an application introduced by the *(app[ts])* rule.
- $e_0 \bullet_{st} e_1$ indicates an application introduced by the *(app[st])* rule.
- $e_0 \bullet_{ss} e_1$ indicates an application introduced by the *(app[ss])* rule.

Finally, x_t indicates that the tag *trivial* was associated with x in the rule *(var)*; x_s indicates that the tag *serious* was associated with x in the rule *(var)*.

Serious Terms:

$$
\begin{array}{rcl}
\mathcal{C}_t\langle\!|\cdot|\!\rangle & : & \mathit{TypingDerivations}[\Lambda_t] \rightarrow \mathit{Terms}[\Lambda^{\sigma^+}] \\
\mathcal{C}_t\langle\!|\mathit{d}\{e : (\sigma\ ,\ \mathit{trivial})\}|\!\rangle & = & \lambda k\,.\,k\,\mathcal{C}_t\langle\mathit{d}\{e : (\sigma\ ,\ \mathit{trivial})\}\rangle \\
\mathcal{C}_t\langle\!|(\mathit{var}) :: x : (\sigma\ ,\ \mathit{serious})|\!\rangle & = & x \\
\mathcal{C}_t\langle\!|(\mathit{opr}) :: \mathsf{op}\, e_1\, e_2 : (\iota\ ,\ \mathit{serious})|\!\rangle & = & \lambda k\,.\,\mathcal{C}_t\langle\!|d_1|\!\rangle\,(\lambda y_1\,.\,\mathcal{C}_t\langle\!|d_2|\!\rangle\,(\lambda y_2\,.\,k\,(\mathsf{op}\, y_1\, y_2))) \\
\mathcal{C}_t\langle\!|(\mathit{app}[tt]) :: e_0\, e_1 : (\sigma_2\ ,\ \mathit{serious})|\!\rangle & = & \lambda k\,.\,\mathcal{C}_t\langle\!|d_0|\!\rangle\,(\lambda y_0\,.\,k\,(y_0\,\mathcal{C}_t\langle d_1\rangle)) \\
\mathcal{C}_t\langle\!|(\mathit{app}[tt]) :: e_0\, e_1 : (\sigma_2\ ,\ \mathit{serious})|\!\rangle & = & \lambda k\,.\,\mathcal{C}_t\langle\!|d_0|\!\rangle\,(\lambda y_0\,.\,k\,(y_0\,\mathcal{C}_t\langle\!|d_1|\!\rangle)) \\
\mathcal{C}_t\langle\!|(\mathit{app}[ts]) :: e_0\, e_1 : (\sigma_2\ ,\ \mathit{serious})|\!\rangle & = & \lambda k\,.\,\mathcal{C}_t\langle\!|d_0|\!\rangle\,(\lambda y_0\,.\,(y_0\,\mathcal{C}_t\langle d_1\rangle)\,k) \\
\mathcal{C}_t\langle\!|(\mathit{app}[ss]) :: e_0\, e_1 : (\sigma_2\ ,\ \mathit{serious})|\!\rangle & = & \lambda k\,.\,\mathcal{C}_t\langle\!|d_0|\!\rangle\,(\lambda y_0\,.\,(y_0\,\mathcal{C}_t\langle\!|d_1|\!\rangle)\,k) \\
\mathcal{C}_t\langle\!|(\mathit{cnd}) :: \mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2 : (\sigma\ ,\ \mathit{serious})|\!\rangle & = & \lambda k\,.\,\mathcal{C}_t\langle\!|d_0|\!\rangle\,(\lambda y_0\,.\,\mathit{if}\ y_0\ \mathit{then}\ \mathcal{C}_t\langle\!|d_1|\!\rangle\ k\ \mathit{else}\ \mathcal{C}_t\langle\!|d_2|\!\rangle\ k) \\
\mathcal{C}_t\langle\!|(\mathit{let}) :: \mathit{let}\ x \Leftarrow e_0\ \mathit{in}\ e_1 : (\sigma_1\ ,\ \mathit{serious})|\!\rangle & = & \lambda k\,.\,\mathcal{C}_t\langle\!|d_0|\!\rangle\,(\lambda x\,.\,\mathcal{C}_t\langle\!|d_1|\!\rangle\ k) \\
\mathcal{C}_t\langle\!|(\mathit{gen}) :: e : (\sigma\ ,\ \mathit{serious})|\!\rangle & = & \lambda k\,.\,k\,\mathcal{C}_t\langle d\rangle
\end{array}
$$

Trivial Terms:

$$
\begin{array}{rcl}
\mathcal{C}_t\langle\cdot\rangle & : & \mathit{TypingDerivations}[\Lambda_t] \rightarrow \mathit{Terms}[\Lambda^{\sigma^+}] \\
\mathcal{C}_t\langle(\mathit{var}) :: x : (\sigma\ ,\ \mathit{trivial})\rangle & = & x \\
\mathcal{C}_t\langle(\mathit{recvar}) :: f : (\sigma\ ,\ \mathit{trivial})\rangle & = & f \\
\mathcal{C}_t\langle(\mathit{const}) :: c : (\iota\ ,\ \mathit{trivial})\rangle & = & c \\
\mathcal{C}_t\langle(\mathit{opr}) :: \mathsf{op}\, e_1\, e_2 : (\iota\ ,\ \mathit{trivial})\rangle & = & \mathsf{op}\,\mathcal{C}_t\langle d_1\rangle\,\mathcal{C}_t\langle d_2\rangle \\
\mathcal{C}_t\langle(\mathit{abs}[tt]) :: \lambda x\,.\,e : (\sigma_1 \rightarrow_{tt} \sigma_2\ ,\ \mathit{trivial})\rangle & = & \lambda x\,.\,\mathcal{C}_t\langle d\rangle \\
\mathcal{C}_t\langle(\mathit{abs}[st]) :: \lambda x\,.\,e : (\sigma_1 \rightarrow_{st} \sigma_2\ ,\ \mathit{trivial})\rangle & = & \lambda x\,.\,\mathcal{C}_t\langle d\rangle \\
\mathcal{C}_t\langle(\mathit{abs}[ts]) :: \lambda x\,.\,e : (\sigma_1 \rightarrow_{ts} \sigma_2\ ,\ \mathit{trivial})\rangle & = & \lambda x\,.\,\mathcal{C}_t\langle\!|d|\!\rangle \\
\mathcal{C}_t\langle(\mathit{abs}[ss]) :: \lambda x\,.\,e : (\sigma_1 \rightarrow_{ss} \sigma_2\ ,\ \mathit{trivial})\rangle & = & \lambda x\,.\,\mathcal{C}_t\langle\!|d|\!\rangle \\
\mathcal{C}_t\langle(\mathit{rec}[tt]) :: \mathit{rec}\, f(x).\, e : (\sigma_1 \rightarrow_{tt} \sigma_2\ ,\ \mathit{trivial})\rangle & = & \mathit{rec}\, f(x).\,\mathcal{C}_t\langle d\rangle \\
\mathcal{C}_t\langle(\mathit{rec}[st]) :: \mathit{rec}\, f(x).\, e : (\sigma_1 \rightarrow_{st} \sigma_2\ ,\ \mathit{trivial})\rangle & = & \mathit{rec}\, f(x).\,\mathcal{C}_t\langle d\rangle \\
\mathcal{C}_t\langle(\mathit{rec}[ts]) :: \mathit{rec}\, f(x).\, e : (\sigma_1 \rightarrow_{ts} \sigma_2\ ,\ \mathit{trivial})\rangle & = & \mathit{rec}\, f(x).\,\mathcal{C}_t\langle\!|d|\!\rangle \\
\mathcal{C}_t\langle(\mathit{rec}[ss]) :: \mathit{rec}\, f(x).\, e : (\sigma_1 \rightarrow_{ss} \sigma_2\ ,\ \mathit{trivial})\rangle & = & \mathit{rec}\, f(x).\,\mathcal{C}_t\langle\!|d|\!\rangle \\
\mathcal{C}_t\langle(\mathit{app}[tt]) :: e_0\, e_1 : (\sigma_2\ ,\ \mathit{trivial})\rangle & = & \mathcal{C}_t\langle d_0\rangle\,\mathcal{C}_t\langle d_1\rangle \\
\mathcal{C}_t\langle(\mathit{app}[st]) :: e_0\, e_1 : (\sigma_2\ ,\ \mathit{trivial})\rangle & = & \mathcal{C}_t\langle d_0\rangle\,\mathcal{C}_t\langle\!|d_1|\!\rangle \\
\mathcal{C}_t\langle(\mathit{cnd}) :: \mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2 : (\sigma\ ,\ \mathit{trivial})\rangle & = & \mathit{if}\ \mathcal{C}_t\langle d_0\rangle\ \mathit{then}\ \mathcal{C}_t\langle d_1\rangle\ \mathit{else}\ \mathcal{C}_t\langle d_2\rangle \\
\mathcal{C}_t\langle(\mathit{let}) :: \mathit{let}\ x \Leftarrow e_0\ \mathit{in}\ e_1 : (\sigma_1\ ,\ \mathit{trivial})\rangle & = & \mathit{let}\ x \Leftarrow \mathcal{C}_t\langle d_0\rangle\ \mathit{in}\ \mathcal{C}_t\langle d_1\rangle
\end{array}
$$

Types and Assumptions:

$$
\begin{array}{rcl}
\mathcal{C}_t\langle\!|\cdot|\!\rangle & : & \mathit{Types}[\Lambda_t] \rightarrow \mathit{Types}[\Lambda^{\sigma^+}] \\
\mathcal{C}_t\langle\!|\sigma|\!\rangle & = & \neg\neg\mathcal{C}_t\langle\sigma\rangle \\
\mathcal{C}_t\langle\iota\rangle & = & \iota \\
\mathcal{C}_t\langle\sigma_0 \rightarrow_{tt} \sigma_1\rangle & = & \mathcal{C}_t\langle\sigma_0\rangle \rightarrow \mathcal{C}_t\langle\sigma_1\rangle \\
\mathcal{C}_t\langle\sigma_0 \rightarrow_{ts} \sigma_1\rangle & = & \mathcal{C}_t\langle\sigma_0\rangle \rightarrow \mathcal{C}_t\langle\!|\sigma_1|\!\rangle \\
\mathcal{C}_t\langle\sigma_0 \rightarrow_{st} \sigma_1\rangle & = & \mathcal{C}_t\langle\!|\sigma_0|\!\rangle \rightarrow \mathcal{C}_t\langle\sigma_1\rangle \\
\mathcal{C}_t\langle\sigma_0 \rightarrow_{ss} \sigma_1\rangle & = & \mathcal{C}_t\langle\!|\sigma_0|\!\rangle \rightarrow \mathcal{C}_t\langle\!|\sigma_1|\!\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{C}_t\langle\!|\cdot|\!\rangle & : & \mathit{Assums}[\Lambda_t] \rightarrow \mathit{Assums}[\Lambda^{\sigma^+}] \\
\mathcal{C}_t\langle\!|\Gamma, x : (\sigma\ ,\ \mathit{serious})|\!\rangle & = & \mathcal{C}_t\langle\!|\Gamma|\!\rangle, x : \mathcal{C}_t\langle\!|\sigma|\!\rangle \\
\mathcal{C}_t\langle\!|\Gamma, x : (\sigma\ ,\ \mathit{trivial})|\!\rangle & = & \mathcal{C}_t\langle\!|\Gamma|\!\rangle, x : \mathcal{C}_t\langle\sigma\rangle \\
\mathcal{C}_t\langle\!|\Gamma, f : \sigma|\!\rangle & = & \mathcal{C}_t\langle\!|\Gamma|\!\rangle, f : \mathcal{C}_t\langle\sigma\rangle
\end{array}
$$

Figure 5.8: Call-by-name CPS transformation $\mathcal{C}_t$ for the termination annotated language Λ_t

5.6 A CPS transformation directed by termination properties

Figure 5.8 displays the new call-by-name transformation $\mathcal{C}_t$. The transformation is directed by termination information captured in the annotated type system. $\mathcal{C}_t\langlebar\cdot\ranglebar$ transforms serious terms; $\mathcal{C}_t\langle\cdot\rangle$ transforms trivial terms. Note that no continuation-passing is required for evaluation-order independence of trivial terms.

The following proposition states the relationship between Λ_t typing derivations and the types of terms in the image of $\mathcal{C}_t$.

Property 5.2 *For all* $d \in \mathit{TypingDerivations}[\Lambda_t]$,

- *If* $d\{\Gamma \vdash_t e : (\sigma\ ,\ \mathit{serious})\}$ *then* $\mathcal{C}_t\langlebar[\![\Gamma]\!]\ranglebar \vdash \mathcal{C}_t\langlebar[\![d]\!]\ranglebar : \mathcal{C}_t\langlebar[\![\sigma]\!]\ranglebar$.
- *If* $d\{\Gamma \vdash_t e : (\sigma\ ,\ \mathit{trivial})\}$ *then* $\mathcal{C}_t\langlebar[\![\Gamma]\!]\ranglebar \vdash \mathcal{C}_t\langle d\rangle : \mathcal{C}_t\langle\sigma\rangle$.

Proof: by induction over the structure of d. ∎

The following theorem expresses the correctness of $\mathcal{C}_t$.

Theorem 5.1 (Simulation and Indifference for $\mathcal{C}_t$) *For all programs* $\cdot \vdash e : \sigma$ *and for all correct annotation-assigning functions* $\mathcal{A}$,

1. **Indifference:** $\mathit{eval}_v((\mathcal{C}_t \circ \mathcal{A})\langlebar[\![e]\!]\ranglebar\ (\lambda y\ .\ y)) \simeq_{obs} \mathit{eval}_n((\mathcal{C}_t \circ \mathcal{A})\langlebar[\![e]\!]\ranglebar\ (\lambda y\ .\ y))$
2. **Simulation:** $\mathit{eval}_n(e) \simeq_{obs} \mathit{eval}_v((\mathcal{C}_t \circ \mathcal{A})\langlebar[\![e]\!]\ranglebar\ (\lambda y\ .\ y))$

Proof: (These proofs remain to be typed in.) ∎

5.6.1 Assessment

Restricting the new transformation $\mathcal{C}_t$ to serious λ-terms yields the call-by-name CPS transformation $\mathcal{C}_n$ (see Figure 2.9). Conversely, restricting $\mathcal{C}_t$ to trivial λ-terms yields the identity transformation — no continuations are needed at all (*e.g.*, the denotational semantics of a strongly-normalizing language or of the language of Figure 5.2 without the "while" statement). Thus, $\mathcal{C}_t$ generalizes both the call-by-name transformation $\mathcal{C}_n$ and the identity transformation.

On a practical note, the definition of $\mathcal{C}_t$ in Figure 5.8 does not take full advantage of the termination information. For example, in the serious version of op, both arguments are treated as serious even if only one argument is serious. This has the effect of introducing extra administrative redexes. The introduction of these redexes for op and similar constructs can be avoided by adding extra cases to the translation.

5.7 Annotating the direct semantics specification

Now we return to the direct semantics specification of the simple imperative language (Figure 5.3) and analyze its termination properties.

Property 5.3 *The function types of* $\mathcal{Z}_d[\![\text{Program}]\!]$, $\mathcal{C}_d[\![\text{Command}]\!]$, *and* $\mathcal{E}_d[\![\text{Expression}]\!]$ *are argument-trivial and result-trivial.*

Justification: Each is argument-trivial because it is not possible for an argument expression of type *Env* to diverge. Each is result-trivial because a λ-abstraction (a value) is always returned. ∎

Property 5.4 *The function type* Com_d *is argument-trivial and result-serious.*

[7] A formalization is tangential to our purpose and omitted.

Justification: Due to the eager binding of the *let* construct, each command is passed a reduced store value. Therefore, the function type can be classified as argument-trivial. Com_d is result-serious because looping may occur in the *while*. ∎

Property 5.5 *The function type Exp_d is argument-trivial and result-trivial.*

Justification: Due to argument-trivial property of Com_d, each expression is passed a reduced store value. Therefore, the function type can be classified as argument-trivial. No expression contains components which may loop. Therefore Exp_d are result-trivial. ∎

Property 5.6 *$Proc_d$ and Fun_d are argument-trivial and result-trivial.*

Justification: The arguments to procedures and functions originate from the evaluation of expressions which can never loop. Therefore, $Proc_d$ and Fun_d function spaces are classified as argument-trivial. They are result trivial because they both return abstractions (values). ∎

Figure 5.9 presents a version of the direct semantics (Figure 5.3) that is annotated according to the properties above. We use the abbreviations for annotations outlined in Section 5.5.4. Any reasonable implementation of $\mathcal{A}$ as outlined in Section 5.5.3 would assign such annotations automatically.

5.8 Transforming the annotated direct specification

The following theorem captures the fact that transforming the annotated direct specification in Figure 5.9 into continuation-passing-style does yield the the continuation specification in Figure 5.4, after administrative reductions.

Theorem 5.2 *For all $z \in$ Program, $c \in$ Command, and $e \in$ Expression,*

$$\begin{array}{lcl} \mathcal{C}_t\langle\mathcal{Z}_d[\![z]\!]\rangle & \longrightarrow\!\!\!\!\!\rightarrow_{\beta adm} & \mathcal{Z}_k[\![z]\!] \\ \mathcal{C}_t\langle\mathcal{C}_d[\![c]\!]\rangle & \longrightarrow\!\!\!\!\!\rightarrow_{\beta adm} & \mathcal{C}_k[\![c]\!] \\ \mathcal{C}_t\langle\mathcal{E}_d[\![e]\!]\rangle & = & \mathcal{E}_k[\![e]\!] \end{array}$$

Proof: by simultaneous induction over the structure of $z \in$ Program, $c \in$ Command, and $e \in$ Expression. Illustrative cases are given below.

case $z \equiv \text{proc } p\,(m) = c \text{ in } z$:

$$\begin{array}{lll} & \mathcal{C}_t\langle\mathcal{Z}_d[\![\text{proc } p\,(m) = c \text{ in } z]\!]\rangle & \\ = & \mathcal{C}_t\langle\lambda_{tt}\rho.\lambda_{ts}\sigma.\mathcal{Z}_d[\![z]\!]\bullet_{tt}\,(\mathsf{ext}\;\rho_t\;p\;(\lambda_{tt}i.\mathcal{C}_d[\![c]\!]\bullet_{tt}\,(\mathsf{ext}\;\rho_t\;m\;i_t)))\bullet_{ts}\,\sigma_t\rangle & \\ = & \lambda\rho\,.\,\lambda\sigma\,.\,\mathcal{C}_t\langle[\![\mathcal{Z}_d[\![z]\!]\bullet_{tt}\,(\mathsf{ext}\;\rho_t\;p\;(\lambda_{tt}i.\mathcal{C}_d[\![c]\!]\bullet_{tt}\,(\mathsf{ext}\;\rho_t\;m\;i_t)))\bullet_{ts}\,\sigma_t]\!]\rangle & \\ = & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda k\,.\,\mathcal{C}_t\langle[\![\mathcal{Z}_d[\![z]\!]\bullet_{tt}\,(\mathsf{ext}\;\rho_t\;p\;(\lambda_{tt}i.\mathcal{C}_d[\![c]\!]\bullet_{tt}\,(\mathsf{ext}\;\rho_t\;m\;i_t)))]\!]\,(\lambda y\,.\,(y\,\mathcal{C}_t\langle\sigma_t\rangle)\,k) & \\ = & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda k\,.\,(\lambda k\,.\,k\,(\mathcal{C}_t\langle[\![\mathcal{Z}_d[\![z]\!]\bullet_{tt}\,(\mathsf{ext}\;\rho_t\;p\;(\lambda_{tt}i.\mathcal{C}_d[\![c]\!]\bullet_{tt}\,(\mathsf{ext}\;\rho_t\;m\;i_t)))]\!]\rangle))\,(\lambda y\,.\,(y\,\sigma)\,k) & \\ = & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda k\,.\,(\lambda k\,.\,k\,(\mathcal{C}_t\langle\mathcal{Z}_d[\![z]\!]\rangle\,(\mathsf{ext}\;\rho\;p\;(\lambda i\,.\,\mathcal{C}_t\langle\mathcal{C}_d[\![c]\!]\rangle\,(\mathsf{ext}\;\rho\;m\;i)))))\,(\lambda y\,.\,(y\,\sigma)\,k) & \\ \longrightarrow\!\!\!\!\!\rightarrow_{\beta adm} & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda k\,.\,\mathcal{C}_t\langle\mathcal{Z}_d[\![z]\!]\rangle\,(\mathsf{ext}\;\rho\;p\;(\lambda i\,.\,\mathcal{C}_t\langle\mathcal{C}_d[\![c]\!]\rangle\,(\mathsf{ext}\;\rho\;m\;i)))\,\sigma\,k & \\ \longrightarrow\!\!\!\!\!\rightarrow_{\beta adm} & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda k\,.\,\mathcal{Z}_k[\![z]\!]\,(\mathsf{ext}\;\rho\;p\;(\lambda i\,.\,\mathcal{C}_k[\![c]\!]\,(\mathsf{ext}\;\rho\;m\;i)))\,\sigma\,k & \textit{...by ind. hyp.} \\ = & \mathcal{Z}_k[\![\text{proc } p\,(m) = c \text{ in } z]\!] & \end{array}$$

case $c \equiv c_1\ ;\ c_2$:

Valuation Functions:

$$\begin{aligned}
\mathcal{Z}_d[\![\text{Program}]\!] &: Env \to_{tt} Com_d \\
\mathcal{C}_d[\![\text{Command}]\!] &: Env \to_{tt} Com_d \\
\mathcal{E}_d[\![\text{Expression}]\!] &: Env \to_{tt} Exp_d \\
\mathcal{N}_d[\![\text{Numeral}]\!] &: Nat \\
\mathcal{L}_d[\![\text{Location}]\!] &: Loc
\end{aligned}$$

Semantic Domains:

$$\begin{aligned}
Com_d &= Store \to_{ts} Store \\
Exp_d &= Store \to_{tt} Nat \\
Proc_d &= Nat \to_{tt} Com_d \\
Fun_d &= Nat \to_{tt} Exp_d
\end{aligned}$$

Programs:

$$\begin{aligned}
\mathcal{Z}_d[\![\text{proc } p\ (m) = c \text{ in } z]\!] &= \lambda_{tt}\rho.\lambda_{ts}\sigma.\mathcal{Z}_d[\![z]\!] \bullet_{tt} (\mathsf{ext}\ \rho_t\ p\ (\lambda_{tt} i.\mathcal{C}_d[\![c]\!] \bullet_{tt} (\mathsf{ext}\ \rho_t\ m\ i_t))) \bullet_{ts} \sigma_t \\
\mathcal{Z}_d[\![\text{fun } f\ (m) = e \text{ in } z]\!] &= \lambda_{tt}\rho.\lambda_{ts}\sigma.\mathcal{Z}_d[\![z]\!] \bullet_{tt} (\mathsf{ext}\ \rho_t\ f\ (\lambda_{tt} i.\mathcal{E}_d[\![e]\!] \bullet_{tt} (\mathsf{ext}\ \rho_t\ m\ i_t))) \bullet_{ts} \sigma_t \\
\mathcal{Z}_d[\![c.]\!] &= \lambda_{tt}\rho.\lambda_{ts}\sigma.\mathcal{C}_d[\![c]\!] \bullet_{tt} \rho_t \bullet_{ts} \sigma_t
\end{aligned}$$

Commands:

$$\begin{aligned}
\mathcal{C}_d[\![\text{skip}]\!] &= \lambda_{tt}\rho.\lambda_{ts}\sigma.\sigma_t \\
\mathcal{C}_d[\![c_1\ ;\ c_2]\!] &= \lambda_{tt}\rho.\lambda_{ts}\sigma.let_t\ \sigma' = \mathcal{C}_d[\![c_1]\!] \bullet_{tt} \rho_t \bullet_{ts} \sigma_t\ in\ \mathcal{C}_d[\![c_2]\!] \bullet_{tt} \rho_t \bullet_{ts} \sigma'_t \\
\mathcal{C}_d[\![l := e]\!] &= \lambda_{tt}\rho.\lambda_{ts}\sigma.\mathsf{upd}\ \sigma_t\ \mathcal{L}_d[\![l]\!]\ (\mathcal{E}_d[\![e]\!] \bullet_{tt} \rho_t \bullet_{tt} \sigma_t) \\
\mathcal{C}_d[\![\text{if } e \text{ then } c_1 \text{ else } c_2]\!] &= \lambda_{tt}\rho.\lambda_{ts}\sigma.\ \begin{array}{l} if\quad \mathsf{iszero?}\ (\mathcal{E}_d[\![e]\!] \bullet_{tt} \rho_t \bullet_{tt} \sigma_t) \\ then\ (\mathcal{C}_d[\![c_1]\!] \bullet_{tt} \rho_t \bullet_{ts} \sigma_t) \\ else\ \ (\mathcal{C}_d[\![c_2]\!] \bullet_{tt} \rho_t \bullet_{ts} \sigma_t) \end{array} \\
\mathcal{C}_d[\![\text{while } e \text{ do } c]\!] &= \lambda_{tt}\rho.\lambda_{ts}\sigma.(rec_{ts}\, w(\sigma).\ \begin{array}{l} if\quad \mathsf{iszero?}\ (\mathcal{E}_d[\![e]\!] \bullet_{tt} \rho_t \bullet_{tt} \sigma_t) \\ then\ let\ \sigma' = \mathcal{C}_d[\![c]\!] \bullet_{tt} \rho_t \bullet_{ts} \sigma_t\ in\ w_t \bullet_{ts} \sigma'_t \\ else\ \ \sigma_t) \bullet_{ts} \sigma_t \end{array} \\
\mathcal{C}_d[\![\text{call } p\ (e)]\!] &= \lambda_{tt}\rho.\lambda_{ts}\sigma.(\mathsf{lookup}\ \rho_t\ p) \bullet_{tt} (\mathcal{E}_d[\![e]\!] \bullet_{tt} \rho_t \bullet_{tt} \sigma_t) \bullet_{ts} \sigma_t
\end{aligned}$$

Expressions:

$$\begin{aligned}
\mathcal{E}_d[\![n]\!] &= \lambda_{tt}\rho.\lambda_{tt}\sigma.\mathcal{N}_d[\![n]\!] \\
\mathcal{E}_d[\![m]\!] &= \lambda_{tt}\rho.\lambda_{tt}\sigma.\mathsf{lookup}\ \rho_t\ m \\
\mathcal{E}_d[\![\text{succ } e]\!] &= \lambda_{tt}\rho.\lambda_{tt}\sigma.\mathsf{succ}\ (\mathcal{E}_d[\![e]\!] \bullet_{tt} \rho_t \bullet_{tt} \sigma_t) \\
\mathcal{E}_d[\![\text{pred } e]\!] &= \lambda_{tt}\rho.\lambda_{tt}\sigma.\mathsf{pred}\ (\mathcal{E}_d[\![e]\!] \bullet_{tt} \rho_t \bullet_{tt} \sigma_t) \\
\mathcal{E}_d[\![\text{deref } l]\!] &= \lambda_{tt}\rho.\lambda_{tt}\sigma.\mathsf{fetch}\ \sigma_t\ \mathcal{L}_d[\![l]\!] \\
\mathcal{E}_d[\![\text{apply } f\ (e)]\!] &= \lambda_{tt}\rho.\lambda_{tt}\sigma.(\mathsf{lookup}\ \rho_t\ f) \bullet_{tt} (\mathcal{E}_d[\![e]\!] \bullet_{tt} \rho_t \bullet_{tt} \sigma_t) \bullet_{tt} \sigma_t
\end{aligned}$$

Figure 5.9: Annotated direct semantics specification

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{C}_t\langle\mathcal{C}_d[\![c_1\ ;\ c_2]\!]\rangle} \\
= & \mathcal{C}_t\langle\lambda_{tt}\rho.\lambda_{ts}\sigma.let\ \sigma' \Leftarrow \mathcal{C}_d[\![c_1]\!]\bullet_{tt}\rho_t\bullet_{ts}\sigma_t\ in\ \mathcal{C}_d[\![c_2]\!]\bullet_{tt}\rho_t\bullet_{ts}\sigma'_t\rangle \\
= & \lambda\rho\,.\,\lambda\sigma\,.\,\mathcal{C}_t\langle\!\langle let\ \sigma' \Leftarrow \mathcal{C}_d[\![c_1]\!]\bullet_{tt}\rho_t\bullet_{ts}\sigma_t\ in\ \mathcal{C}_d[\![c_2]\!]\bullet_{tt}\rho_t\bullet_{ts}\sigma'_t\rangle\!\rangle \\
= & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda k\,.\,\mathcal{C}_t\langle\!\langle\mathcal{C}_d[\![c_1]\!]\bullet_{tt}\rho_t\bullet_{ts}\sigma_t\rangle\!\rangle\,(\lambda\sigma'\,.\,\mathcal{C}_t\langle\!\langle\mathcal{C}_d[\![c_2]\!]\bullet_{tt}\rho_t\bullet_{ts}\sigma'_t\rangle\!\rangle\,k) \\
= & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda k\,.\,(\lambda k\,.\,\mathcal{C}_t\langle\!\langle\mathcal{C}_d[\![c_1]\!]\bullet_{tt}\rho_t\rangle\!\rangle\,(\lambda y\,.\,(y\,\mathcal{C}_t\langle\sigma_t\rangle)\,k))\,(\lambda\sigma'\,.\,\mathcal{C}_t\langle\!\langle\mathcal{C}_d[\![c_2]\!]\bullet_{tt}\rho_t\bullet_{ts}\sigma'_t\rangle\!\rangle\,k) \\
= & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda k\,.\,(\lambda k\,.\,(\lambda k\,.\,k\,(\mathcal{C}_t\langle\mathcal{C}_d[\![c_1]\!]\rangle\,\rho))\,(\lambda y\,.\,(y\,\sigma)\,k))\,(\lambda\sigma'\,.\,\mathcal{C}_t\langle\!\langle\mathcal{C}_d[\![c_2]\!]\bullet_{tt}\rho_t\bullet_{ts}\sigma'_t\rangle\!\rangle\,k) \\
\longrightarrow\!\!\!\!\!\rightarrow_{\beta_{adm}} & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda k\,.\,(\lambda k\,.\,\mathcal{C}_t\langle\mathcal{C}_d[\![c_1]\!]\rangle\,\rho\,\sigma\,k)\,(\lambda\sigma'\,.\,\mathcal{C}_t\langle\!\langle\mathcal{C}_d[\![c_2]\!]\bullet_{tt}\rho_t\bullet_{ts}\sigma'_t\rangle\!\rangle\,k) \\
\longrightarrow\!\!\!\!\!\rightarrow_{\beta_{adm}} & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda k\,.\,(\lambda k\,.\,\mathcal{C}_t\langle\mathcal{C}_d[\![c_1]\!]\rangle\,\rho\,\sigma\,k)\,(\lambda\sigma'\,.\,\mathcal{C}_t\langle\mathcal{C}_d[\![c_2]\!]\rangle\,\rho\,\sigma'\,k) \\
\longrightarrow_{\beta_{adm}} & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda k\,.\,\mathcal{C}_t\langle\mathcal{C}_d[\![c_1]\!]\rangle\,\rho\,\sigma\,(\lambda\sigma'\,.\,\mathcal{C}_t\langle\mathcal{C}_d[\![c_2]\!]\rangle\,\rho\,\sigma'\,k) \\
\longrightarrow\!\!\!\!\!\rightarrow_{\beta_{adm}} & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda k\,.\,\mathcal{C}_k[\![c_1]\!]\,\rho\,\sigma\,(\lambda\sigma'\,.\,\mathcal{C}_k[\![c_2]\!]\,\rho\,\sigma'\,k) \qquad \textit{...by ind. hyp.} \\
= & \mathcal{C}_k[\![c_1\ ;\ c_2]\!]
\end{array}
$$

case $c \equiv l := e$:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{C}_t\langle\mathcal{C}_d[\![l := e]\!]\rangle} \\
= & \mathcal{C}_t\langle\lambda_{tt}\rho.\lambda_{ts}\sigma.\mathsf{upd}\ \sigma_t\ \mathcal{L}_d[\![l]\!]\ (\mathcal{E}_d[\![e]\!]\bullet_{tt}\rho_t\bullet_{tt}\sigma_t)\rangle \\
= & \lambda\rho\,.\,\lambda\sigma\,.\,\mathcal{C}_t\langle\!\langle\mathsf{upd}\ \sigma_t\ \mathcal{L}_d[\![l]\!]\ (\mathcal{E}_d[\![e]\!]\bullet_{tt}\rho_t\bullet_{tt}\sigma_t)\rangle\!\rangle \\
= & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda k\,.\,k\,\mathcal{C}_t\langle\mathsf{upd}\ \sigma_t\ \mathcal{L}_d[\![l]\!]\ (\mathcal{E}_d[\![e]\!]\bullet_{tt}\rho_t\bullet_{tt}\sigma_t)\rangle \\
= & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda k\,.\,k\,(\mathsf{upd}\ \sigma\ \mathcal{L}_d[\![l]\!]\ (\mathcal{C}_t\langle\mathcal{E}_d[\![e]\!]\rangle\,\rho\,\sigma)) \\
= & \lambda\rho\,.\,\lambda\sigma\,.\,\lambda k\,.\,k\,(\mathsf{upd}\ \sigma\ \mathcal{L}_k[\![l]\!]\ (\mathcal{E}_k[\![e]\!]\,\rho\,\sigma)) \qquad \textit{...by ind. hyp.} \\
= & \mathcal{C}_k[\![l := e]\!]
\end{array}
$$

case $e \equiv \mathrm{succ}\ e$:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{C}_t\langle\mathcal{E}_d[\![\mathrm{succ}\ e]\!]\rangle} \\
= & \mathcal{C}_t\langle\lambda_{tt}\rho.\lambda_{tt}\sigma.\mathsf{succ}\ (\mathcal{E}_d[\![e]\!]\bullet_{tt}\rho_t\bullet_{tt}\sigma_t)\rangle \\
= & \lambda\rho\,.\,\lambda\sigma\,.\,\mathcal{C}_t\langle\mathsf{succ}\ (\mathcal{E}_d[\![e]\!]\bullet_{tt}\rho_t\bullet_{tt}\sigma_t)\rangle \\
= & \lambda\rho\,.\,\lambda\sigma\,.\,\mathsf{succ}\ (\mathcal{C}_t\langle\mathcal{E}_d[\![e]\!]\rangle\,\rho\,\sigma) \\
= & \lambda\rho\,.\,\lambda\sigma\,.\,\mathsf{succ}\ (\mathcal{E}_k[\![e]\!]\,\rho\,\sigma) \qquad \textit{...by ind. hyp.} \\
= & \mathcal{E}_k[\![\mathrm{succ}\ e]\!]
\end{array}
$$

∎

Further, we now have the ability to specify other kinds of continuation semantics. For example, we could classify Exp_d to be result-serious (this would happen if recursive functions were allowed in the simple imperative language). This classification gives a continuation semantics where the valuation functions for both commands and expressions are in continuation style.

5.8.1 Continuation style and evaluation-order independence

It is interesting to compare the structure of the terms produced by $\mathcal{C}_t$ with the structure of the terms produced by Plotkin's original continuation-style transformations $\mathcal{C}_n$ and $\mathcal{C}_v$ as presented in the preceding chapters. In addition to satisfying the **Simulation** and **Indifference** theorems, conventional CPS terms (*i.e.*, those produced by $\mathcal{C}_n$ and $\mathcal{C}_v$) have two properties that are often utilized for implementation purposes:

- all function calls are in "tail" position;[8]
- all intermediate values are given names.

In contrast, $\mathcal{C}_t$ inserts just enough continuations to preserve call-by-name meaning under both call-by-name and call-by-value reduction. Thus, $\mathcal{C}_t$ satisfies **Simulation** and **Indifference** theorems, but the two additional properties above are lost because some applications may be trivial.

- Trivial applications may occur as function arguments — not a "tail" position.
- Trivial applications yield intermediate values that are not named — since result-trivial functions are not passed any continuation.

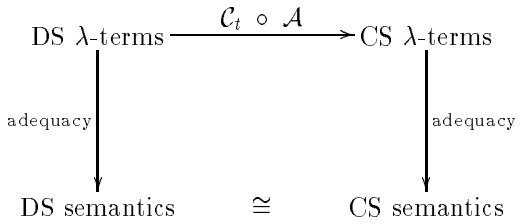
In other words, our transformation $\mathcal{C}_t$ does not produce conventional CPS terms, but it does produce evaluation-order-independent terms that employ some degree of continuation-passing.

[8] As characterized by Meyer and Wand [55], "[Continuation-style] terms are tail-recursive: no argument is an application."

5.9 Conclusion

5.9.1 Summary

We have associated the *relation* between direct and continuation semantics [81] with the *transformation* of λ-terms into continuation-passing style. Our approach has been based on a new call-by-name CPS transformation $\mathcal{C}_t$ which preserves operational properties of semantics specifications but avoids introducing unnecessary continuations. Given an adequacy property relating the operational and mathematical interpretation of simply-typed λ-terms, an adequacy relation between the direct and continuations semantics follows. The situation is summarized by the diagram below.



The transformation $\mathcal{C}_t$ is directed by termination properties captured *via* an annotated type system. An annotation function $\mathcal{A}$ can be automated using existing static analysis techniques. Based on the annotations produced by $\mathcal{A}$, the transformation $\mathcal{C}_t$ introduces just enough continuations to preserve call-by-name meaning under both call-by-name and call-by-value reduction. $\mathcal{C}_t$ generalizes both the call-by-name continuation transformation (should all terms be serious) and the identity transformation (should all terms be trivial).

5.9.2 Issues

Our approach has focused on relating operational properties of direct and continuations semantics specifications. This was due to the fact that the applications we envision are implementation oriented. In fact, the literature [34, 53, 54, 56, 62, 64, 63, 99, 101, 100] indicates that obtaining a definition more suited to implementation seems to be a primary reason for deriving a continuation semantics from a direct semantics.[9]

There are other issues involved if one wishes to focus on mathematical properties. If a tight relationship (*e.g.*, full abstraction) is desired between the operational and mathematical interpretation of the specification, it is almost certain that the automatic transformation to a continuation specification as outlined here would not preserve that relationship. For example, the meta-language of simply-typed λ-terms leaves simple distinctions among domains such as lifting implicit. This is especially important if we apply our technique to semantics for higher-order functional languages *e.g.*, PCF. [10] Any attempt at maintaining full abstraction would require that the correctness result for $\mathcal{C}_t$ be strengthened from adequacy to full abstraction.[11] Obtaining such a result seems problematic since Plotkin showed that his call-by-name CPS transformation $\mathcal{C}_n$ is not fully abstract [76, p. 148].

5.9.3 Applications

We have noted that semantics-directed compiler-generation systems often work on continuation specifications [53, 54, 56, 62], thus forcing one to write a continuation semantics and to prove its congruence with the direct one. The transformation $\mathcal{C}_t$ allows one to produce the continuation specification automatically.

Partial evaluators work better on continuation-passing programs, but again not all continuations are necessary [13, 14]. $\mathcal{C}_t$ makes it possible to reduce the occurrences of continuations in a source program. In addition, it also enables partial evaluation of call-by-name programs with a regular partial evaluator for call-by-value programs.

[9] A continuation semantics is also required when defining control operators — but in these situations there is no corresponding direct semantics. However, one might imagine a move from direct to continuation semantics (and an associated congruence result) as a useful step in preparation for defining control operators. This is part of the motivation given by Reynolds [80].

[10] The computational meta-language presented in the following chapter provides some progress in this direction since it is more explicit about computational properties.

[11] See the work of Riecke [82, 83] for a detailed presentation of fully abstract translations.

It should be noted however, that our transformation assumes that non-termination is the only computational effect in a source program. For semantics-directed compiling, this is sufficient since semantics definitions do not use *e.g.*, state or exceptions. Extending $\mathcal{C}_t$ to allow such effects would require that the notion of *trivial* be extended to mean "effect-free".

5.9.4 Variations

Denotational semantics are written in various fashions. Partial functions are often used in place of total functions and lifted domains when modeling non-termination [72, 95]. Our explanation of totality and partiality in terms of trivial-result and serious-result functions naturally applies to denotational semantics based on partial functions.

Strict functions are often used to model the strictness properties associated with eager (*i.e.*, call-by-value) functions [72, 87]. For simplicity of presentation, we have expressed strictness properties using *let* constructs only. However, the ideas presented in the previous chapter can be applied in a straightforward manner to handle strict functions. In fact, it might be interesting to consider a call-by-value version of $\mathcal{C}_t$ which generalizes the call-by-value CPS transformation $\mathcal{C}_v$ and the identity transformation.

Continuation semantics of imperative languages often express the meaning of commands as "continuation transformers" [87, 95]. Specifically, the functionality of Com_d is given as

$$(\neg Store) \rightarrow \neg Store.$$

It is very simple to specify a CPS transformation that "puts continuations first", as in Fischer's original transformation [31, 85]. Such a transformation would naturally yield the functionality above.

Finally, our work has relied on denotational definitions being stated using a simply-typed meta-language. This meta-language is sufficient for defining simple imperative languages and simply-typed languages such as Algol 60, Pascal, and PCF. Further investigation is needed to determine how the results presented here can be extended to a meta-language with recursive types. This would be necessary for defining untyped languages such as Scheme.

Acknowledgements

An earlier version of this chapter appeared in the *Proceedings of the 9th Conference on Mathematical Foundations of Programming Semantics* [23]. We are grateful to Andrzej Filinski, Julia Lawall, Karoline Malmkjær, David Schmidt, and the MFPS referees for their comments on that version, to Frank Pfenning and Kirsten Solberg for their interest and time, and to Hanne Riis Nielson for a key comment about our style of annotation. Thanks are also due to Patrick Cousot for connecting our proposed termination analysis with Alan Mycroft's $\sharp$ and $\flat$ analyses. The diagram of Section 5.9 was drawn using Kristoffer Rose's XY-pic package.

Chapter 6

A Generic Account of Continuation-Passing Styles

6.1 Introduction

There are a variety of continuation-passing styles — one for each evaluation strategy (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 (and in the preceding chapters of this dissertation), 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* [76], Plotkin first presents call-by-value 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.

In an effort to exploit these similarities, we note 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}) [61].[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 CPS transformation $\mathcal{K}$ from Λ_{ml} to CPS terms. To connect equational theories, we show that the transformation $\mathcal{K}$ (continuation introduction), along with its inverse $\mathcal{K}^{-1}$ (continuation elimination), establishes an equational correspondence between Λ_{ml} and CPS terms. The diagram of Figure 6.1 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 [76]. It also generalizes similar results for Reynolds's call-by-value CPS transformation [81], and for the CPS transformations directed by strictness and totality properties presented in preceding chapters.

We have noted previously that practical use of CPS transformations requires one to characterize administrative reductions. 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 [76, p. 149] and then another colon-translation that performs administrative reductions for call-by-name CPS [76, p. 154]. 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 [18, 25, 32, 86]. However, just like the CPS transformations, they have been studied individually. In contrast, the continuation

[1] Note that Moggi's computational meta-language [61] is a different language than Moggi's computational λ-calculus λ_c [58, 59].

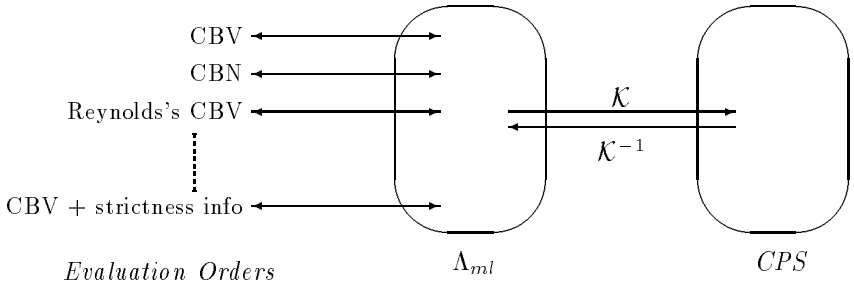


Figure 6.1: Factoring transformations through the computational meta-language

$$
\begin{array}{lcl}
e & \in & \mathit{Terms}[\Lambda_{ml}] \\
e & ::= & c \mid x \mid f \mid \lambda x\,.\,e \mid \mathit{rec}\, f(x).\, e \mid e_0\, e_1 \mid [e] \mid \mathit{let}\, x \Leftarrow e_1\, \mathit{in}\, e_2 \\
 & & \\
\sigma & \in & \mathit{Types}[\Lambda_{ml}] \\
\sigma & ::= & \iota \mid \sigma_1 \rightarrow \widetilde{\sigma_2} \mid \widetilde{\sigma} \\
 & & \\
\Gamma & \in & \mathit{Assums}[\Lambda_{ml}] \\
\Gamma & ::= & \cdot \mid \Gamma, x:\sigma \mid \Gamma, f:\sigma
\end{array}
$$

Figure 6.2: The computational meta-language Λ_{ml}

elimination $\mathcal{K}^{-1}$ 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.

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 6.8 relates the present work to other recent applications of Moggi's framework [86, 97, 98].

The rest of this chapter is organized as follows. Section 6.2 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 6.3 describes the mappings between Λ_{ml} terms and CPS terms. Section 6.4 formalizes administrative reductions. Section 6.5 addresses DS transformations. Section 6.6 describes how data structures are dealt within the framework. Section 6.7 applies the framework to compilation. Section 6.8 addresses related work. Section 6.9 concludes.

6.2 Representing computational properties of λ-terms

6.2.1 The computational meta-language Λ_{ml}

Syntax

$$\textit{(const)} \qquad \Gamma \vdash_{ml} c : \iota$$

$$\textit{(var)} \qquad \Gamma \vdash_{ml} x : \Gamma(x)$$

$$\textit{(recvar)} \qquad \Gamma \vdash_{ml} f : \Gamma(f)$$

$$\textit{(abs)} \qquad \frac{\Gamma, x : \sigma_1 \vdash_{ml} e : \widetilde{\sigma_2}}{\Gamma \vdash_{ml} \lambda x \,.\, e : \sigma_1 \rightarrow \widetilde{\sigma_2}}$$

$$\textit{(rec)} \qquad \frac{\Gamma, f : \sigma_1 \rightarrow \widetilde{\sigma_2}, x : \sigma_1 \vdash_{ml} e : \widetilde{\sigma_2}}{\Gamma \vdash_{ml} \mathit{rec}\, f(x).\, e : \sigma_1 \rightarrow \widetilde{\sigma_2}}$$

$$\textit{(app)} \qquad \frac{\Gamma \vdash_{ml} e_0 : \sigma_1 \rightarrow \widetilde{\sigma_2} \qquad \Gamma \vdash_{ml} e_1 : \sigma_1}{\Gamma \vdash_{ml} e_0\, e_1 : \widetilde{\sigma_2}}$$

$$\textit{(unit)} \qquad \frac{\Gamma \vdash_{ml} e : \sigma}{\Gamma \vdash_{ml} [e] : \widetilde{\sigma}}$$

$$\textit{(let)} \qquad \frac{\Gamma \vdash_{ml} e_1 : \widetilde{\sigma_1} \qquad \Gamma, x : \sigma_1 \vdash_{ml} e_2 : \widetilde{\sigma_2}}{\Gamma \vdash_{ml} \mathit{let}\, x \Leftarrow e_1\ \mathit{in}\ e_2 : \widetilde{\sigma_2}}$$

Figure 6.3: Typing rules for Λ_{ml}

Figure 6.2 presents terms, types, and assumptions for the language Λ_{ml} based on Moggi's *computational meta-language* [61]. Figure 6.3 presents typing rules for Λ_{ml}.

The key feature of the computational meta-language is that its typing system captures the distinction between values and computations and values which is often been used to justify the structure of continuation-passing styles and[76, 80]. Types of the form ι and $\sigma_1 \rightarrow \sigma_2$ are called *value types* and are ranged over by the meta-variable ν. Accordingly, the rules for constants and abstractions are among the introduction rules for value types. Types of the form $\widetilde{\sigma}$ are called *computation types*. Following Moggi, we assume that all functions have computational co-domains. Thus, applications (as typed by the rule *(app)*) are typed as computations.

The *monadic constructs*[2] are used to make the computational process explicit. Intuitively, $[e]$ simply returns the value of e while $\mathit{let}\, x \Leftarrow e_1\ \mathit{in}\ e_2$ first evaluates e_1 and binds the result to x, and then evaluates e_2.

Operational semantics and equational reasoning

Figure 6.4 presents notions of reduction 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 and thus β_{ml} corresponds to β. rec_{ml} can be viewed in a similar manner.

The *monadic reductions* $R_{mon} = \{\mathrm{let}.\beta, \mathrm{let}.\eta, \mathrm{let.assoc}\}$ 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^{σ} programs. We exploit this property when constructing correctness proofs for transformations factored through the meta-language.

To capture the property formally, Figure 6.5 gives two sets of single-step evaluation rules which are used to define operational semantics for Λ_{ml} programs (closed terms of type $\widetilde{\nu}$).[3] In both sets of rules, the R_{ml} reductions

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

[3] Moggi gives a categorical semantics for the meta-language. However, an operational semantics is sufficient here. The rules of Figure 6.5 describe two *leftmost, outermost* reduction strategies over Λ_{ml} reductions.

$$
\begin{array}{rll}
(\lambda x\,.\,e_0)\,e_1 & \rightsquigarrow_{\beta_{ml}} & e_0[x := e_1] \\
(rec\,f(x).\,e_0)\,e_1 & \rightsquigarrow_{\mathrm{rec}_{ml}} & e_0[f := rec\,f(x).\,e_0\,,\,x := e_1] \\
let\,x \Leftarrow [e_1]\;in\,e_2 & \rightsquigarrow_{\mathrm{let}.\beta} & e_2[x := e_1] \\
let\,x \Leftarrow e\;in\,[x] & \rightsquigarrow_{\mathrm{let}.\eta} & e \\
let\,x_2 \Leftarrow (let\,x_1 \Leftarrow e_1\;in\,e_2)\;in\,e_3 & \rightsquigarrow_{\mathrm{let.assoc}} & let\,x_1 \Leftarrow e_1\;in\,(let\,x_2 \Leftarrow e_2\;in\,e_3) \qquad x_1 \notin FV(e_3)
\end{array}
$$

Figure 6.4: Notions of reduction R_{ml} for Λ_{ml}

"Direct-Style" Evaluation Pattern:

$$(\lambda x\,.\,e_0)\,e_1 \longmapsto_{ml.D} e_0[x := e_1] \qquad (rec\,f(x).\,e_0)\,e_1 \longmapsto_{ml.D} e_0[f := rec\,f(x).\,e_0\,,\,x := e_1]$$

$$let\,x \Leftarrow [e_1]\;in\,e_2 \longmapsto_{ml.D} e_2[x := e_1] \qquad \frac{e_1 \longmapsto_{ml.D} e_1'}{let\,x \Leftarrow e_1\;in\,e_2 \longmapsto_{ml.D} let\,x \Leftarrow e_1'\;in\,e_2}$$

"Continuation-Passing Style" Evaluation Pattern:

$$(\lambda x\,.\,e_0)\,e_1 \longmapsto_{ml.C} e_0[x := e_1] \qquad (rec\,f(x).\,e_0)\,e_1 \longmapsto_{ml.C} e_0[f := rec\,f(x).\,e_0\,,\,x := e_1]$$

$$let\,x \Leftarrow [e_1]\;in\,e_2 \longmapsto_{ml.C} e_2[x := e_1]$$

$$\frac{e_1 \longmapsto_{ml.C} e_1'}{let\,x \Leftarrow e_1\;in\,e_2 \longmapsto_{ml.C} let\,x \Leftarrow e_1'\;in\,e_2} \text{ where } e_1 \not\equiv let\,y \Leftarrow e_a\;in\,e_b.$$

$$let\,x_2 \Leftarrow (let\,x_1 \Leftarrow e_1\;in\,e_2)\;in\,e_3 \longmapsto_{ml.C} let\,x_1 \Leftarrow e_1\;in\,(let\,x_2 \Leftarrow e_2\;in\,e_3)$$

Figure 6.5: Single-step evaluation rules for Λ_{ml}

β_{ml}, rec, and let.β 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 [27].

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.

Property 6.1 *For all* $\cdot \vdash_{ml} e : \widetilde{\nu}$,

$$e \longmapsto^*_{ml.D} \texttt{[}v\texttt{]} \quad \textit{iff} \quad e \longmapsto^*_{ml.C} \texttt{[}v\texttt{]}$$

Proof: One method relies on what can be thought of as a generalized version of Plotkin's colon translation [76] which unrolls reductions in the continuation-passing style pattern until a reduction corresponding to a direct-style reduction is exposed (See Section 6.10.2). ∎

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

$$eval_{ml}(e) \;=\; v \;\; \textit{iff} \;\; e \longmapsto^*_{ml.D} \texttt{[}v\texttt{]} \;\; \textit{iff} \;\; e \longmapsto^*_{ml.C} \texttt{[}v\texttt{]}$$

6.2.2 Encoding evaluation orders of Λ^σ in Λ_{ml}

Figures 6.6 and 6.7 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 well-typedness of terms.

Property 6.2

- *If* $\Gamma \vdash e : \sigma$ *then* $\mathcal{E}_n\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{E}_n\langle\!|e|\!\rangle : \mathcal{E}_n\langle\!|\sigma|\!\rangle$.
- *If* $\Gamma \vdash e : \sigma$ *then* $\mathcal{E}_v\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{E}_v\langle\!|e|\!\rangle : \mathcal{E}_v\langle\!|\sigma|\!\rangle$.

Proof: by induction on the structure of the typing derivation for $\Gamma \vdash e : \sigma$ (see Section 6.10.3). ∎

The two encodings differ in that call-by-name functions receive computations as arguments (hence the typing $\mathcal{E}_n\langle\!|\sigma_1|\!\rangle \to \mathcal{E}_n\langle\!|\sigma_2|\!\rangle$) while call-by-value functions receive values as arguments (hence the typing $\mathcal{E}_v\langle\sigma_1\rangle \to \mathcal{E}_v\langle\!|\sigma_2|\!\rangle$). 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 2.2.1. Correctness is captured as follows.

Property 6.3 *For all programs* $\cdot \vdash_{ml} e : \sigma$,

- $\mathcal{E}_n\langle eval_n(e)\rangle \simeq eval_{ml}(\mathcal{E}_n\langle\!|e|\!\rangle)$
- $\mathcal{E}_v\langle eval_v(e)\rangle \simeq eval_{ml}(\mathcal{E}_v\langle\!|e|\!\rangle)$

Proof: The proof takes advantage of the fact that $\longmapsto_{ml.D}$ reductions describe direct-style evaluation. For example, for call-by-name, the proof relies on the fact that $e \longmapsto_n e'$ implies $\mathcal{E}_n\langle\!|e|\!\rangle \longmapsto^*_{ml.D} \mathcal{E}_n\langle\!|e'|\!\rangle$ (see Section 6.10.3). ∎

6.2.3 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^σ terms. Chapter 7 presents encodings of other evaluation strategies. The following section formally connects Λ_{ml} terms and CPS terms.

$$
\begin{array}{rcl}
\mathcal{E}_n\langle\![\cdot]\!\rangle & : & \mathit{Terms}[\Lambda^\sigma] \to \mathit{Terms}[\Lambda_{ml}] \\
\mathcal{E}_n\langle\![v]\!\rangle & = & [\mathcal{E}_n\langle v\rangle] \quad \mathit{...where}\ v \in \mathit{Values}_n[\Lambda^\sigma]. \\
\mathcal{E}_n\langle\![e_0\, e_1]\!\rangle & = & \mathit{let}\, x_0 \Leftarrow \mathcal{E}_n\langle\![e_0]\!\rangle \\
 & & \mathit{in}\, x_0\, \mathcal{E}_n\langle\![e_1]\!\rangle \\
\mathcal{E}_n\langle\![x]\!\rangle & = & x
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{E}_n\langle\cdot\rangle & : & \mathit{Values}_n[\Lambda^\sigma] \to \mathit{Terms}[\Lambda_{ml}] \\
\mathcal{E}_n\langle c\rangle & = & c \\
\mathcal{E}_n\langle\lambda x\,.\,e\rangle & = & \lambda x\,.\,\mathcal{E}_n\langle\![e]\!\rangle \\
\mathcal{E}_n\langle \mathit{rec}\, f(x).\, e\rangle & = & \mathit{rec}\, f(x).\, \mathcal{E}_n\langle\![e]\!\rangle \\
\mathcal{E}_n\langle f\rangle & = & f
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}_n\langle\!|\cdot|\!\rangle & : & \mathit{Types}[\Lambda^\sigma] \to \mathit{Types}[\Lambda_{ml}] \\
\mathcal{E}_n\langle\!|\sigma|\!\rangle & = & \widetilde{\mathcal{E}_n\langle\sigma\rangle} \\
\mathcal{E}_n\langle\iota\rangle & = & \iota \\
\mathcal{E}_n\langle\sigma_1 \to \sigma_2\rangle & = & \mathcal{E}_n\langle\!|\sigma_1|\!\rangle \to \mathcal{E}_n\langle\!|\sigma_2|\!\rangle
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{E}_n\langle\![\cdot]\!\rangle & : & \mathit{Assums}[\Lambda^\sigma] \to \mathit{Assums}[\Lambda_{ml}] \\
\mathcal{E}_n\langle\![\Gamma, x:\sigma]\!\rangle & = & \mathcal{E}_n\langle\![\Gamma]\!\rangle, x:\mathcal{E}_n\langle\!|\sigma|\!\rangle \\
\mathcal{E}_n\langle\![\Gamma, f:\sigma]\!\rangle & = & \mathcal{E}_n\langle\![\Gamma]\!\rangle, f:\mathcal{E}_n\langle\sigma\rangle
\end{array}
$$

Figure 6.6: Call-by-name encoding into Λ_{ml}

$$
\begin{array}{rcl}
\mathcal{E}_v\langle\![\cdot]\!\rangle & : & \mathit{Terms}[\Lambda^\sigma] \to \mathit{Terms}[\Lambda_{ml}] \\
\mathcal{E}_v\langle\![v]\!\rangle & = & [\mathcal{E}_v\langle v\rangle] \quad \mathit{...where}\ v \in \mathit{Values}_v[\Lambda^\sigma]. \\
\mathcal{E}_v\langle\![e_0\, e_1]\!\rangle & = & \mathit{let}\, x_0 \Leftarrow \mathcal{E}_v\langle\![e_0]\!\rangle \\
 & & \mathit{in}\ \mathit{let}\, x_1 \Leftarrow \mathcal{E}_v\langle\![e_1]\!\rangle \\
 & & \quad \mathit{in}\, x_0\, x_1
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{E}_v\langle\cdot\rangle & : & \mathit{Values}_v[\Lambda^\sigma] \to \mathit{Terms}[\Lambda_{ml}] \\
\mathcal{E}_v\langle c\rangle & = & c \\
\mathcal{E}_v\langle\lambda x\,.\,e\rangle & = & \lambda x\,.\,\mathcal{E}_v\langle\![e]\!\rangle \\
\mathcal{E}_v\langle \mathit{rec}\, f(x).\, e\rangle & = & \mathit{rec}\, f(x).\, \mathcal{E}_v\langle\![e]\!\rangle \\
\mathcal{E}_v\langle x\rangle & = & x \\
\mathcal{E}_v\langle f\rangle & = & f
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}_v\langle\!|\cdot|\!\rangle & : & \mathit{Types}[\Lambda^\sigma] \to \mathit{Types}[\Lambda_{ml}] \\
\mathcal{E}_v\langle\!|\sigma|\!\rangle & = & \widetilde{\mathcal{E}_v\langle\sigma\rangle} \\
\mathcal{E}_v\langle\iota\rangle & = & \iota \\
\mathcal{E}_v\langle\sigma_1 \to \sigma_2\rangle & = & \mathcal{E}_v\langle\sigma_1\rangle \to \mathcal{E}_v\langle\!|\sigma_2|\!\rangle
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{E}_v\langle\![\cdot]\!\rangle & : & \mathit{Assums}[\Lambda^\sigma] \to \mathit{Assums}[\Lambda_{ml}] \\
\mathcal{E}_v\langle\![\Gamma, x:\sigma]\!\rangle & = & \mathcal{E}_v\langle\![\Gamma]\!\rangle, x:\mathcal{E}_v\langle\sigma\rangle \\
\mathcal{E}_v\langle\![\Gamma, f:\sigma]\!\rangle & = & \mathcal{E}_v\langle\![\Gamma]\!\rangle, f:\mathcal{E}_v\langle\sigma\rangle
\end{array}
$$

Figure 6.7: Call-by-value encoding into Λ_{ml}

$$
\begin{array}{rcl}
\mathcal{K}\langle\![\cdot]\!\rangle & : & \mathit{Terms}[\Lambda_{ml}] \rightarrow \mathit{Values}_v[\Lambda^\sigma] \\
\mathcal{K}\langle\![c]\!\rangle & = & c \\
\mathcal{K}\langle\![x]\!\rangle & = & x \\
\mathcal{K}\langle\![f]\!\rangle & = & f \\
\mathcal{K}\langle\![\lambda x\,.\,e]\!\rangle & = & \lambda x\,.\,\mathcal{K}\langle\![e]\!\rangle \\
\mathcal{K}\langle\![\mathit{rec}\,f(x).\,e]\!\rangle & = & \mathit{rec}\,f(x).\,\mathcal{K}\langle\![e]\!\rangle \\
\mathcal{K}\langle\![e_0\,\widetilde{{}^{\sigma_1 \rightarrow \sigma_2}}\,e_1]\!\rangle & = & \lambda k^{\neg\mathcal{K}\langle\![\sigma_2]\!\rangle}\,.\,(\mathcal{K}\langle\![e_0]\!\rangle\,\mathcal{K}\langle\![e_1]\!\rangle)\,k \\
\mathcal{K}\langle\![\,[e]\,\widetilde{{}^{\sigma}}]\!\rangle & = & \lambda k^{\neg\mathcal{K}\langle\![\sigma]\!\rangle}\,.\,k\,\mathcal{K}\langle\![e]\!\rangle \\
\mathcal{K}\langle\![\mathit{let}\,x \Leftarrow e_1\,\widetilde{{}^{\sigma_1}}\;\mathit{in}\;e_2\,\widetilde{{}^{\sigma_2}}]\!\rangle & = & \lambda k^{\neg\mathcal{K}\langle\![\sigma_2]\!\rangle}\,.\,\mathcal{K}\langle\![e_1]\!\rangle\,(\lambda x^{\mathcal{K}\langle\![\sigma_1]\!\rangle}\,.\,\mathcal{K}\langle\![e_2]\!\rangle\,k)
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{K}\langle\![\cdot]\!\rangle & : & \mathit{Types}[\Lambda_{ml}] \rightarrow \mathit{Types}[\Lambda^\sigma] \\
\mathcal{K}\langle\![\iota]\!\rangle & = & \iota \\
\mathcal{K}\langle\![\sigma_1 \rightarrow \widetilde{\sigma_2}]\!\rangle & = & \mathcal{K}\langle\![\sigma_1]\!\rangle \rightarrow \neg\neg\mathcal{K}\langle\![\sigma_2]\!\rangle \\
\mathcal{K}\langle\![\widetilde{\sigma}]\!\rangle & = & \neg\neg\mathcal{K}\langle\![\sigma]\!\rangle
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{K}\langle\![\cdot]\!\rangle & : & \mathit{Assums}[\Lambda_{ml}] \rightarrow \mathit{Assums}[\Lambda^\sigma] \\
\mathcal{K}\langle\![\Gamma, x:\sigma]\!\rangle & = & \mathcal{K}\langle\![\Gamma]\!\rangle, x:\mathcal{K}\langle\![\sigma]\!\rangle \\
\mathcal{K}\langle\![\Gamma, f:\sigma]\!\rangle & = & \mathcal{K}\langle\![\Gamma]\!\rangle, f:\mathcal{K}\langle\![\sigma]\!\rangle
\end{array}
$$

Figure 6.8: Continuation introduction — translation from Λ_{ml} into CPS

6.3 Computations as continuation-passing terms

6.3.1 Introducing continuations

Figure 6.8 presents the translation $\mathcal{K}$ from Λ_{ml} to continuation-passing terms of Λ^σ. $\mathcal{K}$ relies on a term representation of the *monad of continuations* [61, 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 $\mathcal{K}$ preserves well-typedness of terms.

Property 6.4 *If* $\Gamma \vdash_{ml} e : \sigma$, *then* $\mathcal{K}\langle\![\Gamma]\!\rangle \vdash \mathcal{K}\langle\![e]\!\rangle : \mathcal{K}\langle\![\sigma]\!\rangle$.

Proof: by induction over the structure of the derivation of $\Gamma \vdash_{ml} e : \sigma$ (see Section 6.10.4). ∎

The translation on computation types $\widetilde{\sigma}$ shows that computations correspond to continuation-passing terms. 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 $\mathcal{K}\langle\![e]\!\rangle$, and
- $\mathit{let}\,x \Leftarrow e_1\;\mathit{in}\;e_2$ abstracts the composition of computations (continuation-passing terms) by forming the continuation $\lambda x\,.\,\mathcal{K}\langle\![e_2]\!\rangle\,k$ and passing it to $\mathcal{K}\langle\![e_1]\!\rangle$.

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

To formalize these properties we establish an equational correspondence between the R_{ml} calculus of Λ_{ml} and CPS terms under the $\beta_a\mathrm{rec}_a\eta_v$ calculi. η_v is sound for reasoning about call-by-name evaluation of CPS terms since identifiers only will bind to abstractions and constants (which are call-by-name values) during call-by-name evaluation. Therefore, $\beta_a\mathrm{rec}_a\eta_v$ calculi are sound for reasoning about *both* call-by-name and call-by-value evaluation of CPS terms.

First, we formalize the language of CPS terms and define a translation $\mathcal{K}^{-1}$ from CPS terms to Λ_{ml}.

Values

(val-const) $$\Gamma \vdash_{val} c : \iota$$

(val-var) $$\Gamma \vdash_{val} x : \Gamma(x)$$

(val-recvar) $$\Gamma \vdash_{val} f : \Gamma(f)$$

(val-abs) $$\frac{\Gamma, x:\sigma_1 \vdash_{val} v : \neg\neg\sigma_2}{\Gamma \vdash_{val} \lambda x\,.\,v : \sigma_1 \rightarrow \neg\neg\sigma_2}$$

(val-rec) $$\frac{\Gamma, f:\sigma_1 \rightarrow \neg\neg\sigma_2, x:\sigma_1 \vdash_{val} v : \neg\neg\sigma_2}{\Gamma \vdash_{val} rec\, f(x).\, v : \sigma_1 \rightarrow \neg\neg\sigma_2}$$

(val-comp) $$\frac{\langle \Gamma\,;\, k : \neg\sigma\rangle \vdash_{ans} \alpha : ans}{\Gamma \vdash_{val} \lambda k\,.\,\alpha : \neg\neg\sigma}$$

Expressions

(exp-app) $$\frac{\Gamma \vdash_{val} v_0 : \sigma_1 \rightarrow \neg\neg\sigma_2 \qquad \Gamma \vdash_{val} v_1 : \sigma_1}{\Gamma \vdash_{exp} v_0\, v_1 : \neg\neg\sigma_2}$$

(exp-val) $$\frac{\Gamma \vdash_{val} v : \neg\neg\sigma}{\Gamma \vdash_{exp} v : \neg\neg\sigma}$$

Continuations

(cont-var) $$\langle \Gamma\,;\, k : \neg\sigma\rangle \vdash_{cont} k : \neg\sigma$$

(cont-abs) $$\frac{\langle \Gamma, x:\sigma_1\,;\, k : \neg\sigma_0\rangle \vdash_{ans} \alpha : ans}{\langle \Gamma\,;\, k : \neg\sigma_0\rangle \vdash_{cont} \lambda x\,.\,\alpha : \neg\sigma_1}$$

Answers

(ans-app.1) $$\frac{\langle \Gamma\,;\, k : \neg\sigma_0\rangle \vdash_{cont} \kappa : \neg\sigma_1 \qquad \Gamma \vdash_{val} v : \sigma_1}{\langle \Gamma\,;\, k : \neg\sigma_0\rangle \vdash_{ans} \kappa\, v : ans}$$

(ans-app.2) $$\frac{\Gamma \vdash_{exp} w : \neg\neg\sigma_0 \qquad \langle \Gamma\,;\, k : \neg\sigma_1\rangle \vdash_{cont} \kappa : \neg\sigma_0}{\langle \Gamma\,;\, k : \neg\sigma_1\rangle \vdash_{ans} w\, \kappa : ans}$$

Types and Assumptions

$$\begin{array}{rcl} \sigma & ::= & \iota \mid \sigma_1 \rightarrow \neg\neg\sigma_2 \mid \neg\neg\sigma \\ \Gamma & ::= & \cdot \mid \Gamma, x:\sigma \mid \Gamma, f:\sigma \end{array}$$

Figure 6.9: Typing rules for Λ_{cps}, the language of CPS terms

$$
\begin{array}{rll}
(\lambda x\,.\,v_0)\,v_1 & \leadsto_{\beta_{exp}} & v_0[x := v_1] \\
(rec\,f(x).\,v_0)\,v_1 & \leadsto_{\mathrm{rec}_{exp}} & v_0[f := rec\,f(x).\,v_0\,,\; x := v_1] \\
(\lambda x\,.\,\alpha)\,v & \leadsto_{\beta_{ans.1}} & \alpha[x := v] \\
(\lambda k\,.\,\alpha)\,\kappa & \leadsto_{\beta_{ans.2}} & \alpha[k := \kappa]
\end{array}
$$

$$
\begin{array}{rll}
\lambda k\,.\,v\,k & \leadsto_{\eta_{val}} & v \quad k \notin FV(v)\ where\ v\ is\ a\ value \\
\lambda x\,.\,\kappa\,x & \leadsto_{\eta_{cont}} & \kappa \quad x \notin FV(\kappa)\ where\ \kappa\ is\ a\ value
\end{array}
$$

Figure 6.10: The set of reductions R_{cps} for Λ_{cps}

Figure 6.9 presents the language Λ_{cps} of CPS λ-terms closed under $\beta_a \mathrm{rec}_a \eta_v$ reduction. (See Section 6.10 for a formal statement of correctness and proofs). Note that Λ_{cps} is a sublanguage of Λ^σ. The judgement $\vdash_{val}$ enforces the property that terms in the image of $\mathcal{K}$ are values (this property is discussed in detail in Section 6.3.4). The judgements $\vdash_{ans}$ and $\vdash_{cont}$ rely on type assumptions that include a distinguished identifier $k \notin \Gamma$.

Figure 6.10 presents all possible $\beta\mathrm{rec}\eta_v$ reductions on Λ_{cps} terms. It is easy to show that each reduction is also a $\beta_v \mathrm{rec}_v \eta_v$ reduction. 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 $\vdash_{val}$ (see Section 6.10.4 for details).

6.3.2 Eliminating continuations

Figure 6.11 presents the translation $\mathcal{K}^{-1}$ from the language of CPS terms Λ_{cps} back to Λ_{ml}. A key component of $\mathcal{K}^{-1}$ is the transformation of continuations to what we call *reduction contexts*.[4] Reduction contexts $\Gamma \vdash_{ml} \varphi : \widetilde{\sigma_2[\sigma_1]}$ of type $\widetilde{\sigma}_2$ with holes of type $\widetilde{\sigma}_1$ are described by the following syntax rules.

$$\textit{(ctxt-triv)} \qquad \Gamma \vdash_{ml} [\cdot] : \widetilde{\sigma}[\widetilde{\sigma}]$$

$$\textit{(ctxt-let)} \qquad \frac{\Gamma, x:\sigma_1 \vdash_{ml} e : \widetilde{\sigma_2}}{\Gamma \vdash_{ml} let\ x \Leftarrow [\cdot]\ in\ e : \widetilde{\sigma_2[\sigma_1]}}$$

$$\textit{(ctxt-fill)} \qquad \frac{\Gamma \vdash_{ml} \varphi : \widetilde{\sigma_2[\sigma_1]} \qquad e : \widetilde{\sigma_1}}{\Gamma \vdash_{ml} \varphi[e] : \widetilde{\sigma_2}}$$

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

Property 6.5

- *If* $\Gamma \vdash_{val} v : \sigma$ *then* $\mathcal{K}^{-1}(\llbracket\Gamma\rrbracket) \vdash_{ml} \mathcal{K}^{-1}_{val}(\llbracket v\rrbracket) : \mathcal{K}^{-1}(\llbracket\sigma\rrbracket)$.
- *If* $\Gamma \vdash_{exp} w : \neg\neg\sigma$ *then* $\mathcal{K}^{-1}(\llbracket\Gamma\rrbracket) \vdash_{ml} \mathcal{K}^{-1}_{exp}(\llbracket w\rrbracket) : \mathcal{K}^{-1}(\llbracket\neg\neg\sigma\rrbracket)$.
- *If* $\langle \Gamma\ ;\ k : \neg\sigma\rangle \vdash_{ans} \alpha : ans$
  then $\mathcal{K}^{-1}(\llbracket\Gamma\rrbracket) \vdash_{ml} \mathcal{K}^{-1}_{ans}(\llbracket\alpha\rrbracket) : \widetilde{\mathcal{K}^{-1}(\llbracket\sigma\rrbracket)}$.
- *If* $\langle \Gamma\ ;\ k : \neg\sigma_0\rangle \vdash_{cont} \kappa : \neg\sigma_1$
  then $\mathcal{K}^{-1}(\llbracket\Gamma\rrbracket) \vdash_{ml} \mathcal{K}^{-1}_{cont}(\llbracket\kappa\rrbracket) : \widetilde{\mathcal{K}^{-1}(\llbracket\sigma_0\rrbracket)}[\widetilde{\mathcal{K}^{-1}(\llbracket\sigma_1\rrbracket)}]$.

[4] Felleisen and Friedman first pointed out that continuations in CPS correspond to *evaluation contexts* in direct-style terms (*e.g.*, terms from the language Λ^σ) [27]. 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.

$$
\begin{array}{rcl}
\mathcal{K}^{-1}\langle\!|\cdot|\!\rangle & : & \mathit{Terms}[\Lambda_{cps}] \rightarrow \mathit{Terms}[\Lambda_{ml}] \\
\mathcal{K}^{-1}_{val}\langle\!|c|\!\rangle & = & c \\
\mathcal{K}^{-1}_{val}\langle\!|x|\!\rangle & = & x \\
\mathcal{K}^{-1}_{val}\langle\!|f|\!\rangle & = & f \\
\mathcal{K}^{-1}_{val}\langle\!|\lambda x\,.\,v|\!\rangle & = & \lambda x\,.\,\mathcal{K}^{-1}_{val}\langle\!|v|\!\rangle \\
\mathcal{K}^{-1}_{val}\langle\!|\mathit{rec}\, f(x).\, v|\!\rangle & = & \mathit{rec}\, f(x).\, \mathcal{K}^{-1}_{val}\langle\!|v|\!\rangle \\
\mathcal{K}^{-1}_{val}\langle\!|\lambda k\,.\,\alpha|\!\rangle & = & \mathcal{K}^{-1}_{ans}\langle\!|\alpha|\!\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{K}^{-1}_{exp}\langle\!|v_0\ v_1|\!\rangle & = & \mathcal{K}^{-1}_{val}\langle\!|v_0|\!\rangle\ \mathcal{K}^{-1}_{val}\langle\!|v_1|\!\rangle \\
\mathcal{K}^{-1}_{exp}\langle\!|v|\!\rangle & = & \mathcal{K}^{-1}_{val}\langle\!|v|\!\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{K}^{-1}_{ans}\langle\!|w\,\kappa|\!\rangle & = & \mathcal{K}^{-1}_{cont}\langle\!|\kappa|\!\rangle[\mathcal{K}^{-1}_{exp}\langle\!|w|\!\rangle] \\
\mathcal{K}^{-1}_{ans}\langle\!|\kappa\ v|\!\rangle & = & \mathcal{K}^{-1}_{cont}\langle\!|\kappa|\!\rangle[\,[\mathcal{K}^{-1}_{val}\langle\!|v|\!\rangle]\,]
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{K}^{-1}_{cont}\langle\!|k|\!\rangle & = & [\cdot] \\
\mathcal{K}^{-1}_{cont}\langle\!|\lambda x\,.\,\alpha|\!\rangle & = & \mathit{let}\ x \Leftarrow [\cdot]\ \mathit{in}\ \mathcal{K}^{-1}_{ans}\langle\!|\alpha|\!\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{K}^{-1}\langle\!|\cdot|\!\rangle & : & \mathit{Types}[\Lambda_{cps}] \rightarrow \mathit{Types}[\Lambda_{ml}] \\
\mathcal{K}^{-1}\langle\!|\iota|\!\rangle & = & \iota \\
\mathcal{K}^{-1}\langle\!|\sigma_1 \rightarrow \sigma_2|\!\rangle & = & \mathcal{K}^{-1}\langle\!|\sigma_1|\!\rangle \rightarrow \mathcal{K}^{-1}\langle\!|\sigma_2|\!\rangle \\
\mathcal{K}^{-1}\langle\!|\neg\neg\sigma|\!\rangle & = & \widetilde{\mathcal{K}^{-1}\langle\!|\sigma|\!\rangle}
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{K}^{-1}\langle\!|\cdot|\!\rangle & : & \mathit{Assums}[\Lambda_{cps}] \rightarrow \mathit{Assums}[\Lambda_{ml}] \\
\mathcal{K}^{-1}\langle\!|\Gamma, x:\sigma|\!\rangle & = & \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle, x:\mathcal{K}^{-1}\langle\!|\sigma|\!\rangle \\
\mathcal{K}^{-1}\langle\!|\Gamma, f:\sigma|\!\rangle & = & \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle, f:\mathcal{K}^{-1}\langle\!|\sigma|\!\rangle
\end{array}
$$

Figure 6.11: Continuation elimination — translation from CPS back to Λ_{ml}

Proof: by simultaneous induction on the typing derivations for $\Gamma \vdash_{val} v : \sigma$, $\Gamma \vdash_{exp} w : \sigma$, $\langle \Gamma\,;\, k : \neg\sigma\rangle \vdash_{ans} \alpha : \mathit{ans}$, and $\langle \Gamma\,;\, k : \neg\sigma_0\rangle \vdash_{cont} \kappa : \neg\sigma_1$ (see Section 6.10.5). ∎

6.3.3 Relating operational semantics and equational theories

We can now state an adequacy property for the translations $\mathcal{K}$ and $\mathcal{K}^{-1}$. The following theorem recasts Plotkin's Simulation and Indifference theorems for call-by-name and call-by-value CPS [76, Section 6] in terms of the generic introduction of continuations by $\mathcal{K}$.

Theorem 6.1 (Simulation and Indifference for $\mathcal{K}$)
For all $\cdot \vdash_{ml} e : \widetilde{\nu}$,

$$\mathit{eval}_{ml}(e) \ \simeq_{obs}\ \mathit{eval}_n(\mathcal{K}\langle\!|e|\!\rangle\,(\lambda x\,.\,x)) \ \simeq_{obs}\ \mathit{eval}_v(\mathcal{K}\langle\!|e|\!\rangle\,(\lambda x\,.\,x))$$

Proof: The proof follows from the equational correspondence between Λ_{ml} and CPS terms given below (see Section 6.10.5). ∎

The corresponding property for $\mathcal{K}^{-1}$ is as follows.

Theorem 6.2 *For all* $\cdot \vdash_{val} v : \neg\neg\nu$,

$$\mathit{eval}_{ml}(\mathcal{K}^{-1}_{val}\langle\!|v|\!\rangle) \ \simeq_{obs}\ \mathit{eval}_n(v\,(\lambda x\,.\,x)) \ \simeq_{obs}\ \mathit{eval}_v(v\,(\lambda x\,.\,x))$$

Proof: Similar to the proof for $\mathcal{K}$ (see Section 6.10.5). ∎

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 [76]. Following a strategy used in Chapter 3, we relate the equational theories of the meta-language and CPS terms by showing an *equational correspondence* between Λ_{ml} terms under the R_{ml} calculus and Λ_{cps} terms under the R_{cps} calculus.

Theorem 6.3 (Equational Correspondence) *For all* $\Gamma \vdash_{ml} e, e_1, e_2 : \sigma$ *and* $\Gamma \vdash_{val} v, v_1, v_2 : \sigma'$,

1. $\lambda R_{ml} \vdash e = (\mathcal{K}_{val}^{-1} \circ \mathcal{K})(\!|e|\!)$
2. $\lambda R_{cps} \vdash v = (\mathcal{K} \circ \mathcal{K}_{val}^{-1})(\!|v|\!)$
3. $\lambda R_{ml} \vdash e_1 = e_2$ *iff* $\lambda R_{cps} \vdash \mathcal{K}(\!|e_1|\!) = \mathcal{K}(\!|e_2|\!)$
4. $\lambda R_{cps} \vdash v_1 = v_2$ *iff* $\lambda R_{ml} \vdash \mathcal{K}_{val}^{-1}(\!|v_1|\!) = \mathcal{K}_{val}^{-1}(\!|v_2|\!)$

Proof: Follows the same outline as the proofs of equational correspondence in Chapter 3 (see Section 6.10.5). ∎

This allows one to reason about CPS programs without moving to continuation-passing-style. In practical terms, any optimization provable in the meta-language theory is guaranteed to be sound on CPS terms under both call-by-name and call-by-value evaluation (since the $\beta_a \mathrm{rec}_a \eta_v$ calculus is sound for both call-by-name and call-by-value evaluation of CPS terms). Furthermore, the meta-language theory is complete in the sense that any CPS optimization provable under $\beta_a \mathrm{rec}_a \eta_v$ reasoning is expressible within the meta-language theory.

6.3.4 Assessment

Evaluation-order independence for all terms in the image of $\mathcal{K}$ holds because all Λ_{cps} function arguments are values. Specifically,

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

Obtaining evaluation-order independence requires slightly more than simply instantiating the monadic constructs $[\cdot]$ and *let* with the continuation monad (witness the η-redex in $\mathcal{K}(\!|e_0\, e_1|\!)$ of Figure 6.8). 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 $\mathcal{K}''$ be a translation that only instantiates $[\cdot]$ and *let*. Following this strategy gives $\mathcal{K}''(\!|e_0\, e_1|\!) = \mathcal{K}''(\!|e_0|\!)\, \mathcal{K}''(\!|e_1|\!)$. Now, if $e_0 \equiv \lambda z \,.\, [c_0]$ and $e_1 \equiv (\mathit{rec}\, f(y).\, f\, y)\, c_1$, then

$$\mathcal{K}''(\!|e_0\, e_1|\!)\, (\lambda x.x) = ((\lambda z.\lambda k.k\, c_0)\, ((\mathit{rec}\, f(y).\, f\, y)\, c_1))\, (\lambda x.x)$$

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 Λ_{cps} terms under call-by-value evaluation. For example, $\mathcal{K}(\!|e_0\, e_1|\!)\, (\lambda x \,.\, x)$ terminates under call-by-value but η-reduces to $\mathcal{K}''(\!|e_0\, e_1|\!)\, (\lambda x \,.\, x)$ which has just been shown to diverge under call-by-value.[5] In reality, problems are encountered only when one attempts to generalize η_{val} redexes to η — all η redexes of the form given by η_{cont} and are also η_v redexes.

6.3.5 An alternate transformation

The discussion above reveals that the λk's in CPS terms serve two purposes:

- they assist in passing continuations;
- they "delay" the call-by-value evaluation of function arguments as required when *e.g.*, encoding call-by-name CPS.

In some sense, the simple transformation $\mathcal{K}$ obtained directly from the CPS monad is too pessimistic — it assumes that every term of computation type needs to be delayed. Thus, many uninteresting redexes *e.g.*, of the form $(\lambda k \,.\, ...)\, k$ get introduced. Figure 6.12 defines a useful optimized version of $\mathcal{K}$. $\mathcal{K}'$ is specialized to the syntactic categories of CPS terms and avoids introducing uninteresting redexes of the form mentioned above. The claim that only non-interesting redexes are optimized away is formalized by the following property.

[5] Similar examples of η being unsound exist for "traditional" untyped call-by-name CPS terms under call-by-value evaluation.

$$
\begin{array}{rcl}
\mathcal{K}'\langle\!|\cdot|\!\rangle & : & Terms[\Lambda_{ml}] \rightarrow Terms[\Lambda^{\sigma}] \\
\mathcal{K}'_{val}\langle\!|c|\!\rangle & = & c \\
\mathcal{K}'_{val}\langle\!|x|\!\rangle & = & x \\
\mathcal{K}'_{val}\langle\!|f|\!\rangle & = & f \\
\mathcal{K}'_{val}\langle\!|\lambda x\,.\,e|\!\rangle & = & \lambda x\,.\,\mathcal{K}'_{val}\langle\!|e|\!\rangle \\
\mathcal{K}'_{val}\langle\!|rec\,f(x).\,e|\!\rangle & = & rec\,f(x).\,\mathcal{K}'_{val}\langle\!|e|\!\rangle \\
\mathcal{K}'_{val}\langle\!|e_0\;e_1|\!\rangle & = & \lambda k\,.\,\mathcal{K}'_{ans}\langle\!|e_0\;e_1|\!\rangle \\
\mathcal{K}'_{val}\langle\!|\texttt{[}e\texttt{]}|\!\rangle & = & \lambda k\,.\,\mathcal{K}'_{ans}\langle\!|\texttt{[}e\texttt{]}|\!\rangle \\
\mathcal{K}'_{val}\langle\!|let\;x \Leftarrow e_1\;in\;e_2|\!\rangle & = & \lambda k\,.\,\mathcal{K}'_{ans}\langle\!|let\;x \Leftarrow e_1\;in\;e_2|\!\rangle \\
 & & \\
\mathcal{K}'_{ans}\langle\!|\texttt{[}e\texttt{]}|\!\rangle & = & k\,\mathcal{K}'_{val}\langle\!|e|\!\rangle \\
\mathcal{K}'_{ans}\langle\!|let\;x \Leftarrow e_1\;in\;e_2|\!\rangle & = & \mathcal{K}'_{exp}\langle\!|e_1|\!\rangle\;(\lambda x\,.\,\mathcal{K}'_{ans}\langle\!|e_2|\!\rangle) \\
\mathcal{K}'_{ans}\langle\!|e|\!\rangle & = & \mathcal{K}'_{exp}\langle\!|e|\!\rangle\;k \\
 & & \\
\mathcal{K}'_{exp}\langle\!|e_0\;e_1|\!\rangle & = & \mathcal{K}'_{val}\langle\!|e_0|\!\rangle\;\mathcal{K}'_{val}\langle\!|e_1|\!\rangle \\
\mathcal{K}'_{exp}\langle\!|e|\!\rangle & = & \mathcal{K}'_{val}\langle\!|e|\!\rangle
\end{array}
$$

Figure 6.12: Optimized continuation introduction $\mathcal{K}'$

Property 6.6 *For $e \in \Lambda_{ml}$,*

$$e \equiv (\mathcal{K}^{-1} \circ \mathcal{K}')\langle\!|e|\!\rangle$$

That is, while $\mathcal{K}'$ optimizes $\mathcal{K}$, $\mathcal{K}'\langle\!|e|\!\rangle$ is still within the class of terms abstractly represented by e. The relationship between $\mathcal{K}$ and $\mathcal{K}'$ is formalized as follows.

Property 6.7 *For all $e \in \Lambda_{ml}$,*

$$\mathcal{K}\langle\!|e|\!\rangle \longrightarrow\!\!\!\!\!\rightarrow_{\beta_{ans.2}} \mathcal{K}'_{val}\langle\!|e|\!\rangle$$

Proof: by induction on the structure of e (see Section 6.10.6). ∎

6.3.6 Generalizing the notion of value

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

$$\sigma \quad ::= \quad \iota \;\mid\; \sigma_1 \rightarrow \sigma_2 \;\mid\; \widetilde{\sigma}$$

This typing generalizes the notion of value to include applications of functions that always terminate when applied. We noted in Chapter 5 that such functions do not need to be passed continuations to achieve evaluation-order independence. We conjecture that Theorem 6.1 and 6.2 hold for a language with this generalized type system and that 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.

Section 7.3 suggests how this generalized notion of value could be used in meta-language encodings.

6.3.7 CPS transformations from encodings of computational properties

Previous applications of Λ_{ml} focus exclusively on call-by-value or call-by-name [61, 98]. 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 6.4), and a correct mapping from CPS back to Λ_{ml} (see Section 6.5) follow as corollaries from simply identifying the appropriate computational properties.

For each evaluation strategy, the corresponding CPS transformation is constructed by composing the encoding $\mathcal{E}$ (of the evaluation order into Λ_{ml}) with the continuation introduction $\mathcal{K}$ (or preferably, the slightly optimized introduction $\mathcal{K}'$ of Section 6.3). In general, the correctness of the constructed CPS transformations follows from the correctness of an encoding $\mathcal{E}$ and the correctness of $\mathcal{K}$ (Theorem 6.1). For instance, the following property shows that $\mathcal{E}_n$ and $\mathcal{E}_v$ gives rise to Plotkin's CPS transformations $\mathcal{C}_n$ and $\mathcal{C}_v$.

Property 6.8 *For all* $\Gamma \vdash e : \sigma$,

$$\begin{aligned} \mathcal{C}_n\langle\!|e|\!\rangle &\equiv (\mathcal{K}' \circ \mathcal{E}_n)\langle\!|e|\!\rangle \\ \mathcal{C}_v\langle\!|e|\!\rangle &\equiv (\mathcal{K}' \circ \mathcal{E}_v)\langle\!|e|\!\rangle \end{aligned}$$

Proof: by induction over the structure of e (see Section 6.10.7). ∎

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^σ with the theory R_{ml} of Λ_{ml}.

Other CPS transformations such as Reynolds's call-by-value CPS transformation and the transformations $\mathcal{C}_s$ and $\mathcal{C}_t$ of Chapters 4 and 5 are addressed in Chapter 7.

6.4 A generic account of administrative reductions

As we have previously noted, 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. The transformation $\mathcal{K}$ introduces administrative abstractions for $e_0\, e_1$, $[\![e]\!]$, and $let\, x \Leftarrow e_0\, in\, e_1$. An examination of CPS terms (Figure 6.9) and CPS reductions (Figure 6.10) reveals that CPS reductions $R_{adm} = \{\beta_{ans.1}, \beta_{ans.2}, \eta_{val}, \eta_{cont}\}$ only reduce administrative abstractions whereas CPS reductions $\{\beta_{exp}, \mathrm{rec}_{exp}\}$ only reduce "source" abstractions (*i.e.*, abstractions already present in Λ_{ml} terms).

A pleasing property of Λ_{ml} is that monadic reductions abstract CPS administrative reductions. Specifically, the following property shows that a monadic reduction corresponds to a series of administrative conversions.

Property 6.9 *For* $e \in \Lambda_{ml}$,

$$e \longrightarrow_{R_{mon}} e' \quad \Rightarrow \quad \mathcal{K}\langle\!|e|\!\rangle =_{R_{adm}} \mathcal{K}\langle\!|e'|\!\rangle$$

Proof: Follows from an examination of the proofs for the equational correspondence (see Section 6.10.6). ∎

Now, let $\mathcal{N}$ be a function mapping each Λ_{ml} term to its R_{mon} normal form.[6] The following theorem states that eliminating monadic redexes (by taking a term to its monadic normal form) corresponds to eliminating administrative redexes.

Theorem 6.4 *For* $\Gamma \vdash_{ml} e : \sigma$, *If* e *is in* R_{mon}*-normal form (i.e.,* $\{\mathrm{let}.\beta, \mathrm{let}.\eta, \mathrm{let.assoc}\}$*-normal form) then* $\mathcal{K}'_{val}\langle\!|e|\!\rangle$ *is in* R_{adm}*-normal form (i.e.,* $\{\beta_{ans.1}, \beta_{ans.2}, \eta_{val}, \eta_{cont}\}$*-normal form).*

Proof: see Section 6.10.8. ∎

In practical terms, this allows a generic description of optimized CPS transformations which produce terms containing no administrative redexes. For example, let $\mathcal{C}_v^{opt}$ denote a version of Plotkin's call-by-value CPS transformation that carries out administrative reductions on the fly [3, 20, 102] as discussed in Section 2.5.4.

[6] This function is well-defined since the reductions R_{mon} are confluent and strongly normalizing [61].

Property 6.10 *For* $\Gamma \vdash e : \sigma$, $\mathcal{C}_v^{opt}\langle\!|e|\!\rangle \equiv (\mathcal{K}' \circ \mathcal{N} \circ \mathcal{E}_v)\langle\!|e|\!\rangle$.

As expected, similar properties hold for the corresponding one-pass call-by-name CPS transformation, and for the corresponding CPS transformations after static analyses [22, 23, 73]. This staging and the account of administrative reductions prior to introducing continuations have been recently noted [17, 18, 32, 52, 86]. 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 $\mathcal{K}$).

The last step is provably reversible, and Lawall automated that proof for another meta-language than Λ_{ml} [50].

Recently, Sabry and Felleisen have identified an additional optimization made possible by administrative reductions on call-by-value CPS terms [86]. The optimization corresponds to relocating evaluation contexts (reduction contexts, continuations) inside abstractions in β-redexes. The following R_{ml} equivalence characterizes this optimization.[7]

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

where $\varphi \not\equiv [\cdot]$ and $x \notin FV(\varphi)$. This generalizes the optimization to any evaluation strategy encoded in the meta-language.

We have characterized administrative reductions abstractly in terms of normalization of Λ_{ml} terms. In practice, one would define an optimized version of $\mathcal{K}$ that performs administrative reductions "on the fly" [3, 20, 102]. This can either be achieved using brute force [65, 86] or with a two-level specification [20, 22, 73].

6.5 DS transformations from encodings of computational properties

Direct-style transformations mapping CPS terms back to direct-style Λ^σ terms are potentially useful in their own right. Our transformation $\mathcal{K}^{-1}$ forms the core of a generic DS transformation, thus generalizing previous work in the absence of computational effects other than non-termination [18, 85]. Direct-style transformations $\mathcal{D}$ are obtained by composing "inverse" encodings $\mathcal{E}^{-1}$ (mapping Λ_{ml} terms to direct-style Λ^σ terms) with the transformation $\mathcal{K}^{-1}$.

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

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 [18, 25, 51, 52]. Alternatively, one may adapt the techniques of Sabry and Felleisen [86] and map reduction contexts to evaluation contexts in Λ^σ.

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^σ which explicitly direct computation (*e.g.*, strict *let*'s, thunks) without resorting to full continuation-passing style.

6.6 Products and co-products

This section outlines how products and co-products may be incorporated into the generic framework.

Figure 6.13 extends the syntax of Λ_{ml} to include products and co-products.[8] The set of *value types* is extended to include types $\sigma_1 \times \sigma_2$ and $\sigma_1 + \sigma_2$. The reductions for products and co-products are as follows.

[7] 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.

[8] The presentation of products follows Moggi [61, Section 3.1].

$$\frac{\Gamma \vdash_{ml} e : \sigma_1 \times \sigma_2}{\Gamma \vdash_{ml} proj_i\, e : \sigma_i} \quad (i = 1,2) \qquad \frac{\Gamma \vdash_{ml} e : \sigma_i}{\Gamma \vdash_{ml} inj_i^{\sigma_1+\sigma_2}\, e : \sigma_1 + \sigma_2} \quad (i = 1,2)$$

$$\frac{\Gamma \vdash_{ml} e_1 : \sigma_1 \qquad \Gamma \vdash_{ml} e_2 : \sigma_2}{\Gamma \vdash_{ml} pair\, e_1\, e_2 : \sigma_1 \times \sigma_2} \qquad \frac{\Gamma \vdash_{ml} e : \sigma_1 + \sigma_2 \qquad \Gamma, x_1 : \sigma_1 \vdash_{ml} e_1 : \widetilde{\sigma} \qquad \Gamma, x_2 : \sigma_2 \vdash_{ml} e_2 : \widetilde{\sigma}}{\Gamma \vdash_{ml} case\, e\, of\, (x_1.e_1) \mid (x_2.e_2) : \widetilde{\sigma}}$$

$$\begin{array}{rcl} \sigma & \in & Types[\Lambda_{ml}] \\ \sigma & ::= & \ldots \mid \sigma_1 \times \sigma_2 \mid \sigma_1 + \sigma_2 \end{array}$$

Figure 6.13: Products and coproducts for the computational meta-language Λ_{ml}

$$\begin{array}{rcl} \mathcal{K}\langle\!|\cdot|\!\rangle & : & Terms[\Lambda_{ml}] \rightarrow Terms[\Lambda^{\sigma^+}] \\ & \ldots & \\ \mathcal{K}\langle\!|proj_i\, e|\!\rangle & = & proj_i\, \mathcal{K}\langle\!|e|\!\rangle \\ \mathcal{K}\langle\!|pair\, e_1\, e_2|\!\rangle & = & pair\, \mathcal{K}\langle\!|e_1|\!\rangle\, \mathcal{K}\langle\!|e_2|\!\rangle \\ \mathcal{K}\langle\!|inj_i\, e|\!\rangle & = & inj_i\, \mathcal{K}\langle\!|e|\!\rangle \\ \mathcal{K}\langle\!|case\, e\, of\, (x_1.e_1) \mid (x_2.e_2)|\!\rangle & = & case\, \mathcal{K}\langle\!|e|\!\rangle\, of\, (x_1.\mathcal{K}\langle\!|e_1|\!\rangle) \mid (x_2.\mathcal{K}\langle\!|e_2|\!\rangle) \end{array}$$

$$\begin{array}{rcl} \mathcal{K}\langle\!|\cdot|\!\rangle & : & Types[\Lambda_{ml}] \rightarrow Types[\Lambda^{\sigma^+}] \\ & \ldots & \\ \mathcal{K}\langle\!|\sigma_1 \times \sigma_2|\!\rangle & = & \mathcal{K}\langle\!|\sigma_1|\!\rangle \times \mathcal{K}\langle\!|\sigma_2|\!\rangle \\ \mathcal{K}\langle\!|\sigma_1 + \sigma_2|\!\rangle & = & \mathcal{K}\langle\!|\sigma_1|\!\rangle + \mathcal{K}\langle\!|\sigma_2|\!\rangle \end{array}$$

Figure 6.14: Continuation introduction for products and co-products

$$\begin{array}{rcl} \mathit{proj}_i\ \mathit{pair}\ e_1\ e_2 & \leadsto_{\times_i.\beta} & e_i \\ \mathit{case}\ (\mathit{inj}_i\ e)\ \mathit{of}\ (x_1.e_1) \mid (x_2.e_2) & \leadsto_{+_i.\beta} & e_i[x_i := e] \end{array}$$

As with function spaces, the structure of Λ_{ml} types and terms provides a description of constructors with differing computational properties (*e.g.*, eager or lazy). For example, eager (*i.e.*, call-by-value) pairing can be expressed via products of values.

$$\begin{array}{rcl} \mathcal{E}_v\langle\sigma_1 \times \sigma_2\rangle & = & \mathcal{E}_v\langle\sigma_1\rangle \times \mathcal{E}_v\langle\sigma_2\rangle \\ \mathcal{E}_v\langle\!|\mathit{pair}\ e_1\ e_2|\!\rangle & = & \mathit{let}\ x_1 \Leftarrow \mathcal{E}_v\langle\!|e_1|\!\rangle\ \mathit{in}\ \mathit{let}\ x_2 \Leftarrow \mathcal{E}_v\langle\!|e_2|\!\rangle\ \mathit{in}\ [\mathit{pair}\ x_1\ x_2] \\ \mathcal{E}_v\langle\!|\mathit{proj}_i\ e|\!\rangle & = & \mathit{let}\ x \Leftarrow \mathcal{E}_v\langle\!|e|\!\rangle\ \mathit{in}\ [\mathit{proj}_i\ x] \end{array}$$

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

$$\begin{array}{rcl} \mathcal{E}_n\langle\sigma_1 + \sigma_2\rangle & = & \mathcal{E}_n\langle\!|\sigma_1|\!\rangle + \mathcal{E}_n\langle\!|\sigma_2|\!\rangle \\ \mathcal{E}_n\langle\!|\mathit{inj}_i\ e|\!\rangle & = & [\mathit{inj}_i\ \mathcal{E}_n\langle\!|e|\!\rangle] \\ \mathcal{E}_n\langle\!|\mathit{case}\ e\ \mathit{of}\ (x_1.e_1) \mid (x_2.e_2)|\!\rangle & = & \mathit{let}\ x \Leftarrow \mathcal{E}_n\langle\!|e|\!\rangle\ \mathit{in}\ \mathit{case}\ x\ \mathit{of}\ (x_1.\mathcal{E}_n\langle\!|e_1|\!\rangle) \mid (x_2.\mathcal{E}_n\langle\!|e_2|\!\rangle) \end{array}$$

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 6.14 extends $\mathcal{K}$ to products and co-products. As before, correct CPS transformations for the extended language are obtained by composing the encodings $\mathcal{E}$ with $\mathcal{K}$. 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 $\lambda k \ldots$. corresponding to meta-language computations). This generalizes the earlier work in Chapter 4 where we presented a CPS transformation $\mathcal{C}_s$ directed by strictness properties and handling both strict and non-strict products.

Note that the definition of $\mathcal{K}$ in Figure 6.14 relies on the monadic constructs to structure continuation-passing properly. However, the definition below gives an alternate structure commonly used when transforming conditional expressions.

$$\begin{array}{l} \mathcal{K}\langle\!|\mathit{case}\ e\ \mathit{of}\ (x_1.e_1) \mid (x_2.e_2)|\!\rangle \\ \qquad = \quad \lambda k\ .\ \mathit{case}\ \mathcal{K}\langle\!|e|\!\rangle\ \mathit{of}\ (x_1.\mathcal{K}\langle\!|e_1|\!\rangle\ k) \mid (x_2.\mathcal{K}\langle\!|e_2|\!\rangle\ k) \end{array}$$

The latter definition allows reduction contexts (in the form of continuations) to be relocated inside *case* constructs — which duplicates the contexts.[9] Indeed, this definition requires adding the following reduction to the set of Λ_{ml} reductions to obtain an equational correspondence with CPS terms:[10]

$$\begin{array}{l} \varphi[\mathit{case}\ e\ \mathit{of}\ (x_1.e_1) \mid (x_2.e_2)] \\ \qquad \longrightarrow_{+.\mathrm{ctxt}} \quad \mathit{case}\ e\ \mathit{of}\ (x_1.\varphi[e_1]) \mid (x_2.\varphi[e_2]) \end{array}$$

where $\varphi \not\equiv [\cdot]$ and $x_1, x_2 \notin FV(\varphi)$.

6.7 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.* [32] and studied for untyped, call-by-value λ-terms. In particular, "A-reductions" provide the following standard compiler optimizations [32, page 243]:

1. code segments are merged across declarations and conditions;

[9] This duplication can be avoided inserting a β-redex when introducing continuations [20, 86].

[10] Sabry and Felleisen [86] give a similar reduction for conditional expressions.

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.

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.)

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 [10]. This observation applies here as well — computation types correspond to boxed values, and value types correspond to unboxed values.

6.8 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 [61, page 58].

Wadler illustrated the usefulness of Moggi's ideas when applied to functional programming [98]. 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 [86], 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 [58] — 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 [31], 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 [86], 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* [61] 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, it seems that the continuation/state monad (obtained *e.g.*, by applying the state-monad constructor to the continuation monad [60]) offers a generic relation between implicit and explicit representations of both control and state.

6.9 Conclusion

We have characterized CPS transformationsadministrative reductions, and 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 $\mathcal{K}$. 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.

Acknowledgements

An earlier version of this chapter appeared the *Proceedings of the Twenty-first Annual ACM Symposium on Principles of Programming Languages* [40]. We are grateful to Andrzej Filinski and Bob Harper for fundamental observations and encouragements early in the development 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 for comments.

6.10 Proofs

6.10.1 Meta-language evaluation characteristics

Property 6.11 (Single-step evaluation characteristics for Λ_{ml}) *For all* $\cdot \vdash_{ml} e : \widetilde{\sigma}$,

$$e \longmapsto_{ml.D} e' \text{ or } e \equiv [\![e']\!]$$

Proof: by induction on the structure of e.

case $e \equiv [\![e']\!]$: immediate.

case $e \equiv e_0\, e_1$:

case $e_0 \equiv x$: disallowed since the property applies to closed terms
case $e_0 \equiv \lambda x\,.\,e_a$: $(\lambda x\,.\,e_a)\, e_1 \longmapsto_{ml.D} e_a[x := e_1]$
case $e_0 \equiv rec\, f(x).\, e_a$: $(rec\, f(x).\, e_a)\, e_1 \longmapsto_{ml.D} e_0[f := rec\, f(x).\, e_a\,,\, x := e_1]$

case $e \equiv let\, x_0 \Leftarrow e_0\ in\ e_1$: by the ind. hyp., $e_0 \longmapsto_{ml.D} e_0'$ or $e_0 \equiv [\![e_0']\!]$.

case $e_0 \longmapsto_{ml.D} e_0'$: by def. of $\longmapsto_{ml.D}$, $let\, x_0 \Leftarrow e_0\ in\ e_1 \longmapsto_{ml.D} let\, x_0 \Leftarrow e_0'\ in\ e_1$
case $e_0 \equiv [\![e_0']\!]$: by def. of $\longmapsto_{ml.D}$, $let\, x_0 \Leftarrow [\![e_0']\!]\ in\ e_1 \longmapsto_{ml.D} e_1[x_0 := e_0']$

∎

Property 6.12 (Program evaluation characteristics for Λ_{ml}) *For all $\cdot \vdash_{ml} e : \widetilde{\nu}$, either $e \longmapsto^*_{ml.D} [v]$, or for all e_n such that $e \longmapsto^*_{ml.D} e_n$, there exists an e_{n+1} such that $e_n \longmapsto_{ml.D} e_{n+1}$ (i.e., there is an infinite reduction sequence beginning with e).*

Proof: follows from Property 6.11. ∎

6.10.2 Correspondence of DS and CPS evaluation patterns in Λ_{ml}

Definition 6.1 (Generic colon translation)
For all $\Gamma \vdash_{ml} e : \widetilde{\sigma_0}$ and $\Gamma \vdash_{ml} \varphi : \widetilde{\sigma_1}[\widetilde{\sigma_0}]$, the translation $e : \varphi$ is defined as follows:

$$\begin{aligned} e : \varphi &= \varphi[e] \text{ if } e \not\equiv \mathit{let}\, x \Leftarrow e_0 \,\mathit{in}\, e_1 \\ \mathit{let}\, x \Leftarrow e_0 \,\mathit{in}\, e_1 : \varphi &= e_0 : \mathit{let}\, x \Leftarrow [\cdot] \,\mathit{in}\, \varphi[e_1] \end{aligned}$$

Property 6.13 (Correctness of colon translation)
For all $\cdot \vdash_{ml} e : \widetilde{\sigma_0}$ and $\cdot \vdash_{ml} \varphi : \widetilde{\sigma_1}[\widetilde{\sigma_0}]$,

$$\varphi[e] \longmapsto^*_{ml.C} e : \varphi$$

Proof: by induction over the structure of e.

case $e \not\equiv \mathit{let}\, x_0 \Leftarrow e_0 \,\mathit{in}\, e_1$: immediate, since $\varphi[e] = e : \varphi$.
case $e \equiv \mathit{let}\, x_0 \Leftarrow e_0 \,\mathit{in}\, e_1$: Show

$$\begin{aligned} \varphi[\mathit{let}\, x_0 \Leftarrow e_0 \,\mathit{in}\, e_1] &\longmapsto^*_{ml.C} \mathit{let}\, x_0 \Leftarrow e_0 \,\mathit{in}\, e_1 : \varphi \\ &= e_0 : \mathit{let}\, x_0 \Leftarrow [\cdot] \,\mathit{in}\, \varphi[e_1]. \end{aligned}$$

case $\varphi \equiv [\cdot]$:

$$\begin{aligned} &\varphi[\mathit{let}\, x_0 \Leftarrow e_0 \,\mathit{in}\, e_1] \\ &\quad\equiv\quad \mathit{let}\, x_0 \Leftarrow [e_0] \,\mathit{in}\, e_1 \\ &\longmapsto^*_{ml.C}\quad e_0 : \mathit{let}\, x_0 \Leftarrow [\cdot] \,\mathit{in}\, e_1 \qquad \textit{...by ind. hyp.} \\ &\quad\equiv\quad e_0 : \mathit{let}\, x_0 \Leftarrow [\cdot] \,\mathit{in}\, \varphi[e_1] \end{aligned}$$

case $\varphi \equiv \mathit{let}\, x_c \Leftarrow [\cdot] \,\mathit{in}\, e_c$:

$$\begin{aligned} &\varphi[\mathit{let}\, x_0 \Leftarrow e_0 \,\mathit{in}\, e_1] \\ &\quad\equiv\quad \mathit{let}\, x_c \Leftarrow (\mathit{let}\, x_0 \Leftarrow e_0 \,\mathit{in}\, e_1) \,\mathit{in}\, e_c \\ &\longmapsto_{ml.C}\quad \mathit{let}\, x_0 \Leftarrow e_0 \,\mathit{in}\, \mathit{let}\, x_c \Leftarrow e_1 \,\mathit{in}\, e_c \\ &\longmapsto^*_{ml.C}\quad e_0 : \mathit{let}\, x_0 \Leftarrow [\cdot] \,\mathit{in}\, \mathit{let}\, x_c \Leftarrow e_1 \,\mathit{in}\, e_c \qquad \textit{...by ind. hyp.} \\ &\quad=\quad e_0 : \mathit{let}\, x_0 \Leftarrow [\cdot] \,\mathit{in}\, \varphi[e_1] \end{aligned}$$

∎

Property 6.14 *For all $\cdot \vdash_{ml} e : \widetilde{\sigma}$,*

$$e \longmapsto_{ml.D} e' \;\Rightarrow\; e : \varphi \longmapsto^+_{ml.C} e' : \varphi$$

Proof: by induction over the structure of the $\longmapsto_{ml.D}$ rules.

case $(\lambda x\,.\,e_0)\,e_1 \longmapsto_{ml.D} e_0[x := e_1]$: Note $(\lambda x\,.\,e_0)\,e_1 : \varphi = \varphi[(\lambda x\,.\,e_0)\,e_1]$

case $\varphi \equiv [\cdot]$:

$$\begin{aligned} &\varphi[(\lambda x\,.\,e_0)\,e_1] \\ &\quad\equiv\quad (\lambda x\,.\,e_0)\,e_1 \\ &\longmapsto_{ml.C}\quad e_0[x := e_1] \\ &\longmapsto^*_{ml.C}\quad e_0[x := e_1] : [\cdot] \qquad \textit{...by Property 6.13} \end{aligned}$$

case $\varphi \equiv \mathit{let}\, x_c \Leftarrow [\cdot]\, \mathit{in}\, e_c$:

$$\begin{array}{lll} \varphi[(\lambda x\,.\,e_0)\,e_1] & & \\ \quad \equiv & \mathit{let}\, x_c \Leftarrow (\lambda x\,.\,e_0)\,e_1\, \mathit{in}\, e_c & \\ \quad \longmapsto_{ml.C} & \mathit{let}\, x_c \Leftarrow e_0[x := e_1]\, \mathit{in}\, e_c & \\ \quad \longmapsto^*_{ml.C} & e_0[x := e_1] \,:\, \mathit{let}\, x_c \Leftarrow [\cdot]\, \mathit{in}\, e_c & \ldots \mathit{by\ Property\ 6.13} \end{array}$$

case $(\mathit{rec}\, f(x).\, e_0)\, e_1 \longmapsto_{ml.D} e_0[f := \mathit{rec}\, f(x).\, e_0\,,\, x := e_1]$: Note $(\mathit{rec}\, f(x).\, e_0)\, e_1 \,:\, \varphi \;=\; \varphi[(\mathit{rec}\, f(x).\, e_0)\, e_1]$

case $\varphi \equiv [\cdot]$:

$$\begin{array}{lll} \varphi[(\mathit{rec}\, f(x).\, e_0)\, e_1] & & \\ \quad \equiv & (\mathit{rec}\, f(x).\, e_0)\, e_1 & \\ \quad \longmapsto_{ml.C} & e_0[f := \mathit{rec}\, f(x).\, e_0\,,\, x := e_1] & \\ \quad \longmapsto^*_{ml.C} & e_0[f := \mathit{rec}\, f(x).\, e_0\,,\, x := e_1] \,:\, [\cdot] & \ldots \mathit{by\ Property\ 6.13} \end{array}$$

case $\varphi \equiv \mathit{let}\, x_c \Leftarrow [\cdot]\, \mathit{in}\, e_c$:

$$\begin{array}{lll} \varphi[(\mathit{rec}\, f(x).\, e_0)\, e_1] & & \\ \quad \equiv & \mathit{let}\, x_c \Leftarrow (\mathit{rec}\, f(x).\, e_0)\, e_1\, \mathit{in}\, e_c & \\ \quad \longmapsto_{ml.C} & \mathit{let}\, x_c \Leftarrow e_0[f := \mathit{rec}\, f(x).\, e_0\,,\, x := e_1]\, \mathit{in}\, e_c & \\ \quad \longmapsto^*_{ml.C} & e_0[f := \mathit{rec}\, f(x).\, e_0\,,\, x := e_1] \,:\, \mathit{let}\, x_c \Leftarrow [\cdot]\, \mathit{in}\, e_c & \ldots \mathit{by\ Property\ 6.13} \end{array}$$

case $\mathit{let}\, x_0 \Leftarrow [\![e_0]\!]\, \mathit{in}\, e_1 \longmapsto_{ml.D} e_1[x_0 := e_0]$:
Note $\mathit{let}\, x_0 \Leftarrow [\![e_0]\!]\, \mathit{in}\, e_1 \,:\, \varphi \;=\; [\![e_0]\!] \,:\, \mathit{let}\, x_0 \Leftarrow [\cdot]\, \mathit{in}\, \varphi[e_1] \;=\; \mathit{let}\, x_0 \Leftarrow [\![e_0]\!]\, \mathit{in}\, \varphi[e_1]$.

case $\varphi \equiv [\cdot]$:

$$\begin{array}{lll} \mathit{let}\, x_0 \Leftarrow [\![e_0]\!]\, \mathit{in}\, \varphi[e_1] & & \\ \quad \equiv & \mathit{let}\, x_0 \Leftarrow [\![e_0]\!]\, \mathit{in}\, e_1 & \\ \quad \longmapsto_{ml.C} & e_1[x_0 := e_0] & \\ \quad \longmapsto^*_{ml.C} & e_1[x_0 := e_0] \,:\, [\cdot] & \ldots \mathit{by\ Property\ 6.13} \end{array}$$

case $\varphi \equiv \mathit{let}\, x_c \Leftarrow [\cdot]\, \mathit{in}\, e_c$:

$$\begin{array}{lll} \mathit{let}\, x_0 \Leftarrow [\![e_0]\!]\, \mathit{in}\, \varphi[e_1] & & \\ \quad \equiv & \mathit{let}\, x_0 \Leftarrow [\![e_0]\!]\, \mathit{in}\, \mathit{let}\, x_c \Leftarrow e_1\, \mathit{in}\, e_c & \\ \quad \longmapsto_{ml.C} & (\mathit{let}\, x_c \Leftarrow e_1\, \mathit{in}\, e_c)[x_0 := e_0] & \\ \quad = & \mathit{let}\, x_c \Leftarrow e_1[x_0 := e_0]\, \mathit{in}\, e_c & \ldots x_0 \notin FV(e_c) \\ \quad \longmapsto^*_{ml.C} & e_1[x_0 := e_0] \,:\, \mathit{let}\, x_c \Leftarrow [\cdot]\, \mathit{in}\, e_c & \ldots \mathit{by\ Property\ 6.13} \end{array}$$

case $\mathit{let}\, x_0 \Leftarrow e_0\, \mathit{in}\, e_1 \longmapsto_{ml.D} \mathit{let}\, x_0 \Leftarrow e_0'\, \mathit{in}\, e_1$ because $e_0 \longmapsto_{ml.D} e_0'$:

$$\begin{array}{lll} \mathit{let}\, x_0 \Leftarrow e_0\, \mathit{in}\, e_1 \,:\, \varphi & & \\ \quad = & e_0 \,:\, \mathit{let}\, x_0 \Leftarrow [\cdot]\, \mathit{in}\, \varphi[e_1] & \\ \quad \longmapsto^+_{ml.C} & e_0' \,:\, \mathit{let}\, x_0 \Leftarrow [\cdot]\, \mathit{in}\, \varphi[e_1] & \ldots \mathit{by\ ind.\ hyp.} \\ \quad = & \mathit{let}\, x_0 \Leftarrow e_0'\, \mathit{in}\, e_1 \,:\, \varphi & \end{array}$$

∎

Property 6.15 *For all* $\cdot \vdash_{ml} e_0 : \widetilde{\sigma}$,

$$e_0 \longmapsto^*_{ml.D} e_n \;\;\Rightarrow\;\; e_0 \,:\, \varphi \longmapsto^*_{ml.C} e_n \,:\, \varphi$$

Proof: by induction on the number of steps n in the reduction sequence.

case $n = 0$: immediate

case $n = i + 1$: by the ind. hyp. $e_0 \,:\, \varphi \longmapsto^*_{ml.C} e_i \,:\, \varphi$ and by Property 6.14, $e_i \,:\, \varphi \longmapsto^+_{ml.C} e_{i+1} \,:\, \varphi$.

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.

Property 6.16 *For all* $\cdot \vdash_{ml} e : \widetilde{\nu}$,

$$e \longmapsto^*_{ml.D} \texttt{[}v\texttt{]} \quad \textit{iff} \quad e \longmapsto^*_{ml.C} \texttt{[}v\texttt{]}$$

Proof:

1. show $e \longmapsto^*_{ml.D} \texttt{[}v\texttt{]} \Rightarrow e \longmapsto^*_{ml.C} \texttt{[}v\texttt{]}$.

$$\begin{array}{lll} e & \longmapsto^*_{ml.C} & e : [\cdot] \qquad \textit{...by Property 6.13} \\ & \longmapsto^*_{ml.C} & \texttt{[}v\texttt{]} : [\cdot] \qquad \textit{...by Property 6.15} \\ & = & \texttt{[}v\texttt{]} \qquad \textit{...by def. of :} \end{array}$$

2. show $\neg(e \longmapsto^*_{ml.D} \texttt{[}v\texttt{]}) \Rightarrow \neg(e \longmapsto^*_{ml.C} \texttt{[}v\texttt{]})$.

   By Property 6.12, $\neg(e \longmapsto^*_{ml.D} \texttt{[}v\texttt{]})$ implies there exists a non-terminating reduction sequence $e \longmapsto_{ml.D} e_1 \longmapsto_{ml.D} \ldots$. Therefore,

$$\begin{array}{lll} e & \longmapsto^*_{ml.C} & e : [\cdot] \qquad \textit{...by Property 6.13} \\ & \longmapsto^+_{ml.C} & e_1 : [\cdot] \qquad \textit{...by Property 6.14} \\ & \longmapsto^+_{ml.C} & \ldots \qquad \textit{...a non-terminating reduction sequence by Property 6.15} \end{array}$$

6.10.3 Correctness of encodings into Λ_{ml}

In this section we prove the correctness of the call-by-name meta-language encoding $\mathcal{C}_n$. The proofs for $\mathcal{C}_v$ are very similar and omitted. In general, the strategy for proving the correctness of encodings is to show that:

- the encoding preserves well-typedness of terms,
- the encoding commutes with relevant substitutions,
- one Λ^σ evaluation step implies one or more Λ_{ml} evaluation steps,
- a Λ^σ evaluation sequence implies an Λ_{ml} sequence, and
- based on the above conclude that the encoding is a correct simulation.

The following property shows that $\mathcal{E}_n$ preserves well-typedness of terms.

Property 6.17 *If* $\Gamma \vdash e : \sigma$ *then* $\mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} \mathcal{E}_n\langle\![e]\!\rangle : \mathcal{E}_n\langle\![\sigma]\!\rangle$

Proof: by induction over the structure of the typing derivation of $\Gamma \vdash e : \sigma$.

case Λ.*(const)*: $\Gamma \vdash c : \iota$:

$$\begin{array}{lll} \mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} c : \iota & & \ldots\Lambda_{ml}.\textit{(const)} \\ \Rightarrow & \mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} \texttt{[}c\texttt{]} : \widetilde{\iota} & \\ \Rightarrow & \mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} \mathcal{E}_n\langle\![c]\!\rangle : \mathcal{E}_n\langle\![\iota]\!\rangle & \end{array}$$

case Λ.*(var)*: $\Gamma \vdash x : \Gamma(x)$:

$$\begin{array}{lll} \mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} x : \mathcal{E}_n\langle\![\Gamma]\!\rangle(x) & & \ldots\Lambda_{ml}.\textit{(var)} \\ \Rightarrow & \mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} x : \mathcal{E}_n\langle\![\Gamma(x)]\!\rangle & \textit{...by def. of } \mathcal{E}_n \textit{ on type assumptions} \\ \Rightarrow & \mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} \mathcal{E}_n\langle\![x]\!\rangle : \mathcal{E}_n\langle\![\Gamma(x)]\!\rangle & \end{array}$$

case Λ.*(var)*: $\Gamma \vdash f : \Gamma(f)$:

$$
\begin{array}{lll}
\mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} f : \mathcal{E}_n\langle\![\Gamma]\!\rangle(f) & & \ldots\Lambda_{ml}.(var) \\
\quad\Rightarrow\quad \mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} f : \mathcal{E}_n\langle\widetilde{\Gamma(f)}\rangle & & \ldots by\ def.\ of\ \mathcal{E}_n\ on\ type\ assumptions \\
\quad\Rightarrow\quad \mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} [f] : \mathcal{E}_n\langle\widetilde{\Gamma(f)}\rangle & & \ldots\Lambda_{ml}.(unit) \\
\quad\Rightarrow\quad \mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} \mathcal{E}_n\langle\![f]\!\rangle : \mathcal{E}_n\langle\![\Gamma(f)]\!\rangle &&
\end{array}
$$

case Λ.*(abs)*: $\Gamma, x:\sigma_1 \vdash e : \sigma_2 \;\Rightarrow\; \Gamma \vdash \lambda x\,.\,e : \sigma_1 \rightarrow \sigma_2$:

$$
\begin{array}{lll}
\Gamma, x:\sigma_1 \vdash e : \sigma_2 & & \\
\quad\Rightarrow\quad \mathcal{E}_n\langle\![\Gamma, x:\sigma_1]\!\rangle \vdash_{ml} \mathcal{E}_n\langle\![e]\!\rangle : \mathcal{E}_n\langle\![\sigma_2]\!\rangle & & \ldots by\ ind.\ hyp. \\
\quad\Rightarrow\quad \mathcal{E}_n\langle\![\Gamma]\!\rangle, x:\mathcal{E}_n\langle\![\sigma_1]\!\rangle \vdash_{ml} \mathcal{E}_n\langle\![e]\!\rangle : \widetilde{\mathcal{E}_n\langle\![\sigma_2]\!\rangle} & & \ldots by\ def.\ of\ \mathcal{E}_n\ on\ type\ assumptions \\
\quad\Rightarrow\quad \mathcal{E}_n\langle\![\Gamma]\!\rangle, x:\mathcal{E}_n\langle\![\sigma_1]\!\rangle \vdash_{ml} \mathcal{E}_n\langle\![e]\!\rangle : \widetilde{\mathcal{E}_n\langle\![\sigma_2]\!\rangle} & & \\
\quad\Rightarrow\quad \mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} \lambda x\,.\,\mathcal{E}_n\langle\![e]\!\rangle : \mathcal{E}_n\langle\![\sigma_1]\!\rangle \rightarrow \widetilde{\mathcal{E}_n\langle\![\sigma_2]\!\rangle} & & \ldots\Lambda_{ml}.(abs) \\
\quad\Rightarrow\quad \mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} \lambda x\,.\,\mathcal{E}_n\langle\![e]\!\rangle : \widetilde{\mathcal{E}_n\langle\![\sigma_1 \rightarrow \sigma_2]\!\rangle} & & \\
\quad\Rightarrow\quad \mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} [\lambda x\,.\,\mathcal{E}_n\langle\![e]\!\rangle] : \widetilde{\mathcal{E}_n\langle\![\sigma_1 \rightarrow \sigma_2]\!\rangle} & & \ldots\Lambda_{ml}.(unit) \\
\quad\Rightarrow\quad \mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} \mathcal{E}_n\langle\![\lambda x\,.\,e]\!\rangle : \mathcal{E}_n\langle\![\sigma_1 \rightarrow \sigma_2]\!\rangle &&
\end{array}
$$

case Λ.*(rec)*: $\Gamma, f:\sigma_1 \rightarrow \sigma_2, x:\sigma_1 \vdash e : \sigma_2 \;\Rightarrow\; \Gamma \vdash rec\, f(x).\, e : \sigma_1 \rightarrow \sigma_2$:

$$
\begin{array}{lll}
\Gamma, f:\sigma_1 \rightarrow \sigma_2, x:\sigma_1 \vdash e : \sigma_2 & & \\
\quad\Rightarrow\quad \mathcal{E}_n\langle\![\Gamma, f:\sigma_1 \rightarrow \sigma_2, x:\sigma_1]\!\rangle \vdash_{ml} \mathcal{E}_n\langle\![e]\!\rangle : \mathcal{E}_n\langle\![\sigma_2]\!\rangle & & \ldots by\ ind.\ hyp. \\
\quad\Rightarrow\quad \mathcal{E}_n\langle\![\Gamma]\!\rangle, f:\mathcal{E}_n\langle\sigma_1 \rightarrow \sigma_2\rangle, x:\mathcal{E}_n\langle\![\sigma_1]\!\rangle \vdash_{ml} \mathcal{E}_n\langle\![e]\!\rangle : \widetilde{\mathcal{E}_n\langle\![\sigma_2]\!\rangle} & & \ldots by\ def.\ of\ \mathcal{E}_n\ on\ type\ assumptions \\
\quad\Rightarrow\quad \mathcal{E}_n\langle\![\Gamma]\!\rangle, f:\mathcal{E}_n\langle\![\sigma_1]\!\rangle \rightarrow \widetilde{\mathcal{E}_n\langle\sigma_2\rangle}, x:\mathcal{E}_n\langle\![\sigma_1]\!\rangle \vdash_{ml} \mathcal{E}_n\langle\![e]\!\rangle : \widetilde{\mathcal{E}_n\langle\sigma_2\rangle} & & \\
\quad\Rightarrow\quad \mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} rec\, f(x).\, \mathcal{E}_n\langle\![e]\!\rangle : \mathcal{E}_n\langle\![\sigma_1]\!\rangle \rightarrow \widetilde{\mathcal{E}_n\langle\sigma_2\rangle} & & \ldots\Lambda_{ml}.(rec) \\
\quad\Rightarrow\quad \mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} rec\, f(x).\, \mathcal{E}_n\langle\![e]\!\rangle : \widetilde{\mathcal{E}_n\langle\sigma_1 \rightarrow \sigma_2\rangle} & & \\
\quad\Rightarrow\quad \mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} [rec\, f(x).\, \mathcal{E}_n\langle\![e]\!\rangle] : \widetilde{\mathcal{E}_n\langle\sigma_1 \rightarrow \sigma_2\rangle} & & \ldots\Lambda_{ml}.(unit) \\
\quad\Rightarrow\quad \mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} \mathcal{E}_n\langle\![rec\, f(x).\, e]\!\rangle : \mathcal{E}_n\langle\![\sigma_1 \rightarrow \sigma_2]\!\rangle &&
\end{array}
$$

case Λ.*(app)*: $\Gamma \vdash e_0 : \sigma_1 \rightarrow \sigma_2,\ \Gamma \vdash e_1 : \sigma_1 \;\Rightarrow\; \Gamma \vdash e_0\, e_1 : \sigma_2$:

$$
\begin{array}{lll}
\mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} \mathcal{E}_n\langle\![e_0]\!\rangle : \mathcal{E}_n\langle\![\sigma_1 \rightarrow \sigma_2]\!\rangle, \quad \widetilde{\mathcal{E}_n\langle\![\Gamma]\!\rangle} \vdash_{ml} \mathcal{E}_n\langle\![e_1]\!\rangle : \mathcal{E}_n\langle\![\sigma_1]\!\rangle & & \ldots by\ ind.\ hyp. \\
\quad\Rightarrow\quad \mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} \mathcal{E}_n\langle\![e_0]\!\rangle : \widetilde{\mathcal{E}_n\langle\![\sigma_1]\!\rangle \rightarrow \mathcal{E}_n\langle\![\sigma_2]\!\rangle} & & \\
\text{Now}\quad \mathcal{E}_n\langle\![\Gamma]\!\rangle, x:\mathcal{E}_n\langle\![\sigma_1]\!\rangle \rightarrow \mathcal{E}_n\langle\![\sigma_2]\!\rangle \vdash_{ml} x : \mathcal{E}_n\langle\![\sigma_1]\!\rangle \rightarrow \mathcal{E}_n\langle\![\sigma_2]\!\rangle & & \ldots\Lambda_{ml}.(var) \\
\quad\Rightarrow\quad \mathcal{E}_n\langle\![\Gamma]\!\rangle, x:\mathcal{E}_n\langle\![\sigma_1]\!\rangle \rightarrow \mathcal{E}_n\langle\![\sigma_2]\!\rangle \vdash_{ml} x\, \mathcal{E}_n\langle\![e_1]\!\rangle : \mathcal{E}_n\langle\![\sigma_2]\!\rangle & & \ldots\Lambda_{ml}.(app) \\
\quad\Rightarrow\quad \mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} let\ x \Leftarrow \mathcal{E}_n\langle\![e_0]\!\rangle\ in\ x\, \mathcal{E}_n\langle\![e_1]\!\rangle : \mathcal{E}_n\langle\![\sigma_2]\!\rangle & & \ldots\Lambda_{ml}.(let) \\
\quad\Rightarrow\quad \mathcal{E}_n\langle\![\Gamma]\!\rangle \vdash_{ml} \mathcal{E}_n\langle\![e_0\, e_1]\!\rangle : \mathcal{E}_n\langle\![\sigma_2]\!\rangle &&
\end{array}
$$

∎

The following property shows how $\mathcal{E}_n$ interacts with relevant substitutions.

Property 6.18 *For all* $e, e' \in TypedTerms[\Lambda^\sigma]$,

$$
\begin{array}{rcl}
\mathcal{E}_n\langle\![e[x := e']]\!\rangle & = & \mathcal{E}_n\langle\![e]\!\rangle[x := \mathcal{E}_n\langle\![e']\!\rangle] \\
\mathcal{E}_n\langle\![e[f := rec\, f(x).\, e']]\!\rangle & = & \mathcal{E}_n\langle\![e]\!\rangle[f := \mathcal{E}_n\langle rec\, f(x).\, e'\rangle]
\end{array}
$$

Proof: by induction over the structure of e. A few of the more interesting cases are as follows.

case $e \equiv c$:

$$
\begin{array}{rcl}
\mathcal{E}_n\langle\![c[x := e']]\!\rangle & = & \mathcal{E}_n\langle\![c]\!\rangle \\
& = & [c] \\
& = & [c][x := \mathcal{E}_n\langle\![e']\!\rangle] \\
& = & \mathcal{E}_n\langle\![c]\!\rangle[x := \mathcal{E}_n\langle\![e']\!\rangle]
\end{array}
$$

case $e \equiv x$

$$\begin{array}{lcl}\mathcal{E}_n\langle\!|x[x := e']|\!\rangle & = & \mathcal{E}_n\langle\!|e'|\!\rangle \\ & = & x[x := \mathcal{E}_n\langle\!|e'|\!\rangle] \\ & = & \mathcal{E}_n\langle\!|x|\!\rangle[x := \mathcal{E}_n\langle\!|e'|\!\rangle]\end{array}$$

case $e \equiv f$:

$$\begin{array}{lcl}\mathcal{E}_n\langle\!|f[f := rec\, f(x).\, e']|\!\rangle & = & \mathcal{E}_n\langle\!|rec\, f(x).\, e'|\!\rangle \\ & = & [\mathcal{E}_n\langle rec\, f(x).\, e'\rangle] \\ & = & [f][f := \mathcal{E}_n\langle rec\, f(x).\, e'\rangle] \\ & = & \mathcal{E}_n\langle\!|f|\!\rangle[f := \mathcal{E}_n\langle rec\, f(x).\, e'\rangle]\end{array}$$

The rest of the cases follow immediately from the ind. hyp. and the definition of substitution.

∎

We now show that one Λ^σ evaluation step implies one or more Λ_{ml} evaluation steps.

Property 6.19 *For all* $\cdot \vdash e : \sigma$,

$$e \longmapsto_n e' \;\Rightarrow\; \mathcal{E}_n\langle\!|e|\!\rangle \longmapsto^+_{ml.D} \mathcal{E}_n\langle\!|e'|\!\rangle$$

Proof: by induction over the structure of the $\longmapsto_n$ rules.

case $(\lambda x\,.\,e_0)\,e_1 \longmapsto_n e_0[x := e_1]$:

$$\begin{array}{lcll}\mathcal{E}_n\langle\!|(\lambda x\,.\,e_0)\,e_1|\!\rangle & = & let\ y \Leftarrow [\lambda x\,.\,\mathcal{E}_n\langle\!|e_0|\!\rangle]\ in\ y\,\mathcal{E}_n\langle\!|e_1|\!\rangle & \\ \longmapsto_{ml.D} & & (\lambda x\,.\,\mathcal{E}_n\langle\!|e_0|\!\rangle)\,\mathcal{E}_n\langle\!|e_1|\!\rangle & \\ \longmapsto_{ml.D} & & \mathcal{E}_n\langle\!|e_0|\!\rangle[x := \mathcal{E}_n\langle\!|e_1|\!\rangle] & \\ & = & \mathcal{E}_n\langle\!|e_0[x := e_1]|\!\rangle & \textit{...by Property 6.18}\end{array}$$

case $(rec\, f(x).\, e_0)\,e_1 \longmapsto_n e_0[f := rec\, f(x).\, e_0\ ,\ x := e_1]$:

$$\begin{array}{lcll}\mathcal{E}_n\langle\!|(rec\, f(x).\, e_0)\,e_1|\!\rangle & = & let\ y \Leftarrow [rec\, f(x).\, \mathcal{E}_n\langle\!|e_0|\!\rangle]\ in\ y\,\mathcal{E}_n\langle\!|e_1|\!\rangle & \\ \longmapsto_{ml.D} & & (rec\, f(x).\, \mathcal{E}_n\langle\!|e_0|\!\rangle)\,\mathcal{E}_n\langle\!|e_1|\!\rangle & \\ \longmapsto_{ml.D} & & \mathcal{E}_n\langle\!|e_0|\!\rangle[f := rec\, f(x).\, \mathcal{E}_n\langle\!|e_0|\!\rangle\ ,\ x := \mathcal{E}_n\langle\!|e_1|\!\rangle] & \\ & = & \mathcal{E}_n\langle\!|e_0|\!\rangle[f := \mathcal{E}_n\langle rec\, f(x).\, e_0\rangle\ ,\ x := \mathcal{E}_n\langle\!|e_1|\!\rangle] & \\ & = & \mathcal{E}_n\langle\!|e_0[f := rec\, f(x).\, e_0\ ,\ x := e_1]|\!\rangle & \textit{...by Property 6.18}\end{array}$$

case $e_0\,e_1 \longmapsto_n e_0'\,e_1$ because $e_0 \longmapsto_n e_0'$:

$$\begin{array}{lcll}\mathcal{E}_n\langle\!|e_0\,e_1|\!\rangle & = & let\ y \Leftarrow \mathcal{E}_n\langle\!|e_0|\!\rangle\ in\ y\,\mathcal{E}_n\langle\!|e_1|\!\rangle & \\ \longmapsto^+_{ml.D} & & let\ y \Leftarrow \mathcal{E}_n\langle\!|e_0'|\!\rangle\ in\ y\,\mathcal{E}_n\langle\!|e_1|\!\rangle & \textit{...since by ind. hyp. } \mathcal{E}_n\langle\!|e_0|\!\rangle \longmapsto^+_{ml.D} \mathcal{E}_n\langle\!|e_0'|\!\rangle \\ & = & \mathcal{E}_n\langle\!|e_0'\,e_1|\!\rangle & \end{array}$$

∎

The following property shows that a Λ^σ evaluation sequence implies an Λ_{ml} sequence.

Property 6.20 *For all* $\cdot \vdash e_0 : \sigma$,

$$e_0 \longmapsto^*_n e_n \;\Rightarrow\; \mathcal{E}_n\langle\!|e_0|\!\rangle \longmapsto^*_{ml.D} \mathcal{E}_n\langle\!|e_n|\!\rangle$$

Proof: by induction on the number of steps n in the reduction sequence.

case $n = 0$: immediate

case $n = i + 1$: by the ind. hyp. $\mathcal{E}_n\langle\!|e_0|\!\rangle \longmapsto^*_{ml.D} \mathcal{E}_n\langle\!|e_i|\!\rangle$ and by Property 6.19, $\mathcal{E}_n\langle\!|e_i|\!\rangle \longmapsto^+_{ml.D} \mathcal{E}_n\langle\!|e_{i+1}|\!\rangle$.

∎

Finally, we show the encoding correctly simulates evaluation.

Property 6.21 *For all* $\cdot \vdash e : \sigma$,

$$\mathcal{E}_n\langle eval_n(e)\rangle \simeq eval_{ml}(\mathcal{E}_n\langle\!\langle e\rangle\!\rangle)$$

Proof:

1. show $e \longmapsto_n^* v \Rightarrow \mathcal{E}_n\langle\!\langle e\rangle\!\rangle \longmapsto_{ml.D}^* \mathtt{[}\mathcal{E}_n\langle v\rangle\mathtt{]}$: by Property 6.20, $\mathcal{E}_n\langle\!\langle e\rangle\!\rangle \longmapsto_{ml.D}^* \mathcal{E}_n\langle\!\langle v\rangle\!\rangle \equiv \mathtt{[}\mathcal{E}_n\langle v\rangle\mathtt{]}$.

2. show $eval_n(e)\uparrow \Rightarrow eval_{ml}(\mathcal{E}_n\langle\!\langle e\rangle\!\rangle)\uparrow$.

   By Property 2.6, $eval_n(e) \uparrow$ implies there exists a non-terminating reduction sequence $e \longmapsto_n e_1 \longmapsto_n \ldots$. Therefore,

   $$\begin{array}{lll} \mathcal{E}_n\langle\!\langle e\rangle\!\rangle & \longmapsto_{ml.D}^+ \mathcal{E}_n\langle\!\langle e_1\rangle\!\rangle & \textit{...by Property 6.19} \\ & \longmapsto_{ml.D}^+ \mathcal{E}_n\langle\!\langle e_2\rangle\!\rangle & \textit{...by Property 6.19} \\ & \longmapsto_{ml.D}^+ \ldots & \textit{...a non-terminating reduction sequence by Property 6.19} \end{array}$$

   and so $eval_{ml}(\mathcal{E}_n\langle\!\langle e\rangle\!\rangle)\uparrow$.

∎

6.10.4 Correctness of the language of CPS terms

This section shows that the language Λ_{cps} is exactly the set K of terms in the image of $\mathcal{K}$ closed under $\beta_a \mathrm{rec}_a \eta_v$-reduction (*i.e.*, R_{cps}-reduction). In other words, $\Lambda_{cps} = K$. K is defined formally as follows.

Definition 6.2

$$K \stackrel{\text{def}}{=} \{v \in \Lambda^\sigma \mid \exists e \in \mathit{Typed\,Terms}[\Lambda_{ml}] \,.\, \mathcal{K}\langle\!\langle e\rangle\!\rangle \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} v\}$$

We begin by showing that terms produced by $\mathcal{K}$ lie within the language Λ_{cps}. As indicated by the definition of K, CPS terms form a subset of Λ^σ. Accordingly, we show that the language Λ_{cps} is indeed a subset of Λ^σ. Then we show that Λ_{cps} obeys a subject-reduction property respect to R_{cps} reductions (*i.e.*, Λ_{cps} is closed under R_{cps} reductions). This allows us to conclude that $K \subseteq \Lambda_{cps}$.

To show that $\Lambda_{cps} \subseteq K$, we prove that all terms in Λ_{cps} are reachable from terms in the image of $\mathcal{K}$. It follows that $\Lambda_{cps} = K$.

The following property states the Λ_{cps} contains the terms in the image of $\mathcal{K}$.

Property 6.22 *If* $\Gamma \vdash_{ml} e : \sigma$ *then* $\mathcal{K}\langle\!\langle\Gamma\rangle\!\rangle \vdash_{val} \mathcal{K}\langle\!\langle e\rangle\!\rangle : \mathcal{K}\langle\!\langle\sigma\rangle\!\rangle$.

Proof: by induction over the structure of the typing derivation $\Gamma \vdash_{ml} e : \sigma$.

case Λ_{ml}.*(const)*: $\Gamma \vdash_{ml} c : \iota$:

$$\begin{array}{ll} \mathcal{K}\langle\!\langle\Gamma\rangle\!\rangle \vdash_{val} c : \iota & \textit{...by (val-const)} \\ \quad\Rightarrow\quad \mathcal{K}\langle\!\langle\Gamma\rangle\!\rangle \vdash_{val} \mathcal{K}\langle\!\langle c\rangle\!\rangle : \mathcal{K}\langle\!\langle\iota\rangle\!\rangle & \textit{...by def. of } \mathcal{K} \end{array}$$

case Λ_{ml}.*(var)*: $\Gamma \vdash_{ml} x : \Gamma(x)$:

$$\begin{array}{ll} \mathcal{K}\langle\!\langle\Gamma\rangle\!\rangle \vdash_{val} x : \mathcal{K}\langle\!\langle\Gamma\rangle\!\rangle(x) & \textit{...by (val-var)} \\ \quad\Rightarrow\quad \mathcal{K}\langle\!\langle\Gamma\rangle\!\rangle \vdash_{val} x : \mathcal{K}\langle\!\langle\Gamma(x)\rangle\!\rangle & \textit{...by def. of } \mathcal{K} \textit{ on type assumptions} \\ \quad\Rightarrow\quad \mathcal{K}\langle\!\langle\Gamma\rangle\!\rangle \vdash_{val} \mathcal{K}\langle\!\langle x\rangle\!\rangle : \mathcal{K}\langle\!\langle\Gamma(x)\rangle\!\rangle & \end{array}$$

case Λ_{ml}.*(abs)*: $\Gamma, x:\sigma_1 \vdash_{ml} e : \widetilde{\sigma_2} \Rightarrow \Gamma \vdash_{ml} \lambda x\,.\,e : \sigma_1 \rightarrow \widetilde{\sigma_2}$:

$$\begin{array}{ll} \Gamma, x:\sigma_1 \vdash_{ml} e : \widetilde{\sigma_2} & \\ \quad\Rightarrow\quad \mathcal{K}\langle\!\langle\Gamma, x:\sigma_1\rangle\!\rangle \vdash_{val} \mathcal{K}\langle\!\langle e\rangle\!\rangle : \mathcal{K}\langle\!\langle\widetilde{\sigma_2}\rangle\!\rangle & \textit{...by ind. hyp.} \\ \quad\Rightarrow\quad \mathcal{K}\langle\!\langle\Gamma\rangle\!\rangle, x:\mathcal{K}\langle\!\langle\sigma_1\rangle\!\rangle \vdash_{val} \mathcal{K}\langle\!\langle e\rangle\!\rangle : \mathcal{K}\langle\!\langle\widetilde{\sigma_2}\rangle\!\rangle & \textit{...by def. of } \mathcal{K} \textit{ on type assumptions} \\ \quad\Rightarrow\quad \mathcal{K}\langle\!\langle\Gamma\rangle\!\rangle \vdash_{val} \lambda x\,.\,\mathcal{K}\langle\!\langle e\rangle\!\rangle : \mathcal{K}\langle\!\langle\sigma_1\rangle\!\rangle \rightarrow \mathcal{K}\langle\!\langle\widetilde{\sigma_2}\rangle\!\rangle & \textit{...by (val-abs)} \\ \quad\Rightarrow\quad \mathcal{K}\langle\!\langle\Gamma\rangle\!\rangle \vdash_{val} \mathcal{K}\langle\!\langle\lambda x\,.\,e\rangle\!\rangle : \mathcal{K}\langle\!\langle\sigma_1 \rightarrow \widetilde{\sigma_2}\rangle\!\rangle & \end{array}$$

case Λ_{ml}.*(rec)*: $\Gamma, f:\sigma_1 \to \widetilde{\sigma_2}, x:\sigma_1 \vdash_{ml} e : \widetilde{\sigma_2} \;\Rightarrow\; \Gamma \vdash_{ml} rec\, f(x).\, e : \sigma_1 \to \widetilde{\sigma_2}$:

$\Gamma, f:\sigma_1 \to \widetilde{\sigma_2}, x:\sigma_1 \vdash_{ml} e : \widetilde{\sigma_2}$
$\Rightarrow$ $\mathcal{K}\langle\![\Gamma, f:\sigma_1 \to \widetilde{\sigma_2}, x:\sigma_1]\!\rangle \vdash_{val} \mathcal{K}\langle\![e]\!\rangle : \mathcal{K}\langle\![\widetilde{\sigma_2}]\!\rangle$ *...by ind. hyp.*
$\Rightarrow$ $\mathcal{K}\langle\![\Gamma]\!\rangle, f:\mathcal{K}\langle\![\sigma_1 \to \widetilde{\sigma_2}]\!\rangle, x:\mathcal{K}\langle\![\sigma_1]\!\rangle \vdash_{val} \mathcal{K}\langle\![e]\!\rangle : \mathcal{K}\langle\![\widetilde{\sigma_2}]\!\rangle$ *...by def. of $\mathcal{K}$ on type assumptions*
$\Rightarrow$ $\mathcal{K}\langle\![\Gamma]\!\rangle \vdash_{val} rec\, f(x).\, \mathcal{K}\langle\![e]\!\rangle : \mathcal{K}\langle\![\sigma_1]\!\rangle \to \mathcal{K}\langle\![\widetilde{\sigma_2}]\!\rangle$ *...by (val-rec)*
$\Rightarrow$ $\mathcal{K}\langle\![\Gamma]\!\rangle \vdash_{val} \mathcal{K}\langle\![rec\, f(x).\, e]\!\rangle : \mathcal{K}\langle\![\sigma_1 \to \widetilde{\sigma_2}]\!\rangle$

case Λ_{ml}.*(app)*: $\Gamma \vdash_{ml} e_0 : \sigma_1 \to \widetilde{\sigma_2},\ \Gamma \vdash_{ml} e_1 : \sigma_1 \;\Rightarrow\; \Gamma \vdash_{ml} e_0\, e_1 : \widetilde{\sigma_2}$:

$\Gamma \vdash_{ml} e_0 : \sigma_1 \to \widetilde{\sigma_2},\ \Gamma \vdash_{ml} e_1 : \sigma_1$
$\Rightarrow$ $\mathcal{K}\langle\![\Gamma]\!\rangle \vdash_{val} \mathcal{K}\langle\![e_0]\!\rangle : \mathcal{K}\langle\![\sigma_1 \to \widetilde{\sigma_2}]\!\rangle,\ \mathcal{K}\langle\![\Gamma]\!\rangle \vdash_{val} \mathcal{K}\langle\![e_1]\!\rangle : \mathcal{K}\langle\![\sigma_1]\!\rangle$ *...by ind. hyp.*
$\Rightarrow$ $\mathcal{K}\langle\![\Gamma]\!\rangle \vdash_{val} \mathcal{K}\langle\![e_0]\!\rangle : \mathcal{K}\langle\![\sigma_1]\!\rangle \to \neg\neg\mathcal{K}\langle\![\sigma_2]\!\rangle,\ \mathcal{K}\langle\![\Gamma]\!\rangle \vdash_{val} \mathcal{K}\langle\![e_1]\!\rangle : \mathcal{K}\langle\![\sigma_1]\!\rangle$
$\Rightarrow$ $\mathcal{K}\langle\![\Gamma]\!\rangle \vdash_{exp} \mathcal{K}\langle\![e_0]\!\rangle\, \mathcal{K}\langle\![e_1]\!\rangle : \neg\neg\mathcal{K}\langle\![\sigma_2]\!\rangle$ *...by (exp-app)*
$\Rightarrow$ $\langle \mathcal{K}\langle\![\Gamma]\!\rangle\, ;\, k:\neg\mathcal{K}\langle\![\sigma_2]\!\rangle \rangle \vdash_{ans} (\mathcal{K}\langle\![e_0]\!\rangle\, \mathcal{K}\langle\![e_1]\!\rangle)\, k : ans$ *...by (cont-var),(ans-app.2)*
$\Rightarrow$ $\mathcal{K}\langle\![\Gamma]\!\rangle \vdash_{val} \lambda k\,.\, (\mathcal{K}\langle\![e_0]\!\rangle\, \mathcal{K}\langle\![e_1]\!\rangle)\, k : \neg\mathcal{K}\langle\![\sigma_2]\!\rangle \to ans$ *...by (val-comp)*
$\Rightarrow$ $\mathcal{K}\langle\![\Gamma]\!\rangle \vdash_{val} \mathcal{K}\langle\![e_0\, e_1]\!\rangle : \mathcal{K}\langle\![\widetilde{\sigma_2}]\!\rangle$

case Λ_{ml}.*(unit)*: $\Gamma \vdash_{ml} e : \sigma \;\Rightarrow\; \Gamma \vdash_{ml} [e] : \widetilde{\sigma}$:

$\Gamma \vdash_{ml} e : \sigma$
$\Rightarrow$ $\mathcal{K}\langle\![\Gamma]\!\rangle \vdash_{val} \mathcal{K}\langle\![e]\!\rangle : \mathcal{K}\langle\![\sigma]\!\rangle$ *...by ind. hyp.*
$\Rightarrow$ $\langle \mathcal{K}\langle\![\Gamma]\!\rangle\, ;\, k:\neg\mathcal{K}\langle\![\sigma]\!\rangle \rangle \vdash_{ans} k\, \mathcal{K}\langle\![e]\!\rangle : ans$ *...by (cont-var), (ans-app.1)*
$\Rightarrow$ $\mathcal{K}\langle\![\Gamma]\!\rangle \vdash_{val} \lambda k\,.\, k\, \mathcal{K}\langle\![e]\!\rangle : \neg\neg\mathcal{K}\langle\![\sigma]\!\rangle$
$\Rightarrow$ $\mathcal{K}\langle\![\Gamma]\!\rangle \vdash_{val} \mathcal{K}\langle\![[e]]\!\rangle : \mathcal{K}\langle\![\widetilde{\sigma}]\!\rangle$

case Λ_{ml}.*(let)*: $\Gamma \vdash_{ml} e_1 : \widetilde{\sigma_1},\ \Gamma, x:\sigma_1 \vdash_{ml} e_2 : \widetilde{\sigma_2} \;\Rightarrow\; \Gamma \vdash_{ml} let\, x \Leftarrow e_1\, in\, e_2 : \widetilde{\sigma}$:

$\Gamma, x:\sigma_1 \vdash_{ml} e_2 : \widetilde{\sigma_2}$
$\Rightarrow$ $\mathcal{K}\langle\![\Gamma, x:\sigma_1]\!\rangle \vdash_{val} \mathcal{K}\langle\![e_2]\!\rangle : \mathcal{K}\langle\![\widetilde{\sigma_2}]\!\rangle$ *...by ind. hyp.*
$\Rightarrow$ $\mathcal{K}\langle\![\Gamma]\!\rangle, x:\mathcal{K}\langle\![\sigma_1]\!\rangle \vdash_{val} \mathcal{K}\langle\![e_2]\!\rangle : \neg\neg\mathcal{K}\langle\![\sigma_2]\!\rangle$
$\Rightarrow$ $\langle \mathcal{K}\langle\![\Gamma]\!\rangle, x:\mathcal{K}\langle\![\sigma_1]\!\rangle\, ;\, k:\neg\mathcal{K}\langle\![\sigma_2]\!\rangle \rangle \vdash_{ans} \mathcal{K}\langle\![e_2]\!\rangle\, k : ans$ *...by (cont-var),(ans-app.2)*
$\Rightarrow$ $\langle \mathcal{K}\langle\![\Gamma]\!\rangle\, ;\, k:\neg\mathcal{K}\langle\![\sigma_2]\!\rangle \rangle \vdash_{cont} \lambda x\,.\, \mathcal{K}\langle\![e_2]\!\rangle\, k : \neg\mathcal{K}\langle\![\sigma_1]\!\rangle$ (†) *...by (cont-abs)*
Now $\Gamma \vdash_{ml} e_1 : \widetilde{\sigma_1}$
$\Rightarrow$ $\mathcal{K}\langle\![\Gamma]\!\rangle \vdash_{val} \mathcal{K}\langle\![e_1]\!\rangle : \mathcal{K}\langle\![\widetilde{\sigma_1}]\!\rangle$ *...by ind. hyp.*
$\Rightarrow$ $\mathcal{K}\langle\![\Gamma]\!\rangle \vdash_{val} \mathcal{K}\langle\![e_1]\!\rangle : \neg\neg\mathcal{K}\langle\![\sigma_1]\!\rangle$
$\Rightarrow$ $\langle \mathcal{K}\langle\![\Gamma]\!\rangle\, ;\, k:\neg\mathcal{K}\langle\![\sigma_2]\!\rangle \rangle \vdash_{ans} \mathcal{K}\langle\![e_1]\!\rangle\, (\lambda x\,.\, \mathcal{K}\langle\![e_2]\!\rangle\, k) : ans$ *...by (†) and (ans-app.2)*
$\Rightarrow$ $\mathcal{K}\langle\![\Gamma]\!\rangle \vdash_{val} \lambda k\,.\, \mathcal{K}\langle\![e_1]\!\rangle\, (\lambda x\,.\, \mathcal{K}\langle\![e_2]\!\rangle\, k) : \neg\neg\mathcal{K}\langle\![\sigma_2]\!\rangle$ *...by (val-comp)*
$\Rightarrow$ $\mathcal{K}\langle\![\Gamma]\!\rangle \vdash_{val} \mathcal{K}\langle\![let\, x \Leftarrow e_1\, in\, e_2]\!\rangle : \mathcal{K}\langle\![\widetilde{\sigma_2}]\!\rangle$

∎

The following property states that Λ_{cps} is a sub-language of Λ^{σ}.

Property 6.23

$$
\begin{array}{lrcl}
(a) & \Gamma \vdash_{val} v : \sigma & \Rightarrow & \Gamma \vdash v : \sigma \\
(b) & \Gamma \vdash_{exp} w : \sigma & \Rightarrow & \Gamma \vdash w : \sigma \\
(c) & \langle \Gamma\, ;\, k:\neg\sigma \rangle \vdash_{ans} \alpha : ans & \Rightarrow & \Gamma, k:\neg\sigma \vdash \alpha : ans \\
(d) & \langle \Gamma\, ;\, k:\neg\sigma_0 \rangle \vdash_{cont} \kappa : \neg\sigma_1 & \Rightarrow & \Gamma, k:\neg\sigma_0 \vdash \kappa : \neg\sigma_1
\end{array}
$$

Proof: by a straightforward simultaneous induction over the typing derivations of $\Gamma \vdash_{val} v : \sigma$, $\Gamma \vdash_{exp} w : \sigma$, $\langle \Gamma\, ;\, k:\neg\sigma \rangle \vdash_{ans} \alpha : ans$, and $\langle \Gamma\, ;\, k:\neg\sigma_0 \rangle \vdash_{cont} \kappa : \neg\sigma_1$. ∎

The two following properties state that Λ_{cps} is closed under relevant substitutions.

Property 6.24 *Given* $\Gamma \vdash_{val} v_1 : \sigma_1$,

$$\begin{array}{lrcl}
(a) & \Gamma, x:\sigma_1 \vdash_{val} v_0 : \sigma_0 & \Rightarrow & \Gamma \vdash_{val} v_0[x := v_1] : \sigma_1 \\
(b) & \Gamma, x:\sigma_1 \vdash_{exp} w : \sigma_0 & \Rightarrow & \Gamma \vdash_{exp} w[x := v_1] : \sigma_1 \\
(c) & \langle \Gamma, x:\sigma_1 \,;\, k:\neg\sigma_0 \rangle \vdash_{ans} \alpha : ans & \Rightarrow & \langle \Gamma \,;\, k:\neg\sigma_0 \rangle \vdash_{ans} \alpha[x := v_1] : ans \\
(d) & \langle \Gamma, x:\sigma_1 \,;\, k:\neg\sigma_0 \rangle \vdash_{cont} \kappa : \neg\sigma_2 & \Rightarrow & \langle \Gamma \,;\, k:\neg\sigma_0 \rangle \vdash_{cont} \kappa[x := v_1] : \neg\sigma_2
\end{array}$$

Proof: by simultaneous induction over typing derivations.

case $\Gamma, x:\sigma_1 \vdash_{val} c : \iota$: immediate, since $c[x := v_1] \equiv c$.

case $\Gamma, x:\sigma_1 \vdash_{val} x : \sigma_1$: immediate, since $x[x := v_1] \equiv v_1$ and $\Gamma \vdash_{val} v_1 : \sigma_1$.

case $\Gamma, x:\sigma_1, y:\sigma_2 \vdash_{val} v_0' : \neg\neg\sigma_3 \;\Rightarrow\; \Gamma, x:\sigma_1 \vdash_{val} \lambda y\,.\,v_0' : \sigma_2 \to \neg\neg\sigma_3$:

$$\begin{array}{lll}
\Gamma, x:\sigma_1, y:\sigma_2 \vdash_{val} v_0' : \neg\neg\sigma_3 & & \\
\quad\Rightarrow\quad \Gamma, y:\sigma_2 \vdash_{val} v_0'[x := v_1] : \neg\neg\sigma_3 & \textit{...by ind. hyp.} \\
\quad\Rightarrow\quad \Gamma \vdash_{val} \lambda y\,.\,(v_0'[x := v_1]) : \neg\neg\sigma_3 & \textit{...by (val-abs)} \\
\quad\Rightarrow\quad \Gamma \vdash_{val} (\lambda y\,.\,v_0')[x := v_1] : \neg\neg\sigma_3 & \textit{...by def. of capture-free substitution}
\end{array}$$

The rest of the cases follow from the inductive hypotheses and definition of substitution in a similar manner.

∎

Property 6.25 *Given* $\langle \Gamma \,;\, k_1:\neg\sigma_1 \rangle \vdash_{cont} \kappa_0 : \neg\sigma_0$,

$$\begin{array}{lrcl}
(a) & \langle \Gamma \,;\, k_0:\neg\sigma_0 \rangle \vdash_{ans} \alpha : ans & \Rightarrow & \langle \Gamma \,;\, k_1:\neg\sigma_1 \rangle \vdash_{ans} \alpha[k_0 := \kappa_0] : ans \\
(b) & \langle \Gamma \,;\, k_0:\neg\sigma_0 \rangle \vdash_{cont} \kappa_2 : \neg\sigma_2 & \Rightarrow & \langle \Gamma \,;\, k_1:\neg\sigma_1 \rangle \vdash_{cont} \kappa_2[k_0 := \kappa_0] : \neg\sigma_2
\end{array}$$

Proof: by simultaneous induction over typing derivations.

case $\langle \Gamma \,;\, k_0:\neg\sigma_0 \rangle \vdash_{cont} k_0 : \neg\sigma_0$: immediate, since $k_0[k_0 := \kappa_0] \equiv \kappa_0$ and $\langle \Gamma \,;\, k_1:\neg\sigma_1 \rangle \vdash_{cont} \kappa_0 : \neg\sigma_0$.

case $\langle \Gamma, x:\sigma_2 \,;\, k_0:\neg\sigma_0 \rangle \vdash_{ans} \alpha : ans \;\Rightarrow\; \langle \Gamma \,;\, k_0:\neg\sigma_0 \rangle \vdash_{cont} \lambda x\,.\,\alpha : \neg\sigma_2$:

$$\begin{array}{lll}
\langle \Gamma, x:\sigma_2 \,;\, k_0:\neg\sigma_0 \rangle \vdash_{ans} \alpha : ans & & \\
\quad\Rightarrow\quad \langle \Gamma, x:\sigma_2 \,;\, k_1:\neg\sigma_1 \rangle \vdash_{cont} \alpha[k_0 := \kappa_0] : ans & \textit{...by ind. hyp.} \\
\quad\Rightarrow\quad \langle \Gamma \,;\, k_1:\neg\sigma_1 \rangle \vdash_{cont} \lambda x\,.\,(\alpha[k_0 := \kappa_0]) : \neg\sigma_2 & \textit{...by (cont-abs)} \\
\quad\Rightarrow\quad \langle \Gamma \,;\, k_1:\neg\sigma_1 \rangle \vdash_{cont} (\lambda x\,.\,\alpha)[k_0 := \kappa_0] : \neg\sigma_2 & \textit{...by def. of capture-free substitution}
\end{array}$$

case $\langle \Gamma \,;\, k_0:\neg\sigma_0 \rangle \vdash_{cont} \kappa_2 : \neg\sigma_2, \Gamma \vdash_{val} v : \sigma_2 \;\Rightarrow\; \langle \Gamma \,;\, k_0:\neg\sigma_0 \rangle \vdash_{cont} \kappa_2\, v : ans$:

$$\begin{array}{lll}
\langle \Gamma \,;\, k_0:\neg\sigma_0 \rangle \vdash_{cont} \kappa_2 : \neg\sigma_2 & & \\
\quad\Rightarrow\quad \langle \Gamma \,;\, k_1:\neg\sigma_1 \rangle \vdash_{cont} \kappa_2[k_0 := \kappa_0] : \neg\sigma_2 & \textit{...by ind. hyp.} \\
\quad\Rightarrow\quad \langle \Gamma \,;\, k_1:\neg\sigma_1 \rangle \vdash_{cont} (\kappa_2[k_0 := \kappa_0])\, v : ans & \textit{...by (ans-app.1)} \\
\quad\Rightarrow\quad \langle \Gamma \,;\, k_1:\neg\sigma_1 \rangle \vdash_{cont} (\kappa_2\, v)[k_0 := \kappa_0] : ans & \textit{...since } k_0 \notin FV(v)
\end{array}$$

case $\Gamma \vdash_{exp} w : \neg\neg\sigma_2, \langle \Gamma \,;\, k_0:\neg\sigma_0 \rangle \vdash_{cont} \kappa_2 : \neg\sigma_2 \;\Rightarrow\; \langle \Gamma \,;\, k_0:\neg\sigma_0 \rangle \vdash_{cont} w\,\kappa_2 : ans$:

$$\begin{array}{lll}
\langle \Gamma \,;\, k_0:\neg\sigma_0 \rangle \vdash_{cont} \kappa_2 : \neg\sigma_2 & & \\
\quad\Rightarrow\quad \langle \Gamma \,;\, k_1:\neg\sigma_1 \rangle \vdash_{cont} \kappa_2[k_0 := \kappa_0] : \neg\sigma_2 & \textit{...by ind. hyp.} \\
\quad\Rightarrow\quad \langle \Gamma \,;\, k_1:\neg\sigma_1 \rangle \vdash_{cont} w\,(\kappa_2[k_0 := \kappa_0]) : ans & \textit{...by (ans-app.2)} \\
\quad\Rightarrow\quad \langle \Gamma \,;\, k_1:\neg\sigma_1 \rangle \vdash_{cont} (w\,\kappa_2)[k_0 := \kappa_0] : ans & \textit{...since } k_0 \notin FV(w)
\end{array}$$

∎

The following property establishes that Λ_{cps} is closed under R_{cps} reductions.

Property 6.26 (Subject reduction for Λ_{cps})

$$
\begin{array}{lrcl}
(a) & \Gamma \vdash_{val} v : \sigma \quad and \quad v \longrightarrow_{R_{cps}} v' & \Rightarrow & \Gamma \vdash_{val} v' : \sigma \\
(b) & \Gamma \vdash_{exp} w : \sigma \quad and \quad w \longrightarrow_{R_{cps}} w' & \Rightarrow & \Gamma \vdash_{exp} w' : \sigma \\
(c) & \langle \Gamma \,;\, k : \neg\sigma \rangle \vdash_{ans} \alpha : ans \quad and \quad \alpha \longrightarrow_{R_{cps}} \alpha' & \Rightarrow & \langle \Gamma \,;\, k : \neg\sigma \rangle \vdash_{ans} \alpha' : ans \\
(d) & \langle \Gamma \,;\, k : \neg\sigma_0 \rangle \vdash_{cont} \kappa : \neg\sigma_1 \quad and \quad \kappa \longrightarrow_{R_{cps}} \kappa' & \Rightarrow & \langle \Gamma \,;\, k : \neg\sigma_0 \rangle \vdash_{cont} \kappa' : \neg\sigma_1
\end{array}
$$

Proof: by simultaneous induction over the structure of the typing derivations. Only the details of the prime cases (*i.e.*, where reductions occur at the root of terms) are given. The rest of the cases are trivial (*i.e.*, no reductions are possible) or follow from the induction hypotheses and compatibility.

(a): *val* syntactic category:

case *(val-comp)*: $\langle \Gamma \,;\, k : \neg\sigma \rangle \vdash_{ans} \alpha : ans \;\Rightarrow\; \Gamma \vdash_{val} \lambda k \,.\, \alpha : \neg\neg\sigma$:

case $\alpha \equiv v_a\, k$:

Now $\lambda k \,.\, v_a\, k \longrightarrow_{\eta_{val}} v_a$ since it is straightforward to show that if $\Gamma \vdash_{val} \lambda k \,.\, v_a\, k : \neg\neg\sigma$ holds then $k \notin FV(v_a)$ and $\Gamma \vdash_{val} v_a : \neg\neg\sigma$ holds as well.

Cases *(val-const)* and *(val-var)* are trivially true (no reductions are possible). The rest of the cases follow the inductive hypotheses and compatibility of $\longrightarrow$.

(b): *exp* syntactic category:

case *(exp-app)*: $\Gamma \vdash_{val} v_0 : \sigma_1 \rightarrow \neg\neg\sigma_2,\ \Gamma \vdash_{val} v_1 : \sigma_1 \;\Rightarrow\; \Gamma \vdash_{exp} v_0\, v_1 : \neg\neg\sigma_2$:

case $v_0 \equiv \lambda x \,.\, v_0'$:

Now $(\lambda x \,.\, v_0')\, v_1 \longrightarrow_{\beta_{exp}} v_0'[x := v_1]$.
It is straightforward to show that if $\Gamma \vdash_{val} \lambda x \,.\, v_0' : \sigma_1 \rightarrow \neg\neg\sigma_2$ holds then $\Gamma, x : \sigma_1 \vdash_{val} v_0' : \neg\neg\sigma_2$ holds as well. By Property 6.24 we have $\Gamma \vdash_{val} v_0'[x := v_1] : \neg\neg\sigma_2$ and applying typing rule *(exp-val)* we have $\Gamma \vdash_{exp} v_0'[x := v_1] : \neg\neg\sigma_2$.

case $v_0 \equiv rec\, f(x).\, v_0'$:

Now $(rec\, f(x).\, v_0')\, v_1 \longrightarrow_{\mathrm{rec}\, exp} v_0'[f := rec\, f(x).\, v_0',\ x := v_1]$. It is straightforward to show that if $\Gamma \vdash_{val} rec\, f(x).\, v_0' : \sigma_1 \rightarrow \neg\neg\sigma_2$ holds then $\Gamma, x : \sigma_1 \vdash_{val} v_0' : \neg\neg\sigma_2$ holds as well. By Property 6.24 (twice) we have $\Gamma \vdash_{val} v_0'[f := rec\, f(x).\, v_0',\ x := v_1] : \neg\neg\sigma_2$ and applying typing rule *(exp-val)* we have $\Gamma \vdash_{exp} v_0'[f := rec\, f(x).\, v_0',\ x := v_1] : \neg\neg\sigma_2$.

(c): *cont* syntactic category:

case *(cont-abs)*: $\langle \Gamma, x : \sigma_1 \,;\, k : \neg\sigma_0 \rangle \vdash_{ans} \alpha : ans \;\Rightarrow\; \langle \Gamma \,;\, k : \neg\sigma_0 \rangle \vdash_{cont} \lambda x \,.\, \alpha : \neg\sigma_1$:

case $\alpha \equiv \kappa\, x$ where $x \notin FV(\kappa)$:

Now $\lambda x \,.\, \kappa\, x \longrightarrow_{\eta_{cont}} \kappa$. It is straightforward to show that if $\langle \Gamma \,;\, k : \neg\sigma_0 \rangle \vdash_{cont} \lambda x \,.\, \kappa\, x : \neg\sigma_1$ holds then $\langle \Gamma \,;\, k : \neg\sigma_0 \rangle \vdash_{cont} \kappa : \neg\sigma_1$ holds as well.

(d): *ans* syntactic category:

case *(ans-app.1)*: $\langle \Gamma \,;\, k : \neg\sigma_0 \rangle \vdash_{cont} \kappa : \neg\sigma_1, \Gamma \vdash_{val} v : \sigma_1 \;\Rightarrow\; \langle \Gamma \,;\, k : \neg\sigma_0 \rangle \vdash_{ans} \kappa\, v : ans$:

case $\kappa \equiv \lambda x \,.\, \alpha$:

Now $(\lambda x \,.\, \alpha)\, v \longrightarrow_{\beta_{ans.1}} \alpha[x := v]$. It is straightforward to show that if $\langle \Gamma \,;\, k : \neg\sigma_0 \rangle \vdash_{cont} \lambda x \,.\, \alpha : \neg\sigma_1$ holds then $\langle \Gamma, x : \sigma_1 \,;\, k : \neg\sigma_0 \rangle \vdash_{ans} \alpha : ans$ holds as well. By Property 6.24 we have $\langle \Gamma \,;\, k : \neg\sigma_0 \rangle \vdash_{ans} \alpha[x := v] : ans$.

case *(ans-app.2)*: $\Gamma \vdash_{exp} w : \neg\neg\sigma_0, \langle \Gamma \,;\, k_1 : \neg\sigma_1 \rangle \vdash_{cont} \kappa : \neg\sigma_0$ *implies* $\langle \Gamma \,;\, k_1 : \neg\sigma_1 \rangle \vdash_{ans} w\, \kappa : ans$:

case $w \equiv \lambda k_0 \,.\, \alpha$:

Now $(\lambda k_0 \,.\, \alpha)\, \kappa \longrightarrow_{\beta_{ans.2}} \alpha[k_0 := \kappa]$. It is straightforward to show that if $\Gamma \vdash_{exp} \lambda k_0 \,.\, \alpha : \neg\neg\sigma_0$ holds then $\langle \Gamma \,;\, k_0 : \neg\sigma_0 \rangle \vdash_{ans} \alpha : ans$ holds as well. By Property 6.25 we have $\langle \Gamma \,;\, k_1 : \neg\sigma_1 \rangle \vdash_{ans} \alpha[k_0 := \kappa] : ans$.

Lemma 6.1 $K \subseteq \Lambda_{cps}$

Proof: Let $s \in K$. From the definition of K, there exists an $e \in \Lambda_{ml}$ such that $\mathcal{K}\langle\!|e|\!\rangle \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} s$ in n steps. Now show $s \in \Lambda_{cps}$ by induction on n.

case $n = 0$: then $s \equiv \mathcal{K}\langle\!|e|\!\rangle \in \Lambda_{cps}$ by Property 6.22.

case $n = i + 1$: then $\mathcal{K}\langle\!|e|\!\rangle \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} r \longrightarrow_{R_{cps}} s$.

By the induction hypothesis, $r \in \Lambda_{cps}$ and therefore $s \in \Lambda_{cps}$ by Property 6.26.

Lemma 6.2 $\Lambda_{cps} \subseteq K$.

Proof: Based on the definition of K it is sufficient to show the following

$$
\begin{array}{lrcl}
(a) & \Gamma \vdash_{val} v : \sigma & \Rightarrow & \exists e \in \Lambda_{ml} \,.\, \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} e : \mathcal{K}^{-1}\langle\!|\sigma|\!\rangle \quad \text{and} \quad \mathcal{K}\langle\!|e|\!\rangle \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} v \\
(b) & \Gamma \vdash_{exp} w : \neg\neg\sigma & \Rightarrow & \exists e \in \Lambda_{ml} \,.\, \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} e : \mathcal{K}^{-1}\langle\!|\neg\neg\sigma|\!\rangle \quad \text{and} \quad \mathcal{K}\langle\!|e|\!\rangle \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} \lambda k \,.\, w\, k \\
(c) & \langle \Gamma \,;\, k : \neg\sigma \rangle \vdash_{ans} \alpha : ans & \Rightarrow & \exists e \in \Lambda_{ml} \,.\, \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} e : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma|\!\rangle} \quad \text{and} \quad \mathcal{K}\langle\!|e|\!\rangle \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} \lambda k \,.\, \alpha
\end{array}
$$

The proof proceeds by induction over the structure of derivations.

case Λ_{cps}.*(val-const)*: $\Gamma \vdash_{val} c : \iota$:

Take $e \equiv c$ since $\mathcal{K}\langle\!|c|\!\rangle = c$, $\mathcal{K}^{-1}\langle\!|\iota|\!\rangle = \iota$ and $\mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} c : \iota$ by Λ_{ml}.*(const)*.

case Λ_{cps}.*(val-var)*: $\Gamma \vdash_{val} x : \Gamma(x)$:

Take $e \equiv x$ since $\mathcal{K}\langle\!|x|\!\rangle = x$, $\mathcal{K}^{-1}\langle\!|\Gamma(x)|\!\rangle = \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle(x)$ and $\mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} x : \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle(x)$ by Λ_{ml}.*(var)*.

case Λ_{cps}.*(val-abs)*: $\Gamma, x : \sigma_1 \vdash_{val} v : \neg\neg\sigma_2 \;\Rightarrow\; \Gamma \vdash_{val} \lambda x \,.\, v_a : \sigma_1 \rightarrow \neg\neg\sigma_2$:

By the inductive hypothesis $\exists e_a \in \Lambda_{ml} \;.\; \mathcal{K}^{-1}\langle\!|\Gamma, x : \sigma_1|\!\rangle \vdash_{ml} e_a : \mathcal{K}^{-1}\langle\!|\neg\neg\sigma_2|\!\rangle$ and $\mathcal{K}\langle\!|e_a|\!\rangle \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} v_a$. So take $e \equiv \lambda x \,.\, e_a$ since $\mathcal{K}\langle\!|\lambda x \,.\, e_a|\!\rangle = \lambda x \,.\, \mathcal{K}\langle\!|e_a|\!\rangle \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} \lambda x \,.\, v_a$ and note that $\mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \lambda x \,.\, e_a : \mathcal{K}^{-1}\langle\!|\sigma_1 \rightarrow \neg\neg\sigma_2|\!\rangle$ follows from the definition of $\mathcal{K}^{-1}$ and Λ_{ml}.*(abs)*.

case Λ_{cps}.*(val-rec)*: $\Gamma, f : \sigma_1 \rightarrow \neg\neg\sigma_2, x : \sigma_1 \vdash_{val} v : \neg\neg\sigma_2 \;\Rightarrow\; \Gamma \vdash_{val} rec\, f(x).\, v : \sigma_1 \rightarrow \neg\neg\sigma_2$:

similar to case Λ_{cps}.*(val-abs)* above.

case Λ_{cps}.*(val-comp)*: $\langle \Gamma \,;\, k : \neg\sigma \rangle \vdash_{ans} \alpha : ans \;\Rightarrow\; \Gamma \vdash_{val} \lambda k \,.\, \alpha : \neg\neg\sigma$:

by inductive hypothesis (c), $\exists e_a \in \Lambda_{cps} \;.\; \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} e_a : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma|\!\rangle}$ and $\mathcal{K}\langle\!|e_a|\!\rangle \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} \lambda k \,.\, \alpha$.
So take $e \equiv e_a$ and note that $\mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} e_a : \mathcal{K}^{-1}\langle\!|\neg\neg\sigma|\!\rangle$ since $\mathcal{K}^{-1}\langle\!|\neg\neg\sigma|\!\rangle = \widetilde{\mathcal{K}^{-1}\langle\!|\sigma|\!\rangle}$.

case Λ_{cps}.*(exp-app)*: $\Gamma \vdash_{val} v_0 : \sigma_1 \rightarrow \neg\neg\sigma_2,\; \Gamma \vdash_{val} v_1 : \sigma_1 \;\Rightarrow\; \Gamma \vdash_{exp} v_0\, v_1 : \neg\neg\sigma_2$:

by inductive hypothesis (a), $\exists e_0, e_1 \in \Lambda_{ml} \;.\; \mathcal{K}\langle\!|e_i|\!\rangle \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} v_i \;(i = 0, 1)$
and $\mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} e_0 : \mathcal{K}^{-1}\langle\!|\sigma_1 \rightarrow \neg\neg\sigma_2|\!\rangle$ and $\mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} e_1 : \mathcal{K}^{-1}\langle\!|\sigma_1|\!\rangle$. So take $e \equiv e_0\, e_1$ since $\mathcal{K}\langle\!|e_0\, e_1|\!\rangle = \lambda k \,.\, (\mathcal{K}\langle\!|e_0|\!\rangle\, \mathcal{K}\langle\!|e_1|\!\rangle)\, k \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} \lambda k \,.\, (v_0\, v_1)\, k$ and note $\mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} e_0\, e_1 : \mathcal{K}^{-1}\langle\!|\neg\neg\sigma_2|\!\rangle$ follows from the definition of $\mathcal{K}^{-1}$ and Λ_{ml}.*(app)*.

case Λ_{cps}.*(exp-val)*: $\Gamma \vdash_{val} v : \neg\neg\sigma \;\Rightarrow\; \Gamma \vdash_{exp} v : \neg\neg\sigma$:

by inductive hypothesis (a), $\exists e_a \in \Lambda_{ml} \;.\; \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} e_a : \mathcal{K}^{-1}\langle\!|\sigma|\!\rangle$ and $\mathcal{K}\langle\!|e_a|\!\rangle \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} v$.
So take $e \equiv let\, x \Leftarrow e_a\, in\, [x]$ since

$$\begin{array}{lll}
\mathcal{K}\langle\!\langle \mathit{let}\, x \Leftarrow e_a \;\mathit{in}\; \texttt{[}x\texttt{]} \rangle\!\rangle & = & \lambda k\,.\,\mathcal{K}\langle\!\langle e_a \rangle\!\rangle\,(\lambda x\,.\,(\lambda k\,.\,k\,x)\,k) \\
& \longrightarrow_{\beta_{ans.2}} & \lambda k\,.\,\mathcal{K}\langle\!\langle e_a \rangle\!\rangle\,(\lambda x\,.\,k\,x) \\
& \longrightarrow_{\eta_{cont}} & \lambda k\,.\,\mathcal{K}\langle\!\langle e_a \rangle\!\rangle\,k \\
& \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} & \lambda k\,.\,v\,k
\end{array}$$

and note $\mathcal{K}^{-1}\langle\!\langle \Gamma \rangle\!\rangle \vdash_{ml} \mathit{let}\, x \Leftarrow e_a \;\mathit{in}\; \texttt{[}x\texttt{]} : \mathcal{K}^{-1}\langle\!\langle \neg\neg\sigma \rangle\!\rangle$ follows from the definition of $\mathcal{K}^{-1}$ and $\Lambda_{ml}.\mathit{(var)}$,$\Lambda_{ml}.\mathit{(unit)}$, and $\Lambda_{ml}.\mathit{(let)}$.

case $\Lambda_{cps}.\mathit{(ans\text{-}app.1)}$: $\langle \Gamma\,;\, k : \neg\sigma_0 \rangle \vdash_{cont} \kappa : \neg\sigma_1, \Gamma \vdash_{val} v : \sigma_1 \;\Rightarrow\; \langle \Gamma\,;\, k : \neg\sigma_0 \rangle \vdash_{ans} \kappa\, v : \mathit{ans}$:

Cases of κ are unfolded below.
case $\Lambda_{cps}.\mathit{(cont\text{-}var)}$: $\langle \Gamma\,;\, k : \neg\sigma_1 \rangle \vdash_{cont} k : \neg\sigma_1$ where $\sigma_0 \equiv \sigma_1$ in the instantiation of $\Lambda_{cps}.\mathit{(cont\text{-}var)}$ above.:

By the inductive hypothesis (a), $\exists e_a, \in \Lambda_{ml}\;.\; \mathcal{K}^{-1}\langle\!\langle \Gamma \rangle\!\rangle \vdash_{ml} e_a : \mathcal{K}^{-1}\langle\!\langle \sigma_1 \rangle\!\rangle$ and $\mathcal{K}\langle\!\langle e_a \rangle\!\rangle \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} v$.
So take $e \equiv \texttt{[}e_a\texttt{]}$ since $\mathcal{K}\langle\!\langle \texttt{[}e_a\texttt{]} \rangle\!\rangle = \lambda k\,.\,k\,\mathcal{K}\langle\!\langle e_a \rangle\!\rangle \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} \lambda k\,.\,k\,v$ and note
$\mathcal{K}^{-1}\langle\!\langle \Gamma \rangle\!\rangle \vdash_{ml} \texttt{[}e_a\texttt{]} : \neg\neg\mathcal{K}^{-1}\langle\!\langle \sigma_1 \rangle\!\rangle$ follows from the definition of $\mathcal{K}^{-1}$ and $\Lambda_{ml}.\mathit{(unit)}$.

case $\Lambda_{cps}.\mathit{(cont\text{-}abs)}$: $\langle \Gamma, x : \sigma_1\,;\, k : \neg\sigma_0 \rangle \vdash_{ans} \alpha : \mathit{ans} \;\Rightarrow\; \langle \Gamma\,;\, k : \neg\sigma_0 \rangle \vdash_{cont} \lambda x\,.\,\alpha : \neg\sigma_1$:

By the inductive hypothesis (c), $\exists e_0, \in \Lambda_{ml}\;.\; \mathcal{K}^{-1}\langle\!\langle \Gamma, x : \sigma_1 \rangle\!\rangle \vdash_{ml} e_0 : \widetilde{\mathcal{K}^{-1}\langle\!\langle \sigma_0 \rangle\!\rangle}$ and
$\mathcal{K}\langle\!\langle e_0 \rangle\!\rangle \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} \lambda k\,.\,\alpha$. By the inductive hypothesis (a), $\exists e_1, \in \Lambda_{ml}\;.\; \mathcal{K}^{-1}\langle\!\langle \Gamma \rangle\!\rangle \vdash_{ml} e_1 : \mathcal{K}^{-1}\langle\!\langle \sigma_1 \rangle\!\rangle$ and
$\mathcal{K}\langle\!\langle e_1 \rangle\!\rangle \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} v$.
So take $e \equiv \mathit{let}\, x \Leftarrow \texttt{[}e_1\texttt{]}\; \mathit{in}\, e_0$ since

$$\begin{array}{lll}
\mathcal{K}\langle\!\langle \mathit{let}\, x \Leftarrow \texttt{[}e_1\texttt{]}\; \mathit{in}\, e_0 \rangle\!\rangle & = & \lambda k\,.\,(\lambda k\,.\,k\,\mathcal{K}\langle\!\langle e_1 \rangle\!\rangle)\,(\lambda x\,.\,\mathcal{K}\langle\!\langle e_0 \rangle\!\rangle\,k) \\
& \longrightarrow_{\beta_{ans.2}} & \lambda k\,.\,(\lambda x\,.\,\mathcal{K}\langle\!\langle e_0 \rangle\!\rangle\,k)\,\mathcal{K}\langle\!\langle e_1 \rangle\!\rangle \\
& \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} & \lambda k\,.\,(\lambda x\,.\,(\lambda k\,.\,\alpha)\,k)\,\mathcal{K}\langle\!\langle e_1 \rangle\!\rangle \\
& \longrightarrow_{\beta_{ans.2}} & \lambda k\,.\,(\lambda x\,.\,\alpha)\,\mathcal{K}\langle\!\langle e_1 \rangle\!\rangle \\
& \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} & \lambda k\,.\,(\lambda x\,.\,\alpha)\,v
\end{array}$$

and note $\mathcal{K}^{-1}\langle\!\langle \Gamma \rangle\!\rangle \vdash_{ml} \mathit{let}\, x \Leftarrow \texttt{[}e_1\texttt{]}\; \mathit{in}\, e_0 : \widetilde{\mathcal{K}^{-1}\langle\!\langle \sigma_0 \rangle\!\rangle}$ follows from the definition of $\mathcal{K}^{-1}$
and $\Lambda_{ml}.\mathit{(unit)}$,$\Lambda_{ml}.\mathit{(let)}$.

case $\Lambda_{cps}.\mathit{(ans\text{-}app.2)}$: $\Gamma \vdash_{exp} w : \neg\neg\sigma_0, \langle \Gamma\,;\, k : \neg\sigma_1 \rangle \vdash_{cont} \kappa : \neg\sigma_0 \;\Rightarrow\; \langle \Gamma\,;\, k : \neg\sigma_1 \rangle \vdash_{ans} w\,\kappa : \mathit{ans}$:

Cases of κ are unfolded below.
case $\Lambda_{cps}.\mathit{(cont\text{-}var)}$: $\langle \Gamma\,;\, k : \neg\sigma_1 \rangle \vdash_{cont} k : \neg\sigma_1$ where $\sigma_0 \equiv \sigma_1$ when instantiating $\Lambda_{cps}.\mathit{(ans\text{-}app.2)}$ above.:

By the inductive hypothesis (b), $\exists e_a, \in \Lambda_{ml}\;.\; \mathcal{K}^{-1}\langle\!\langle \Gamma \rangle\!\rangle \vdash_{ml} e_a : \mathcal{K}^{-1}\langle\!\langle \neg\neg\sigma_1 \rangle\!\rangle$ and $\mathcal{K}\langle\!\langle e_a \rangle\!\rangle \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}}$ $\lambda k\,.\,w\,k$. So take $e \equiv e_a$.

case $\Lambda_{cps}.\mathit{(cont\text{-}abs)}$: $\langle \Gamma, x : \sigma_1\,;\, k : \neg\sigma_0 \rangle \vdash_{ans} \alpha : \mathit{ans} \;\Rightarrow\; \langle \Gamma\,;\, k : \neg\sigma_0 \rangle \vdash_{cont} \lambda x\,.\,\alpha : \neg\sigma_1$:

By the inductive hypothesis (b), $\exists e_0, \in \Lambda_{ml}\;.\; \mathcal{K}^{-1}\langle\!\langle \Gamma \rangle\!\rangle \vdash_{ml} e_0 : \mathcal{K}^{-1}\langle\!\langle \neg\neg\sigma_0 \rangle\!\rangle$ and $\mathcal{K}\langle\!\langle e_0 \rangle\!\rangle \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}}$
$\lambda k\,.\,w\,k$. By the inductive hypothesis (c), $\exists e_1, \in \Lambda_{ml}\;.\; \mathcal{K}^{-1}\langle\!\langle \Gamma, x : \sigma_0 \rangle\!\rangle \vdash_{ml} e_1 : \widetilde{\mathcal{K}^{-1}\langle\!\langle \sigma_1 \rangle\!\rangle}and\mathcal{K}\langle\!\langle e_1 \rangle\!\rangle \longrightarrow$
$\longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} \lambda k\,.\,\alpha$. So take $e \equiv \mathit{let}\, x \Leftarrow e_0\; \mathit{in}\, e_1$ since

$$\begin{array}{lll}
\mathcal{K}\langle\!\langle \mathit{let}\, x \Leftarrow e_0\; \mathit{in}\, e_1 \rangle\!\rangle & = & \lambda k\,.\,\mathcal{K}\langle\!\langle e_0 \rangle\!\rangle\,(\lambda x\,.\,\mathcal{K}\langle\!\langle e_1 \rangle\!\rangle\,k) \\
& \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} & \lambda k\,.\,(\lambda k\,.\,w\,k)\,(\lambda x\,.\,\mathcal{K}\langle\!\langle e_1 \rangle\!\rangle\,k) \\
& \longrightarrow_{\beta_{ans.2}} & \lambda k\,.\,w\,(\lambda x\,.\,\mathcal{K}\langle\!\langle e_1 \rangle\!\rangle\,k) \\
& \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} & \lambda k\,.\,w\,(\lambda x\,.\,(\lambda k\,.\,\alpha)\,k) \\
& \longrightarrow_{\beta_{ans.2}} & \lambda k\,.\,w\,(\lambda x\,.\,\alpha)
\end{array}$$

and note $\mathcal{K}^{-1}\langle\!\langle \Gamma \rangle\!\rangle \vdash_{ml} \mathit{let}\, x \Leftarrow e_0\; \mathit{in}\, e_1 : \widetilde{\mathcal{K}^{-1}\langle\!\langle \sigma_1 \rangle\!\rangle}$ follows from the definition of $\mathcal{K}^{-1}$ and $\Lambda_{ml}.\mathit{(let)}$.

∎

6.10.5 Correctness of $\mathcal{K}$ and $\mathcal{K}^{-1}$

This section establishes the correctness of $\mathcal{K}$ and $\mathcal{K}^{-1}$. We begin by considering typing properties. Following this, we show that $\mathcal{K}$ and $\mathcal{K}^{-1}$ establish a equational correspondence between Λ_{ml} terms under the R_{ml}-calculus and Λ_{cps} terms under the R_{cps}-calculus (*i.e.*, the $\beta_a \mathrm{rec}_a \eta_v$-calculi). The equational correspondence result is used to prove adequacy results for $\mathcal{K}$ and $\mathcal{K}^{-1}$.

Preservation of well-typedness

Property 6.22 showed that $\mathcal{K}$ preserved well-typedness of terms. The following property shows that $\mathcal{K}^{-1}$ preserves well-typedness of terms.

Property 6.27

$$
\begin{array}{lrcl}
(a) & \Gamma \vdash_{val} v : \sigma & \Rightarrow & \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}_{val}\langle\!|v|\!\rangle : \mathcal{K}^{-1}\langle\!|\sigma|\!\rangle \\
(b) & \Gamma \vdash_{exp} w : \sigma & \Rightarrow & \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}_{exp}\langle\!|w|\!\rangle : \mathcal{K}^{-1}\langle\!|\sigma|\!\rangle \\
(c) & \langle \Gamma \,;\, k : \neg\sigma \rangle \vdash_{ans} \alpha : ans & \Rightarrow & \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}_{ans}\langle\!|\alpha|\!\rangle : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma|\!\rangle} \\
(d) & \langle \Gamma \,;\, k : \neg\sigma_0 \rangle \vdash_{cont} \kappa : \neg\sigma_1 & \Rightarrow & \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}_{cont}\langle\!|\kappa|\!\rangle : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma_0|\!\rangle}[\widetilde{\mathcal{K}^{-1}\langle\!|\sigma_1|\!\rangle}]
\end{array}
$$

Proof: by simultaneous induction over the structure of the typing derivations.

case Λ_{cps}.*(val-const)*: $\Gamma \vdash_{val} c : \iota$:

$$\mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} c : \iota \quad \Rightarrow \quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|c|\!\rangle : \mathcal{K}^{-1}\langle\!|\iota|\!\rangle$$

case Λ_{cps}.*(val-var)*: $\Gamma \vdash_{val} x : \Gamma(x)$:

$$
\begin{array}{l}
\mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} x : \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle(x) \\
\quad \Rightarrow \quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} x : \mathcal{K}^{-1}\langle\!|\Gamma(x)|\!\rangle \qquad \textit{...by def. of } \mathcal{K}^{-1} \textit{ on type assumptions} \\
\quad \Rightarrow \quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|x|\!\rangle : \mathcal{K}^{-1}\langle\!|\Gamma(x)|\!\rangle
\end{array}
$$

case Λ_{cps}.*(val-abs)*: $\Gamma, x : \sigma_1 \vdash_{val} v : \neg\neg\sigma_2 \;\Rightarrow\; \Gamma \vdash_{val} \lambda x \,.\, v : \sigma_1 \rightarrow \neg\neg\sigma_2$:

$$
\begin{array}{l}
\Gamma, x : \sigma_1 \vdash_{val} v : \neg\neg\sigma_2 \\
\quad \Rightarrow \quad \mathcal{K}^{-1}\langle\!|\Gamma, x : \sigma_1|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|v|\!\rangle : \mathcal{K}^{-1}\langle\!|\neg\neg\sigma_2|\!\rangle \qquad \textit{...by ind. hyp.} \\
\quad \Rightarrow \quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle, x : \mathcal{K}^{-1}\langle\!|\sigma_1|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|v|\!\rangle : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma_2|\!\rangle} \qquad \textit{...by def. of } \mathcal{K}^{-1} \textit{ on types and assumptions} \\
\quad \Rightarrow \quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \lambda x \,.\, \mathcal{K}^{-1}\langle\!|v|\!\rangle : \mathcal{K}^{-1}\langle\!|\sigma_1|\!\rangle \rightarrow \widetilde{\mathcal{K}^{-1}\langle\!|\sigma_2|\!\rangle} \\
\quad \Rightarrow \quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|\lambda x \,.\, v|\!\rangle : \mathcal{K}^{-1}\langle\!|\sigma_1 \rightarrow \neg\neg\sigma_2|\!\rangle \qquad \textit{...by def. of } \mathcal{K}^{-1} \textit{ on types and terms}
\end{array}
$$

case Λ_{cps}.*(val-rec)*: $\Gamma, f : \sigma_1 \rightarrow \neg\neg\sigma_2, x : \sigma_1 \vdash_{val} v : \neg\neg\sigma_2 \;\Rightarrow\; \Gamma \vdash_{val} rec\, f(x).\, v : \sigma_1 \rightarrow \neg\neg\sigma_2$:

$$
\begin{array}{l}
\Gamma, f : \sigma_1 \rightarrow \neg\neg\sigma_2, x : \sigma_1 \vdash_{val} v : \neg\neg\sigma_2 \\
\quad \Rightarrow \quad \mathcal{K}^{-1}\langle\!|\Gamma, f : \sigma_1 \rightarrow \neg\neg\sigma_2, x : \sigma_1|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|v|\!\rangle : \mathcal{K}^{-1}\langle\!|\neg\neg\sigma_2|\!\rangle \qquad \textit{...by ind. hyp.} \\
\quad \Rightarrow \quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle, f : \mathcal{K}^{-1}\langle\!|\sigma_1 \rightarrow \neg\neg\sigma_2|\!\rangle, x : \mathcal{K}^{-1}\langle\!|\sigma_1|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|e|\!\rangle : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma_2|\!\rangle} \\
\qquad\qquad \textit{...by def. of } \mathcal{K}^{-1} \textit{ on types and type assumptions} \\
\quad \Rightarrow \quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} rec\, f(x).\, \mathcal{K}^{-1}\langle\!|v|\!\rangle : \mathcal{K}^{-1}\langle\!|\sigma_1|\!\rangle \rightarrow \widetilde{\mathcal{K}^{-1}\langle\!|\sigma_2|\!\rangle} \\
\quad \Rightarrow \quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|rec\, f(x).\, v|\!\rangle : \mathcal{K}^{-1}\langle\!|\sigma_1 \rightarrow \neg\neg\sigma_2|\!\rangle
\end{array}
$$

case Λ_{cps}.*(val-comp)*: $\langle \Gamma \,;\, k : \neg\sigma \rangle \vdash_{ans} \alpha : ans \;\Rightarrow\; \Gamma \vdash_{val} \lambda k \,.\, \alpha : \neg\neg\sigma$:

$$
\begin{array}{l}
\langle \Gamma \,;\, k : \neg\sigma \rangle \vdash_{ans} \alpha : ans \\
\quad \Rightarrow \quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|\alpha|\!\rangle : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma|\!\rangle} \qquad \textit{...by ind. hyp.} \\
\quad \Rightarrow \quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|\lambda k \,.\, \alpha|\!\rangle : \mathcal{K}^{-1}\langle\!|\neg\neg\sigma|\!\rangle \qquad \textit{...by def. of } \mathcal{K}^{-1} \textit{ on terms and types}
\end{array}
$$

case Λ_{cps}.*(exp-val)*: $\Gamma \vdash_{val} v : \sigma \;\Rightarrow\; \Gamma \vdash_{exp} v : \sigma$: follows immediately from ind. hyp.

case Λ_{cps}.*(exp-app)*: $\Gamma \vdash_{val} v_0 : \sigma_1 \rightarrow \neg\neg\sigma_2,\ \Gamma \vdash_{val} v_1 : \sigma_1 \;\Rightarrow\; \Gamma \vdash_{exp} v_0\, v_1 : \neg\neg\sigma_2$:

$\Gamma \vdash_{val} v_0 : \sigma_1 \rightarrow \neg\neg\sigma_2,\ \Gamma \vdash_{val} v_1 : \sigma_1$
$\Rightarrow\quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|v_0|\!\rangle : \mathcal{K}^{-1}\langle\!|\sigma_1 \rightarrow \neg\neg\sigma_2|\!\rangle,\ \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|v_1|\!\rangle : \mathcal{K}^{-1}\langle\!|\sigma_1|\!\rangle$ *...by ind. hyp.*
$\Rightarrow\quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|v_0|\!\rangle : \mathcal{K}^{-1}\langle\!|\sigma_1|\!\rangle \rightarrow \widetilde{\mathcal{K}^{-1}\langle\!|\sigma_2|\!\rangle},\ \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|v_1|\!\rangle : \mathcal{K}^{-1}\langle\!|\sigma_1|\!\rangle$
$\Rightarrow\quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|v_0|\!\rangle\ \mathcal{K}^{-1}\langle\!|v_1|\!\rangle : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma_2|\!\rangle}$
$\Rightarrow\quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|v_0\ v_1|\!\rangle : \mathcal{K}^{-1}\langle\!|\neg\neg\sigma_2|\!\rangle$

case Λ_{cps}.*(cont-var)*: $\langle \Gamma\ ;\ k : \neg\sigma \rangle \vdash_{cont} k : \neg\sigma$:

$$\mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} [\cdot] : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma|\!\rangle}[\widetilde{\mathcal{K}^{-1}\langle\!|\sigma|\!\rangle}] \quad\Rightarrow\quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|k|\!\rangle : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma|\!\rangle}[\widetilde{\mathcal{K}^{-1}\langle\!|\sigma|\!\rangle}]$$

case Λ_{cps}.*(cont-abs)*: $\langle \Gamma, x:\sigma_1\ ;\ k : \neg\sigma_0 \rangle \vdash_{ans} \alpha : ans \ \Rightarrow\ \langle \Gamma\ ;\ k : \neg\sigma_0 \rangle \vdash_{cont} \lambda x\,.\,\alpha : \neg\sigma_1$:

$\langle \Gamma, x:\sigma_1\ ;\ k : \neg\sigma_0 \rangle \vdash_{ans} \alpha : ans$
$\Rightarrow\quad \mathcal{K}^{-1}\langle\!|\Gamma, x:\sigma_1|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|\alpha|\!\rangle : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma_0|\!\rangle}$ *...by ind. hyp.*
$\Rightarrow\quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle, x:\mathcal{K}^{-1}\langle\!|\sigma_1|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|\alpha|\!\rangle : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma_0|\!\rangle}$ *...def. of $\mathcal{K}^{-1}$ on type assumptions*
$\Rightarrow\quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathit{let}\ x \Leftarrow [\cdot]\ \mathit{in}\ \mathcal{K}^{-1}\langle\!|\alpha|\!\rangle : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma_0|\!\rangle}[\widetilde{\mathcal{K}^{-1}\langle\!|\sigma_1|\!\rangle}]$ *...introduction rule for contexts*
$\Rightarrow\quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|\lambda x\,.\,\alpha|\!\rangle : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma_0|\!\rangle}[\widetilde{\mathcal{K}^{-1}\langle\!|\sigma_1|\!\rangle}]$

case Λ_{cps}.*(ans-app.1)*: $\langle \Gamma\ ;\ k : \neg\sigma_0 \rangle \vdash_{cont} \kappa : \neg\sigma_1, \Gamma \vdash_{val} v : \sigma_1 \ \Rightarrow\ \langle \Gamma\ ;\ k : \neg\sigma_0 \rangle \vdash_{ans} \kappa\, v : ans$:

$\langle \Gamma\ ;\ k : \neg\sigma_0 \rangle \vdash_{cont} \kappa : \neg\sigma_1, \Gamma \vdash_{val} v : \sigma_1$
$\Rightarrow\quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|\kappa|\!\rangle : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma_0|\!\rangle}[\widetilde{\mathcal{K}^{-1}\langle\!|\sigma_1|\!\rangle}], \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|v|\!\rangle : \mathcal{K}^{-1}\langle\!|\sigma_1|\!\rangle$ *...by ind. hyp.*
$\Rightarrow\quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|\kappa|\!\rangle : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma_0|\!\rangle}[\widetilde{\mathcal{K}^{-1}\langle\!|\sigma_1|\!\rangle}], \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} [\mathcal{K}^{-1}\langle\!|v|\!\rangle] : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma_1|\!\rangle}$
$\Rightarrow\quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|\kappa|\!\rangle[[\mathcal{K}^{-1}\langle\!|v|\!\rangle]] : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma_0|\!\rangle}$
$\Rightarrow\quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|\kappa\, v|\!\rangle : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma_1|\!\rangle}$

case Λ_{cps}.*(ans-app.2)*: $\Gamma \vdash_{exp} w : \neg\neg\sigma_0, \langle \Gamma\ ;\ k : \neg\sigma_1 \rangle \vdash_{cont} \kappa : \neg\sigma_0$ *implies* $\langle \Gamma\ ;\ k : \neg\sigma_1 \rangle \vdash_{ans} w\,\kappa : ans$:

$\Gamma \vdash_{exp} w : \neg\neg\sigma_0, \langle \Gamma\ ;\ k : \neg\sigma_1 \rangle \vdash_{cont} \kappa : \neg\sigma_0$
$\Rightarrow\quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|w|\!\rangle : \mathcal{K}^{-1}\langle\!|\neg\neg\sigma_0|\!\rangle, \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|\kappa|\!\rangle : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma_1|\!\rangle}[\widetilde{\mathcal{K}^{-1}\langle\!|\sigma_0|\!\rangle}]$ *...by ind. hyp.*
$\Rightarrow\quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|w|\!\rangle : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma_0|\!\rangle}, \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|\kappa|\!\rangle : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma_1|\!\rangle}[\widetilde{\mathcal{K}^{-1}\langle\!|\sigma_0|\!\rangle}]$
$\Rightarrow\quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|\kappa|\!\rangle[\mathcal{K}^{-1}\langle\!|w|\!\rangle] : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma_1|\!\rangle}$
$\Rightarrow\quad \mathcal{K}^{-1}\langle\!|\Gamma|\!\rangle \vdash_{ml} \mathcal{K}^{-1}\langle\!|w\,\kappa|\!\rangle : \widetilde{\mathcal{K}^{-1}\langle\!|\sigma_1|\!\rangle}$

∎

Equational Correspondence

In this section we prove the equational correspondence between theories associated with Λ_{ml} and Λ_{cps}. The outline of the proof follows the outline of the equational correspondence dealing with thunks in Section 3.8.5. We will only give the necessary properties here. The reader is referred to Chapter 3 for the overall strategy.

The following property shows how $\mathcal{K}$ interacts with substitution.

Property 6.28 *For all $e \in \Lambda_{ml}$,*

$$\mathcal{K}\langle\!|e[x := e']|\!\rangle \quad = \quad \mathcal{K}\langle\!|e|\!\rangle[x := \mathcal{K}\langle\!|e'|\!\rangle]$$

Proof: by a simple induction over the structure of e. ∎

We now consider how $\mathcal{K}^{-1}$ interacts with substitution. There are two kinds of relevant substitutions: the substitution of terms from the Λ_{cps} syntactic category of *values*, and the substitution of terms from the Λ_{cps} syntactic category of *continuations*. The following property treats the first kind.

Property 6.29

$$\begin{array}{rcl}
\Gamma, x:\sigma' \vdash_{val} v : \sigma \ and \ \Gamma \vdash_{val} v' : \sigma' & \Rightarrow & \mathcal{K}^{-1}_{val}(\!|v[x := v']|\!) = \mathcal{K}^{-1}_{val}(\!|v|\!)[x := \mathcal{K}^{-1}_{val}(\!|v'|\!)] \\
\Gamma, x:\sigma' \vdash_{exp} w : \sigma \ and \ \Gamma \vdash_{val} v' : \sigma' & \Rightarrow & \mathcal{K}^{-1}_{exp}(\!|w[x := v']|\!) = \mathcal{K}^{-1}_{exp}(\!|w|\!)[x := \mathcal{K}^{-1}_{val}(\!|v'|\!)] \\
\langle \Gamma, x:\sigma'; k:\neg\sigma'' \rangle \vdash_{ans} \alpha : ans \ and \ \Gamma \vdash_{val} v' : \sigma' & \Rightarrow & \mathcal{K}^{-1}_{ans}(\!|\alpha[x := v']|\!) = \mathcal{K}^{-1}_{ans}(\!|\alpha|\!)[x := \mathcal{K}^{-1}_{val}(\!|v'|\!)]
\end{array}$$

Proof: by simultaneous induction over the structure of v, w, α.

case $v \equiv c$:

$$\begin{array}{rcll}
\mathcal{K}^{-1}(\!|c[x := v']|\!) & = & \mathcal{K}^{-1}(\!|c|\!) & \\
& = & c & \\
& = & c[x := \mathcal{K}^{-1}(\!|v'|\!)] & \\
& = & \mathcal{K}^{-1}(\!|c|\!)[x := \mathcal{K}^{-1}(\!|v'|\!)] &
\end{array}$$

case $v \equiv x$:

$$\begin{array}{rcll}
\mathcal{K}^{-1}(\!|x[x := v']|\!) & = & \mathcal{K}^{-1}(\!|v'|\!) & \\
& = & x[x := \mathcal{K}^{-1}(\!|v'|\!)] & \\
& = & \mathcal{K}^{-1}(\!|x|\!)[x := \mathcal{K}^{-1}(\!|v'|\!)] &
\end{array}$$

case $v \equiv y$: similar to case $v \equiv c$.

case $v \equiv \lambda y \,.\, v_0$:

$$\begin{array}{rcll}
\mathcal{K}^{-1}(\!|(\lambda y \,.\, v_0)[x := v']|\!) & = & \mathcal{K}^{-1}(\!|\lambda y \,.\, (v_0[x := v'])|\!) & \\
& = & \lambda y \,.\, \mathcal{K}^{-1}(\!|v_0[x := v']|\!) & \\
& = & \lambda y \,.\, (\mathcal{K}^{-1}(\!|v_0|\!)[x := \mathcal{K}^{-1}(\!|v'|\!)]) & \ldots by\ ind.\ hyp. \\
& = & (\lambda y \,.\, \mathcal{K}^{-1}(\!|v_0|\!))[x := \mathcal{K}^{-1}(\!|v'|\!)] & \\
& = & \mathcal{K}^{-1}(\!|\lambda y \,.\, v_0|\!)[x := \mathcal{K}^{-1}(\!|v'|\!)] &
\end{array}$$

case $v \equiv rec\, f(y).\, v_0$: similar to case above.

case $v \equiv \lambda k \,.\, \alpha$:

$$\begin{array}{rcll}
\mathcal{K}^{-1}(\!|(\lambda k \,.\, \alpha)[x := v']|\!) & = & \mathcal{K}^{-1}(\!|\lambda k \,.\, (\alpha[x := v'])|\!) & \\
& = & \mathcal{K}^{-1}(\!|\alpha[x := v']|\!) & \\
& = & \mathcal{K}^{-1}(\!|\alpha|\!)[x := \mathcal{K}^{-1}(\!|v'|\!)] & \ldots by\ ind.\ hyp. \\
& = & \mathcal{K}^{-1}(\!|\lambda k \,.\, \alpha|\!)[x := \mathcal{K}^{-1}(\!|v'|\!)] &
\end{array}$$

case $w \equiv v$: immediate, by ind. hyp. for v.

case $w \equiv v_0\, v_1$:

$$\begin{array}{rcll}
\mathcal{K}^{-1}(\!|(v_0\, v_1)[x := v']|\!) & = & \mathcal{K}^{-1}(\!|(v_0[x := v'])\,(v_1[x := v'])|\!) & \\
& = & \mathcal{K}^{-1}(\!|v_0[x := v']|\!)\, \mathcal{K}^{-1}(\!|v_1[x := v']|\!) & \\
& = & (\mathcal{K}(\!|v_0|\!)[x := \mathcal{K}^{-1}(\!|v'|\!)])\,(\mathcal{K}(\!|v_1|\!)[x := \mathcal{K}^{-1}(\!|v'|\!)]) & \ldots by\ ind.\ hyp. \\
& = & (\mathcal{K}^{-1}(\!|v_0|\!)\, \mathcal{K}^{-1}(\!|v_1|\!))[x := v'] & \\
& = & \mathcal{K}^{-1}(\!|v_0\, v_1|\!)[x := v'] &
\end{array}$$

case $\alpha \equiv w\, k$:

$$\begin{array}{rcll}
\mathcal{K}^{-1}(\!|(w\, k)[x := v']|\!) & = & \mathcal{K}^{-1}(\!|(w[x := v'])\, k|\!) & \\
& = & \mathcal{K}^{-1}(\!|w[x := v']|\!) & \\
& = & \mathcal{K}^{-1}(\!|w|\!)[x := \mathcal{K}^{-1}(\!|v'|\!)] & \ldots by\ ind.\ hyp. \\
& = & \mathcal{K}^{-1}(\!|w\, k|\!)[x := \mathcal{K}^{-1}(\!|v'|\!)] &
\end{array}$$

case $\alpha \equiv k\, v$: similar to case above.

case $\alpha \equiv w\, (\lambda y \,.\, \alpha_0)$:

$$\begin{array}{rcl}
\mathcal{K}^{-1}\langle\!|[(w\,(\lambda y\,.\,\alpha_0))[x\;:=\;v']]\rangle\!| & = & \mathcal{K}^{-1}\langle\!|[(w[x\;:=\;v'])\,(\lambda y\,.\,\alpha_0[x\;:=\;v'])]\rangle\!| \\
& = & let\; y \Leftarrow \mathcal{K}^{-1}\langle\!|[w[x\;:=\;v']]\rangle\!|\; in\; \mathcal{K}^{-1}\langle\!|[\alpha_0[x\;:=\;v']]\rangle\!| \\
& = & let\; y \Leftarrow (\mathcal{K}^{-1}\langle\!|[w]\rangle\!|[x\;:=\;\mathcal{K}^{-1}\langle\!|[v']\rangle\!|])\; in\; (\mathcal{K}^{-1}\langle\!|[\alpha_0]\rangle\!|[x\;:=\;\mathcal{K}^{-1}\langle\!|[v']\rangle\!|]) \\
& & \qquad ...by\; ind.\; hyp. \\
& = & (let\; y \Leftarrow \mathcal{K}^{-1}\langle\!|[w]\rangle\!|\; in\; \mathcal{K}^{-1}\langle\!|[\alpha_0]\rangle\!|)[x\;:=\;\mathcal{K}^{-1}\langle\!|[v']\rangle\!|] \\
& = & \mathcal{K}^{-1}\langle\!|[w\,(\lambda y\,.\,\alpha_0)]\rangle\!|[x\;:=\;\mathcal{K}^{-1}\langle\!|[v']\rangle\!|]
\end{array}$$

case $\alpha \equiv \kappa\, v$: similar to case above.

∎

The interaction of $\mathcal{K}^{-1}$ with the substitution of continuations is more complicated since continuation identifiers k do not appear in Λ_{ml} terms. The proof requires the following observation.

Observation 6.1 *For all* $\Gamma \vdash_{ml} \varphi : \widetilde{\sigma_2}[\widetilde{\sigma_1}]$ *and* $\Gamma \vdash_{ml} let\; x \Leftarrow e_1\; in\; e_2 : \widetilde{\sigma_2}$,

$$\varphi[let\; x \Leftarrow e_1\; in\; e_2] \quad \longrightarrow\!\!\!\!\!\rightarrow_{\text{let.assoc}} \quad let\; x \Leftarrow e_1\; in\; \varphi[e_2]$$

Proof:

case $\varphi \equiv [\cdot]$: immediate

case $\varphi \equiv let\; y \Leftarrow [\cdot]\; in\; e$:

$$let\; y \Leftarrow (let\; x \Leftarrow e_1\; in\; e_2)\; in\; e \quad \longrightarrow_{\text{let.assoc}} \quad let\; x \Leftarrow e_1\; in\; let\; y \Leftarrow e_2\; in\; e$$

∎

Property 6.30 *For all* $\langle \Gamma\,;\, k : \neg\sigma_1\rangle \vdash_{ans} \alpha : ans$ *and* $\langle \Gamma\,;\, k : \neg\sigma_0\rangle \vdash_{cont} \kappa : \neg\sigma_1$,

$$\mathcal{K}^{-1}\langle\!|[\kappa]\rangle\!|[\mathcal{K}^{-1}\langle\!|[\alpha]\rangle\!|] \quad \longrightarrow\!\!\!\!\!\rightarrow_{\text{let.assoc}} \quad \mathcal{K}^{-1}\langle\!|[\alpha[k\;:=\;\kappa]]\rangle\!|$$

Proof: by induction over the structure of α.

case $\alpha \equiv w\,\kappa_0$:

case $\kappa_0 \equiv k$:

$$\begin{array}{rcl}
\mathcal{K}^{-1}\langle\!|[\kappa]\rangle\!|[\mathcal{K}^{-1}\langle\!|[w\,k]\rangle\!|] & = & \mathcal{K}^{-1}\langle\!|[\kappa]\rangle\!|[\mathcal{K}^{-1}\langle\!|[w]\rangle\!|] \\
& = & \mathcal{K}^{-1}\langle\!|[w\,\kappa]\rangle\!| \\
& = & \mathcal{K}^{-1}\langle\!|[(w\,k)[k\;:=\;\kappa]]\rangle\!|
\end{array}$$

case $\kappa_0 \equiv \lambda x\,.\,\alpha_0$:

$$\begin{array}{rcll}
\mathcal{K}^{-1}\langle\!|[\kappa]\rangle\!|[\mathcal{K}^{-1}\langle\!|[w\,(\lambda x\,.\,\alpha_0)]\rangle\!|] & = & \mathcal{K}^{-1}\langle\!|[\kappa]\rangle\!|[let\; x \Leftarrow \mathcal{K}^{-1}\langle\!|[w]\rangle\!|\; in\; \mathcal{K}^{-1}\langle\!|[\alpha_0]\rangle\!|] & \\
& \longrightarrow\!\!\!\!\!\rightarrow_{\text{let.assoc}} & let\; x \Leftarrow \mathcal{K}^{-1}\langle\!|[w]\rangle\!|\; in\; \mathcal{K}^{-1}\langle\!|[\kappa]\rangle\!|[\mathcal{K}^{-1}\langle\!|[\alpha_0]\rangle\!|] & ...by\; Obs.\; 6.1 \\
& \longrightarrow\!\!\!\!\!\rightarrow_{\text{let.assoc}} & let\; x \Leftarrow \mathcal{K}^{-1}\langle\!|[w]\rangle\!|\; in\; \mathcal{K}^{-1}\langle\!|[\alpha_0[k\;:=\;\kappa]]\rangle\!| & ...by\; ind.\; hyp. \\
& = & \lambda x\,.\,(\alpha_0[k\;:=\;\kappa])[\mathcal{K}^{-1}\langle\!|[w]\rangle\!|] & \\
& = & \mathcal{K}^{-1}\langle\!|[w\,(\lambda x\,.\,(\alpha_0[k\;:=\;\kappa]))]\rangle\!| & \\
& = & \mathcal{K}^{-1}\langle\!|[w\,(\lambda x\,.\,\alpha_0)[k\;:=\;\kappa]]\rangle\!| &
\end{array}$$

case $\alpha \equiv \kappa_0\, v$:

case $\kappa_0 \equiv k$:

$$\begin{array}{rcl}
\mathcal{K}^{-1}\langle\!|[\kappa]\rangle\!|[\mathcal{K}^{-1}\langle\!|[k\,v]\rangle\!|] & = & \mathcal{K}^{-1}\langle\!|[\kappa]\rangle\!|[[\mathcal{K}^{-1}\langle\!|[v]\rangle\!|]] \\
& = & \mathcal{K}^{-1}\langle\!|[\kappa\,v]\rangle\!| \\
& = & \mathcal{K}^{-1}\langle\!|[(k\,v)[k\;:=\;\kappa]]\rangle\!|
\end{array}$$

case $\kappa_0 \equiv \lambda x\,.\,\alpha_0$:

$$
\begin{array}{rcl}
\mathcal{K}^{-1}\langle\!|\kappa|\!\rangle[\mathcal{K}^{-1}\langle\!|(\lambda x\,.\,\alpha_0)\,v|\!\rangle] & = & \mathcal{K}^{-1}\langle\!|\kappa|\!\rangle[\mathit{let}\,x \Leftarrow \texttt{[}\mathcal{K}^{-1}\langle\!|v|\!\rangle\texttt{]}\;\mathit{in}\;\mathcal{K}^{-1}\langle\!|\alpha_0|\!\rangle] \\
& \longrightarrow\!\!\!\!\!\rightarrow_{\text{let.assoc}} & \mathit{let}\,x \Leftarrow \texttt{[}\mathcal{K}^{-1}\langle\!|v|\!\rangle\texttt{]}\;\mathit{in}\;\mathcal{K}^{-1}\langle\!|\kappa|\!\rangle[\mathcal{K}^{-1}\langle\!|\alpha_0|\!\rangle] \\
& & \quad \mathit{...by\ Observation\ 6.1} \\
& \longrightarrow\!\!\!\!\!\rightarrow_{\text{let.assoc}} & \mathit{let}\,x \Leftarrow \texttt{[}\mathcal{K}^{-1}\langle\!|v|\!\rangle\texttt{]}\;\mathit{in}\;\mathcal{K}^{-1}\langle\!|\alpha_0[k := \kappa]|\!\rangle\ \mathit{...by\ ind.\ hyp.} \\
& = & (\lambda x\,.\,\alpha_0[k := \kappa])[\texttt{[}\mathcal{K}^{-1}\langle\!|v|\!\rangle\texttt{]}] \\
& = & \mathcal{K}^{-1}\langle\!|(\lambda x\,.\,\alpha_0[k := \kappa])\,v|\!\rangle \\
& = & \mathcal{K}^{-1}\langle\!|((\lambda x\,.\,\alpha_0)\,v)[k := \kappa]|\!\rangle
\end{array}
$$

∎

The following two properties establish components (1) and (2) of the equational correspondence. The property below states that the introduction of continuations *via* $\mathcal{K}$ followed the elimination of continuations *via* $\mathcal{K}^{-1}$ gives identical terms (up to α-equivalence).

Property 6.31 *For all* $e \in \Lambda_{ml}$,

$$(\mathcal{K}^{-1} \circ \mathcal{K})\langle\!|e|\!\rangle \;\equiv\; e$$

Proof: by induction over the structure of e. A few of the cases are given.

case $e \equiv e_0\,e_1$:

$$
\begin{array}{rcll}
(\mathcal{K}^{-1} \circ \mathcal{K})\langle\!|e_0\,e_1|\!\rangle & = & \mathcal{K}^{-1}\langle\!|\lambda k\,.\,(\mathcal{K}\langle\!|e_0|\!\rangle\,\mathcal{K}\langle\!|e_1|\!\rangle)\,k|\!\rangle & \\
& = & \mathcal{K}^{-1}\langle\!|(\mathcal{K}\langle\!|e_0|\!\rangle\,\mathcal{K}\langle\!|e_1|\!\rangle)\,k|\!\rangle & \\
& = & \mathcal{K}^{-1}\langle\!|\mathcal{K}\langle\!|e_0|\!\rangle\,\mathcal{K}\langle\!|e_1|\!\rangle|\!\rangle & \\
& = & (\mathcal{K}^{-1} \circ \mathcal{K})\langle\!|e_0|\!\rangle\,(\mathcal{K}^{-1} \circ \mathcal{K})\langle\!|e_1|\!\rangle & \\
& = & e_0\,e_1 & \mathit{...by\ ind.\ hyp.}
\end{array}
$$

case $e \equiv \texttt{[}e_0\texttt{]}$:

$$
\begin{array}{rcll}
(\mathcal{K}^{-1} \circ \mathcal{K})\langle\!|\texttt{[}e_0\texttt{]}|\!\rangle & = & \mathcal{K}^{-1}\langle\!|\lambda k\,.\,k\,\mathcal{K}\langle\!|e_0|\!\rangle|\!\rangle & \\
& = & \mathcal{K}^{-1}\langle\!|k\,\mathcal{K}\langle\!|e_0|\!\rangle|\!\rangle & \\
& = & \texttt{[}(\mathcal{K}^{-1} \circ \mathcal{K})\langle\!|e_0|\!\rangle\texttt{]} & \\
& = & \texttt{[}e_0\texttt{]} & \mathit{...by\ ind.\ hyp.}
\end{array}
$$

case $e \equiv \mathit{let}\,x \Leftarrow e_0\;\mathit{in}\,e_1$:

$$
\begin{array}{rcll}
(\mathcal{K}^{-1} \circ \mathcal{K})\langle\!|\mathit{let}\,x \Leftarrow e_0\;\mathit{in}\,e_1|\!\rangle & = & \mathcal{K}^{-1}\langle\!|\lambda k\,.\,\mathcal{K}\langle\!|e_0|\!\rangle\,(\lambda x\,.\,\mathcal{K}\langle\!|e_1|\!\rangle\,k)|\!\rangle & \\
& = & \mathcal{K}^{-1}\langle\!|\mathcal{K}\langle\!|e_0|\!\rangle\,(\lambda x\,.\,\mathcal{K}\langle\!|e_1|\!\rangle\,k)|\!\rangle & \\
& = & \mathit{let}\,x \Leftarrow (\mathcal{K}^{-1} \circ \mathcal{K})\langle\!|e_0|\!\rangle\;\mathit{in}\;\mathcal{K}^{-1}\langle\!|\mathcal{K}\langle\!|e_1|\!\rangle\,k|\!\rangle & \\
& = & \mathit{let}\,x \Leftarrow (\mathcal{K}^{-1} \circ \mathcal{K})\langle\!|e_0|\!\rangle\;\mathit{in}\;(\mathcal{K}^{-1} \circ \mathcal{K})\langle\!|e_1|\!\rangle & \mathit{...by\ ind.\ hyp.}
\end{array}
$$

∎

Property 6.32 *For* $\Gamma \vdash_{val} v : \sigma$, $\Gamma \vdash_{exp} w : \sigma$, *and* $\langle \Gamma\,;\, k : \neg\sigma\rangle \vdash_{ans} \alpha : ans$,

$$
\begin{array}{lrcl}
(a) & (\mathcal{K} \circ \mathcal{K}^{-1})\langle\!|v|\!\rangle & =_{R_{cps}} & v \\
(b) & (\mathcal{K} \circ \mathcal{K}^{-1})\langle\!|w|\!\rangle & =_{R_{cps}} & \lambda k\,.\,w\,k \\
(c) & (\mathcal{K} \circ \mathcal{K}^{-1})\langle\!|\alpha|\!\rangle & =_{R_{cps}} & \lambda k\,.\,\alpha
\end{array}
$$

Proof: by induction over the structure of v, w, and α.

case $v \equiv c$:

$$
\begin{array}{rcl}
(\mathcal{K} \circ \mathcal{K}^{-1})\langle\!|c|\!\rangle & = & \mathcal{K}\langle\!|c|\!\rangle \\
& = & c
\end{array}
$$

case $v \equiv x$:

$$
\begin{array}{rcl}
(\mathcal{K} \circ \mathcal{K}^{-1})\langle\!|c|\!\rangle & = & \mathcal{K}\langle\!|x|\!\rangle \\
& = & x
\end{array}
$$

case $v \equiv \lambda x\,.\,v_0$:

$$\begin{array}{lll}(\mathcal{K}\circ\mathcal{K}^{-1})(\!|[\lambda x\,.\,v_0]\!|) & = & \mathcal{K}(\!|[\lambda x\,.\,\mathcal{K}^{-1}(\!|[v_0]\!|)]\!|)\\ & = & \lambda x\,.\,(\mathcal{K}\circ\mathcal{K}^{-1})(\!|[v_0]\!|)\\ & =_{R_{cps}} & \lambda x\,.\,v_0 \qquad \textit{...by ind. hyp. (a)}\end{array}$$

case $v \equiv rec\,f(x).\,v_0$: similar to case above.

case $v \equiv \lambda k\,.\,\alpha$:

$$\begin{array}{lll}(\mathcal{K}\circ\mathcal{K}^{-1})(\!|[\lambda k\,.\,\alpha]\!|) & = & (\mathcal{K}\circ\mathcal{K}^{-1})(\!|[\alpha]\!|)\\ & =_{R_{cps}} & \lambda k\,.\,\alpha \qquad \textit{...by ind. hyp. (c)}\end{array}$$

case $w \equiv v$:

$$\begin{array}{lll}(\mathcal{K}\circ\mathcal{K}^{-1})(\!|[v]\!|) & =_{R_{cps}} & v \qquad \textit{...by ind. hyp. (a)}\\ & \longleftarrow_{\eta_{val}} & \lambda k\,.\,v\,k\end{array}$$

case $w \equiv v_0\,v_1$:

$$\begin{array}{lll}(\mathcal{K}\circ\mathcal{K}^{-1})(\!|[v_0\,v_1]\!|) & = & \mathcal{K}(\!|[\mathcal{K}^{-1}(\!|[v_0]\!|)\;\mathcal{K}^{-1}(\!|[v_1]\!|)]\!|)\\ & = & \lambda k\,.\,((\mathcal{K}\circ\mathcal{K}^{-1})(\!|[v_0]\!|)\;(\mathcal{K}\circ\mathcal{K}^{-1})(\!|[v_1]\!|))\,k\\ & =_{R_{cps}} & \lambda k\,.\,(v_0\,v_1)\,k \qquad \textit{...by ind. hyp. (a) (twice)}\end{array}$$

case $\alpha \equiv \kappa\,v$: cases of κ are unfolded below.

case $\kappa \equiv k$:

$$\begin{array}{lll}(\mathcal{K}\circ\mathcal{K}^{-1})(\!|[k\,v]\!|) & = & \mathcal{K}(\!|[\mathtt{[}\mathcal{K}^{-1}(\!|[v]\!|)\mathtt{]}]\!|)\\ & = & \lambda k\,.\,k\,(\mathcal{K}\circ\mathcal{K}^{-1})(\!|[v]\!|)\\ & =_{R_{cps}} & \lambda k\,.\,k\,v \qquad \textit{...by ind. hyp. (a)}\end{array}$$

case $\kappa \equiv \lambda x\,.\,\alpha_0$:

$$\begin{array}{lll}(\mathcal{K}\circ\mathcal{K}^{-1})(\!|[(\lambda x\,.\,\alpha_0)\,v]\!|) & = & \mathcal{K}(\!|[let\;x \Leftarrow \mathtt{[}\mathcal{K}^{-1}(\!|[v]\!|)\mathtt{]}\;in\;\mathcal{K}^{-1}(\!|[\alpha_0]\!|)]\!|)\\ & = & \lambda k\,.\,(\lambda k\,.\,k\,(\mathcal{K}\circ\mathcal{K}^{-1})(\!|[v]\!|))\,(\lambda x\,.\,(\mathcal{K}\circ\mathcal{K}^{-1})(\!|[\alpha_0]\!|)\,k)\\ & =_{R_{cps}} & \lambda k\,.\,(\lambda k\,.\,k\,v)\,(\lambda x\,.\,(\lambda k\,.\,\alpha_0)\,k) \qquad \textit{...by ind. hyp. (a) and (c)}\\ & \longrightarrow_{\beta_{ans.2}} & \lambda k\,.\,(\lambda x\,.\,(\lambda k\,.\,\alpha_0)\,k)\,v\\ & \longrightarrow_{\beta_{ans.2}} & \lambda k\,.\,(\lambda x\,.\,\alpha_0)\,v\end{array}$$

case $\alpha \equiv w\,\kappa$: cases of κ are unfolded below.

case $\kappa \equiv k$:

$$\begin{array}{lll}(\mathcal{K}\circ\mathcal{K}^{-1})(\!|[w\,k]\!|) & = & (\mathcal{K}\circ\mathcal{K}^{-1})(\!|[w]\!|)\\ & =_{R_{cps}} & \lambda k\,.\,w\,k \qquad \textit{...by ind. hyp. (b)}\end{array}$$

case $\kappa \equiv \lambda x\,.\,\alpha_0$:

$$\begin{array}{lll}(\mathcal{K}\circ\mathcal{K}^{-1})(\!|[w\,(\lambda x\,.\,\alpha_0)]\!|) & = & \mathcal{K}(\!|[let\;x \Leftarrow w\;in\;\mathcal{K}^{-1}(\!|[\alpha_0]\!|)]\!|)\\ & = & \lambda k\,.\,(\mathcal{K}\circ\mathcal{K}^{-1})(\!|[w]\!|)\,(\lambda x\,.\,(\mathcal{K}\circ\mathcal{K}^{-1})(\!|[\alpha_0]\!|)\,k)\\ & =_{R_{cps}} & \lambda k\,.\,(\lambda k\,.\,w\,k)\,(\lambda x\,.\,(\lambda k\,.\,\alpha_0)\,k) \qquad \textit{...by ind. hyp. (b) and (c)}\\ & \longrightarrow_{\beta_{ans.2}} & \lambda k\,.\,w\,(\lambda x\,.\,(\lambda k\,.\,\alpha_0)\,k)\\ & \longrightarrow_{\beta_{ans.2}} & \lambda k\,.\,w\,(\lambda x\,.\,\alpha_0)\end{array}$$

∎

We now show that each R_{ml} reduction induces an R_{cps} equivalence.

Property 6.33 *For all* $\Gamma \vdash_{ml} e : \sigma$,

$$e \longrightarrow_{R_{ml}} e' \;\Rightarrow\; \mathcal{K}(\!|[e]\!|) =_{R_{cps}} \mathcal{K}(\!|[e']\!|)$$

Proof:

case $(\lambda x\,.\,e_0)\,e_1 \longrightarrow_{\beta_{ml}} e_0[x := e_1]$:

$$\begin{array}{lcll}
\mathcal{K}\langle\!|(\lambda x\,.\,e_0)\,e_1|\!\rangle & = & \lambda k\,.\,((\lambda x\,.\,\mathcal{K}\langle\!|e_0|\!\rangle)\,\mathcal{K}\langle\!|e_1|\!\rangle)\,k & \\
& \longrightarrow_{\beta_{exp}} & \lambda k\,.\,(\mathcal{K}\langle\!|e_0|\!\rangle[x := \mathcal{K}\langle\!|e_1|\!\rangle])\,k & \\
& = & \lambda k\,.\,\mathcal{K}\langle\!|e_0[x := e_1]|\!\rangle\,k & \textit{...by Property 6.28} \\
& \longrightarrow_{\eta_{val}} & \mathcal{K}\langle\!|e_0[x := e_1]|\!\rangle &
\end{array}$$

case $(rec\,f(x).\,e_0)\,e_1 \longrightarrow_{\mathrm{rec}_{ml}} e_0[f := rec\,f(x).\,e_0\,,\,x := e_1]$:

$$\begin{array}{lcll}
\mathcal{K}\langle\!|(rec\,f(x).\,e_0)\,e_1|\!\rangle & = & \lambda k\,.\,((rec\,f(x).\,\mathcal{K}\langle\!|e_0|\!\rangle)\,\mathcal{K}\langle\!|e_1|\!\rangle)\,k & \\
& \longrightarrow_{\mathrm{rec}\,exp} & \lambda k\,.\,(\mathcal{K}\langle\!|e_0|\!\rangle[f := rec\,f(x).\,\mathcal{K}\langle\!|e_0|\!\rangle\,,\,x := \mathcal{K}\langle\!|e_1|\!\rangle])\,k & \\
& = & \lambda k\,.\,(\mathcal{K}\langle\!|e_0|\!\rangle[f := \mathcal{K}\langle\!|rec\,f(x).\,e_0|\!\rangle\,,\,x := \mathcal{K}\langle\!|e_1|\!\rangle])\,k & \\
& = & \lambda k\,.\,\mathcal{K}\langle\!|e_0[f := rec\,f(x).\,e_0\,,\,x := e_1]|\!\rangle\,k & \textit{...by Property 6.28} \\
& \longrightarrow_{\eta_{val}} & \mathcal{K}\langle\!|e_0[f := rec\,f(x).\,e_0\,,\,x := e_1]|\!\rangle &
\end{array}$$

case $let\,x \Leftarrow [\![e_1]\!]\,in\,e_2 \longrightarrow_{\mathrm{let}.\beta} e_2[x := e_1]$:

$$\begin{array}{lcll}
\mathcal{K}\langle\!|let\,x \Leftarrow [\![e_1]\!]\,in\,e_2|\!\rangle & = & \lambda k\,.\,(\lambda k\,.\,k\,\mathcal{K}\langle\!|e_1|\!\rangle)\,(\lambda x\,.\,\mathcal{K}\langle\!|e_2|\!\rangle\,k) & \\
& \longrightarrow_{\beta_{ans.2}} & \lambda k\,.\,(\lambda x\,.\,\mathcal{K}\langle\!|e_2|\!\rangle\,k)\,\mathcal{K}\langle\!|e_0|\!\rangle & \\
& \longrightarrow_{\beta_{ans.1}} & \lambda k\,.\,(\mathcal{K}\langle\!|e_2|\!\rangle\,k)[x := \mathcal{K}\langle\!|e_1|\!\rangle] & \\
& = & \lambda k\,.\,(\mathcal{K}\langle\!|e_2|\!\rangle[x := \mathcal{K}\langle\!|e_1|\!\rangle])\,k & \\
& = & \lambda k\,.\,\mathcal{K}\langle\!|e_2[x := e_1]|\!\rangle\,k & \textit{...by Property 6.28} \\
& \longrightarrow_{\eta_{val}} & \mathcal{K}\langle\!|e_2[x := e_1]|\!\rangle &
\end{array}$$

case $let\,x \Leftarrow e\,in\,[\![x]\!] \longrightarrow_{\mathrm{let}.\eta} e$:

$$\begin{array}{lcll}
\mathcal{K}\langle\!|let\,x \Leftarrow e\,in\,[\![x]\!]|\!\rangle & = & \lambda k\,.\,\mathcal{K}\langle\!|e|\!\rangle\,(\lambda x\,.\,(\lambda k\,.\,k\,x)\,k) & \\
& \longrightarrow_{\beta_{ans.2}} & \lambda k\,.\,\mathcal{K}\langle\!|e|\!\rangle\,(\lambda x\,.\,k\,x) & \\
& \longrightarrow_{\eta_{cont}} & \lambda k\,.\,\mathcal{K}\langle\!|e|\!\rangle\,k & \\
& \longrightarrow_{\eta_{val}} & \mathcal{K}\langle\!|e|\!\rangle &
\end{array}$$

case $let\,x_2 \Leftarrow (let\,x_1 \Leftarrow e_1\,in\,e_2)\,in\,e_3 \longrightarrow_{\mathrm{let.assoc}} let\,x_1 \Leftarrow e_1\,in\,(let\,x_2 \Leftarrow e_2\,in\,e_3)$:

$$\begin{array}{lcll}
\mathcal{K}\langle\!|let\,x_2 \Leftarrow (let\,x_1 \Leftarrow e_1\,in\,e_2)\,in\,e_3|\!\rangle & = & \lambda k\,.\,(\lambda k\,.\,\mathcal{K}\langle\!|e_1|\!\rangle\,(\lambda x_1\,.\,\mathcal{K}\langle\!|e_2|\!\rangle\,k))\,(\lambda x_2\,.\,\mathcal{K}\langle\!|e_3|\!\rangle\,k) & \\
& \longrightarrow_{\beta_{ans.2}} & \lambda k\,.\,\mathcal{K}\langle\!|e_1|\!\rangle\,(\lambda x_1\,.\,\mathcal{K}\langle\!|e_2|\!\rangle\,(\lambda x_2\,.\,\mathcal{K}\langle\!|e_3|\!\rangle\,k)) & \\
& \longleftarrow_{\beta_{ans.2}} & \lambda k\,.\,\mathcal{K}\langle\!|e_1|\!\rangle\,(\lambda x_1\,.\,(\lambda k\,.\,\mathcal{K}\langle\!|e_2|\!\rangle\,(\lambda x_2\,.\,\mathcal{K}\langle\!|e_3|\!\rangle\,k))\,k) & \\
& = & \mathcal{K}\langle\!|let\,x_1 \Leftarrow e_1\,in\,(let\,x_2 \Leftarrow e_2\,in\,e_3)|\!\rangle &
\end{array}$$

∎

In the other direction, each R_{cps} reduction induces R_{ml} reductions.

Property 6.34 *For* $\Gamma \vdash_{val} v : \sigma$, $\Gamma \vdash_{exp} w : \neg\neg\sigma$, $\langle \Gamma\,;\,k : \neg\sigma_0\rangle \vdash_{cont} \kappa : \neg\sigma_1$, $\langle \Gamma\,;\,k : \neg\sigma\rangle \vdash_{ans} \alpha : ans$,

$$\begin{array}{lrcl}
(a) & v \longrightarrow_{R_{cps}} v' & \Rightarrow & \mathcal{K}^{-1}\langle\!|v|\!\rangle \longrightarrow\!\!\!\!\rightarrow_{R_{ml}} \mathcal{K}^{-1}\langle\!|v'|\!\rangle \\
(b) & w \longrightarrow_{R_{cps}} w' & \Rightarrow & \mathcal{K}^{-1}\langle\!|w|\!\rangle \longrightarrow\!\!\!\!\rightarrow_{R_{ml}} \mathcal{K}^{-1}\langle\!|w'|\!\rangle \\
(c) & \kappa \longrightarrow_{R_{cps}} \kappa' & \Rightarrow & \mathcal{K}^{-1}\langle\!|\kappa|\!\rangle \longrightarrow\!\!\!\!\rightarrow_{R_{ml}} \mathcal{K}^{-1}\langle\!|\kappa'|\!\rangle \\
(d) & \alpha \longrightarrow_{R_{cps}} \alpha' & \Rightarrow & \mathcal{K}^{-1}\langle\!|\alpha|\!\rangle \longrightarrow\!\!\!\!\rightarrow_{R_{ml}} \mathcal{K}^{-1}\langle\!|\alpha'|\!\rangle
\end{array}$$

Proof: by simultaneous induction over the structure of terms v, w, κ, and α. Only the details of the prime cases (*i.e.*, where reductions occur at the root of terms) are given. The rest of the cases are trivial (*i.e.*, no reductions are possible) or follow from the induction hypotheses and compatibility.

case $(\lambda x\,.\,v_0)\,v_1 \longrightarrow_{\beta\,exp} v_0[x := v_1]$:

$$\begin{array}{lcll}
\mathcal{K}^{-1}\langle\!|(\lambda x\,.\,v_0)\,v_1|\!\rangle & = & (\lambda x\,.\,\mathcal{K}^{-1}\langle\!|v_0|\!\rangle)\,\mathcal{K}^{-1}\langle\!|v_1|\!\rangle & \\
& = & \mathcal{K}^{-1}\langle\!|v_0|\!\rangle[x := \mathcal{K}^{-1}\langle\!|v_1|\!\rangle] & \\
& = & \mathcal{K}^{-1}\langle\!|v_0[x := v_1]|\!\rangle & \textit{...by Property 6.29}
\end{array}$$

case $(rec\,f(x).\,v_0)\,v_1 \longrightarrow_{\mathrm{rec}\,exp} v_0[f := rec\,f(x).\,v_0\,,\,x := v_1]$:

$$\begin{array}{lcl}
\mathcal{K}^{-1}\langle\!|(\mathit{rec}\, f(x).\, v_0)\, v_1|\!\rangle & = & (\mathit{rec}\, f(x).\, \mathcal{K}^{-1}\langle\!|v_0|\!\rangle)\, \mathcal{K}^{-1}\langle\!|v_1|\!\rangle \\
& = & \mathcal{K}^{-1}\langle\!|v_0|\!\rangle[f := \mathit{rec}\, f(x).\, \mathcal{K}^{-1}\langle\!|v_0|\!\rangle\, ,\, x := \mathcal{K}^{-1}\langle\!|v_1|\!\rangle] \\
& = & \mathcal{K}^{-1}\langle\!|v_0|\!\rangle[f := \mathcal{K}^{-1}\langle\!|\mathit{rec}\, f(x).\, v_0|\!\rangle\, ,\, x := \mathcal{K}^{-1}\langle\!|v_1|\!\rangle] \\
& = & \mathcal{K}^{-1}\langle\!|v_0[f := \mathit{rec}\, f(x).\, v_0\, ,\, x := v_1]|\!\rangle \qquad \textit{...by Property 6.29}
\end{array}$$

case $(\lambda x\,.\,\alpha)\, v \longrightarrow_{\beta_{ans.1}} \alpha[x := v]$:

$$\begin{array}{lcl}
\mathcal{K}^{-1}\langle\!|(\lambda x\,.\,\alpha)\, v|\!\rangle & = & \mathit{let}\, x \Leftarrow [\mathcal{K}^{-1}\langle\!|v|\!\rangle]\ \mathit{in}\ \mathcal{K}^{-1}\langle\!|\alpha|\!\rangle \\
& \longrightarrow_{\mathrm{let}.\beta} & \mathcal{K}^{-1}\langle\!|\alpha|\!\rangle[x := \mathcal{K}^{-1}\langle\!|v|\!\rangle] \\
& = & \mathcal{K}^{-1}\langle\!|\alpha[x := v]|\!\rangle \qquad \textit{...by Property 6.29}
\end{array}$$

case $(\lambda k\,.\,\alpha)\, \kappa \longrightarrow_{\beta_{ans.2}} \alpha[k := \kappa]$:

$$\begin{array}{lcl}
\mathcal{K}^{-1}\langle\!|(\lambda k\,.\,\alpha)\, \kappa|\!\rangle & = & \mathcal{K}^{-1}\langle\!|\kappa|\!\rangle[\mathcal{K}^{-1}\langle\!|\lambda k\,.\,\alpha|\!\rangle] \\
& = & \mathcal{K}^{-1}\langle\!|\kappa|\!\rangle[\mathcal{K}^{-1}\langle\!|\alpha|\!\rangle] \\
& \longrightarrow\!\!\!\!\rightarrow_{R_{ml}} & \mathcal{K}^{-1}\langle\!|\alpha[k := \kappa]|\!\rangle \qquad \textit{...by Property 6.30}
\end{array}$$

case $\lambda k\,.\,v\, k \longrightarrow_{\eta_{val}} v$:

$$\begin{array}{lcl}
\mathcal{K}^{-1}\langle\!|\lambda k\,.\,v\, k|\!\rangle & = & \mathcal{K}^{-1}\langle\!|v\, k|\!\rangle \\
& = & \mathcal{K}^{-1}\langle\!|k|\!\rangle[\mathcal{K}^{-1}\langle\!|v|\!\rangle] \\
& = & \mathcal{K}^{-1}\langle\!|v|\!\rangle
\end{array}$$

case $\lambda x\,.\,\kappa\, x \longrightarrow_{\eta_{cont}} \kappa$ where $x \notin FV(\kappa)$:

$$\begin{array}{lcl}
\mathcal{K}^{-1}\langle\!|\lambda x\,.\,\kappa\, x|\!\rangle & = & \mathit{let}\, x \Leftarrow [\cdot]\ \mathit{in}\ \mathcal{K}^{-1}\langle\!|\kappa\, x|\!\rangle \\
& = & \mathit{let}\, x \Leftarrow [\cdot]\ \mathit{in}\ \mathcal{K}^{-1}\langle\!|\kappa|\!\rangle[[x]]
\end{array}$$

case $\kappa \equiv k$:

$$\begin{array}{lcl}
\mathit{let}\, x \Leftarrow [\cdot]\ \mathit{in}\ \mathcal{K}^{-1}\langle\!|k|\!\rangle[[x]] & = & \mathit{let}\, x \Leftarrow [\cdot]\ \mathit{in}\ [x] \\
& \longrightarrow_{\mathrm{let}.\eta} & [\cdot] \\
& = & \mathcal{K}^{-1}\langle\!|k|\!\rangle
\end{array}$$

case $\kappa \equiv \lambda y\,.\,\alpha$:

$$\begin{array}{lcll}
\mathit{let}\, x \Leftarrow [\cdot]\ \mathit{in}\ \mathcal{K}^{-1}\langle\!|\lambda y\,.\,\alpha|\!\rangle[[x]] & = & \mathit{let}\, x \Leftarrow [\cdot]\ \mathit{in}\ \mathit{let}\, y \Leftarrow [x]\ \mathit{in}\ \mathcal{K}^{-1}\langle\!|\alpha|\!\rangle & \\
& \longrightarrow_{\mathrm{let}.\beta} & \mathit{let}\, x \Leftarrow [\cdot]\ \mathit{in}\ \mathcal{K}^{-1}\langle\!|\alpha|\!\rangle[y := x] & \\
& \equiv_{\alpha} & \mathit{let}\, y \Leftarrow [\cdot]\ \mathit{in}\ \mathcal{K}^{-1}\langle\!|\alpha|\!\rangle & \textit{...since } x \notin FV(\lambda y\,.\,\alpha) \\
& = & \mathcal{K}^{-1}\langle\!|\lambda y\,.\,\alpha|\!\rangle &
\end{array}$$

∎

Components (3) and (4) of the equational correspondence between Λ_{ml} and Λ_{cps} follow from the two properties above as outlined in Chapter 3.

Adequacy for $\mathcal{K}$ and $\mathcal{K}^{-1}$

We now prove computational adequacy results for $\mathcal{K}$ (Theorem 6.1) and for $\mathcal{K}^{-1}$ (Theorem 6.2). For Theorem 6.1, we show for all $\cdot \vdash_{ml} e : \widetilde{\nu}$,

$$\mathit{eval}_{ml}(e) \simeq_{obs} \mathit{eval}_a(\mathcal{K}\langle\!|e|\!\rangle\, (\lambda x\,.\,x)).$$

This is a consequence of showing the following:

1. $\mathit{eval}_{ml}(e) \preceq_{obs} \mathit{eval}_a(\mathcal{K}\langle\!|e|\!\rangle\, (\lambda x\,.\,x))$ — that is, $\mathit{eval}_{ml}(e) = v$ implies $\mathit{eval}_a(\mathcal{K}\langle\!|e|\!\rangle\, (\lambda x\,.\,x)) = v'$ and $\mathit{Obs}(v) = \mathit{Obs}(v')$;
2. $\mathit{eval}_a(\mathcal{K}\langle\!|e|\!\rangle\, (\lambda x\,.\,x)) \preceq_{obs} \mathit{eval}_{ml}(e)$ — given the above, it will only be necessary to show $\mathcal{K}\langle\!|e|\!\rangle\, (\lambda x\,.\,x)\downarrow_a$ implies $e\downarrow_{ml}$.

The proofs make use of the equational correspondence between Λ_{ml} and Λ_{cps} and the fact that reductions in each language preserve observational equivalence. The statement and proof of component 1 above is as follows.

Lemma 6.3 *For all* $\cdot \vdash_{ml} e : \widetilde{\nu}$,

$$eval_{ml}(e) \preceq_{obs} eval_a(\mathcal{K}\langle\!|e|\!\rangle\,(\lambda x\,.\,x))$$

Proof: Assume $eval_{ml}(e) = v$. Then by definition of $eval_{ml}$, $e \longrightarrow\!\!\!\!\!\rightarrow_{R_{ml}} [v]$. Now from the equational correspondence (Theorem 6.3),

$$\begin{array}{rll} \mathcal{K}\langle\!|e|\!\rangle\,(\lambda x\,.\,x) & =_{\beta_a \mathrm{rec}_a \eta_v} & \mathcal{K}\langle\!|[v]|\!\rangle\,(\lambda x\,.\,x) \\ & = & (\lambda k\,.\,k\,\mathcal{K}\langle\!|v|\!\rangle)\,(\lambda x\,.\,x) \\ & \longrightarrow\!\!\!\!\!\rightarrow_{\beta_a} & \mathcal{K}\langle\!|v|\!\rangle \qquad \textit{...which is a value} \end{array}$$

By soundness of $\beta_a \mathrm{rec}_a \eta_v$ it follows that $eval_a(\mathcal{K}\langle\!|e|\!\rangle\,(\lambda x\,.\,x))\downarrow$ and $Obs(eval_a(\mathcal{K}\langle\!|e|\!\rangle\,(\lambda x\,.\,x))) = Obs(\mathcal{K}\langle\!|v|\!\rangle)$. Finally, it is straightforward to show that for all $\cdot \vdash_{ml} v : \nu$, $Obs(v) = Obs(\mathcal{K}\langle\!|v|\!\rangle)$. ∎

For the opposite direction, we show that the evaluation of Λ_{cps} term induces a series of R_{ml} reductions. However, things are not as straightforward here since the *evaluation* of a Λ_{cps} term will take us outside the language of Λ_{cps} terms. Specifically, a Λ_{cps} term $\cdot \vdash_{val} v : \neg\neg\sigma$ is evaluated as

$$v\,(\lambda x\,.\,x) \longmapsto_a e_1 \longmapsto_a e_2 \longmapsto_a \ldots$$

The terms e_i are in the language Λ^σ but not in Λ_{cps} since they contain the continuation $\lambda x\,.\,x$ which is not in Λ_{cps}. Thus, Λ_{cps} terms under evaluation cannot be directly connected with Λ_{ml} *via* $\mathcal{K}$ (since $\mathcal{K}$ is defined over Λ_{cps} terms). Establishing a connection requires that the initial continuation $\lambda x\,.\,x$ be factored out of each of the e_i to produce terms $v_i \in \Lambda_{cps}$. The following definition is the basis of the connection.

Definition 6.3 (Factoring of initial continuation for Λ_{cps}) *For all* $\cdot \vdash_{val} \lambda k\,.\,\alpha : \neg\neg\sigma$ *and for any abstraction* $\cdot \vdash a : \neg\sigma$,

$$(\lambda k\,.\,\alpha) \bullet a \stackrel{\mathrm{def}}{=} \alpha[k := a]$$

The next three properties show that the CPS evaluation

$$\ldots \longmapsto_a e_1 \longmapsto_a e_2 \longmapsto_a e_3 \longmapsto_a \longmapsto_a e_4 \longmapsto_a \ldots$$

which uses the initial continuation $\lambda x\,.\,x$ can be described as

$$\ldots \longmapsto_a v_1 \bullet (\lambda x\,.\,x) \longmapsto_a v_2 \bullet (\lambda x\,.\,x) \longmapsto_a v_3 \bullet (\lambda x\,.\,x) \longmapsto_a v_4 \bullet (\lambda x\,.\,x) \longmapsto_a \ldots$$

where $e_i \equiv v_i \bullet (\lambda x\,.\,x)$ and $v_i \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} v_{i+1}$.

Property 6.35 *For all* $\cdot \vdash_{val} v : \neg\neg\sigma$ *and for all abstractions* $\cdot \vdash a : \neg\sigma$,

$$v\,a \longmapsto_a v \bullet a$$

Proof: All terms $\cdot \vdash_{val} v : \neg\neg\sigma$ take the form $\lambda k\,.\,\alpha$ and $(\lambda k\,.\,\alpha)\,a \longmapsto_a \alpha[k := a]$. ∎

Property 6.36 *For all* $\cdot \vdash_{val} v : \neg\neg\sigma$,

$$v\,(\lambda x\,.\,x) \longmapsto_a e$$

and exactly one of the following conditions holds:

1. $e \notin Values_a[\Lambda^\sigma]$ *and there exists a* $v' \in val[\Lambda_{cps}]$ *such that* $e \equiv v' \bullet (\lambda x\,.\,x)$ *and* $v \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} v'$; *or*
2. $e \in Values_a[\Lambda^\sigma]$ *and* $v \equiv \lambda k\,.\,k\,e$.

Proof: We implicitly make use of Properties 6.24, 6.25, and 6.26 which state that the language Λ_{cps} is closed under relevant substitutions and reductions. Substitution manipulations rely on the fact that k does not occur free in $val[\Lambda_{cps}]$ and $exp[\Lambda_{cps}]$ terms. According to the type requirements, the closed term v can only take the form $\lambda k\,.\,\alpha$. We now proceed with the proof by structural induction — we only need consider the possible forms of α.

case $\alpha \equiv \kappa\, v_0$:

case $\kappa \equiv k$: so $v \equiv \lambda k\,.\,k\, v_0$ and

$$(\lambda k\,.\,k\, v_0) \bullet (\lambda x\,.\,x) \;=\; (\lambda x\,.\,x)\, v_0 \;\longmapsto_a\; v_0$$

and therefore condition 2 holds.

case $\kappa \equiv k$: so $v \equiv \lambda k\,.\,(\lambda y\,.\,\alpha_0)\, v_0$ and

$$\begin{array}{rcl} (\lambda k\,.\,(\lambda y\,.\,\alpha_0)\, v_0) \bullet (\lambda x\,.\,x) & = & ((\lambda y\,.\,\alpha_0)\, v_0)[k := \lambda x\,.\,x] \\ & = & (\lambda y\,.\,\alpha_0[k := \lambda x\,.\,x])\, v_0 \\ & \longmapsto_a & (\alpha_0[k := \lambda x\,.\,x])[y := v_0] \\ & \equiv & e \end{array}$$

Now condition 1 holds since

$$e \;\equiv\; (\alpha_0[y := v_0])[k := \lambda x\,.\,x] \;=\; \lambda k\,.\,\alpha_0[y := v_0] \bullet \lambda x\,.\,x$$

and

$$v \;\equiv\; \lambda k\,.\,(\lambda x\,.\,\alpha_0)\, v_0 \;\longrightarrow\!\!\!\!\!\rightarrow_{\beta ans.1}\; \lambda k\,.\,\alpha_0[y := v_0] \;\equiv\; v'.$$

case $\alpha \equiv w\,\kappa$: only the following cases are applicable.

case $w \equiv (\lambda y\,.\,v_0)\, v_1$: so $v \equiv \lambda k\,.\,((\lambda y\,.\,v_0)\, v_1)\,\kappa$ and

$$\begin{array}{rcl} \lambda k\,.\,((\lambda y\,.\,v_0)\, v_1)\,\kappa \bullet \lambda x\,.\,x & = & ((\lambda y\,.\,v_0)\, v_1)\,(\kappa[k := \lambda x\,.\,x]) \\ & \longmapsto_a & (v_0[x := v_1])\,(\kappa[k := \lambda x\,.\,x]) \\ & \equiv & e \end{array}$$

Now condition 1 holds since

$$e \;\equiv\; ((v_0[y := v_1])\,\kappa)[k := \lambda x\,.\,x] \;=\; \lambda k\,.\,(v_0[y := v_1])\,\kappa \bullet \lambda x\,.\,x$$

and

$$v \;\equiv\; \lambda k\,.\,((\lambda y\,.\,v_0)\, v_1)\,\kappa \;\longrightarrow\!\!\!\!\!\rightarrow_{\beta exp}\; \lambda k\,.\,(v_0[y := v_1])\,\kappa \;\equiv\; v'.$$

case $w \equiv (rec\, f(y).\, v_0)\, v_1$: similar to case above.

case $w \equiv \lambda k\,.\,\alpha_0$: so $v \equiv \lambda k\,.\,(\lambda k\,.\,\alpha_0)\,\kappa$ and

$$\begin{array}{rcl} \lambda k\,.\,(\lambda k\,.\,\alpha_0)\,\kappa \bullet \lambda x\,.\,x & = & (\lambda k\,.\,\alpha_0)\,(\kappa[k := \lambda x\,.\,x]) \\ & \longmapsto_a & \alpha_0[k := (\kappa[k := \lambda x\,.\,x])] \\ & \equiv & e \end{array}$$

Now condition 1 holds since

$$e \;\equiv\; (\alpha_0[k := \kappa])[k := \lambda y\,.\,y] \;=\; \lambda k\,.\,(\alpha_0[k := \kappa]) \bullet \lambda x\,.\,x$$

and

$$v \;\equiv\; \lambda k\,.\,(\lambda k\,.\,\alpha_0)\,\kappa \;\longrightarrow\!\!\!\!\!\rightarrow_{\beta ans.2}\; \lambda k\,.\,\alpha_0[k := \kappa] \;\equiv\; v'.$$

∎

We now consider a series of evaluation steps.

Property 6.37 *For all* $\cdot \vdash_{val} v : \neg\neg\sigma$,

1. $v \bullet \lambda x\,.\,x \longmapsto_a^* e$ *where* $e \notin Values_a[\Lambda^\sigma]$ *implies there exists a* v' *such that* $e \equiv v' \bullet \lambda x\,.\,x$ *and* $v \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} v'$.

2. $v \bullet \lambda x\,.\,x \longmapsto_a^* e \longmapsto_a v'$ *implies* $v \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} \lambda k\,.\,k\, v'$.

Proof: Component 1 is proved by induction on the number of evaluation steps (applying Property 6.36 at each step). For component 2, note that by component 1, there exists a v'' such that $e \equiv v'' \bullet \lambda x\,.\,x$ and $v \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} v''$. Since $v'' \bullet \lambda x\,.\,x \longmapsto_a v'$, $v'' \equiv \lambda k\,.\,k\,v'$ by Property 6.36. ∎

Now we show that a terminating Λ_{cps} evaluation implies that the corresponding Λ_{ml} evaluation is terminating.

Lemma 6.4 *For all* $\cdot \vdash_{ml} e : \widetilde{\nu}$,

$$\mathcal{K}\langle\!|e|\!\rangle\,(\lambda x\,.\,x)\!\downarrow_a \quad \textit{implies} \quad e\!\downarrow_{ml}$$

Proof: The evaluation of $\mathcal{K}\langle\!|e|\!\rangle\,(\lambda x\,.\,x)$ must take the form...

$$\begin{array}{lll} \mathcal{K}\langle\!|e|\!\rangle\,(\lambda x\,.\,x) & \longmapsto_a & \mathcal{K}\langle\!|e|\!\rangle \bullet \lambda x\,.\,x \qquad \textit{...by Property 6.35} \\ & \longmapsto_a^+ & v \qquad \textit{...one or more steps are required to reach a value by Property 6.36.} \end{array}$$

Now by Property 6.37, $\mathcal{K}\langle\!|e|\!\rangle \longrightarrow\!\!\!\!\!\rightarrow_{R_{cps}} \lambda k\,.\,k\,v$, so by equational correspondence, $(\mathcal{K}^{-1} \circ \mathcal{K})\langle\!|e|\!\rangle \longrightarrow\!\!\!\!\!\rightarrow_{R_{ml}} \mathcal{K}^{-1}\langle\!|\lambda k\,.\,k\,v|\!\rangle$. This implies $e =_{R_{ml}} [\![\mathcal{K}^{-1}\langle\!|v|\!\rangle]\!]$ which implies $e\!\downarrow_{ml}$. ∎

6.10.6 Correctness of the optimized continuation introduction $\mathcal{K}'$

In this section, we establish the correctness of the optimized continuation introduction $\mathcal{K}'$.

The following property states that $\mathcal{K}'$ preserves well-typedness of terms. Note how the subscripted versions of $\mathcal{K}'$ produce terms in the appropriate Λ_{cps} syntactic category.

Property 6.38

$$\begin{array}{lll} (a) & \Gamma \vdash_{ml} e : \sigma & \Rightarrow \quad \mathcal{K}\langle\!|\Gamma|\!\rangle \vdash_{val} \mathcal{K}'_{val}\langle\!|e|\!\rangle : \mathcal{K}\langle\!|\sigma|\!\rangle \\ (b) & \Gamma \vdash_{ml} e : \sigma & \Rightarrow \quad \mathcal{K}\langle\!|\Gamma|\!\rangle \vdash_{exp} \mathcal{K}'_{exp}\langle\!|e|\!\rangle : \mathcal{K}\langle\!|\sigma|\!\rangle \\ (c) & \Gamma \vdash_{ml} e : \sigma & \Rightarrow \quad \langle \mathcal{K}\langle\!|\Gamma|\!\rangle\,;\, k : \neg\mathcal{K}\langle\!|\sigma|\!\rangle \rangle \vdash_{ans} \mathcal{K}'_{ans}\langle\!|e|\!\rangle : ans \end{array}$$

Proof: Similar to the proof of Property 6.22. ∎

Property 6.39 *For all* $e \in \Lambda_{ml}$,

$$\mathcal{K}\langle\!|e|\!\rangle \longrightarrow\!\!\!\!\!\rightarrow_{\beta ans.2} \mathcal{K}'_{val}\langle\!|e|\!\rangle$$

Proof: by induction over the structure of e.

case $e \equiv c$:

$$\begin{array}{lll} \mathcal{K}\langle\!|c|\!\rangle & = & c \\ & = & \mathcal{K}'_{val}\langle\!|c|\!\rangle \end{array}$$

case $e \equiv x$:

$$\begin{array}{lll} \mathcal{K}\langle\!|x|\!\rangle & = & x \\ & = & \mathcal{K}'_{val}\langle\!|x|\!\rangle \end{array}$$

case $e \equiv \lambda x\,.\,e_0$:

$$\begin{array}{lll} \mathcal{K}\langle\!|\lambda x\,.\,e_0|\!\rangle & = & \lambda x\,.\,\mathcal{K}\langle\!|e_0|\!\rangle \\ & \longrightarrow\!\!\!\!\!\rightarrow_{\beta ans.2} & \lambda x\,.\,\mathcal{K}'_{val}\langle\!|e_0|\!\rangle \qquad \textit{...by ind. hyp. and compatibility} \\ & = & \mathcal{K}'_{val}\langle\!|\lambda x\,.\,e_0|\!\rangle \end{array}$$

case $e \equiv rec\,f(x).\,e_0$: as above.

case $e \equiv e_0\,e_1$:

$$\begin{array}{lll} \mathcal{K}\langle\!|e_0\,e_1|\!\rangle & = & \lambda k\,.\,(\mathcal{K}\langle\!|e_0|\!\rangle\,\mathcal{K}\langle\!|e_1|\!\rangle)\,k \\ & \longrightarrow\!\!\!\!\!\rightarrow_{\beta ans.2} & \lambda k\,.\,(\mathcal{K}'_{val}\langle\!|e_0|\!\rangle\,\mathcal{K}'_{val}\langle\!|e_1|\!\rangle)\,k \qquad \textit{...by ind. hyp. and compatibility} \\ & = & \lambda k\,.\,\mathcal{K}'_{exp}\langle\!|e_0\,e_1|\!\rangle\,k \\ & = & \mathcal{K}'_{val}\langle\!|e_0\,e_1|\!\rangle \end{array}$$

case $e \equiv [\![e_0]\!]$:

$$\begin{array}{lcll} \mathcal{K}\langle\!|[\![e_0]\!]|\!\rangle & = & \lambda k\,.\,k\,\mathcal{K}\langle\!|[\![e_0]\!]|\!\rangle & \\ & \twoheadrightarrow_{\beta ans.2} & \lambda k\,.\,k\,\mathcal{K}'_{val}\langle\!|[\![e_0]\!]|\!\rangle & \textit{...by ind. hyp. and compatibility} \\ & = & \lambda k\,.\,\mathcal{K}'_{ans}\langle\!|[\![e_0]\!]|\!\rangle & \\ & = & \mathcal{K}'_{val}\langle\!|[\![e_0]\!]|\!\rangle & \end{array}$$

case $e \equiv let\ x \Leftarrow e_1\ in\ e_2$:

We need to show $\mathcal{K}\langle\!|[\![let\ x \Leftarrow e_1\ in\ e_2]\!]|\!\rangle \twoheadrightarrow_{\beta ans.2} \lambda k\,.\,\mathcal{K}'_{exp}\langle\!|[\![e_1]\!]|\!\rangle\,(\lambda x\,.\,\mathcal{K}'_{ans}\langle\!|[\![e_2]\!]|\!\rangle)$. First note that

$$\begin{array}{lcll} \mathcal{K}\langle\!|[\![let\ x \Leftarrow e_1\ in\ e_2]\!]|\!\rangle & = & \lambda k\,.\,\mathcal{K}\langle\!|[\![e_1]\!]|\!\rangle\,(\lambda x\,.\,\mathcal{K}\langle\!|[\![e_2]\!]|\!\rangle\,k) & \\ & \twoheadrightarrow_{\beta ans.2} & \lambda k\,.\,\mathcal{K}'_{val}\langle\!|[\![e_1]\!]|\!\rangle\,(\lambda x\,.\,\mathcal{K}'_{val}\langle\!|[\![e_2]\!]|\!\rangle\,k) & \textit{...by ind. hyp. and compatibility} \end{array}$$

Next, show $\lambda k\,.\,\mathcal{K}'_{val}\langle\!|[\![e_1]\!]|\!\rangle\,(\lambda x\,.\,\mathcal{K}'_{val}\langle\!|[\![e_2]\!]|\!\rangle\,k) \twoheadrightarrow_{\beta ans.2} \lambda k\,.\,\mathcal{K}'_{exp}\langle\!|[\![e_1]\!]|\!\rangle\,(\lambda x\,.\,\mathcal{K}'_{val}\langle\!|[\![e_2]\!]|\!\rangle\,k)$:

case $e_1 \equiv e_a\,e_b$:

$$\begin{array}{lcl} \lambda k\,.\,\mathcal{K}'_{val}\langle\!|[\![e_a\,e_b]\!]|\!\rangle\,(\lambda x\,.\,\mathcal{K}'_{val}\langle\!|[\![e_2]\!]|\!\rangle\,k) & = & \lambda k\,.\,(\lambda k\,.\,\mathcal{K}'_{exp}\langle\!|[\![e_a\,e_b]\!]|\!\rangle\,k)\,(\lambda x\,.\,\mathcal{K}'_{val}\langle\!|[\![e_2]\!]|\!\rangle\,k) \\ & \twoheadrightarrow_{\beta ans.2} & \lambda k\,.\,\mathcal{K}'_{exp}\langle\!|[\![e_a\,e_b]\!]|\!\rangle\,(\lambda x\,.\,\mathcal{K}'_{val}\langle\!|[\![e_2]\!]|\!\rangle\,k) \end{array}$$

case $e_1 \not\equiv e_a\,e_b$:

$$\lambda k\,.\,\mathcal{K}'_{val}\langle\!|[\![e_1]\!]|\!\rangle\,(\lambda x\,.\,\mathcal{K}'_{val}\langle\!|[\![e_2]\!]|\!\rangle\,k) \quad = \quad \lambda k\,.\,\mathcal{K}'_{exp}\langle\!|[\![e_1]\!]|\!\rangle\,(\lambda x\,.\,\mathcal{K}'_{val}\langle\!|[\![e_2]\!]|\!\rangle\,k)$$

Now show $\lambda k\,.\,\mathcal{K}'_{exp}\langle\!|[\![e_1]\!]|\!\rangle\,(\lambda x\,.\,\mathcal{K}'_{val}\langle\!|[\![e_2]\!]|\!\rangle\,k) \twoheadrightarrow_{\beta ans.2} \lambda k\,.\,\mathcal{K}'_{exp}\langle\!|[\![e_1]\!]|\!\rangle\,(\lambda x\,.\,\mathcal{K}'_{ans}\langle\!|[\![e_2]\!]|\!\rangle)$:

cases $e_2 \equiv e_a\,e_b, [\![e_a]\!]$, and $let\ y \Leftarrow e_a\ in\ e_b$:

$$\begin{array}{lcl} \lambda k\,.\,\mathcal{K}'_{exp}\langle\!|[\![e_1]\!]|\!\rangle\,(\lambda x\,.\,\mathcal{K}'_{val}\langle\!|[\![e_2]\!]|\!\rangle\,k) & = & \lambda k\,.\,\mathcal{K}'_{exp}\langle\!|[\![e_1]\!]|\!\rangle\,(\lambda x\,.\,(\lambda k\,.\,\mathcal{K}'_{ans}\langle\!|[\![e_2]\!]|\!\rangle)\,k) \\ & \twoheadrightarrow_{\beta ans.2} & \lambda k\,.\,\mathcal{K}'_{exp}\langle\!|[\![e_1]\!]|\!\rangle\,(\lambda x\,.\,\mathcal{K}'_{ans}\langle\!|[\![e_2]\!]|\!\rangle[k := k]) \\ & = & \lambda k\,.\,\mathcal{K}'_{exp}\langle\!|[\![e_1]\!]|\!\rangle\,(\lambda x\,.\,\mathcal{K}'_{ans}\langle\!|[\![e_2]\!]|\!\rangle) \end{array}$$

cases $e_2 \not\equiv e_a\,e_b, [\![e_a]\!], let\ y \Leftarrow e_a\ in\ e_b$: (note $e_2 \equiv y$ is the only possible type-correct case)

$$\begin{array}{lcl} \lambda k\,.\,\mathcal{K}'_{exp}\langle\!|[\![e_1]\!]|\!\rangle\,(\lambda x\,.\,\mathcal{K}'_{val}\langle\!|[\![y]\!]|\!\rangle\,k) & = & \lambda k\,.\,\mathcal{K}'_{exp}\langle\!|[\![e_1]\!]|\!\rangle\,(\lambda x\,.\,\mathcal{K}'_{exp}\langle\!|[\![y]\!]|\!\rangle\,k) \\ & = & \lambda k\,.\,\mathcal{K}'_{exp}\langle\!|[\![e_1]\!]|\!\rangle\,(\lambda x\,.\,\mathcal{K}'_{ans}\langle\!|[\![y]\!]|\!\rangle) \end{array}$$

∎

6.10.7 Construction of $\mathcal{C}_n$ and $\mathcal{C}_v$

This section illustrates how Plotkin's CPS transformation $\mathcal{C}_n$ and $\mathcal{C}_v$ are constructed from Λ_{ml} encodings and the generic CPS transformation $\mathcal{K}'$. Only proofs for $\mathcal{C}_n$ are given. The proofs for $\mathcal{C}_v$ are similar.

Plotkin's call-by-name CPS transformation

We first consider the transformation $\mathcal{C}_n$ on terms. The following property is required for the main proof.

Property 6.40 *For all $e \in \Lambda^\sigma$,*

$$(\mathcal{K}'_{val} \circ \mathcal{E}_n)\langle\!|[\![e]\!]|\!\rangle \quad \equiv \quad (\mathcal{K}'_{exp} \circ \mathcal{E}_n)\langle\!|[\![e]\!]|\!\rangle$$

Proof: First note that for $e \not\equiv e_0\,e_1$, $\mathcal{K}'_{val}\langle\!|[\![e]\!]|\!\rangle \equiv \mathcal{K}'_{exp}\langle\!|[\![e]\!]|\!\rangle$. The property follows since $\mathcal{E}_n$ never generates a term of the form $e_0\,e_1$. ∎

Property 6.41 *For all $e \in \Lambda^\sigma$,*

$$(\mathcal{K}'_{val} \circ \mathcal{E}_n)\langle\!|[\![e]\!]|\!\rangle \quad \equiv \quad \mathcal{C}_n\langle\!|[\![e]\!]|\!\rangle$$

Proof: by induction over the structure of e.

case $e \equiv c$:

$$\begin{array}{lcl}(\mathcal{K}'_{val} \circ \mathcal{E}_n)\langle\!|c|\!\rangle & = & \mathcal{K}'_{val}\langle\!|[\![c]\!]|\!\rangle \\ & = & \lambda k \,.\, k\, c \\ & = & \mathcal{C}_n\langle\!|c|\!\rangle\end{array}$$

case $e \equiv x$:

$$\begin{array}{lcl}(\mathcal{K}'_{val} \circ \mathcal{E}_n)\langle\!|x|\!\rangle & = & \mathcal{K}'_{val}\langle\!|x|\!\rangle \\ & = & x \\ & = & \mathcal{C}_n\langle\!|x|\!\rangle\end{array}$$

case $e \equiv f$:

$$\begin{array}{lcl}(\mathcal{K}'_{val} \circ \mathcal{E}_n)\langle\!|f|\!\rangle & = & \mathcal{K}'_{val}\langle\!|[\![f]\!]|\!\rangle \\ & = & \lambda k \,.\, k\, f \\ & = & \mathcal{C}_n\langle\!|f|\!\rangle\end{array}$$

case $e \equiv \lambda x \,.\, e_0$:

$$\begin{array}{lcll}(\mathcal{K}'_{val} \circ \mathcal{E}_n)\langle\!|\lambda x \,.\, e_0|\!\rangle & = & \mathcal{K}'_{val}\langle\!|[\![\lambda x \,.\, \mathcal{E}_n\langle\!|e_0|\!\rangle]\!]|\!\rangle & \\ & = & \lambda k \,.\, k\, \mathcal{K}'_{val}\lambda x \,.\, \mathcal{E}_n\langle\!|e_0|\!\rangle & \\ & = & \lambda k \,.\, k\, (\lambda x \,.\, (\mathcal{K}'_{val} \circ \mathcal{E}_n)\langle\!|e_0|\!\rangle) & \\ & = & \lambda k \,.\, k\, (\lambda x \,.\, \mathcal{C}_n\langle\!|e_0|\!\rangle) & \textit{...by ind. hyp.} \\ & = & \mathcal{C}_n\langle\!|\lambda x \,.\, e_0|\!\rangle & \end{array}$$

case $e \equiv rec\, f(x).\, e_0$:

$$\begin{array}{lcll}(\mathcal{K}'_{val} \circ \mathcal{E}_n)\langle\!|rec\, f(x).\, e_0|\!\rangle & = & \mathcal{K}'_{val}\langle\!|[\![rec\, f(x).\, \mathcal{E}_n\langle\!|e_0|\!\rangle]\!]|\!\rangle & \\ & = & \lambda k \,.\, k\, \mathcal{K}'_{val} rec\, f(x).\, \mathcal{E}_n\langle\!|e_0|\!\rangle & \\ & = & \lambda k \,.\, k\, (rec\, f(x).\, (\mathcal{K}'_{val} \circ \mathcal{E}_n)\langle\!|e_0|\!\rangle) & \\ & = & \lambda k \,.\, k\, (rec\, f(x).\, \mathcal{C}_n\langle\!|e_0|\!\rangle) & \textit{...by ind. hyp.} \\ & = & \mathcal{C}_n\langle\!|rec\, f(x).\, e_0|\!\rangle & \end{array}$$

case $e \equiv e_0\, e_1$:

$$\begin{array}{lcll}(\mathcal{K}'_{val} \circ \mathcal{E}_n)\langle\!|e_0\, e_1|\!\rangle & = & \mathcal{K}'_{val}\langle\!|let\, y_0 \Leftarrow \mathcal{E}_n\langle\!|e_0|\!\rangle\ in\ y_0\, \mathcal{E}_n\langle\!|e_1|\!\rangle|\!\rangle & \\ & = & \lambda k \,.\, (\mathcal{K}'_{exp} \circ \mathcal{E}_n)\langle\!|e_0|\!\rangle\, (\lambda y_0 \,.\, \mathcal{K}'_{ans}\langle\!|y_0\, \mathcal{E}_n\langle\!|e_1|\!\rangle|\!\rangle) & \\ & = & \lambda k \,.\, (\mathcal{K}'_{val} \circ \mathcal{E}_n)\langle\!|e_0|\!\rangle\, (\lambda y_0 \,.\, \mathcal{K}'_{ans}\langle\!|y_0\, \mathcal{E}_n\langle\!|e_1|\!\rangle|\!\rangle) & \textit{...Property 6.40} \\ & = & \lambda k \,.\, (\mathcal{K}'_{val} \circ \mathcal{E}_n)\langle\!|e_0|\!\rangle\, (\lambda y_0 \,.\, \mathcal{K}'_{exp}\langle\!|y_0\, \mathcal{E}_n\langle\!|e_1|\!\rangle|\!\rangle\, k) & \\ & = & \lambda k \,.\, (\mathcal{K}'_{val} \circ \mathcal{E}_n)\langle\!|e_0|\!\rangle\, (\lambda y_0 \,.\, (\mathcal{K}'_{val}\langle\!|y_0|\!\rangle\, (\mathcal{K}'_{val} \circ \mathcal{E}_n)\langle\!|e_1|\!\rangle)\, k) & \\ & = & \lambda k \,.\, (\mathcal{K}'_{val} \circ \mathcal{E}_n)\langle\!|e_0|\!\rangle\, (\lambda y_0 \,.\, (y_0\, (\mathcal{K}'_{val} \circ \mathcal{E}_n)\langle\!|e_1|\!\rangle)\, k) & \\ & = & \lambda k \,.\, \mathcal{C}_n\langle\!|e_0|\!\rangle\, (\lambda y_0 \,.\, (y_0\, \mathcal{C}_n\langle\!|e_1|\!\rangle)\, k) & \textit{...by ind. hyp.} \\ & = & \mathcal{C}_n\langle\!|e_0\, e_1|\!\rangle & \end{array}$$

∎

We now consider the transformation on types.

Property 6.42 *For all $\sigma \in Types[\Lambda^\sigma]$,*

$$\begin{array}{rcl}\mathcal{K}\langle\!|\mathcal{E}_n\langle\sigma\rangle|\!\rangle & = & \mathcal{C}_n\langle\sigma\rangle \\ \mathcal{K}\langle\!|[\![\mathcal{E}_n\langle\!|\sigma|\!\rangle]\!]|\!\rangle & = & \mathcal{C}_n\langle\!|\sigma|\!\rangle \\ \mathcal{K}\langle\!|[\![\mathcal{E}_n\langle\!|\Gamma|\!\rangle]\!]|\!\rangle & = & \mathcal{C}_n\langle\!|\Gamma|\!\rangle\end{array}$$

Proof: The first two statements are proved by simultaneous induction over the structure of σ. We begin with the first statement.

case $\sigma \equiv \iota$:

$$\begin{array}{rcl}\mathcal{K}\langle\![\mathcal{E}_n\langle\iota\rangle]\!\rangle & = & \mathcal{K}\langle\![\iota]\!\rangle \\ & = & \iota \\ \mathcal{C}_n\langle\iota\rangle & & \end{array}$$

case $\sigma \equiv \sigma_1 \rightarrowtail \sigma_2$:

$$\begin{array}{rcll}\mathcal{K}\langle\![\mathcal{E}_n\langle\sigma_1 \rightarrowtail \sigma_2\rangle]\!\rangle & = & \mathcal{K}\langle\![\mathcal{E}_n\langle\![\sigma_1]\!\rangle \rightarrowtail \mathcal{E}_n\langle\![\sigma_2]\!\rangle]\!\rangle & \\ & = & \mathcal{K}\langle\![\mathcal{E}_n\langle\![\sigma_1]\!\rangle]\!\rangle \rightarrowtail \mathcal{K}\langle\![\mathcal{E}_n\langle\![\sigma_2]\!\rangle]\!\rangle & \\ & = & \mathcal{C}_n\langle\![\sigma_1]\!\rangle \rightarrowtail \mathcal{C}_n\langle\![\sigma_2]\!\rangle & \textit{...by ind. hyp.} \\ & = & \mathcal{C}_n\langle\sigma_1 \rightarrowtail \sigma_2\rangle & \end{array}$$

Continuing with the second statement.

For all σ,

$$\begin{array}{rcl}\mathcal{K}\langle\![\mathcal{E}_n\langle\![\sigma]\!\rangle]\!\rangle & = & \mathcal{K}\langle\![\widetilde{\mathcal{E}_n\langle\sigma\rangle}]\!\rangle \\ & = & \neg\neg\mathcal{K}\langle\![\mathcal{E}_n\langle\sigma\rangle]\!\rangle \\ & = & \neg\neg\mathcal{C}_n\langle\sigma\rangle \\ & = & \mathcal{C}_n\langle\![\sigma]\!\rangle \end{array}$$

For the third statement,

case $\Gamma, x:\sigma$:

$$\begin{array}{rcl}\mathcal{K}\langle\![\mathcal{E}_n\langle\![\Gamma, x:\sigma]\!\rangle]\!\rangle & = & \mathcal{K}\langle\![\mathcal{E}_n\langle\![\Gamma]\!\rangle, x:\mathcal{E}_n\langle\![\sigma]\!\rangle]\!\rangle \\ & = & \mathcal{K}\langle\![\mathcal{E}_n\langle\![\Gamma]\!\rangle]\!\rangle, x:\mathcal{K}\langle\![\mathcal{E}_n\langle\![\sigma]\!\rangle]\!\rangle \\ & = & \mathcal{C}_n\langle\![\Gamma]\!\rangle, x:\mathcal{C}_n\langle\![\sigma]\!\rangle \\ & = & \mathcal{C}_n\langle\![\Gamma, x:\sigma]\!\rangle \end{array}$$

case $\Gamma, f:\sigma$:

$$\begin{array}{rcl}\mathcal{K}\langle\![\mathcal{E}_n\langle\![\Gamma, f:\sigma]\!\rangle]\!\rangle & = & \mathcal{K}\langle\![\mathcal{E}_n\langle\![\Gamma]\!\rangle, f:\mathcal{E}_n\langle\sigma\rangle]\!\rangle \\ & = & \mathcal{K}\langle\![\mathcal{E}_n\langle\![\Gamma]\!\rangle]\!\rangle, f:\mathcal{K}\langle\![\mathcal{E}_n\langle\sigma\rangle]\!\rangle \\ & = & \mathcal{C}_n\langle\![\Gamma]\!\rangle, f:\mathcal{C}_n\langle\sigma\rangle \\ & = & \mathcal{C}_n\langle\![\Gamma, f:\sigma]\!\rangle \end{array}$$

∎

6.10.8 A generic account of administrative reductions

Property 6.43 *For all* $\Gamma \vdash_{ml} e : \sigma$, $\mathcal{K}'_{val}\langle\!|e|\!\rangle$ *is in* $\{\beta_{ans.1}\}$*-normal form.*

Proof: by induction over the structure of e. The only interesting case is where $e \equiv \mathit{let}\, x \Leftarrow e_1\ \mathit{in}\ e_2$ as this is the only case where abstractions the form $\kappa \equiv \lambda x\,.\,\alpha$ are introduced. Now

$$\mathcal{K}'_{val}\langle\!|\mathit{let}\, x \Leftarrow e_1\ \mathit{in}\ e_2|\!\rangle \;=\; \lambda k\,.\,\mathcal{K}'_{exp}\langle\!|e_1|\!\rangle\,(\lambda x\,.\,\mathcal{K}'_{ans}\langle\!|e_2|\!\rangle)$$

and here there is no $\beta_{ans.1}$ redex. The rest of the cases follow immediately from the inductive hypothesis. ∎

Property 6.44 *For all* $\Gamma \vdash_{ml} e : \sigma$, $\mathcal{K}'_{val}\langle\!|e|\!\rangle$ *is in* $\{\eta_{val}\}$*-normal form.*

Proof: by induction over the structure of e. We consider only cases where redexes may be directly introduced (*i.e.*, where abstractions of the form $\lambda k\,.\,\alpha$ are introduced). The rest of the cases follow immediately from the inductive hypothesis.

case $e \equiv e_0\, e_1$: $\mathcal{K}'_{val}\langle\!|e_0\, e_1|\!\rangle \;=\; \lambda k\,.\,(\mathcal{K}'_{val}\langle\!|e_0|\!\rangle\,\mathcal{K}'_{val}\langle\!|e_1|\!\rangle)\,k$ — not a η_{val} redex.

case $e \equiv [\![e_0]\!]$: $\mathcal{K}'_{val}\langle\!|[\![e_0]\!]|\!\rangle \;=\; \lambda k\,.\,k\,\mathcal{K}'_{val}\langle\!|e_0|\!\rangle$ — not a η_{val} redex.

case $e \equiv \mathit{let}\, x \Leftarrow e_1\ \mathit{in}\, e_2$: $\mathcal{K}'_{val}\langle\!|\mathit{let}\, x \Leftarrow e_1\ \mathit{in}\, e_2|\!\rangle \;=\; \lambda k\,.\,\mathcal{K}'_{exp}\langle\!|e_1|\!\rangle\,(\lambda x\,.\,\mathcal{K}'_{ans}\langle\!|e_2|\!\rangle)$ — not a η_{val} redex.

■

Property 6.45 *For all* $\Gamma \vdash_{ml} e : \sigma$, *If* e *is in* $\{\text{let}.\beta, \text{let.assoc}\}$*-normal form then* $\mathcal{K}'_{val}\langle\!|e|\!\rangle$ *is in* $\{\beta_{ans.2}\}$*-normal form.*

Proof: by induction over the structure e. We consider only the prime cases by noting that a $\beta_{ans.2}$-redex must take the form $\alpha \equiv w\,\kappa$ — a form that only $\mathcal{K}'_{ans}$ introduces directly. $\mathcal{K}'_{ans}$ only acts on terms e which may have a computation type $(x, e_0\, e_1, \texttt{[}e_0\texttt{]}, let)$. Also we must have $w \equiv \lambda k\,.\,\alpha_0$. By examining $\mathcal{K}'_{ans}\langle\!|e|\!\rangle$ for the possibilities of e given above, we see that only $\mathcal{K}'_{ans}\langle\!|let\, x \Leftarrow e_1\ in\ e_2|\!\rangle$ can directly produce a term of the form $\alpha \equiv (\lambda k\,.\,\alpha_0)\,\kappa$. Now

$$\mathcal{K}'_{ans}\langle\!|let\, x \Leftarrow e_1\ in\ e_2|\!\rangle \;=\; \mathcal{K}'_{exp}\langle\!|e_1|\!\rangle\,(\lambda x\,.\,\mathcal{K}'_{ans}\langle\!|e_2|\!\rangle).$$

For a $\beta_{ans.2}$-redex to exist, $\mathcal{K}'_{exp}\langle\!|e_1|\!\rangle \equiv \lambda k\,.\,\alpha_0$. By examination of the definition of $\mathcal{K}'_{exp}$, this only occurs when $e_1 \equiv \texttt{[}e'_1\texttt{]}$ or $e_1 \equiv let\, x' \Leftarrow e'_1\ in\ e'_2$, *i.e.*, $e \equiv let\, x \Leftarrow \texttt{[}e'_1\texttt{]}\ in\ e_2$ (a let.β-redex) or $e \equiv let\, x \Leftarrow (let\, x' \Leftarrow e'_1\ in\ e'_2)\ in\ e_2$ (a let.assoc-redex). However, these cases for e are disallowed by the hypothesis that e is in $\{\text{let}.\beta, \text{let.assoc}\}$-normal form. ■

Property 6.46 *For all* $\Gamma \vdash_{ml} e : \sigma$, *If* e *is in* $\{\text{let}.\eta\}$*-normal form then* $\mathcal{K}'_{val}\langle\!|e|\!\rangle$ *is in* $\{\eta_{cont}\}$*-normal form.*

Proof: by induction over the structure of e. The only interesting case is where $e \equiv let\, x \Leftarrow e_1\ in\ e_2$ as this is the only case where abstractions the form $\kappa \equiv \lambda x\,.\,\alpha$ are introduced directly. The rest of the cases follow immediately from the inductive hypothesis.

$$\mathcal{K}'_{val}\langle\!|let\, x \Leftarrow e_1\ in\ e_2|\!\rangle \;=\; \lambda k\,.\,\mathcal{K}'_{exp}\langle\!|e_1|\!\rangle\,(\lambda x\,.\,\mathcal{K}'_{ans}\langle\!|e_2|\!\rangle)$$

Now $\lambda x\,.\,\mathcal{K}'_{ans}\langle\!|e_2|\!\rangle$ will only be a η_{cont} redex if $\mathcal{K}'_{ans}\langle\!|e_2|\!\rangle \equiv \kappa\, x$. By examination of the definition $\mathcal{K}'_{ans}$, this occurs only if $e_2 \equiv \texttt{[}x\texttt{]}$ (*i.e.*, $\mathcal{K}'_{ans}\langle\!|\texttt{[}x\texttt{]}|\!\rangle = k\, x$). But this means $e \equiv let\, x \Leftarrow e_1\ in\ \texttt{[}x\texttt{]}$ which is a let.η redex thus e is not in $\{\text{let}.\eta\}$-normal form. ■

Theorem 6.5 *For all* $\Gamma \vdash_{ml} e : \sigma$, *If* e *is in monadic normal-form (i.e.,* $\{\text{let}.\beta, \text{let}.\eta, \text{let.assoc}\}$*-normal form) then* $\mathcal{K}'_{val}\langle\!|e|\!\rangle$ *is in* $\{\beta_{ans.1}, \beta_{ans.2}, \eta_{val}, \eta_{cont}\}$*-normal form.*

Proof: By Properties 6.43 and 6.44, $\mathcal{K}'_{val}\langle\!|e|\!\rangle$ is always in $\{\beta_{ans.1}, \eta_{val}\}$-normal form. Furthermore, e is in $\{\text{let}.\beta, \text{let}.\eta, \text{let.assoc}\}$-normal form implies $\mathcal{K}'_{val}\langle\!|e|\!\rangle$ is in $\{\beta_{ans.2}, \eta_{cont}\}$-normal form by Properties 6.45 and 6.46. ■

Chapter 7

Applying the Computational Meta-Language

In this chapter we give further applications of the generic framework introduced in Chapter 6. We focus on recasting the main results of Chapters 3,4, and 5 by factoring transformations through the computational meta-language. This illustrates that the generic framework is capable of describing many interestings aspects of continuation-passing styles.

7.1 Thunks and the λ-calculus

7.1.1 Encoding thunks in Λ_{ml}

We begin by considering how thunks can be encoded into Λ_{ml}. Our strategy has been to encode each Λ^σ term of type σ as a Λ_{ml} computation of type $\widetilde{\sigma}$. A special case is when a Λ^σ value of type σ is encoded *via* $\texttt{[}\cdot\texttt{]}$ as a trivial computation of type $\widetilde{\sigma}$. When thunks were added Λ^σ to obtain Λ^σ_τ, *delay* provided a way to turn an arbitrary Λ^σ_τ expression of type σ into a value of type $\widetilde{\sigma}$. This suggests that Λ^σ_τ a thunk (*i.e.*, a value of type $\widetilde{\sigma}$) be encoded *via* $\texttt{[}\cdot\texttt{]}$ as a trivial computation of type $\widetilde{\widetilde{\sigma}}$.

Based on these remarks, Figure 7.1 extends the call-by-value encoding $\mathcal{E}_v$ for Λ^σ to the language including thunks Λ^σ_τ. The correctness of the encoding follows from the correctness of $\mathcal{E}_v$ for Λ^σ and the property below.

Property 7.1 *For all programs* $\cdot \vdash e : \sigma$,

$$\mathcal{E}_v\langle\!|force\,(delay\,e)|\!\rangle \longmapsto^*_{ml} \mathcal{E}_v\langle\!|e|\!\rangle$$

Proof:

$$\begin{array}{lll} \mathcal{E}_v\langle\!|force\,(delay\,e)|\!\rangle & = & let\ y \Leftarrow \texttt{[}\mathcal{E}_v\langle\!|e|\!\rangle\texttt{]}\ in\ y \\ & \longmapsto_{ml} & \mathcal{E}_v\langle\!|e|\!\rangle \end{array}$$

∎

7.1.2 Obtaining the CPS transformation of suspension constructs

The call-by-value CPS transformation of suspension constructs *delay* and *force* given in Section 3.3 coincides with the CPS transformation constructed using the generic framework. In other words, the construction of $\mathcal{C}_v$ via $\mathcal{E}_v$ (Property 6.8) extends to the language with thunks Λ^σ_τ.

Property 7.2 *For all* $\Gamma \vdash_\tau e : \sigma$,

$$\mathcal{C}_v\langle\!|e|\!\rangle \quad \equiv \quad (\mathcal{K}' \circ \mathcal{E}_v)\langle\!|e|\!\rangle$$

Proof: by induction over the structure of e. The relevant cases are as follows.

$$
\begin{array}{rcl}
\mathcal{E}_v\langle\!|\cdot|\!\rangle & : & Terms[\Lambda^\sigma_\tau] \to Terms[\Lambda_{ml}] \\
 & \cdots & \\
\mathcal{E}_v\langle\!|force\ e|\!\rangle & = & let\ x \Leftarrow \mathcal{E}_v\langle\!|e|\!\rangle\ in\ x \\
 & & \\
\mathcal{E}_v\langle\cdot\rangle & : & Values_v[\Lambda^\sigma_\tau] \to Terms[\Lambda_{ml}] \\
 & \cdots & \\
\mathcal{E}_v\langle delay\ e\rangle & = & \mathcal{E}_v\langle\!|e|\!\rangle \\
 & & \\
\mathcal{E}_v\langle\cdot\rangle & : & Types[\Lambda^\sigma_\tau] \to Types[\Lambda_{ml}] \\
 & \cdots & \\
\mathcal{E}_v\langle\widetilde{\sigma}\rangle & = & \widetilde{\mathcal{E}_v\langle\sigma\rangle}
\end{array}
$$

Figure 7.1: Call-by-value encoding of thunks into Λ_{ml}

case $e \equiv delay\ e$:

$$
\begin{array}{rcll}
(\mathcal{K}' \circ \mathcal{E}_v)\langle\!|delay\ e|\!\rangle & & & \\
 = & \mathcal{K}'\langle\!|[\![\mathcal{E}_v\langle\!|e|\!\rangle]\!]|\!\rangle & & \\
 = & \lambda k\,.\,k\,(\mathcal{K}' \circ \mathcal{E}_v)\langle\!|e|\!\rangle & & \\
 = & \lambda k\,.\,k\,\mathcal{C}_v\langle\!|e|\!\rangle & & ...by\ ind.\ hyp. \\
 = & \mathcal{C}_v\langle\!|delay\ e|\!\rangle & &
\end{array}
$$

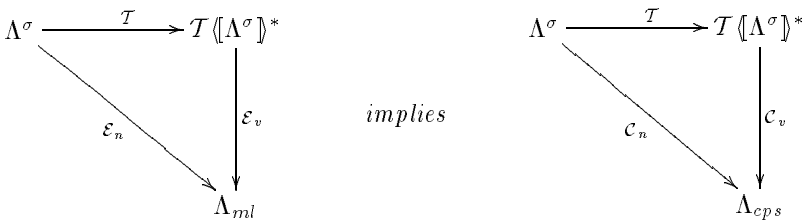
case $e \equiv force\ e$:

$$
\begin{array}{rcll}
(\mathcal{K}' \circ \mathcal{E}_v)\langle\!|force\ e|\!\rangle & & & \\
 = & \mathcal{K}'\langle\!|let\ x \Leftarrow \mathcal{E}_v\langle\!|e|\!\rangle\ in\ x|\!\rangle & & \\
 = & \lambda k\,.\,(\mathcal{K}' \circ \mathcal{E}_v)\langle\!|e|\!\rangle\ (\lambda x\,.\,\mathcal{K}'\langle\!|x|\!\rangle\ k) & & \\
 = & \lambda k\,.\,(\mathcal{K}' \circ \mathcal{E}_v)\langle\!|e|\!\rangle\ (\lambda x\,.\,x\ k) & & \\
 = & \lambda k\,.\,\mathcal{C}_v\langle\!|e|\!\rangle\ (\lambda x\,.\,x\ k) & & ...by\ ind.\ hyp. \\
 = & \mathcal{C}_v\langle\!|force\ e|\!\rangle & &
\end{array}
$$

∎

7.1.3 Connecting the thunk-based and the continuation-based simulations

Section 3.3 showed how the call-by-name CPS transformation $\mathcal{C}_n$ could be factored into the call-by-value CPS transformation $\mathcal{C}_v$ and the thunk-introducing transformation $\mathcal{T}$. Interestingly, this factorization can be captured completely in the Λ_{ml} (the CPS factorization then follows as a corollary). In pictures,



The following property formalizes the factorization *via* the meta-language.

Property 7.3 *For all* $e \in \Lambda^{\sigma}$, $(\mathcal{E}_v \circ \mathcal{T})(\!|e|\!) =_{R_{ml}} \mathcal{E}_n(\!|e|\!)$

Proof: by induction over the structure of e.

case $e \equiv c$:

$$\begin{array}{ll} & (\mathcal{E}_v \circ \mathcal{T})(\!|c|\!) \\ = & \mathcal{E}_v(\!|c|\!) \\ = & [\![c]\!] \\ = & \mathcal{E}_n(\!|c|\!) \end{array}$$

case $e \equiv x$:

$$\begin{array}{ll} & (\mathcal{E}_v \circ \mathcal{T})(\!|x|\!) \\ = & \mathcal{E}_v(\!|force\ x|\!) \\ = & let\ y \Leftarrow \mathcal{E}_v(\!|x|\!)\ in\ y \\ = & let\ y \Leftarrow [\![x]\!]\ in\ y \\ \longrightarrow_{\mathrm{let}.\beta} & x \\ = & \mathcal{E}_n(\!|x|\!) \end{array}$$

case $e \equiv f$:

$$\begin{array}{ll} & (\mathcal{E}_v \circ \mathcal{T})(\!|f|\!) \\ = & \mathcal{E}_v(\!|f|\!) \\ = & [\![f]\!] \\ = & \mathcal{E}_n(\!|f|\!) \end{array}$$

case $e \equiv \lambda x\,.\,e_0$:

$$\begin{array}{ll} & (\mathcal{E}_v \circ \mathcal{T})(\!|\lambda x\,.\,e_0|\!) \\ = & \mathcal{E}_v(\!|\lambda x\,.\,\mathcal{T}(\!|e_0|\!)|\!) \\ = & [\![\lambda x\,.\,(\mathcal{E}_v \circ \mathcal{T})(\!|e_0|\!)]\!] \\ =_{R_{ml}} & [\![\lambda x\,.\,\mathcal{E}_n(\!|e_0|\!)]\!] \qquad \textit{...by ind. hyp.} \\ = & \mathcal{E}_n(\!|\lambda x\,.\,e_0|\!) \end{array}$$

case $e \equiv rec\,f(x).\,e_0$:

$$\begin{array}{ll} & (\mathcal{E}_v \circ \mathcal{T})(\!|rec\,f(x).\,e_0|\!) \\ = & \mathcal{E}_v(\!|rec\,f(x).\,\mathcal{T}(\!|e_0|\!)|\!) \\ = & [\![rec\,f(x).\,(\mathcal{E}_v \circ \mathcal{T})(\!|e_0|\!)]\!] \\ =_{R_{ml}} & [\![rec\,f(x).\,\mathcal{E}_n(\!|e_0|\!)]\!] \qquad \textit{...by ind. hyp.} \\ = & \mathcal{E}_n(\!|rec\,f(x).\,e_0|\!) \end{array}$$

case $e \equiv e_0\,e_1$:

$$\begin{array}{ll} & (\mathcal{E}_v \circ \mathcal{T})(\!|e_0\,e_1|\!) \\ = & \mathcal{E}_v(\!|\mathcal{T}(\!|e_0|\!)\,(delay\,\mathcal{T}(\!|e_1|\!))|\!) \\ = & let\ y_0 \Leftarrow (\mathcal{E}_v \circ \mathcal{T})(\!|e_0|\!)\ in\ let\ y_1 \Leftarrow \mathcal{E}_v(\!|delay\,\mathcal{T}(\!|e_1|\!)|\!)\ in\ y_0\ y_1 \\ = & let\ y_0 \Leftarrow (\mathcal{E}_v \circ \mathcal{T})(\!|e_0|\!)\ in\ let\ y_1 \Leftarrow [\![(\mathcal{E}_v \circ \mathcal{T})(\!|e_1|\!)]\!]\ in\ y_0\ y_1 \\ \longrightarrow_{\mathrm{let}.\beta} & let\ y_0 \Leftarrow (\mathcal{E}_v \circ \mathcal{T})(\!|e_0|\!)\ in\ y_0\,(\mathcal{E}_v \circ \mathcal{T})(\!|e_1|\!) \\ =_{R_{ml}} & let\ y_0 \Leftarrow \mathcal{E}_n(\!|e_0|\!)\ in\ y_0\,\mathcal{E}_n(\!|e_1|\!) \qquad \textit{...by ind. hyp.} \\ = & \mathcal{E}_n(\!|e_0\,e_1|\!) \end{array}$$

∎

$$
\begin{array}{rcl}
\mathcal{C}_{\mathrm{R}}\langle\!|\cdot|\!\rangle & : & \mathit{Terms}[\Lambda^{\sigma}] \rightarrow \mathit{Terms}[\Lambda_{ml}] \\
\mathcal{E}_{\mathrm{R}}\langle\!|v|\!\rangle & = & [\mathcal{C}_{\mathrm{R}}\langle v\rangle] \qquad \mathit{...where}\ v \not\equiv x \\
\mathcal{E}_{\mathrm{R}}\langle\!|x|\!\rangle & = & x \\
\mathcal{E}_{\mathrm{R}}\langle\!|e_0\ e_1|\!\rangle & = & \mathit{let}\ y_0 \Leftarrow \mathcal{C}_{\mathrm{R}}\langle\!|e_0|\!\rangle\ \mathit{in}\ y_0\ \mathcal{C}_{\mathrm{R}}\langle\!|e_1|\!\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}_{\mathrm{R}}\langle\cdot\rangle & : & \mathit{Values}_v[\Lambda^{\sigma}] \rightarrow \mathit{Terms}[\Lambda_{ml}] \\
\mathcal{E}_{\mathrm{R}}\langle c\rangle & = & c \\
\mathcal{E}_{\mathrm{R}}\langle f\rangle & = & f \\
\mathcal{E}_{\mathrm{R}}\langle \lambda x\,.\,e\rangle & = & \lambda t\,.\,\mathit{let}\ y \Leftarrow t\ \mathit{in}\ (\lambda x\,.\,\mathcal{E}_{\mathrm{R}}\langle\!|e|\!\rangle)\ [y] \\
\mathcal{E}_{\mathrm{R}}\langle \mathit{rec}\ f(x).\,e\rangle & = & \lambda t\,.\,\mathit{let}\ y \Leftarrow t\ \mathit{in}\ (\mathit{rec}\ f(x).\,\mathcal{E}_{\mathrm{R}}\langle\!|e|\!\rangle)\ [y]
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}_{\mathrm{R}}\langle\!|\cdot|\!\rangle & : & \mathit{Types}[\Lambda^{\sigma}] \rightarrow \mathit{Types}[\Lambda_{ml}] \\
\mathcal{E}_{\mathrm{R}}\langle\!|\sigma|\!\rangle & = & \widetilde{\mathcal{E}_{\mathrm{R}}\langle\sigma\rangle} \\
\mathcal{E}_{\mathrm{R}}\langle\iota\rangle & = & \iota \\
\mathcal{E}_{\mathrm{R}}\langle\sigma_1 \rightarrow \sigma_2\rangle & = & \mathcal{E}_{\mathrm{R}}\langle\!|\sigma_1|\!\rangle \rightarrow \mathcal{E}_{\mathrm{R}}\langle\!|\sigma_2|\!\rangle
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{E}_{\mathrm{R}}\langle\!|\cdot|\!\rangle & : & \mathit{Assums}[\Lambda^{\sigma}] \rightarrow \mathit{Assums}[\Lambda_{ml}] \\
\mathcal{E}_{\mathrm{R}}\langle\!|\Gamma, x:\sigma|\!\rangle & = & \mathcal{E}_{\mathrm{R}}\langle\!|\Gamma|\!\rangle, x:\mathcal{E}_{\mathrm{R}}\langle\!|\sigma|\!\rangle \\
\mathcal{E}_{\mathrm{R}}\langle\!|\Gamma, f:\sigma|\!\rangle & = & \mathcal{E}_{\mathrm{R}}\langle\!|\Gamma|\!\rangle, f:\mathcal{E}_{\mathrm{R}}\langle\sigma\rangle
\end{array}
$$

Figure 7.2: Encoding of Reynolds's CPS transformation into Λ_{ml}

7.1.4 Reynolds's call-by-value CPS transformation

Plotkin's call-by-value CPS transformation $\mathcal{C}_v$ encodes call-by-value (eager evaluation) by forcing the evaluation of an argument before function application. Section 3.6.2 noted that call-by-value could also be encoded by forcing the evaluation of an argument *immediately after* it is received by a function. This is the strategy used in Reynolds's call-by-value CPS transformation [81].[1]

Figure 7.2 expresses this strategy *via* the Λ_{ml} encoding $\mathcal{E}_{\mathrm{R}}$. The encoding yields Reynolds's call-by-value transformation given in Figure 3.10 (page 59).

Property 7.4 *For all* $e \in \Lambda^{\sigma}$, $\quad \mathcal{C}_{\mathrm{R}}\langle\!|e|\!\rangle \equiv (\mathcal{K}' \circ \mathcal{E}_{\mathrm{R}})\langle\!|e|\!\rangle$

Proof: by induction over the structure of e. ∎

The two strategies for encoding call-by-value ($\mathcal{E}_v$ *vs.* $\mathcal{E}_{\mathrm{R}}$) corresponds to the two styles of describing call-by-value in denotational semantics:

1. using a strictness check in the applicative structure (corresponding to $\mathcal{E}_v$), or
2. forming strict functions to create a strict function space (corresponding to $\mathcal{E}_{\mathrm{R}}$).

Note that the strategy encoded by $\mathcal{E}_{\mathrm{R}}$ requires treating identifiers as computations even though they belong to the set $\mathit{Values}_v[\Lambda^{\sigma}]$. This is reflected in the transformation on assumptions. However, the transformed identifiers will only be bound to trivial computations — and thus they coincide with values.

Griffin [38, Footnote 3] pointed out that the typing of identifiers and function spaces in Reynolds's CPS transformation $\mathcal{C}_{\mathrm{R}}$ matches that of Plotkin's CPS transformation $\mathcal{C}_n$, *i.e.*,

$$\mathcal{C}_{\mathrm{R}}\langle\!|\Gamma, x:\sigma|\!\rangle = \mathcal{C}_{\mathrm{R}}\langle\!|\Gamma|\!\rangle, x:\mathcal{C}_{\mathrm{R}}\langle\!|\sigma|\!\rangle \qquad \mathcal{C}_{\mathrm{R}}\langle\sigma_1 \rightarrow \sigma_2\rangle = \mathcal{C}_{\mathrm{R}}\langle\!|\sigma_1|\!\rangle \rightarrow \mathcal{C}_{\mathrm{R}}\langle\!|\sigma_2|\!\rangle$$

[1] When we refer to Reynolds's call-by-value CPS transformation, we mean the transformation of Figure 3.10 where "call-by-name abstractions" are omitted from the language.

$$
\begin{array}{rcl}
\mathcal{C}_{\mathrm{R}}\langle\!|\cdot|\!\rangle & : & \mathit{Terms}[\Lambda^{\sigma}] \rightarrow \mathit{Terms}[\Lambda_{ml}] \\
\mathcal{E}_{\mathrm{R}'}\langle\!|v|\!\rangle & = & [\![\mathcal{C}_{\mathrm{R}}\langle v\rangle]\!] \\
\mathcal{E}_{\mathrm{R}'}\langle\!|e_0\, e_1|\!\rangle & = & \mathit{let}\ y_0 \Leftarrow \mathcal{C}_{\mathrm{R}}\langle\!|e_0|\!\rangle\ \mathit{in}\ y_0\, \mathcal{C}_{\mathrm{R}}\langle\!|e_1|\!\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}_{\mathrm{R}}\langle\cdot\rangle & : & \mathit{Values}_v[\Lambda^{\sigma}] \rightarrow \mathit{Terms}[\Lambda_{ml}] \\
\mathcal{E}_{\mathrm{R}'}\langle c\rangle & = & c \\
\mathcal{E}_{\mathrm{R}'}\langle x\rangle & = & x \\
\mathcal{E}_{\mathrm{R}'}\langle f\rangle & = & f \\
\mathcal{E}_{\mathrm{R}'}\langle \lambda x\,.\,e\rangle & = & \lambda t\,.\,\mathit{let}\ x \Leftarrow t\ \mathit{in}\ \mathcal{E}_{\mathrm{R}}\langle\!|e|\!\rangle \\
\mathcal{E}_{\mathrm{R}}\langle \mathit{rec}\, f(x).\, e\rangle & = & \lambda t\,.\,\mathit{let}\ x \Leftarrow t\ \mathit{in}\ (\mathit{rec}\, f(x).\, \mathcal{E}_{\mathrm{R}}\langle\!|e|\!\rangle)\, x
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}_{\mathrm{R}}\langle\!|\cdot|\!\rangle & : & \mathit{Types}[\Lambda^{\sigma}] \rightarrow \mathit{Types}[\Lambda_{ml}] \\
\mathcal{E}_{\mathrm{R}'}\langle\!|\sigma|\!\rangle & = & \widetilde{\mathcal{E}_{\mathrm{R}'}\langle\sigma\rangle} \\
\mathcal{E}_{\mathrm{R}'}\langle\iota\rangle & = & \iota \\
\mathcal{E}_{\mathrm{R}'}\langle\sigma_1 \rightarrow \sigma_2\rangle & = & \mathcal{E}_{\mathrm{R}'}\langle\!|\sigma_1|\!\rangle \rightarrow \mathcal{E}_{\mathrm{R}'}\langle\!|\sigma_2|\!\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}_{\mathrm{R}}\langle\!|\cdot|\!\rangle & : & \mathit{Assums}[\Lambda^{\sigma}] \rightarrow \mathit{Assums}[\Lambda_{ml}] \\
\mathcal{E}_{\mathrm{R}'}\langle\!|\Gamma, x:\sigma|\!\rangle & = & \mathcal{E}_{\mathrm{R}'}\langle\!|\Gamma|\!\rangle, x:\mathcal{E}_{\mathrm{R}'}\langle\sigma\rangle \\
\mathcal{E}_{\mathrm{R}'}\langle\!|\Gamma, f:\sigma|\!\rangle & = & \mathcal{E}_{\mathrm{R}'}\langle\!|\Gamma|\!\rangle, f:\mathcal{E}_{\mathrm{R}'}\langle\sigma\rangle
\end{array}
$$

Figure 7.3: Encoding of an optimized version of Reynolds's CPS transformation into Λ_{ml}

$$\mathcal{C}_n\langle\!|\Gamma, x:\sigma|\!\rangle = \mathcal{C}_n\langle\!|\Gamma|\!\rangle, x:\mathcal{C}_n\langle\!|\sigma|\!\rangle \qquad \mathcal{C}_n\langle\sigma_1 \rightarrow \sigma_2\rangle = \mathcal{C}_n\langle\!|\sigma_1|\!\rangle \rightarrow \mathcal{C}_n\langle\!|\sigma_2|\!\rangle$$

This illustrates that the CPS transformation over types does not always determine the CPS transformation over terms. Of course, the typing coincidence is also reflected by the Λ_{ml} encodings — before introducing continuations.

$$\mathcal{E}_{\mathrm{R}}\langle\!|\Gamma, x:\sigma|\!\rangle = \mathcal{E}_{\mathrm{R}}\langle\!|\Gamma|\!\rangle, x:\mathcal{E}_{\mathrm{R}}\langle\!|\sigma|\!\rangle \qquad \mathcal{E}_{\mathrm{R}}\langle\sigma_1 \rightarrow \sigma_2\rangle = \mathcal{E}_{\mathrm{R}}\langle\!|\sigma_1|\!\rangle \rightarrow \mathcal{E}_{\mathrm{R}}\langle\!|\sigma_2|\!\rangle$$

$$\mathcal{E}_n\langle\!|\Gamma, x:\sigma|\!\rangle = \mathcal{E}_n\langle\!|\Gamma|\!\rangle, x:\mathcal{E}_n\langle\!|\sigma|\!\rangle \qquad \mathcal{E}_n\langle\sigma_1 \rightarrow \sigma_2\rangle = \mathcal{E}_n\langle\!|\sigma_1|\!\rangle \rightarrow \mathcal{E}_n\langle\!|\sigma_2|\!\rangle$$

7.1.5 Variation on Reynolds's call-by-value CPS transformation

Reynolds's transformation $\mathcal{E}_{\mathrm{R}}$ can be optimized. As before, an argument is passed unevaluated to a function and forced immediately in the body of the function. However, instead of coercing the resulting argument value to a trivial computation, it can remain a value. This restores the treatment of call-by-value identifiers as meta-language values (as in $\mathcal{E}_v$) and reduces the number of steps needed to evaluate the transformed term.

Figure 7.3 gives the optimized encoding $\mathcal{E}_{\mathrm{R}'}$. The optimization is captured in the definitions for abstractions. Note that the typing of the function space in $\mathcal{E}_{\mathrm{R}'}$ still matches the one of $\mathcal{E}_n$.

$$\mathcal{E}_{\mathrm{R}'}\langle\sigma_1 \rightarrow \sigma_2\rangle = \mathcal{E}_{\mathrm{R}'}\langle\!|\sigma_1|\!\rangle \rightarrow \mathcal{E}_{\mathrm{R}'}\langle\!|\sigma_2|\!\rangle\,.$$

$$\mathcal{E}_n\langle\!|\sigma_1 \rightarrow \sigma_2|\!\rangle = \mathcal{E}_n\langle\!|\sigma_1|\!\rangle \rightarrow \mathcal{E}_n\langle\!|\sigma_2|\!\rangle\,.$$

However, the transformation on type assumptions Γ is the same as for $\mathcal{E}_v$.

$$\mathcal{E}_{\mathrm{R}'}\langle\!|\Gamma, x:\sigma|\!\rangle = \mathcal{E}_{\mathrm{R}'}\langle\!|\Gamma|\!\rangle, x:\mathcal{E}_{\mathrm{R}'}\langle\sigma\rangle$$

$$\mathcal{E}_v\langle\!|\Gamma, x:\sigma|\!\rangle = \mathcal{E}_v\langle\!|\Gamma|\!\rangle, x:\mathcal{E}_v\langle\sigma\rangle$$

That is, identifiers ($x \in \mathit{Values}_v[\Lambda^{\sigma}]$) are treated as meta-language values by $\mathcal{E}_{\mathrm{R}'}$ instead of as meta-language computations as in $\mathcal{E}_{\mathrm{R}}$.

7.2 A CPS transformation directed by strictness properties

In Chapter 4 we showed how to derive a CPS transformation $\mathcal{C}_s$ directed by strictness properties. In essence, $\mathcal{C}_s$ contains both call-by-value-like (capturing strictness) and call-by-name-like (capturing non-strictness) applications/functions/identifiers. Figure 7.4 presents a meta-language encoding $\mathcal{E}_s$ that yields the CPS transformation $\mathcal{C}_s$ of Figure 4.5 (page 87). This encoding is based on combining the styles of $\mathcal{E}_v$ and $\mathcal{E}_n$. However, a correct encoding directed by strictness information can also be obtained by combining the styles of $\mathcal{E}_{\mathrm{R}}$ and $\mathcal{E}_n$ or of $\mathcal{E}_{\mathrm{R}'}$ and $\mathcal{E}_n$. Again, as long as the computational properties are correctly identified, the correct CPS transformation and accompanying tools follow.

7.3 A CPS transformation directed by termination properties

In Chapter 5 we presented a CPS transformation $\mathcal{C}_t$ directed by totality properties. Given the generalization of the Λ_{ml} typing system discussed in Section 6.3.6, trivial terms are encoded as terms of type σ whereas serious terms are encoded as terms of type $\widetilde{\sigma}$. Figure 7.5 presents the full encoding.

7.4 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 6.7 with the one below gives a call-by-value CPS transformation where the argument is evaluated before the function in an application.

$$\mathcal{E}_v\langle\!\langle [\![e_0\, e_1]\!] \rangle\!\rangle \quad = \quad \mathit{let}\ x_1 \Leftarrow \mathcal{E}_v\langle\!\langle [\![e_1]\!] \rangle\!\rangle\ \mathit{in}\ \mathit{let}\ x_0 \Leftarrow \mathcal{E}_v\langle\!\langle [\![e_0]\!] \rangle\!\rangle\ \mathit{in}\ x_0\, x_1$$

Issues regarding different orders of sub-expression evaluation are discussed in [19, 52]

7.5 Conclusion

We have shown that CPS transformations for a variety of evaluation strategies can be described by simple encodings in Λ_{ml}. The corresponding CPS transformations and correctness proofs follow from the correctness of the encodings. Other tools such as optimizations and DS transformations follow as outlined in Chapter 6.

$$
\begin{array}{rcl}
\mathcal{E}_s & : & \mathit{Terms}[\Lambda_s] \rightarrow \mathit{Terms}[\Lambda_{ml}] \\
\mathcal{E}_s\langle\!|v|\!\rangle & = & [\mathcal{E}_s\langle v\rangle] \\
\mathcal{E}_s\langle\!|x|\!\rangle & = & x \\
\mathcal{E}_s\langle\!|@_s\, e_0\, e_1|\!\rangle & = & \mathit{let}\ y_0 \Leftarrow \mathcal{E}_s\langle\!|e_0|\!\rangle\ \mathit{in}\ \mathit{let}\ y_1 \Leftarrow \mathcal{E}_s\langle\!|e_1|\!\rangle\ \mathit{in}\ y_0\, y_1 \\
\mathcal{E}_s\langle\!|@\, e_0\, e_1|\!\rangle & = & \mathit{let}\ y_0 \Leftarrow \mathcal{E}_s\langle\!|e_0|\!\rangle\ \mathit{in}\ y_0\, \mathcal{E}_s\langle\!|e_1|\!\rangle \\
\mathcal{E}_s\langle\!|\mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2|\!\rangle & = & \mathit{let}\ y_0 \Leftarrow \mathcal{E}_s\langle\!|e_0|\!\rangle\ \mathit{in}\ \mathit{if}\ y_0\ \mathit{then}\ \mathcal{E}_s\langle\!|e_1|\!\rangle\ \mathit{else}\ \mathcal{E}_s\langle\!|e_2|\!\rangle \\
\mathcal{E}_s\langle\!|\mathit{pair}_{ss}\, e_0\, e_1|\!\rangle & = & \mathit{let}\ y_0 \Leftarrow \mathcal{E}_s\langle\!|e_0|\!\rangle\ \mathit{in}\ \mathit{let}\ y_1 \Leftarrow \mathcal{E}_s\langle\!|e_1|\!\rangle\ \mathit{in}\ [\mathit{pair}\, y_0\, y_1] \\
\mathcal{E}_s\langle\!|\mathit{pair}_{s\bullet}\, e_0\, e_1|\!\rangle & = & \mathit{let}\ y_0 \Leftarrow \mathcal{E}_s\langle\!|e_0|\!\rangle\ \mathit{in}\ [\mathit{pair}\, y_0\, \mathcal{E}_s\langle\!|e_1|\!\rangle] \\
\mathcal{E}_s\langle\!|\mathit{pair}_{\bullet s}\, e_0\, e_1|\!\rangle & = & \mathit{let}\ y_1 \Leftarrow \mathcal{E}_s\langle\!|e_1|\!\rangle\ \mathit{in}\ [\mathit{pair}\, \mathcal{E}_s\langle\!|e_0|\!\rangle\, y_1] \\
\mathcal{E}_s\langle\!|\mathit{fst}_s\, e|\!\rangle & = & \mathit{let}\ y \Leftarrow \mathcal{E}_s\langle\!|e|\!\rangle\ \mathit{in}\ [\mathit{fst}\, y] \\
\mathcal{E}_s\langle\!|\mathit{fst}\, e|\!\rangle & = & \mathit{let}\ y \Leftarrow \mathcal{E}_s\langle\!|e|\!\rangle\ \mathit{in}\ \mathit{fst}\, y \\
\mathcal{E}_s\langle\!|\mathit{snd}_s\, e|\!\rangle & = & \mathit{let}\ y \Leftarrow \mathcal{E}_s\langle\!|e|\!\rangle\ \mathit{in}\ [\mathit{snd}\, y] \\
\mathcal{E}_s\langle\!|\mathit{snd}\, e|\!\rangle & = & \mathit{let}\ y \Leftarrow \mathcal{E}_s\langle\!|e|\!\rangle\ \mathit{in}\ \mathit{snd}\, y
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}_s & : & \mathit{Values}[\Lambda_s] \rightarrow \mathit{Terms}[\Lambda_{ml}] \\
\mathcal{E}_s\langle c\rangle & = & c \\
\mathcal{E}_s\langle f\rangle & = & f \\
\mathcal{E}_s\langle x_s\rangle & = & x \\
\mathcal{E}_s\langle \lambda x_s\,.\, e\rangle & = & \lambda x\,.\,\mathcal{E}_s\langle\!|e|\!\rangle \\
\mathcal{E}_s\langle \lambda x\,.\, e\rangle & = & \lambda x\,.\,\mathcal{E}_s\langle\!|e|\!\rangle \\
\mathcal{E}_s\langle \mathit{rec}\, f(x_s).\, e\rangle & = & \mathit{rec}\, f(x).\,\mathcal{E}_s\langle\!|e|\!\rangle \\
\mathcal{E}_s\langle \mathit{rec}\, f(x).\, e\rangle & = & \mathit{rec}\, f(x).\,\mathcal{E}_s\langle\!|e|\!\rangle \\
\mathcal{E}_s\langle \mathit{pair}_{\bullet\bullet}\, e_0\, e_1\rangle & = & \mathit{pair}\, \mathcal{E}_s\langle\!|e_0|\!\rangle\, \mathcal{E}_s\langle\!|e_1|\!\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}_s\langle\!|\cdot|\!\rangle & : & \mathit{Types}[\Lambda_s] \rightarrow \mathit{Types}[\Lambda_{ml}] \\
\mathcal{E}_s\langle\!|\sigma|\!\rangle & = & \widetilde{\mathcal{E}_s\langle\sigma\rangle} \\
\mathcal{E}_s\langle \iota\rangle & = & \iota \\
\mathcal{E}_s\langle \sigma_1 \stackrel{s}{\rightarrow} \sigma_2\rangle & = & \mathcal{E}_s\langle \sigma_1\rangle \rightarrow \mathcal{E}_s\langle\!|\sigma_2|\!\rangle \\
\mathcal{E}_s\langle \sigma_1 \rightarrow \sigma_2\rangle & = & \mathcal{E}_s\langle\!|\sigma_1|\!\rangle \rightarrow \mathcal{E}_s\langle\!|\sigma_2|\!\rangle \\
\mathcal{E}_s\langle \sigma_1 \stackrel{ss}{\times} \sigma_2\rangle & = & \mathcal{E}_s\langle \sigma_1\rangle \times \mathcal{E}_s\langle \sigma_2\rangle \\
\mathcal{E}_s\langle \sigma_1 \stackrel{s\bullet}{\times} \sigma_2\rangle & = & \mathcal{E}_s\langle \sigma_1\rangle \times \mathcal{E}_s\langle\!|\sigma_2|\!\rangle \\
\mathcal{E}_s\langle \sigma_1 \stackrel{\bullet s}{\times} \sigma_2\rangle & = & \mathcal{E}_s\langle\!|\sigma_1|\!\rangle \times \mathcal{E}_s\langle \sigma_2\rangle \\
\mathcal{E}_s\langle \sigma_1 \stackrel{\bullet\bullet}{\times} \sigma_2\rangle & = & \mathcal{E}_s\langle\!|\sigma_1|\!\rangle \times \mathcal{E}_s\langle\!|\sigma_2|\!\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}_s\langle\!|\cdot|\!\rangle & : & \mathit{Assums}[\Lambda_s] \rightarrow \mathit{Assums}[\Lambda_{ml}] \\
\mathcal{E}_s\langle\!|\Gamma, x_s : \sigma|\!\rangle & = & \mathcal{E}_s\langle\!|\Gamma|\!\rangle, x : \mathcal{E}_s\langle\sigma\rangle \\
\mathcal{E}_s\langle\!|\Gamma, x : \sigma|\!\rangle & = & \mathcal{E}_s\langle\!|\Gamma|\!\rangle, x : \mathcal{E}_s\langle\!|\sigma|\!\rangle \\
\mathcal{E}_s\langle\!|\Gamma, f : \sigma|\!\rangle & = & \mathcal{E}_s\langle\!|\Gamma|\!\rangle, f : \mathcal{E}_s\langle\sigma\rangle
\end{array}
$$

Figure 7.4: Encoding of the CPS transformation $\mathcal{C}_s$ directed by strictness properties into Λ_{ml}

Serious Terms:

$$
\begin{array}{rcl}
\mathcal{E}_t\langle\!|\cdot|\!\rangle & : & \mathit{TypingDerivations}[\Lambda_t] \rightarrow \mathit{Terms}[\Lambda_{ml}] \\
\mathcal{E}_t\langle\!|d\{e : (\sigma\ ,\ \mathit{trivial})\}|\!\rangle & = & [\mathcal{E}_t\langle d\{e : (\sigma\ ,\ \mathit{trivial})\}\rangle] \\
\mathcal{E}_t\langle\!|(\mathit{var}) :: x : (\sigma\ ,\ \mathit{serious})|\!\rangle & = & x \\
\mathcal{E}_t\langle\!|(\mathit{opr}) :: \mathsf{op}\, e_1\, e_2 : (\iota\ ,\ \mathit{serious})|\!\rangle & = & \mathit{let}\ y_1 \Leftarrow \mathcal{E}_t\langle\!|d_1|\!\rangle\ \mathit{in}\ \mathit{let}\ y_2 \Leftarrow \mathcal{E}_t\langle\!|d_2|\!\rangle\ \mathit{in}\ [\mathsf{op}\, y_1\, y_2] \\
\mathcal{E}_t\langle\!|(\mathit{app}[tt]) :: e_0\, e_1 : (\sigma_2\ ,\ \mathit{serious})|\!\rangle & = & \mathit{let}\ y_0 \Leftarrow \mathcal{E}_t\langle\!|d_0|\!\rangle\ \mathit{in}\ [y_0\, \mathcal{E}_t\langle d_1\rangle] \\
\mathcal{E}_t\langle\!|(\mathit{app}[tt]) :: e_0\, e_1 : (\sigma_2\ ,\ \mathit{serious})|\!\rangle & = & \mathit{let}\ y_0 \Leftarrow \mathcal{E}_t\langle\!|d_0|\!\rangle\ \mathit{in}\ [y_0\, \mathcal{E}_t\langle\!|d_1|\!\rangle] \\
\mathcal{E}_t\langle\!|(\mathit{app}[ts]) :: e_0\, e_1 : (\sigma_2\ ,\ \mathit{serious})|\!\rangle & = & \mathit{let}\ y_0 \Leftarrow \mathcal{E}_t\langle\!|d_0|\!\rangle\ \mathit{in}\ y_0\, \mathcal{E}_t\langle d_1\rangle \\
\mathcal{E}_t\langle\!|(\mathit{app}[ss]) :: e_0\, e_1 : (\sigma_2\ ,\ \mathit{serious})|\!\rangle & = & \mathit{let}\ y_0 \Leftarrow \mathcal{E}_t\langle\!|d_0|\!\rangle\ \mathit{in}\ y_0\, \mathcal{E}_t\langle\!|d_1|\!\rangle \\
\mathcal{E}_t\langle\!|(\mathit{cnd}) :: \mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2 : (\sigma\ ,\ \mathit{serious})|\!\rangle & = & \mathit{let}\ y_0 \Leftarrow \mathcal{E}_t\langle\!|d_0|\!\rangle\ \mathit{in}\ \mathit{if}\ y_0\ \mathit{then}\ \mathcal{E}_t\langle\!|d_1|\!\rangle\ \mathit{else}\ \mathcal{E}_t\langle\!|d_2|\!\rangle \\
\mathcal{E}_t\langle\!|(\mathit{let}) :: \mathit{let}\ x \Leftarrow e_0\ \mathit{in}\ e_1 : (\sigma_1\ ,\ \mathit{serious})|\!\rangle & = & \mathit{let}\ x \Leftarrow \mathcal{E}_t\langle\!|d_0|\!\rangle\ \mathit{in}\ \mathcal{E}_t\langle\!|d_1|\!\rangle \\
\mathcal{E}_t\langle\!|(\mathit{gen}) :: e : (\sigma\ ,\ \mathit{serious})|\!\rangle & = & [\mathcal{E}_t\langle d\rangle]
\end{array}
$$

Trivial Terms:

$$
\begin{array}{rcl}
\mathcal{E}_t\langle\cdot\rangle & : & \mathit{TypingDerivations}[\Lambda_t] \rightarrow \mathit{Terms}[\Lambda_{ml}] \\
\mathcal{E}_t\langle(\mathit{var}) :: x : (\sigma\ ,\ \mathit{trivial})\rangle & = & x \\
\mathcal{E}_t\langle(\mathit{recvar}) :: f : (\sigma\ ,\ \mathit{trivial})\rangle & = & f \\
\mathcal{E}_t\langle(\mathit{const}) :: c : (\iota\ ,\ \mathit{trivial})\rangle & = & c \\
\mathcal{E}_t\langle(\mathit{opr}) :: \mathsf{op}\, e_1\, e_2 : (\iota\ ,\ \mathit{trivial})\rangle & = & \mathsf{op}\, \mathcal{E}_t\langle d_1\rangle\, \mathcal{E}_t\langle d_2\rangle \\
\mathcal{E}_t\langle(\mathit{abs}[tt]) :: \lambda x\,.\,e : (\sigma_1 \rightarrow_{tt} \sigma_2\ ,\ \mathit{trivial})\rangle & = & \lambda x\,.\,\mathcal{E}_t\langle d\rangle \\
\mathcal{E}_t\langle(\mathit{abs}[st]) :: \lambda x\,.\,e : (\sigma_1 \rightarrow_{st} \sigma_2\ ,\ \mathit{trivial})\rangle & = & \lambda x\,.\,\mathcal{E}_t\langle d\rangle \\
\mathcal{E}_t\langle(\mathit{abs}[ts]) :: \lambda x\,.\,e : (\sigma_1 \rightarrow_{ts} \sigma_2\ ,\ \mathit{trivial})\rangle & = & \lambda x\,.\,\mathcal{E}_t\langle\!|d|\!\rangle \\
\mathcal{E}_t\langle(\mathit{abs}[ss]) :: \lambda x\,.\,e : (\sigma_1 \rightarrow_{ss} \sigma_2\ ,\ \mathit{trivial})\rangle & = & \lambda x\,.\,\mathcal{E}_t\langle\!|d|\!\rangle \\
\mathcal{E}_t\langle(\mathit{rec}[tt]) :: \mathit{rec}\, f(x).\, e : (\sigma_1 \rightarrow_{tt} \sigma_2\ ,\ \mathit{trivial})\rangle & = & \mathit{rec}\, f(x).\, \mathcal{E}_t\langle d\rangle \\
\mathcal{E}_t\langle(\mathit{rec}[st]) :: \mathit{rec}\, f(x).\, e : (\sigma_1 \rightarrow_{st} \sigma_2\ ,\ \mathit{trivial})\rangle & = & \mathit{rec}\, f(x).\, \mathcal{E}_t\langle d\rangle \\
\mathcal{E}_t\langle(\mathit{rec}[ts]) :: \mathit{rec}\, f(x).\, e : (\sigma_1 \rightarrow_{ts} \sigma_2\ ,\ \mathit{trivial})\rangle & = & \mathit{rec}\, f(x).\, \mathcal{E}_t\langle\!|d|\!\rangle \\
\mathcal{E}_t\langle(\mathit{rec}[ss]) :: \mathit{rec}\, f(x).\, e : (\sigma_1 \rightarrow_{ss} \sigma_2\ ,\ \mathit{trivial})\rangle & = & \mathit{rec}\, f(x).\, \mathcal{E}_t\langle\!|d|\!\rangle \\
\mathcal{E}_t\langle(\mathit{app}[tt]) :: e_0\, e_1 : (\sigma_2\ ,\ \mathit{trivial})\rangle & = & \mathcal{E}_t\langle d_0\rangle\, \mathcal{E}_t\langle d_1\rangle \\
\mathcal{E}_t\langle(\mathit{app}[st]) :: e_0\, e_1 : (\sigma_2\ ,\ \mathit{trivial})\rangle & = & \mathcal{E}_t\langle d_0\rangle\, \mathcal{E}_t\langle\!|d_1|\!\rangle \\
\mathcal{E}_t\langle(\mathit{cnd}) :: \mathit{if}\ e_0\ \mathit{then}\ e_1\ \mathit{else}\ e_2 : (\sigma\ ,\ \mathit{trivial})\rangle & = & \mathit{if}\ \mathcal{E}_t\langle d_0\rangle\ \mathit{then}\ \mathcal{E}_t\langle d_1\rangle\ \mathit{else}\ \mathcal{E}_t\langle d_2\rangle \\
\mathcal{E}_t\langle(\mathit{let}) :: \mathit{let}\ x \Leftarrow e_0\ \mathit{in}\ e_1 : (\sigma_1\ ,\ \mathit{trivial})\rangle & = & \mathit{let}\ x \Leftarrow \mathcal{E}_t\langle d_0\rangle\ \mathit{in}\ \mathcal{E}_t\langle d_1\rangle
\end{array}
$$

Types and Assumptions:

$$
\begin{array}{rcl}
\mathcal{E}_t\langle\!|\cdot|\!\rangle & : & \mathit{Types}[\Lambda_t] \rightarrow \mathit{Types}[\Lambda_{ml}] \\
\mathcal{E}_t\langle\!|\sigma|\!\rangle & = & \widetilde{\mathcal{E}_t\langle\sigma\rangle} \\
\mathcal{E}_t\langle\iota\rangle & = & \iota \\
\mathcal{E}_t\langle\sigma_0 \rightarrow_{tt} \sigma_1\rangle & = & \mathcal{E}_t\langle\sigma_0\rangle \rightarrow \mathcal{E}_t\langle\sigma_1\rangle \\
\mathcal{E}_t\langle\sigma_0 \rightarrow_{ts} \sigma_1\rangle & = & \mathcal{E}_t\langle\sigma_0\rangle \rightarrow \mathcal{E}_t\langle\!|\sigma_1|\!\rangle \\
\mathcal{E}_t\langle\sigma_0 \rightarrow_{st} \sigma_1\rangle & = & \mathcal{E}_t\langle\!|\sigma_0|\!\rangle \rightarrow \mathcal{E}_t\langle\sigma_1\rangle \\
\mathcal{E}_t\langle\sigma_0 \rightarrow_{ss} \sigma_1\rangle & = & \mathcal{E}_t\langle\!|\sigma_0|\!\rangle \rightarrow \mathcal{E}_t\langle\!|\sigma_1|\!\rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{E}_t\langle\!|\cdot|\!\rangle & : & \mathit{Assums}[\Lambda_t] \rightarrow \mathit{Assums}[\Lambda_{ml}] \\
\mathcal{E}_t\langle\!|\Gamma, x : (\sigma\ ,\ \mathit{serious})|\!\rangle & = & \mathcal{E}_t\langle\!|\Gamma|\!\rangle, x : \mathcal{E}_t\langle\!|\sigma|\!\rangle \\
\mathcal{E}_t\langle\!|\Gamma, x : (\sigma\ ,\ \mathit{trivial})|\!\rangle & = & \mathcal{E}_t\langle\!|\Gamma|\!\rangle, x : \mathcal{E}_t\langle\sigma\rangle \\
\mathcal{E}_t\langle\!|\Gamma, f : \sigma|\!\rangle & = & \mathcal{E}_t\langle\!|\Gamma|\!\rangle, f : \mathcal{E}_t\langle\sigma\rangle
\end{array}
$$

Figure 7.5: Encoding of the CPS transformation $\mathcal{C}_t$ directed by termination properties into Λ_{ml}

Chapter 8

Conclusion

8.1 Summary of contributions

We have demonstrated how an analysis of the structure of CPS transformations can lead to useful factorizations, optimizations, and simplifications. We began in Chapter 3 by showing that the continuation-passing structure of Plotkin's CPS transformation $\mathcal{C}_n$ is not necessary for simulating call-by-name under call-by-value. Rather, the essence of the simulation is a "delaying" property which can be achieved using thunks instead. Furthermore, the continuation-passing structure of $\mathcal{C}_n$ is not intrinsic to call-by-name — it can be obtained by applying Plotkin's call-by-value CPS transformation $\mathcal{C}_v$ to the result of the thunk transformation $\mathcal{T}$.

The factorization of $\mathcal{C}_n$ by $\mathcal{C}_v$ and $\mathcal{T}$ allowed most of the operational and equational properties associated with $\mathcal{C}_n$ to be derived from the corresponding properties of $\mathcal{C}_v$ and $\mathcal{T}$. This factorization has since been used in compiling applications [22, 73]. Moreover, our experience in presenting this work to others indicates that the factorization has pedagogical value as well. The burden of understanding call-by-name and call-by-value continuation-passing style is reduced to understanding call-by-value continuation-passing style and thunks.

Chapters 4 and 5 demonstrated that the introduction of continuations can be directed by computational properties such as strictness and termination. The corresponding CPS transformations $\mathcal{C}_s$ and $\mathcal{C}_t$ have hybrid structure. The strictness-directed transformation $\mathcal{C}_s$ generalizes the call-by-name and call-by-value CPS transformations. The termination-directed transformation $\mathcal{C}_t$ generalizes the call-by-name CPS transformation and the identity transformation.

Both transformations have proved useful for language implementation. $\mathcal{C}_s$ provides a new technique for compiling call-by-name languages by reusing the backend of existing call-by-value CPS-based compilers. $\mathcal{C}_t$ enables the generation of more realistic continuation-semantics specifications for use in semantics-directed compiling.

Our work culminated in a generic framework for reasoning about continuation-passing styles. We showed how structural similarities in different styles can be exploited to obtain general results and tools. Results for specific evaluation strategies follow as corollaries of the framework. Consequently, the framework generalizes a significant amount of previous work on CPS and DS transformations.

The key to success in this presentation was Moggi's formalization of the distinction between values and computations. As mentioned previously, this distinction has often been used to intuitively justify the structure of CPS transformations. *Moggi's expression of the distinction using an extended type system and his abstraction of computational structure via monads and associated equations elegantly captures the essence of CPS.*

We believe that the presentation of CPS transformations in the meta-language also has pedagogical value. Danvy and Lawall [17, 52] and others [32, 86] have advocated explaining CPS transformations as a staged process. The meta-language framework allows a generalization of this view.

8.2 Limitations and future work

8.2.1 Addressing limitations

Our goals led to a broad study of a variety of CPS transformations as opposed to an in-depth study for a specific general-purpose language or evaluation strategy. Accordingly, the work here can be extended by

addressing particular language features in detail.

Other work on CPS considers the addition of *state* [86] and *control operators* [20, 25, 86] as exemplified by the general purpose languages Scheme [12] and SML [57]. In the conclusion of Chapter 6, we suggested several approaches for the addition of state. The general theme was that Moggi's computational meta-language has been very useful in explaining other computational effects including state and existing work [24, 61, 97, 98] indicates clear directions to follow.

The meta-language also seems well-suited for studying control operators independently of a particular evaluation strategy. This would be a unique approach since previous studies have usually been confined either to call-by-name or to call-by-value. The greatest challenge in adding state or control operators most likely lies in appropriately extending the equational theory of the meta-language. The work of Felleisen and others [26, 28, 29, 86] should provide direction here.

The current framework is limited in that we only consider simply-typed languages. This means the framework cannot be applied directly to languages with extended type systems (including Hindley-Milner polymorphism and inductive types) such as SML, Miranda, and Haskell or to untyped languages such as Scheme. Extending the type system of the meta-language to include Hindley-Milner polymorphism, inductive types, and co-inductive types seems orthogonal to issues concerning calculi and operational semantics. We expect these extensions to be relatively straightforward. It may also be possible to include the recursive types necessary to describe untyped languages.

A drawback to the presentations in Chapter 4 and 5 is that existence of strictness and totality analyses is assumed. This allowed us to illustrate our points independently of a particular analysis technique. However, actually implementing the analyses would make it possible to evaluate the effectiveness of the approaches.

We noted previously that a type inference approach seems better suited to our framework than abstract interpretation. To be as informative as abstract interpretation, type-inference-based analyses often employ *conjunctive types* [44, 45, 75, 90]. Applying our transformations to the output of such analyses would require extending the CPS transformations $\mathcal{C}_s$ and $\mathcal{C}_t$ to handle terms with conjunctive types. These extended transformations would map terms in the conjunctive-type discipline to simply-typed terms. Although we know of no examples of this strategy in the literature, a formalization seems possible based on a preliminary investigation.

8.2.2 Additional applications and directions

Compiling with monadic normal forms

CPS has long been used as an intermediate for compiling [3, 91]. Flanagan *et al.* [32] have pointed out that the properties that CPS-based compilers utilize can be obtained without using continuations. We noted in Section 6.7 the monadic normal forms of Λ_{ml} compare favorably with the alternative intermediate language ("A-normal forms") proposed by Flanagan *et al.*It seems straightforward to adapt their formal study of A-normal forms (which described an untyped call-by-value language) to Λ_{ml}. This would be the first step in justifying a generic compiler back-end based on monadic normal forms. Given the call-by-name and call-by-value Λ_{ml} encodings, such a compiler would be able to implement both call-by-name and call-by-value languages. Furthermore, the Λ_{ml} encoding $\mathcal{E}_s$ directed by strictness information would optimize the monadic-normal-form compilation of call-by-name languages in the same way that the CPS transformation $\mathcal{C}_s$ optimized CPS-based compiling.

Λ_{ml} as an ideal language for unfolding transformations

Transformation-based implementation techniques such as partial-evaluation, deforestation, driving, and supercompilation all center around function-call unfolding [46]. Unfolding a function call means replacing the call with a version of the function body where actual parameters have been substituted for formal parameters. The techniques above are usually much simpler to apply to call-by-name languages (as opposed to call-by-value languages) since call-by-name function calls are processed by the copy rule (see Section 1.1.1). Call unfolding in call-by-value is unsound in general and the total correctness of many existing transformation systems based on call-by-value languages cannot be established.

One alternative for working with call-by-value languages is to transform programs into CPS before processing. The evaluation-order independence property of CPS allows call-by-value function calls to be safely unfolded. However, CPS introduces extra complexities in *e.g.*, closure analysis [88, 89] since the number of closures is

dramatically increased [32]. Again, Λ_{ml} stands as a more suitable alternative since it provides sound call unfolding while avoiding the complexities associated with CPS.

The distinction of termination properties in the Λ_{ml} typing system is also potentially beneficial. Many useful program transformations are unsound in the presence of non-termination. Wadler has suggested that "it would be valuable to explore languages that prohibit recursion, or allow only its restricted use" [96, p. 358]. Λ_{ml} is an instance of the latter since general recursion is only allowed for computation types.

A model-theoretic presentation of the generic framework

While all of our results have been presented in an operational framework, a companion model-theoretic presentation would be beneficial. A usual argument for multiple presentations is that the number of available proof techniques is increased. Indeed, proofs of the soundness of program calculi (as presented in Chapter 2) would probably have been much smoother in a model-theoretic presentation.

For a more significant argument, Riecke [82, 83, 84] and Filinski [30] have illustrated the usefulness of model-theoretic retractions in expressing properties of translations. Riecke [84] shows how monadic-based retractions can be used to delimit the scope of effects. This could provide directions for extending Λ_{ml} with state and contol operators. Filinski's work suggests that retractions might be used to formalize a general relationship between the *strictness monad* and continuation-passing terms.

Moggi's original categorical semantics for the computational meta-language [61] is an obvious starting point for a model-theoretic presentation. Although he did not address recursion, Crole and Pitts [15] have shown how a fixpoint object can be added to Moggi's categorical model.

It is also straightforward to give a denotational semantics (*i.e.*, environment model) for Λ_{ml} using standard techniques [37, 87]. In fact, the appropriate denotational semantics follows mechanically by interpreting Λ_{ml} using the strictness monad. This interpretation identifies $[\cdot]$ with *lifting* (*i.e.*, adding a least element to a cpo) and *let* with the usual strictness check [87, p. 48].

A denotational semantics for Λ_{ml} should allow a more rigorous presentation of the ideas of Chapter 5 since Λ_{ml} could serve as the meta-language in the framework given there. A denotational semantics for Λ_{ml} would also provide another standard of correctness for Λ_{ml} encodings. In this case, the approach to CPS transformations would become "write a direct-style denotational semantics for Λ (by encoding into Λ_{ml}) and get a correct CPS transformation as a corollary".

Connecting classical and intuitionistic logic

There are interesting connections between Λ_{ml} and Griffin's association of double-negation translation with CPS transformations [36] and Murthy's intrepid display of continuation-passing styles from a logical standpoint [66, 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 Ph.D. thesis, it is striking that his "slightly modified" Kuroda translation providing for mixed call-by-value and call-by-name evaluation [66, page 159] corresponds to the mixed CPS transformation $\mathcal{C}_s$ of Chapter 4, and that his "pervasive Kolmogorov translation" [66, pp. 164-167] corresponds to Reynolds's CPS transformation in Section 7.1.4. This suggests that Λ_{ml} might be a useful framework for a general study of connections between natural-deduction presentations of classical and intuitionistic logic.

Bibliography

[1] Samson Abramsky. Abstract interpretation, logical relations, and kan extensions. *Journal of Logic and Computation*, 1(1), 1990.

[2] Samson Abramsky. The lazy lambda calculus. In David A. Turner, editor, *Research Topics in Functional Programming*, The University of Texas Year of Programming Series, chapter 4, pages 65–116. Addison-Wesley, 1990.

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

[4] Arvind, editor. *Proceedings of the Sixth ACM Conference on Functional Programming and Computer Architecture*, Lecture Notes in Computer Science, Copenhagen, Denmark, June 1993.

[5] Andrea Asperti. A categorical understanding of environment machines. *Journal of Functional Programming*, 2(1):23–59, January 1992.

[6] Henk Barendregt. *The Lambda Calculus — Its Syntax and Semantics*. North-Holland, 1984.

[7] Henk Barendregt. Lambda calculi with types. In Samson Abramsky, Dov M. Gabby, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 2*, chapter 2, pages 118–309. Oxford University Press, Oxford, 1992.

[8] Adrienne Bloss, Paul Hudak, and Jonathan Young. Code optimization for lazy evaluation. *LISP and Symbolic Computation*, 1:147–164, 1988.

[9] Hans Boehm, editor. *Proceedings of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994. ACM Press.

[10] 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.

[11] 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.

[12] William Clinger and Jonathan Rees (editors). Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[13] Charles Consel and Olivier Danvy. For a better support of static data flow. In Hughes [42], pages 496–519.

[14] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Graham [35], pages 493–501.

[15] Roy L. Crole and Andrew M. Pitts. New foundations for fixpoint computations: FIX-hyperdoctrines and the FIX-logic. *Information and Computation*, 98:171–210, 1992.

[16] Pierre-Louis Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*, volume 1 of *Research Notes in Theoretical Computer Science*. Pitman, 1986.

[17] Olivier Danvy. Three steps for the CPS transformation. Technical Report CIS-92-2, Kansas State University, Manhattan, Kansas, December 1991.

[18] 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.

[19] Olivier Danvy. Back to direct style (extended version). Technical Report CIS-92-1, Kansas State University, Manhattan, Kansas, 1993.

[20] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. In Wand [103], pages 361–391.

[21] Olivier Danvy and John Hatcliff. Thunks (continued). In *Proceedings of the Workshop on Static Analysis WSA'92*, volume 81-82 of *Bigre Journal*, pages 3–11, Bordeaux, France, September 1992. IRISA, Rennes, France. Extended version available as Technical Report CIS-92-28, Kansas State University.

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

[23] Olivier Danvy and John Hatcliff. On the transformation between direct and continuation semantics. In *Proceedings of the 9th Conference on Mathematical Foundations of Programming Semantics*, number 802 in Lecture Notes in Computer Science, New Orleans, Louisiana, April 1993.

[24] Olivier Danvy, Jürgen Koslowski, and Karoline Malmkjær. Compiling monads. Technical Report CIS-92-3, Kansas State University, Manhattan, Kansas, December 1991.

[25] Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In Clinger [11], pages 299–310.

[26] Matthias Felleisen. *The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* PhD thesis, Department of Computer Science, Indiana University, Bloomington, Indiana, August 1987.

[27] 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.

[28] Matthias Felleisen, Daniel P. Friedman, Eugene Kohlbecker, and Bruce Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.

[29] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992.

[30] Andrzej Filinski. Representing monads. In Boehm [9].

[31] Michael J. Fischer. Lambda-calculus schemata. In Talcott [94]. An earlier version appeared in the *Proceedings of the ACM Conference on Proving Assertions about Programs*, SIGPLAN Notices, Vol. 7, No. 1, January 1972.

[32] 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.

[33] Pascal Fradet and Daniel Le Métayer. Compilation of functional languages by program transformation. *ACM Transactions on Programming Languages and Systems*, 13:21–51, 1991.

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

[35] Susan L. Graham, editor. *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, Charleston, South Carolina, January 1993. ACM Press.

[36] Timothy G. Griffin. A formulae-as-types notion of control. In Frances E. Allen, editor, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, California, January 1990. ACM Press.

[37] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* MIT Press, 1992.

[38] Bob Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. In Talcott [94].

[39] John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. Technical Report CIS-93-15, Kansas State University, Manhattan, Kansas, August 1993.

[40] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm [9].

[41] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*, volume 1 of *London Mathematical Society Student Texts.* Cambridge University Press, 1986.

[42] John Hughes, editor. *Proceedings of the Fifth ACM Conference on Functional Programming and Computer Architecture*, number 523 in Lecture Notes in Computer Science, Cambridge, Massachusetts, August 1991.

[43] P. Z. Ingerman. Thunks, a way of compiling procedure statements with some comments on procedure declarations. *Communications of the ACM*, 4(1):55–58, 1961.

[44] Thomas P. Jensen. Strictness analysis in logical form. In Hughes [42].

[45] Thomas P. Jensen. *Abstract Interpretation in Logical Form.* PhD thesis, Imperial College, London, England, 1992.

[46] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, 1993.

[47] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In Michael J. O'Donnell and Stuart Feldman, editors, *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 281–292, Austin, Texas, January 1989.

[48] Jan W. Klop. *Combinatory Reduction Systems.* Mathematical Centre Tracts 127. Mathematisch Centrum, Amsterdam, 1980.

[49] John Launchbury. A natural semantics for lazy evaluation. In Graham [35], pages 144–154.

[50] 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*, Copenhagen, Denmark, June 1993. ACM Press.

[51] Julia L. Lawall. *Continuation Introduction and Elimination in Higher-order Programming Languages.* PhD thesis, Computer Science Department, Indiana University, Bloomington, Indiana, USA, July 1994.

[52] Julia L. Lawall and Olivier Danvy. Separating stages in the continuation-passing style transformation. In Graham [35], pages 124–136.

[53] Peter Lee. *Realistic Compiler Generation.* MIT Press, 1989.

[54] Peter Lee and Uwe Pleban. On the use of LISP in implementing denotational semantics. In William L. Scherlis and John H. Williams, editors, *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 233–248, Cambridge, Massachusetts, August 1986.

[55] 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.

[56] Robert E. Milne and Christopher Strachey. *A Theory of Programming Language Semantics.* Chapman and Hall, London, and John Wiley, New York, 1976.

[57] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[58] Eugenio Moggi. Computational lambda-calculus and monads. Report ECS-LFCS-88-66, University of Edinburgh, Edinburgh, Scotland, October 1988.

[59] 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.

[60] 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.

[61] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

[62] Margaret Montenyohl and Mitchell Wand. Correct flow analysis in continuation semantics. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 204–218, San Diego, California, January 1988.

[63] Margaret Montenyohl and Mitchell Wand. Correctness of static flow analysis in continuation semantics. *Science of Computer Programming*, 16(1):1–18, 1991.

[64] Margaret Montenyohl and Mitchell Wand. Incorporating static analysis in a combinator-based compiler. *Information and Computation*, 82:151–184, 1991.

[65] Luc Moreau and Daniel Ribbens. Sound rules for parallel evaluation of a functional language with callcc. In Arvind [4], pages 125–135.

[66] Chetan R. Murthy. *Extracting Constructive Content from Classical Proofs*. PhD thesis, Department of Computer Science, Cornell University, 1990.

[67] 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.

[68] Alan Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, University of Edinburgh, Edinburgh, Scotland, 1981.

[69] Peter Naur. Revised report on the algorithmic language Algol 60. *Communications of the ACM*, 6(1):1–17, 1962.

[70] Flemming Nielson. Two-level semantics and abstract interpretation. *Theoretical Computer Science*, 69(2):117–242, 1989.

[71] Flemming Nielson and Hanne Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59–133, January 1988. Special issue on ESOP'86, the First European Symposium on Programming, Saarbrücken, March 17-19, 1986.

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

[73] Chris Okasaki, Peter Lee, and David Tarditi. Call-by-need and continuation-passing style. In Talcott [94].

[74] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International Series in Computer Science. Prentice-Hall International, 1987.

[75] Benjamin C. Pierce. *Programming with Intersection Types and Bounded Polymorphism*. PhD thesis, Carnegie Mellon University, Pittsburgh, Pennsylvania, December 1991.

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

[77] Gordon D. Plotkin. Personal communication, New Orleans, Louisiana, March 1993.

[78] B. Randell and L. J. Russell. *Algol 60 Implementation.* Academic Press, New York, 1964.

[79] Eric Raymond (editor). *The New Hacker's Dictionary.* The MIT Press, 1992.

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

[81] 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.

[82] Jon G. Riecke. Fully abstract translations between functional languages. In Robert (Corky) Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 245–254, Orlando, Florida, January 1991. ACM Press.

[83] Jon G. Riecke. *The Logic and Expressibility of Simply-typed Call-by-value and Lazy Languages.* PhD thesis, MIT, Cambridge, Massachusetts, August 1991.

[84] Jon G. Riecke. Delimiting the scope of effects. In Arvind [4].

[85] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In Clinger [11], pages 288–298.

[86] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In Talcott [94].

[87] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Inc., 1986.

[88] Peter Sestoft. Replacing function parameters by global variables. In Stoy [93], pages 39–53.

[89] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* PhD thesis, CMU, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

[90] Kirsten L. Solberg. Strictness and totality analysis. Technical Report PB-474, Computer Science Department, Aarhus University, 1994.

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

[92] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, 1977.

[93] Joseph E. Stoy, editor. *Proceedings of the Fourth International Conference on Functional Programming and Computer Architecture*, London, England, September 1989. ACM Press.

[94] Carolyn L. Talcott, editor. *Special issue on continuations*, LISP and Symbolic Computation, Vol. 6, Nos. 3/4. Kluwer Academic Publishers, 1993.

[95] Robert D. Tennent. *Semantics of Programming Languages.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1990.

[96] Philip Wadler. Theorems for free! In Stoy [93], pages 347–359.

[97] Philip Wadler. Comprehending monads. In Wand [103], pages 461–493.

[98] 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.

[99] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27(1):164–180, January 1980.

[100] Mitchell Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, 1982.

[101] Mitchell Wand. Semantics-directed machine architecture. In Richard DeMillo, editor, *Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 234–241, January 1982.

[102] Mitchell Wand. Correctness of procedure representations in higher-order assembly language. In Steve 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.

[103] 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.