Metatheorems about Convertibility in Typed Lambda Calculi: Applications to CPS transform and "Free Theorems"

by

Jakov Kučan

B.S.E., University of Pennsylvania (1991)
M.A., University of Pennsylvania (1991)

Submitted to the Department of Mathematics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1997

Author ....

Department of Mathematics
October 10, 1996

Certified by ....

Albert R. Meyer
Hitachi America Professor of Engineering
Thesis Supervisor

Accepted by ....

Hung Cheng
Chairman, Applied Mathematics Committee

Metatheorems about Convertibility in Typed Lambda Calculi: Applications to CPS transform and "Free Theorems"

by

Jakov Kučan

Submitted to the Department of Mathematics
on October 10, 1996, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

In this thesis we present two applications of using types of a typed lambda calculus to derive equation schemas, instances of which are provable by convertibility.

Part 1: Retraction Approach to CPS transform: We study the continuation passing style (CPS) transform and its generalization, the *computational transform*, in which the notion of computation is generalized from continuation passing to an arbitrary one. To establish a relation between *direct style* and *continuation passing style* interpretation of sequential call-by-value programs, we prove the Retraction Theorem which says that a lambda term can be recovered from its continuationized form via a λ-definable retraction. The Retraction Theorem is proved in the logic of computational lambda calculus for the simply typable terms.

Part 2: Another look at "Theorems for Free!": We reexamine Wadler's "free theorems" — theorems about polymorphic functions derivable from the type of the function — in an extension of Mairson's syntactic treatment of logical relations. Logical relations are defined as formulas in second (and higher) order logic over untyped lambda terms, from which we can instantiate equations involving a given term. As a consequence we can formulate a free theorem as an equation schema, each instance of which is provable by convertibility, given typing conditions on the terms involved are met. In our framework datatypes are represented as free algebras generated by constants, and we define sufficient conditions which a representation of a datatype must satisfy to preserve free theorems. As a special case we prove examples of free theorems, as equations by convertibility, using Böhm-Berarducci representation of datatypes.

Thesis Supervisor: Albert R. Meyer
Title: Hitachi America Professor of Engineering

Thanks

I would like to thank to all who, in one way or another, helped and inspired me in this work: to my advisor, Albert Meyer, for bearing with me over the years and teaching me how to say what I think (I'm still learning); to Lotus Development for the "real world" experience; to Harry Mairson for saving me from bankruptcy in the last moment; to fellow students and officemates.

Lastly, thanks to my wife Jadranka, just for being a wife of a graduate student.

Contents

Chapter 1

Retraction Approach to CPS Transform

Abstract: We study the continuation passing style (CPS) transform and its generalization, the *computational transform*, in which the notion of computation is generalized from continuation passing to an arbitrary one. To establish a relation between *direct style* and *continuation passing style* interpretation of sequential call-by-value programs, we prove the Retraction Theorem which says that a lambda term can be recovered from its continuationized form via a λ-definable retraction. The Retraction Theorem is proved in the logic of computational lambda calculus for the simply typable terms.

1.1 Introduction

The notions of a *continuation* and a *continuation passing style* (CPS) transform have been introduced by a number of authors (see [Rey93] for a historical overview). The main motivation for the independent developments of these concepts seemed to be twofold: explaining the behavior of imperative features in functional languages, and compilation of programs with higher order procedures. Further research led to development of CPS denotational semantics [SW74] (see also [Sto77]), and later categorical semantics of computations [Mog89], as well as compilers based on the CPS transform [Ste78] (see also [App92]). In both kinds of applications one of the central goals of the research has been to establish a relationship between original terms and their images under the transform.

Motivation: Plotkin [Plo75] has shown that the CPS transform[1] preserves equalities between lambda terms provable in call-by-value calculus λ_v.

Theorem 1.1.1 (Plotkin). For any two lambda terms M and N,

$$\lambda_v \vdash M = N \quad \text{implies} \quad \lambda_n \vdash \overline{M} = \overline{N},$$

[1]Unless otherwise stated, we are referring to a call-by-value variant of the CPS transform.

where $\overline{M}$ and $\overline{N}$ denote the CPS transforms of M and N, and λ_v and λ_n are call-by-value and call-by-name equational logics. [2]

He has also shown that the converse of this theorem is not true by presenting several examples of equations which ought to hold in a sequential call-by-value language, but are not provable in λ_v. Sabry and Felleisen [SF92] strengthened Plotkin's result by extending the logic of λ_v to what turns out to be Moggi's computational lambda calculus λ_{cp}[3], to prove the equational correspondence result.

Theorem 1.1.2 (Sabry-Felleisen). For any two lambda terms M and N,

$$\lambda_{cp} \vdash M = N \quad \text{iff} \quad \lambda_n \vdash \overline{M} = \overline{N}.$$

Sabry-Felleisen's result says that the CPS transform is a function mapping lambda terms to lambda terms that preserves λ_{cp}-equality, as well as distinctions. To state the problem we are trying to solve precisely, we observe that left-to-right implication of the theorem says that the CPS transform defines a function, call it $\mathcal{B}$, mapping λ_{cp}-equivalence classes to λ_n-equivalence classes of terms. In fact Sabry and Felleisen's proof of the above theorem is based on a construction of such a function. The right-to-left implication says that $\mathcal{B}$ is injective and therefore has a left inverse $\mathcal{B}^{-1}$. To strengthen Sabry-Felleisen's result we ask whether $\mathcal{B}$ or $\mathcal{B}^{-1}$ are lambda definable. More precisely we ask the following questions:

- Is there as lambda term P, such that

$$\lambda_n \vdash (P\ M) = \overline{M}?$$

- Is there a lambda term R, such that

$$\lambda_{cp} \vdash (R\ \overline{M}) = M?$$

The answer to the first question, as shown by Meyer and Riecke [MR], is "no". The Retraction Theorem, which is the main result of this work, answers the second question affirmatively. However, instead of giving a single lambda term R that defines the inverse of the CPS transform, we will show there is a type indexed family of terms, R_σ, such that for every closed term M of type σ, $\lambda_{cp} \vdash M = (R_\sigma\ \overline{M})$.

A version of the Retraction Theorem was first presented by Meyer and Wand [MW85] in the following form.

Theorem 1.1.3 (Meyer-Wand). For any simple type σ, there is a term R_σ, such that for a closed term M, of type σ

$$\lambda_n \vdash M = (R_\sigma\ \overline{M}).$$

[2] In Plotkin's work, λ_v denotes β_v-convertibility, while λ_n denotes β-convertibility. In this paper, we assume λ_v and λ_v also contain axioms (η_v) and (η) respectively.

[3] We use λ_{cp} to denote the pure segment of Moggi's calculus, while λ_c denotes the complete calculus.

While certainly true, as the authors have pointed out to us, this theorem is somewhat misleading. The variant of the CPS transform is the *call-by-value* transform which makes the computational structure of the term M explicit, but under assumption that M is to be evaluated under a call-by-value evaluation strategy. The transform does not tell us anything about how M would behave under a call-by-name evaluation. Therefore, we expect the equality $M = (R_\sigma\ \overline{M})$ to tell us something about the term M and its CPS form in a context of call-by-value evaluation. In Meyer-Wand Theorem, this equality is an equality in λ_n, which is a call-by-name logic and is not sound for call-by-value reasoning. This limits the applications of the theorem to setting in which call-by-name and call-by-value coincide (e.g. for pure simply typed terms). To apply the theorem in a wider range of settings, we need to weaken the logic in which the conclusion of the theorem is proved.

The main contribution of this paper is a stronger and, by above considerations, correct version of the Retraction Theorem where provability in λ_n is replaced with provability in Moggi's λ_{cp}. Note that, while stronger than λ_v, the logic of λ_{cp} is still sound for call-by-value reasoning, that is, a closed term is equivalent to a value (a variable or a lambda abstraction) under λ_{cp} if and only if it is equivalent to a value under λ_v [Mog88]. Our version of the Retraction Theorem reads as follows.

Theorem 1.1.4 (Retraction). For any simple type σ, there is a term R_σ, such that for a closed term M, of type σ
$$\lambda_{cp} \vdash M = (R_\sigma\ \overline{M}).$$

Moreover, we will show that this is the strongest possible version of the Retraction Theorem, meaning λ_{cp} is the weakest logic in which the conclusion of the theorem holds.

Lambda calculi: Before we go any further, to make our exposition easier to read we summarize some of the key concepts and explain the notation we use. A more detailed treatment is given in Section 1.2. The phrase *lambda calculus* refers to a set of terms — the language of the calculus — and a set of provable equations — the logic (or the theory) of the calculus.[4] We will consider several different calculi whose languages will consist of terms closed under lambda abstraction and application, with possible extensions specific to each calculus. The logic of a calculus is intended to model specific computational behavior of the terms of the calculus.

We view the CPS transform, and its generalization, the computational transform, as interpretations of a sequential call-by-value functional programming language with possible "impure" features (continuations, mutable store, exceptions, nondeterminism, etc.). For the programming language we choose Moggi's λ_{cp} calculus for the following reasons: the logic of λ_{cp} is sound for call-by-value reasoning, and it is complete with respect to a class of models in which most commonly used "impure" features can be represented. We concentrate on the the pure simply typed terms first, and later discuss possible extensions that would make λ_{cp} a more realistic programming language.

[4] We also consider a type system to be a part of the calculus, but we postpone the discussion of type systems to Section 1.2.

The language of λ_{cp} consist of terms formed by abstraction, application and `let`-expressions. However, the `let`-expressions can be treated as syntactic sugar. The logic of λ_{cp} contains (β_v) and (η_v), and additional axioms that allow proving equations like $((\lambda x.x)\ M) = M$. The complete definition of λ_{cp} is given in section 1.2.3.

For the codomains of the transforms, we define calculi λ_n, λ_{ml}, and λ_c. Calculus λ_n is a call-by-name calculus, the language of which consists *only* of terms formed by application and abstraction. The logic of λ_n is the $\beta\eta$-theory. We use λ_n as the codomain of the CPS transform. Calculus λ_{ml} is what Moggi calls a "monadic metalanguage". We define it as an extension of λ_n with *monadic* constructs $[M]_T$ and $(\mathtt{let}_T\ x{\Leftarrow}M\ \mathtt{in}\ N)$. The axioms of λ_{ml} extend those of λ_n with axioms corresponding to monad laws (see sections 1.2.2 and 1.2.4). λ_{ml} is the codomain of the *computational transform.* Finally we define calculus λ_c as an extensions of λ_{cp} with constructs $[M]$ and $\mu(M)$ satisfying axioms $[\mu(V)] = V$ and $\mu([M]) = M$, where M is an arbitrary term, and V is a value (a variable, an abstraction, or a term of the form $[M]$). λ_c will serve as a codomain of the *modified computational transform.*

In our discussion, we also refer to the calculus λ_v. The language if λ_v contains terms formed by application and abstraction, and its logic is defined by (β_v) and (η_v) axioms.

We say a calculus λ' is an *extension* of a calculus λ if the theory of λ is a subset of the theory of λ'. In such cases we write $\lambda \subseteq \lambda'$. Since we assume reflexivity is an axiom in every lambda theory, this implies that the language of λ is contained in the language of λ' whenever $\lambda \subseteq \lambda'$.

The five calculi described above are related as follows:

$$\lambda_v \subseteq \lambda_{cp} \subseteq \lambda_n \subseteq \lambda_{ml} \qquad \text{and} \qquad \lambda_{cp} \subseteq \lambda_c\,.$$

Transforms: In addition to the CPS transform, we study its generalization, the *computational transform* mapping λ_{cp} to λ_{ml}. As we explain below, it is not possible to generalize the Retraction Theorem to the computational transform. Thus, we will define the *modified computational transform* mapping λ_{cp} to λ_c. We also study the *modified CPS transform* that can be viewed as a special case of the modified computational transform. If M is a λ_{cp} term, we write $\overline{M}$ to denote the CPS transform of M, $\overline{\overline{M}}$ to denote the computational transform of M, $\overline{\overline{M}}^\sharp$ to denote the modified computational transform of M, and $\overline{M}^\sharp$ for the modified CPS transform of M. Figure 1-1 gives an intuitive picture of the transforms we study. We use symbols $\mathcal{T}_{ml}$, $\mathcal{T}_{ml\sharp}$, $\mathcal{T}_{cps}$, and $\mathcal{T}_{cps\sharp}$ to denote the computational, modified computational, CPS, and modified CPS transform respectively. Transforms $\mathcal{T}_\sharp$ and $\mathcal{T}_\sharp^K$ are the auxiliary transforms modifying $\mathcal{T}_{ml}$ and $\mathcal{T}_{cps}$.

The CPS transform can be viewed as a function mapping lambda terms to lambda terms. However, we believe that this view is overly simplistic and undermines essential properties of the CPS transform. Even though, languages of λ_v, λ_{cp} and λ_n coincide, the logics of these calculi are very different. Looking at the CPS transform as a function mapping a lambda terms to lambda terms, while forgetting associated logic of the calculus, can be confusing and lead to misleading results. To get things straight we define the CPS transform as a function form one *calculus* to another. In other words, we make a distinction between languages of different calculi even though they may contain exactly the same terms. For example, if a term M is considered in a context of λ_{cp} calculus it will be regarded as a λ_{cp}

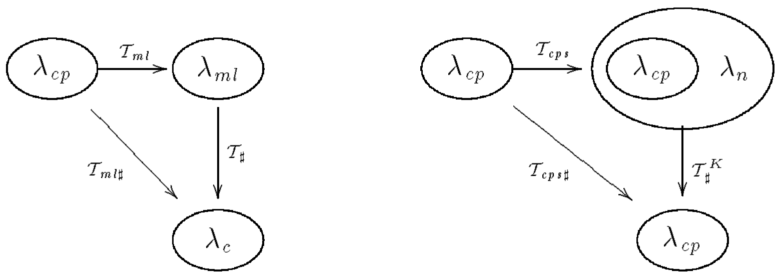

Figure 1-1: Summary of the transforms.

term, but the same term in the context of λ_n calculus will be regarded as a λ_n term. The problem of defining the correct domain and codomain of the CPS transform does not have a unique solution. For instance, Plotkin defined the CPS transform mapping λ_v into λ_n. The precise question we are trying to answer in this work is whether there is a λ_{cp} term R_σ such that for each λ_{cp} term of type σ, the equality $M = (R_\sigma\ \overline{M})$ is provable in λ_{cp}. This formulation requires that we view the CPS transform as a mapping from λ_{cp} to λ_{cp}. We now justify it is correct to do so.

One of the main features of the CPS transform is that it makes the computational structure of a term explicit in terms of continuations. More precisely, the control of how to evaluate the rest of the program is encoded in a continuation which is passed to each function as an explicit argument. In particular we consider the call-by-value variant of the CPS transform which encodes the call-by-value order of evaluation. The logic of λ_{cp} is sound for call-by-value reasoning and, in fact, the CPS transform preserves λ_{cp}-equality, so λ_{cp} is a correct domain for the CPS transform. To show that λ_{cp} is a correct codomain of the transform, consider Plotkin's indifference result:

Theorem 1.1.5 (Plotkin). For any two lambda terms M and N,

$$\lambda_v \vdash \overline{M} = \overline{N} \quad \text{iff} \quad \lambda_n \vdash \overline{M} = \overline{N}.$$

This theorem, as shown in [SW96] can be extended to λ_{cp}.

Theorem 1.1.6 (Sabry-Wadler). For any two lambda terms M and N,

$$\lambda_{cp} \vdash \overline{M} = \overline{N} \quad \text{iff} \quad \lambda_n \vdash \overline{M} = \overline{N}.$$

By these results, λ_n restricted to the image of the CPS transform is a conservative extension of λ_{cp} restricted to the same set of terms. Therefore on terms in the image of the CPS transform logics of λ_n and λ_{cp} coincide. Combined with Theorem 1.1.2, we conclude that

$$\lambda_{cp} \vdash M = N \quad \text{iff} \quad \lambda_{cp} \vdash \overline{M} = \overline{N}.$$

Hence, without loss of generality, the CPS transform can be viewed as a transform from

λ_{cp} into λ_{cp}.

The CPS transform gives purely functional interpretation of the control of the execution of a program. Similarly, one can give a functional interpretation of a program that manipulates mutable store by passing the state of the store explicitly in every function call, yielding a *state passing style* transform. As pointed out by Moggi [Mog88], many other *computational effects*, such as nondeterminism, divergence, interactive I/O, etc., can be treated in a similar way. Abstracting away from particular properties of each computational effect we arrive at what we call the *computational transform*. However, the codomain of the computational transform is λ_{ml} calculus, which makes it impossible to obtain a desired generalization of the Retractions Theorem. For this reason we will modify the computation transform to obtain a transform from λ_{cp} to λ_c, for which we prove the Retraction Theorem in the logic of λ_c.

Results: We have already stated the main result of this work, the Retraction Theorem for the CPS transform (Theorem 1.1.4). We also prove versions of the Retraction Theorem for the modified computational transform and the modified CPS transform. We now discuss these results.

Since our version of the Retraction Theorem for the CPS transform is provable in the logic of λ_{cp}, it is valid in all models of Moggi's computational lambda calculus, which can model sequential programming languages that include a variety of common features like mutable store, exceptions, nondeterminism, etc.

We also claim that our version of the Retraction Theorem is the strongest possible, in the sense that it cannot be proven in a logic, extending λ_v, that is weaker than λ_{cp}. More formally,

Proposition 1.1.7. Let λ be an extension of λ_v in which we can prove the Retraction Theorem,

$$\lambda \vdash M = (R_\sigma\ \overline{M}). \tag{1.1}$$

Then, λ is an extension of λ_{cp}.

Proof: Let σ be a simple type and let M and N be closed terms of type σ such that $\lambda_{cp} \vdash M = N$. By Sabry-Felleisen's equational correspondence we have that $\lambda_n \vdash \overline{M} = \overline{N}$, and by Plotkin's indifference result $\lambda_v \vdash \overline{M} = \overline{N}$, and therefore $\lambda \vdash \overline{M} = \overline{N}$. This, by equation (1.1) and congruence, implies that $\lambda \vdash M = N$, so $\lambda_{cp} \subseteq \lambda$. ■

Viewing the computational transform as a generalization of the CPS transform, a natural question arises whether the Retraction Theorem generalizes to the computational transform $\mathcal{T}_{ml}$. In order to answer this question, that is, to find a definable inverse of $\mathcal{T}_{ml}$, we need to make clear in which calculus the inverse should be definable. As explained in Section 1.3.1, $\mathcal{T}_{ml}$ is a transform from λ_{cp} into λ_{ml}. We want to formulate the retraction theorem in a logic that is sound for call-by-value reasoning. The analysis presented for the CPS transform, in which we show the codomain of the CPS transform is λ_{cp}, does not generalize, since the computational transform, $\overline{\overline{M}}$, of a term M is not a term in the language of λ_{cp}. On the other hand, while it is possible to formulate the Retraction Theorem in the logic of λ_{ml} (since $\lambda_{cp} \subseteq \lambda_{ml}$) to obtain the version below, the logic of λ_{ml} is a call-by-name logic which is not sound for call-by-value reasoning. In fact, the Meyer-Wand formulation of the Retraction Theorem is a special case of the following theorem.

Theorem 1.1.8. For any simple type σ there is a term R_σ, such that for every closed term M of type σ,

$$\lambda_{ml} \vdash M = (R_\sigma\ \overline{\overline{M}}).$$

To get a more meaningful generalization of the Retraction Theorem we use the modified computational transform $\mathcal{T}_{ml\sharp}$. The modified transform maps λ_{cp} to λ_c. We prove the Retraction Theorem for the modified computational transform in the logic of λ_c, which is conservative over λ_{cp}, and is sound for call-by-value reasoning. Namely we show:

Theorem 1.1.9. For any simple type σ there is a λ_c term R_σ, such that for every closed term M of type σ,

$$\lambda_c \vdash M = (R_\sigma\ \overline{\overline{M}}^\sharp).$$

Even though the Retraction Theorem for the CPS transform will not follow as a corollary of the above theorem, we feel that we benefit from studying the modified transform by abstracting away from particulars of the CPS transform in the more general setting.

Our results are pictured in Figure 1-2. Arrows R_σ and R_σ^K denote the definable retractions for the modified computational transform, $\mathcal{T}_{ml\sharp}$, and the CPS transform, $\mathcal{T}_{cps}$, respectively. Note that R_σ^K is also a retraction of the modified CPS transform $\mathcal{T}_{cps\sharp}$. The first diagram presents a general case, and the second diagram can be viewed as a special case for the CPS transform. One could also consider the *state passing style* transform, as another special case.

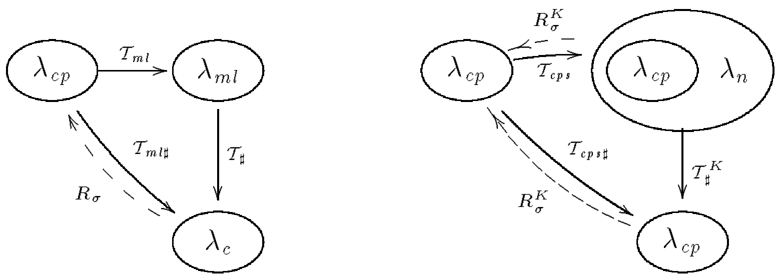

Figure 1-2: Summary of the results.

Outline of the proof: Our proof method is a generalization of the *evaluation semantics* developed by McAllester et al. [MKO95], which in turn can be viewed as defining a Mitchell's model of type inference [Mit88]. The method also has roots in Girard's method of *Candidats de Reducibilité* [Gir71, Gir72]. However, it must be pointed out that fore mentioned proof methods are tuned for proving properties of lambda terms in the $\beta\eta$-logic. Since our results hold in the weaker logic of λ_{cp} (and λ_c), we take a somewhat less structured approach.

We define define interpretations of terms and types and show that the two interpretations are related. More precisely, the meaning of a term is defined to be its CPS transform (or modified CT), while types are interpreted as sets of certain terms in the image of the transform which represent *effect-free* computations. The meanings of terms and types are related by an ε-relation, the purpose of which is to separate the meaning of a term into a part representing its effect-free computation and a part containing its computational effects (if any). The meaning of a term is said to be ε-related to a type set, interpreting the corresponding type, if its effect-free part is an element of the type set.

Remark. There are several flavors of the CPS transform. With respect to evaluation strategy we have call-by-value and call-by-name variants of the transform, and with respect to evaluation order, we have left-to-right and right-to-left variants. In this paper we focus on call-by-value CPS transform, and assume left-to-right evaluation order. Assuming the proper adjustments, viz. axioms of λ_c need to be modified slightly, all of the results apply when right-to-left evaluation order is assumed. We could also consider call-by-name evaluation strategy to obtain analogous results.

1.2 Setup and Definitions

In this section we will review the definitions of the elementary concepts used in this paper.

1.2.1 Lambda Calculi

From a computer scientist's point of view, the simply typed lambda calculus represents the essence of sequential functional programming. In simply typed lambda calculus functions are represented by lambda abstractions, and evaluation is represented by a reduction relation. In programming languages, there are different strategies of evaluating function application (function can be evaluated before of after its argument, the argument may or may not be evaluated before it is passed to a function, etc.). In lambda calculus formulation, these strategies are represented by different variants of reduction relation (e.g. β, β_v, etc.), thus giving us different lambda calculi. Instead of reduction relation, in this work, we study its symmetric closure, i.e. an equivalence relation on lambda terms. Therefore, for our purposes, a lambda calculus is defined by the following parts:

- the *language* of the calculus defined by a grammar for forming untyped terms,
- the *equational logic* of the calculus given by a set of axioms, and
- the *type system* consisting of set of *types* and a deductive system for proving *typing sequents.*

Different lambda calculi will be denoted by the symbol λ with subscripts. We will often use the term calculus and the symbol λ to refer to any part of the calculus.

Language: For each calculus we give a grammar defining the set of terms of the calculus. In call-by-value calculi (for example, λ_{cp} and λ_c), we define a distinguished subset of the terms called *values*. Typically, variables and lambda abstractions will be considered values. Throughout this paper, unless indicated otherwise, letters M, N, P and Q are assumed to range over arbitrary terms, while letters U, V and W range over values.

Terms are defined over an infinite set of variables (we use letters x, y and z to range over variables). A number of constructs are said to bind a variable within a given scope. In particular, in term $\lambda x.M$, λx binds variable x in M. Similarly, in term ($\texttt{let}\ x\texttt{=}M\ \texttt{in}\ N$), x is bound in N, but not in M, and likewise for ($\texttt{let}_T\ x\Leftarrow M\ \texttt{in}\ N$). If a variable x, occurring in M, is not bound we say it is free in M, denoted by $x \in \mathrm{FV}(M)$. We consider two terms the same if they are identical up to the names of bound variables.

A substitution is a function mapping variables to terms. In call-by-value calculi we only consider substitutions mapping variables to values. A substitution ρ is often applied to a term M, yielding the term ρM which is equal to M with $\rho(x)$ substituted in M for all free occurrences of a variable x. We assume the substitution is done in the standard way to avoid incorrect capture of free variables in the term being substituted (renaming bound variables if necessary).

If ρ is a substitution we write $\rho\{V/x\}$ for the substitution that coincides with ρ at all variables, except possibly x where it equals V. We will also write $M\{V/x\}$ for the term obtained by substituting V for all free occurrences of x in M.

For notational convenience we will also use *contexts*. A context can be viewed as a term with a singe free occurrence of a special variable, called a *hole*, denoted by $\langle\!\![\cdot]\!\!\rangle$. The hole in a context $C\langle\!\![\cdot]\!\!\rangle$ can be *replaced* by a term M to obtain the term $C\langle\!\![M]\!\!\rangle$. The difference between substituting a term for a variable, and replacing a term for the hole is that in the latter case, no renaming of bound variables is done to avoid capture of free variables in the term being substituted. Consequently, free variables in M can become bound in $C\langle\!\![M]\!\!\rangle$.

Note that while we define the language without any reference to types, in our work, we assume all terms as typable in the appropriate type inference system.

Type inference system: To define a type system of a calculus we first give a grammar for forming *types*. The type systems we consider consist of simple types with appropriate extensions. Thus types will generally include a base type $\imath$ and functional types $\sigma \rightarrow \tau$, where both σ and τ are types. We use letters σ and τ to range over types.

A *type assertion* is an expression of the form $M\colon\sigma$, where M is a term and σ is a type. A *typing hypothesis*, Γ, is a set of type assertions of the form $x\colon\sigma$, such that no variable occurs in two assertions in Γ. We write $\Gamma, x\colon\sigma$ for the typing hypothesis $\Gamma \cup \{x\colon\sigma\}$, and assume that x does not occur in Γ. Finally, a *typing sequent* is an expression of the form $\Gamma \triangleright M\colon\sigma$, where Γ is a typing hypothesis, M is a term, and σ is a type.

For each calculus we will give a set of axioms and rules for deriving typing sequents. If not specified otherwise, we only consider derivable typing sequents. We will often subscript the symbol $\triangleright$ to disambiguate in which system, the sequent is derivable.

Equational logic: There are two ways of defining the equational logic of the calculus. In first, one considers *equations in context*, i.e. sequents of the form $\Gamma \triangleright M{=}N : \sigma$, where Γ is a typing hypothesis, M and N are terms, and σ is a type. Whenever we write $\Gamma \triangleright M{=}N : \sigma$, we assume that typing sequents $\Gamma \triangleright M{:}\,\sigma$ and $\Gamma \triangleright N{:}\,\sigma$ are derivable in the type system of the calculus.

On the other hand, one can consider *untyped equations*, i.e. formulas of the form $M = N$, where M and N are untyped terms.

We point out that these two approaches give different theories, since the untyped theory proves equations between terms that may not be typable. If an equation in context $\Gamma \triangleright M{=}N : \sigma$ is provable, then the corresponding untyped equation, $M = N$, is provable as well. Conversely, if M and N are terms such that $\Gamma \triangleright M{:}\,\sigma$ and $\Gamma \triangleright N{:}\,\sigma$, and $M = N$ is an equation of the untyped theory, the equation $\Gamma \triangleright M{=}N : \sigma$ is also provable provided the Church-Rosser property[5] and subject reduction property[6] hold for the calculus. For the calculi we study in this work, both properties hold, so the untyped theory does not prove any new equations between terms typable with the same type. Moreover, we only prove equations between terms of the same type, so we can omit the type information from the equations, keeping in mind that all terms in the equations are appropriately typable.

A *lambda congruence* is a set of equations derivable using axioms and rules in Table 1.1. Note that in all the rules and axioms, the terms are assumed to range over the language of a particular calculus.

$M = M$	(refl)	$\dfrac{M = P, \quad P = N}{M = N}$	(trans)
$\dfrac{M = N}{N = M}$	(sym)	$\dfrac{M = N}{C\langle\!\langle M\rangle\!\rangle = C\langle\!\langle N\rangle\!\rangle}$	(congr)

Table 1.1: Axioms and rules of lambda congruence.

With every calculus we associate a set of axioms (in addition to those of lambda congruence) to define the equational logic of the calculus. If an equation is provable form the set of axioms λ, using the axioms and rules of lambda congruence, we write

$$\lambda \vdash M = N,$$

or just $M = N$ when it is understood that the equation is provable.

Notational conventions: When writing terms and types, we use parentheses freely to make expressions easier to read. In doing so we assume application of terms associates to the left, so that $((M\ N)\ P)$ can be written as $(M\ N\ P)$, and the function space type constructor, $\rightarrow$, associates to the right, so that $\sigma{\rightarrow}(\tau{\rightarrow}\upsilon)$ can be written $\sigma{\rightarrow}\tau{\rightarrow}\upsilon$. We may also write $M(N)$ for $(M\ N)$, etc.

[5] A reduction relation $\twoheadrightarrow^*$ is said to be Church-Rosser if whenever $M \twoheadrightarrow^* M_1$ and $M \twoheadrightarrow^* M_2$, there is a term M_3 such that $M_1 \twoheadrightarrow^* M_3$ and $M_2 \twoheadrightarrow^* M_3$ (cf. [Bar84]).

[6] The subject reduction property says that the reduction relation preserves typing sequents, i.e. if $\Gamma \triangleright M{:}\,\sigma$ and $M \twoheadrightarrow^* N$, then $\Gamma \triangleright N{:}\,\sigma$.

Nested let-expressions (let x_1=M_1 in (let x_2=$M_2 \cdots$ in N)) may be abbreviated as (let x_1=M_1, x_2=$M_2, \ldots$ in N).

The identity function $\lambda x.x$ is denoted by id, and for two terms M and N, the *composition* $\lambda x.(M\ (N\ x))$ is abbreviated as $M \circ N$.

1.2.2 Monads and computational effects

Denotational semantics is a powerful tool for describing the meaning of functional programs. What makes this theory even more attractive is that even the programs containing "impure" features, such as jumps or assignment, can be given a purely functional interpretation. For example, in a functional programming language with assignments, programs are interpreted in *state passing style*, where each function is passed an extra argument representing the state of the mutable store. Likewise, each function returns a pair consisting of its result and a new state of the store. A functional program with non-local jumps, can similarly be interpreted in *continuation passing style*.

Monads give a framework, due to Moggi [Mog88], that unifies the ideas used in denotational semantics to handle a particular "impure" feature of functional programs. Strictly speaking, a monad (in a category) is a categorical concept, providing the structure needed to give categorical interpretation of functional programs with features like assignment, divergence, exceptions, etc.

Many concepts we use in this paper are motivated by the monadic framework, but rather than giving a detailed exposition of monads in a category, we find it more useful to present the intuitive ideas in a simplified way. For completeness, we summarize the main definitions in the Section 1.9.1. A reader is also referred to [Mog88], [Mog91], and [ML71] for more details.

For simplicity, assume we want to give a denotational interpretation of a programming language in the category of sets, so that programs are interpreted as functions, and types are interpreted as sets.[7] A monad, T, is defined by a triple $(T, \eta, \_^*)$[8] where: T is a function mapping sets to sets; for each set A, η_A is a function from A to $T(A)$; and for each function $f: A \to T(B)$, f^* is a function from $T(A)$ to $T(B)$. Moreover, the components η and $\_^*$ of the monad satisfy the following *monad laws*:

- *right identity:* $\eta_A^* = \mathrm{id}_{T(A)}$,
- *left identity:* for every $f: A \to T(B)$, $\eta_A; f^* = f$, and
- *associativity:* for every $f: A \to T(B)$, $g: B \to T(C)$ and $h: C \to T(D)$,

$$(f; g^*); h^* = f; (g; h^*)^*,$$

where $\mathrm{id}_T(A)$ denotes the identity function on $T(A)$, and $f; g$ is the composition of f *followed* by g.

[7] A fully general description could be given by describing a monad in an arbitrary Cartesian closed category, instead of in **Set**.

[8] The triple $(T, \eta, \_^*)$ is called *Kleisli triple* and it uniquely defines a monad (see Section 1.9.1 for more details).

To help understanding how a monad is used in giving a denotational semantics of programs, we first present several examples.

Example 1.2.1. The *nondeterministic monad* $(N, \eta, \_^*)$ is defined by:

- function N that, to each set A assigns the set $\mathcal{P}_{\text{fin}}(A)$, the set of all finite subsets of A,
- for each set A, the function η_A, assigning the set $\{a\}$ to each element a of A, and
- for each function $f: A \to \mathcal{P}_{\text{fin}}(B)$, the function f^* is defined by $f^*(S) = \bigcup_{a \in S} f(a)$, for each finite $S \subseteq A$.

This monad can be used to describes nondeterministic computations as follows. A nondeterministic program, takes a value of type A and, in general, returns a set of results of type B (one for each computation path). The interpretation of such a program using the nondeterministic monad is a function from A to $\mathcal{P}_{\text{fin}}(B)$. If f is a function representing a deterministic program, $f; \eta_A$ represents a nondeterministic program that returns a single value produced by f. Let $f: A \to \mathcal{P}_{\text{fin}}(B)$ and $g: B \to \mathcal{P}_{\text{fin}}(C)$ be meanings of two nondeterministic programs. The meaning of a program obtained by sequencing these two programs is given by the function $f; g^*: A \to \mathcal{P}_{\text{fin}}(C)$.

Example 1.2.2. A *continuation monad* $(K, \eta, \_^*)$ is defined by:

- function K that, to each set A assigns the set $(A^R)^R$, where R is a set representing answers,
- for each set A, the function η_A, assigning the function $\lambda k \in A^R.k(a)$ to every element a of A, and
- for each function $f: A \to (B^R)^R$, the function f^* is defined by
$$f^*(c) = \lambda k \in B^R.c(\lambda a \in A.f(a)(k)),$$
for each $c \in (A^R)^R$.

In this monad, one can give the continuation passing style interpretation of programs. In CPS semantics, a computation is a function that takes a continuation and returns a result, and a continuation is a function that takes a value and returns a result. Intuitively, a continuation describes the rest of the program. We interpret a program that takes an argument of type A and returns a value of type B as a function from the set A to the set $(B^R)^R$. An effect-free program is a program that does not contain any "jumps", so its interpretation is a function form A to $(B^R)^R$, that computes a value and passes it directly to a continuation. More precisely, if $f: A \to B$ is a function describing how the program computes the return value from its argument, the function $f; \eta_B = \lambda x \in A.\lambda k \in B^R.k(f(x))$ gives the CPS meaning of the program. Similarly, the composition of two programs whose meaning are given by functions $f: A \to (B^R)^R$ and $g: B \to (C^R)^R$ is given by the function $f; g^* = \lambda a \in A.\lambda k \in C^R.f(a)(\lambda b \in B.g(b)(k))$.

Example 1.2.3. Mutable store can be described by the monad $(St, \eta, \_^*)$ where

- function St assigns, to each set A, the set $(A \times S)^S$, where S is a set representing the store (typically, elements of a S are functions from a set of locations to a set of values that can be stored in the sore),
- for each set A, the function η_A, assigning the function $\lambda s \in S.\langle a, s\rangle$ to every element a of A, and
- for each function $f\colon A \to (A \times S)^S$, the function f^* is defined by

$$f^*(c) = \lambda s \in S.(let\ \langle a, s'\rangle = c(s)\ in\ f(a)(s')),$$

  for each $c \in (A \times S)^S$.

A computation is represented by a function that given a state of the store produces a pair consisting of a value, and a new state of the store. Programs, that denote functions that can manipulate the store are interpreted as functions from a set A to the set $(B \times S)^S$. The meaning of a program which does not access the store is given by the function $f; \eta_B = \lambda a \in A.\lambda s \in S.\langle f(a), s\rangle$, where $f\colon A \to B$ describes the functional behavior of the program. The composition of two programs, the meanings of which are given by functions $f\colon A \to (B \times S)^S$ and $g\colon B \to (C \times S)^S$, is interpreted by the function $f; g^* = \lambda a \in A.\lambda s \in S.(let\ \langle b, s'\rangle = f(a)(s)\ in\ g(b)(s'))$, which says that the first program is evaluated in the current state of the store, followed by the evaluation of the second program in the state of the store that resulted from the evaluation of the first program.

Other examples of monads corresponding to divergence, interactive I/O, exceptions, mutable store, etc. can be found in [Mog88, Mog89, Wad90, Wad95].

To summarize the above examples, we notice that in each of the examples, the components of the monad η and $\_^*$ satisfy the monad laws. Secondly, computations were described by the set $T(A)$, where T was either K, N, or St, and the programs were interpreted as functions from a set A to the set of computations $T(B)$. The effect-free programs were interpreted as functions obtained by composing the function η_B with a function $f\colon A \to B$, describing the functional behavior of the program. Finally, programs were composed using the composition $f; g^*$, where f and g were the meanings of the individual programs.

In the sections that follow, we will concentrate on a general monad, and specifically on the continuation monad, and formalize the above discussed interpretation of programs. For the programming language, we use Moggi's λ_{cp} calculus. The interpretation will be formalized by several transforms. For instance, we define the computational transform to give an interpretation of λ_{cp} using an unspecified monad, and the CPS transform to give the CPS semantics using the continuation monad. For the metalanguage in which we give interpretation we choose calculi λ_{ml}, λ_c, $\lambda_{n\forall}$ and $\lambda_{cp\forall}$ depending on the particular transform that specified the interpretation. Sections 1.2.3, 1.2.4, and 1.2.5 define these calculi.

1.2.3 Computational lambda calculus

Calculus λ_{cp} represents the core of a sequential call-by-value programming language with possible "impure" effects. Even though λ_{cp} does not have any such features as assignment, continuations etc., its logic is sound with respect to models where such effects can be interpreted. Our presentation of λ_{cp} differs slightly form Moggi's and is in a style more familiar to a computer scientist.[9] We will define the *pure* calculus λ_{cp} and its extension, the *full* computational calculus λ_c. Note that while we use λ_{cp} to represent a programming language, we use λ_c to formalize the interpretation of λ_{cp} using the modified computational transform defined in section 1.3.2.

The computational calculus, λ_{cp}, is a call-by-value calculus, and therefore we need to distinguish between values and terms. The language of λ_{cp} is defined by the grammar:

$$\begin{array}{lrcl} \textit{values} & V & ::= & x \mid \lambda x.M \\ \textit{terms} & M & ::= & V \mid (M\ M) \mid (\texttt{let}\ x\texttt{=}M\ \texttt{in}\ M). \end{array}$$

Axioms of λ_{cp} are given in Table 1.2. We consider only simply typed term over a base type

$$((\lambda x.M)\ V) = M\{V/x\} \qquad (\beta_v)$$

$$\lambda x.Vx = V \quad , x \notin \mathrm{FV}(V) \qquad (\eta_v)$$

$$(\texttt{let}\ x\texttt{=}M\ \texttt{in}\ x) = M \qquad (\mathrm{unit})$$

$$\begin{array}{r} (\texttt{let}\ x_2\texttt{=}(\texttt{let}\ x_1\texttt{=}M_1\ \texttt{in}\ M_2)\ \texttt{in}\ M) = \\ (\texttt{let}\ x_1\texttt{=}M_1\ \texttt{in}\ (\texttt{let}\ x_2\texttt{=}M_2\ \texttt{in}\ M)) \end{array} \quad , x_1 \notin \mathrm{FV}(M) \qquad (\mathrm{assoc})$$

$$(M\ N) = (\texttt{let}\ x\texttt{=}M\ \texttt{in}\ (x\ N)) \qquad (\mathrm{let.1})$$

$$(V\ M) = (\texttt{let}\ y\texttt{=}M\ \texttt{in}\ (V\ y)) \qquad (\mathrm{let.2})$$

Table 1.2: Axioms of λ_{cp}.

$\imath$. The λ_{cp} type system is presented in Table 1.3.

The full computational calculus λ_c is an extension of λ_{cp} with the term constructors:

$$\begin{array}{lrcl} \textit{values} & V & ::= & \cdots \mid [M] \\ \textit{terms} & M & ::= & \ldots \mid \mu(M). \end{array}$$

The types of λ_c are extended with the unary type constructor T, and rules

[9] In contrast to [Mog88, Mog91], where λ_{cp} is presented as a calculus with two kinds of predicates, equality and existence, we only consider equality predicates. An existence predicate asserts that a term has a value, meaning that it has no computational effects. We prefer to define values as a syntactic subclass of terms. The two approaches are equivalent, in the sense that an existence predicate is provable in Moggi's setting for exactly the same terms that, in our setting, can be shown provably equal to a value. We should also point out, that in our presentation, we omit products, but as shown in Section 1.5.5, all of our results hold if products were included λ_{cp}.

Types:

$$\sigma ::= \iota \mid \sigma \to \sigma$$

Inference Rules:

$$\Gamma, x\colon \sigma \rhd_{cp} x\colon \sigma \qquad \text{(var)}$$

$$\frac{\Gamma \rhd_{cp} M\colon \sigma \to \tau, \quad \Gamma \rhd_{cp} N\colon \sigma}{\Gamma \rhd_{cp} (M\ N)\colon \tau} \qquad \text{(app)}$$

$$\frac{\Gamma, x\colon \sigma \rhd_{cp} M\colon \tau}{\Gamma \rhd_{cp} \lambda x.M\colon \sigma \to \tau} \qquad \text{(abs)}$$

$$\frac{\Gamma \rhd_{cp} M\colon \sigma, \quad \Gamma, x\colon \sigma \rhd_{cp} N\colon \tau}{\Gamma \rhd_{cp} (\mathtt{let}\ x\mathtt{=}M\ \mathtt{in}\ N)\colon \tau} \qquad \text{(let)}$$

Table 1.3: Type inference rules of λ_{cp}.

$$\frac{\Gamma \rhd_c M\colon \sigma}{\Gamma \rhd_c [M]\colon T(\sigma)} \qquad ([\cdot])$$

and

$$\frac{\Gamma \rhd_c M\colon T(\sigma)}{\Gamma \rhd_c \mu(M)\colon \sigma} \qquad (\mu)$$

type the new term constructors. The logic of λ_c extends the logic of λ_{cp} with the axioms:

$$\mu([M]) = M \qquad (\mu.\beta)$$

and

$$[\mu(V)] = V. \qquad (\mu.\eta)$$

The language of λ_c is certainly more expressive that λ_{cp}. However, its logic does not prove any new equations between λ_{cp} terms, that is λ_c is a conservative extension of λ_{cp}.

Proposition 1.2.4. Let M and N be λ_{cp} terms. Then $\lambda_c \vdash M = N$ iff $\lambda_{cp} \vdash M = N$.

Proof Sketch: We define the reduction relation in λ_c, and show it has the following properties. First the reduction is Church-Rosser, which implies that if $\lambda_c \vdash M = N$, then there is a term P such that both M and N reduce to P, and secondly, if M is a λ_{cp} term that reduces to P, then P is also a λ_{cp} term. The reduction relation is defined by "orienting" the λ_c axioms from left to right. Then the second property is obvious. To show the Church-Rosser property we consider the reduction relation defined by reductions $(\mu.\beta)$ and $(\mu.\eta)$, and show that it commutes with the *let*, β_v and η_v reduction relations as defined in [Mog88]. ■

In this work we do not use the full expressive power of λ_c. Using constructors $[\cdot]$ and

$\mu(\cdot)$, we can define terms

$$\mathcal{E} \stackrel{\text{def}}{=} \lambda x.[x] \quad \text{and} \quad \mathcal{R} \stackrel{\text{def}}{=} \lambda x.\mu(x),$$

for which it is provable that

$$\lambda_c \vdash \mathcal{R} \circ \mathcal{E} = \mathtt{id}. \tag{r-e}$$

In all of our proofs, we only use terms $\mathcal{E}$ and $\mathcal{R}$, and equality (r-e), without any other reference to constructors $[\cdot]$ and $\mu(\cdot)$. Thus when we consider λ_c we are really referring to λ_{cp} extended with terms $\mathcal{E}$ and $\mathcal{R}$ satisfying equation (r-e) and typable with appropriate types. Namely for any type σ we assume sequents $\Gamma \rhd_c \mathcal{E}: \sigma \rightarrow T(\sigma)$ and $\Gamma \rhd_c \mathcal{R}: T(\sigma) \rightarrow \sigma$ are derivable. As a consequence, our reasoning carries over if we replace $\mathcal{E}$ and $\mathcal{R}$ with any terms having the above mentioned abstract properties.

Intuitively, we think of functions $\mathcal{E}$ and $\mathcal{R}$ in a context of a particular monad. Namely, given a value denoted by V of type σ, $\mathcal{E}(V)$ denotes the trivial computation of the meaning of V which has the type $T(\sigma)$. On the other hand, given a computation m, denoted by M, of type $T(\sigma)$, $\mathcal{R}(M)$ denotes the result of the evaluation of m.

For example, when defining the modified CPS transform (see section 1.3.2), we define terms $\mathcal{E}^K$ and $\mathcal{R}^K$ to be

$$\mathcal{E}^K \stackrel{\text{def}}{=} \lambda xk.k\ x \quad \text{and} \quad \mathcal{R}^K \stackrel{\text{def}}{=} \lambda x.x\ \mathtt{id}.$$

For a value v, denoted by V, $\mathcal{E}^K(V)$ denotes a trivial computation of v that, given a continuation, passes v to the continuation. If M denotes a computation m, $\mathcal{R}^K(M)$ denotes the result of evaluating m by passing it a "top-level" continuation $\mathtt{id}$. It is easy to see that $\lambda_{cp} \vdash \mathcal{R}^K \circ \mathcal{E}^K = \mathtt{id}$, so the reasoning in λ_c, using abstract properties of $\mathcal{E}$ and $\mathcal{R}$, applies when we replace $\mathcal{E}$ and $\mathcal{R}$ with terms $\mathcal{E}^K$ and $\mathcal{R}^K$.

Even though `let`-expressions are elementary constructs in λ_c, it is sometimes convenient to treat them as syntactic sugar. The following two lemmas justifies doing so. First we show that a `let`-expression is provably equivalent to its de-sugared from.

Lemma 1.2.5. $(\mathtt{let}\ x\mathtt{=}M\ \mathtt{in}\ N) = ((\lambda x.N)\ M)$.

Proof: By (β_v) and (let.2) we have,

$$(\mathtt{let}\ x\mathtt{=}M\ \mathtt{in}\ N) = (\mathtt{let}\ x\mathtt{=}M\ \mathtt{in}\ ((\lambda x.N)\ x)) = ((\lambda x.N)\ M).$$

■

The next lemma is necessary to justify omitting the (let) case form the proofs by induction on the type derivation in λ_c. Let M and M' be λ_c terms such that M' is obtained from M by de-sugaring all `let`'s in M, that is, by replacing every subterm $(\mathtt{let}\ x\mathtt{=}P\ \mathtt{in}\ Q)$ in M with $((\lambda x.Q)\ P)$. Then the lemma implies that M is typable with σ from hypothesis Γ using type inference rules of λ_c, if and only if M' is typable with σ from Γ without using the (let) typing rule.

Lemma 1.2.6. Assume M and N do not contain any `let`-expressions. Then, sequent $\Gamma \rhd_c (\texttt{let } x\texttt{=}M \texttt{ in } N){:}\,\sigma$ is derivable in λ_c if and only if sequent $\Gamma \rhd_c ((\lambda x.N)\ M){:}\,\sigma$ is derivable in λ_c, but without using the (let) rule.

Proof Sketch: To show one direction, show that from the antecedents of the (let) rule, which must be the last rule in the derivation of $\Gamma \rhd_c (\texttt{let } x\texttt{=}M \texttt{ in } N){:}\,\sigma$, one can derive $\Gamma \rhd_c ((\lambda x.N)\ M){:}\,\sigma$ using (abs) and (app) rules. The other direction is proved similarly. ∎

An equivalent set of axioms for the logic of λ_c can be defined using the notion of an *evaluation context.*

Definition 1.2.7. An *evaluation context* is a λ_c context defined by the following grammar:

$$E⟪\cdot⟫ ::= ⟪\cdot⟫ \mid (V\ E⟪\cdot⟫) \mid (E⟪\cdot⟫\ M) \mid (\texttt{let } x\texttt{=}E⟪\cdot⟫ \texttt{ in } M) \mid (\texttt{let } x\texttt{=}V \texttt{ in } E⟪\cdot⟫),$$

where V ranges over values, and M ranges over terms.

Intuitively, the evaluation context specifies the order of evaluation. Using the evaluation context, Sabry and Felleisen [SF93] have formulated a set of axioms, equivalent to axioms of λ_c, which are sometimes more convenient to use in proving λ_c identities. In their formulation, they use the following equation schema:

$$((\lambda x.E⟪x⟫)\ M) = E⟪M⟫, \qquad x \notin \mathrm{FV}(E⟪\cdot⟫). \tag{β_{sf}}$$

Lemma 1.2.8 (Sabry-Felleisen). Every instance of the above schema is provable in λ_c.

In most of our results we prove equations by convertibility and to simplify the proofs we will refer to equations in Table 1.4. Since λ_c is conservative over λ_{cp}, all of the equalities are provable in λ_{cp} as well. In all of the equations, we assume M, N, P, and Q are arbitrary λ_c terms, and V and W denote λ_c values. The equations in Table 1.4 all follow easily from λ_{cp} axioms, or using equation (β_{sf}). For illustration we show how (C-assoc) is proved. Note that (C-assoc) is not provable in λ_v.

Lemma 1.2.9. In λ_c, composition is associative. Namely, for any values U, V and W

$$((U \circ V) \circ W) = (U \circ (V \circ W)).$$

Proof:

$$\begin{aligned}
((U \circ V) \circ W) &= \lambda x.((U \circ V)\ (W\ x)) && \text{by defn.}\\
&= \lambda x.((\lambda y.(U\ (V\ y)))\ (W\ x)) && \text{by defn.}\\
&= \lambda x.(\texttt{let } y\texttt{=}(W\ x) \texttt{ in } (U\ (V\ y))) && \text{by (let.2) and } (\beta_v)\\
&= \lambda x.(\texttt{let } y\texttt{=}(W\ x) \texttt{ in } (\texttt{let } z\texttt{=}(V\ y) \texttt{ in } (U\ z))) && \text{by (let.2)}\\
&= \lambda x.(\texttt{let } z\texttt{=}(\texttt{let } y\texttt{=}(W\ x) \texttt{ in } (V\ y)) \texttt{ in } (U\ z)) && \text{by (assoc)}\\
&= \lambda x.(U\ (\texttt{let } y\texttt{=}(W\ x) \texttt{ in } (V\ y))) && \text{by (let.2)}\\
&= \lambda x.(U\ (V\ (W\ x))) && \text{by (let.2)}\\
&= \lambda x.(U\ ((\lambda y.(V\ (W\ y)))\ x)) && \text{by } (\beta_v)\\
&= \lambda x.U\ ((V \circ W)\ x) && \text{by defn.}\\
&= (U \circ (V \circ W)) && \text{by defn.}
\end{aligned}$$

$$(\mathtt{id}\ M) = M \tag{C-id}$$

$$\frac{V \circ W = \mathtt{id}}{(V\ (W\ M)) = M} \tag{C-id.1}$$

$$((U \circ V) \circ W) = (U \circ (V \circ W)) \tag{C-assoc}$$

$$(V \circ \mathtt{id}) = (\mathtt{id} \circ V) = V \tag{C-unit}$$

$$(V\ (\mathtt{let}\ x\mathtt{=}M\ \mathtt{in}\ N)) = (\mathtt{let}\ x\mathtt{=}M\ \mathtt{in}\ (V\ N)), \qquad x \notin \mathrm{FV}(V) \tag{C-appl.1}$$

$$(\mathtt{let}\ x\mathtt{=}V\ \mathtt{in}\ M) = M\{V/x\} \tag{C-let.1}$$

$$((\mathtt{let}\ x\mathtt{=}P\ \mathtt{in}\ V)\ (\mathtt{let}\ y\mathtt{=}Q\ \mathtt{in}\ W)) = (\mathtt{let}\ x\mathtt{=}P\ \mathtt{in}\ (\mathtt{let}\ y\mathtt{=}Q\ \mathtt{in}\ (V\ W))) \tag{C-appl.2}$$

$$((\lambda x.x\ M)\ N) = (N\ M), \qquad x \notin \mathrm{FV}(M) \tag{C-appl.3}$$

Table 1.4: Identities provable in λ_c

■

1.2.4 Monadic metalanguage

As a formalization of a metalanguage in which a denotational semantics of λ_{cp} can be defined, we introduce λ_{ml} calculus. We remark that λ_{ml} is what Moggi [Mog91] calls a "monadic metalanguage", but our presentation is more in tune with a functional programming point of view. We define the calculus λ_{ml} to be an extension of the simply typed $\beta\eta$-calculus, λ_n, with a set of constructs corresponding to an arbitrary monad.

The language of λ_{ml} consists of the following terms:

$$M ::= x \mid (M\ M) \mid \lambda x.M \mid [M]_T \mid (\mathtt{let}_T\ x \Leftarrow M\ \mathtt{in}\ M).$$

Note that above term constructs are different from constructs $[\cdot]$ and `let` used in λ_c. The types of λ_{ml} are simple types over a base type $\imath$, extended with types of the form $T(\sigma)$ where T is a unary type constructor, and the type inference rules are added for typing $[\cdot]_T$ and $\mathtt{let}_T$ expressions. The type system is given in Table 1.5.

In addition to (β) and (η) axioms, the logic of λ_{ml} contains the additional axioms that correspond to monad laws. The axioms are given in Table 1.6 below.

Types:

$$\sigma ::= \imath \mid \sigma \to \sigma \mid T(\sigma)$$

Inference Rules:

$$\Gamma, x{:}\,\sigma \rhd_{ml} x{:}\,\sigma \qquad \text{(var)}$$

$$\frac{\Gamma \rhd_{ml} M{:}\,T(\sigma), \quad \Gamma, x{:}\,\sigma \rhd_{ml} N{:}\,T(\tau)}{\Gamma \rhd_{ml} (\mathtt{let}_T\ x \Leftarrow M\ \mathtt{in}\ N){:}\,T(\tau)} \quad \text{(let)}$$

$$\frac{\Gamma \rhd_{ml} M{:}\,\sigma \to \tau, \quad \Gamma \rhd_{ml} N{:}\,\sigma}{\Gamma \rhd_{ml} (M\ N){:}\,\tau} \quad \text{(app)}$$

$$\frac{\Gamma \rhd_{ml} M{:}\,\sigma}{\Gamma \rhd_{ml} [M]_T{:}\,T(\sigma)} \quad \text{(unit)}$$

$$\frac{\Gamma, x{:}\,\sigma \rhd_{ml} M{:}\,\tau}{\Gamma \rhd_{ml} \lambda x.M{:}\,\sigma \to \tau} \quad \text{(abs)}$$

Table 1.5: Type system of λ_{ml}.

1.2.5 Calculi λ_n, $\lambda_{n\forall}$, $\lambda_{cp\forall}$, etc.

Calculus λ_n is intended to capture call-by-name evaluation in programming languages. The language of λ_n is defined by the grammar

$$M ::= x \mid \lambda x.M \mid (M\ M),$$

and its axioms are

$$((\lambda x.M)\ N) = M\{N/x\} \qquad (\beta)$$

and

$$\lambda x.Mx = M \quad , x \notin \mathrm{FV}(M). \qquad (\eta)$$

In λ_n, We consider only simply typed terms over a base type $\imath$. The type system is described in Table 1.7.

In order to define a typed CPS transform we need to extend λ_n with polymorphic types. Let $\lambda_{n\forall}$ denote this extension. The types of $\lambda_{n\forall}$ are extended with type variables and universal types. Namely we have

$$\sigma ::= \imath \mid t \mid \sigma \to \sigma \mid \forall t.\sigma.$$

We will use letters s and t to range over type variables. The type inference rules, in addition to those of λ_n, are given in Table 1.8 below. In $\lambda_{n\forall}$, it is possible to define constructors

$$((\lambda x.M)\ N) = M\{N/x\} \tag{β}$$

$$\lambda x.Mx = M \quad , x \notin \mathrm{FV}(M) \tag{η}$$

$$(\mathtt{let}_T\ x \Leftarrow M\ \mathtt{in}\ [x]_T) = M \tag{R-unit}$$

$$(\mathtt{let}_T\ x \Leftarrow [M]_T\ \mathtt{in}\ N) = N\{M/x\} \tag{L-unit}$$

$$\begin{array}{r} (\mathtt{let}_T\ x_2 \Leftarrow (\mathtt{let}_T\ x_1 \Leftarrow M_1\ \mathtt{in}\ M_2)\ \mathtt{in}\ M) = \\ (\mathtt{let}_T\ x_1 \Leftarrow M_1\ \mathtt{in}\ (\mathtt{let}_T\ x_2 \Leftarrow M_2\ \mathtt{in}\ M)) \end{array}, \quad x_1 \notin \mathrm{FV}(M) \tag{assoc}$$

Table 1.6: Axioms of λ_{ml}.

corresponding to a continuation monad. Let us define

$$K(\sigma) \stackrel{\mathrm{def}}{=} \forall s.(\sigma \rightarrow s) \rightarrow s$$

$$\begin{array}{rcl} [M]_K & \stackrel{\mathrm{def}}{=} & \lambda k.k\ M \\ (\mathtt{let}_K\ x \Leftarrow M\ \mathtt{in}\ N) & \stackrel{\mathrm{def}}{=} & \lambda k.(M\ (\lambda x.(N\ k))) \end{array}$$

Note that K, $[\cdot]_K$, and $\mathtt{let}_K$ is to be understood as abbreviation for the corresponding terms on the right-hand side.

One can give an intuitive interpretation for the above constructs, similar to the intuitive explanation of the continuation monad given in section 1.2.2. A computation in a continuation monad is a function that takes a *continuation* and returns a *result*. A continuation represents the rest of the program, and can be described as a function that maps values to results. Therefore, if m is a value denoted by M, the trivial computation of m will take a continuation and pass m to it. Hence the definition of $[M]_K = \lambda k.k\ M$. If N denotes a computation, and k is (bound to) a continuation, $\lambda x.(N\ k)$ denotes a new continuation that, given a value bound to x, returns a result of computing N with continuation k. Computations denoted by M and N can thus be composed by passing the continuation $\lambda x.(N\ k)$ to M as done in the definition of $(\mathtt{let}_K\ x \Leftarrow M\ \mathtt{in}\ N)$.

To determine a type of a computation of values of type σ, let s denote the type of results. Then a continuation must be of type $\sigma \rightarrow s$ and a computation will have type $(\sigma \rightarrow s) \rightarrow s$. Many authors, e.g. [MW85, Wad90, Mog91, HL92, HD94], define the type of results to be a fixed (observable) type o. We believe this is not general enough for our purposes. Instead we define the type of computations to be $K(\sigma) = \forall s.(\sigma \rightarrow s) \rightarrow s$ as above. The reason for defining the type of computations in this way is that we want to show that type σ is a retract of type $K(\sigma)$ (see definition 1.4.1). With our definition of $K(\sigma)$, the functions $\mathcal{E}^K$ and $\mathcal{R}^K$, defined in Section 1.2.3, provide the desired retraction-embedding pair. On the other hand,

Types:

$$\sigma ::= \iota \mid \sigma \to \sigma$$

Inference rules:

$$\Gamma, x\colon\sigma \rhd_n x\colon\sigma \qquad \text{(var)}$$

$$\frac{\Gamma \rhd_n M\colon\sigma\to\tau, \quad \Gamma \rhd_n N\colon\tau}{\Gamma \rhd_n (M\ N)\colon\tau} \qquad \text{(app)}$$

$$\frac{\Gamma, x\colon\sigma \rhd_n M\colon\tau}{\Gamma \rhd_n \lambda x.M\colon\sigma\to\tau} \qquad \text{(abs)}$$

Table 1.7: Type system of simple types for λ_n.

Types:

$$\sigma ::= \cdots \mid t \mid \forall t.\sigma$$

Inference rules:

$$\frac{\Gamma \rhd_\forall M\colon\sigma}{\Gamma \rhd_\forall M\colon\forall t.\sigma}, \qquad \text{if } x \text{ is not free in } \Gamma \qquad \text{(gen)}$$

$$\frac{\Gamma \rhd_\forall M\colon\forall t.\sigma}{\Gamma \rhd_\forall M\colon\sigma\{\tau/t\}} \qquad \text{(inst)}$$

Table 1.8: Polymorphic extension of a type system.

to define a retraction-embedding pair from σ to the type $(\sigma\to o)\to o$, one would require that each type σ is a retract of the type of answers o. It is also possible to give a proper typing for the functions $\mathcal{E}^K$ and $\mathcal{R}^K$ using recursive types, but we prefer the polymorphic solution. note that we don't really use the full power of polymorphism, since the only polymorphic types occur in the form $\forall s.(\sigma\to s)\to s$, where s is not free in σ.

To restrict the codomain of the CPS transform to a call-by-value language, we also define the calculus $\lambda_{cp\forall}$, to be an extension of λ_{cp} with polymorphic types.

In addition to polymorphic extension, we will also consider an extension of the type system of a calculus with products. This extension is discussed in Section 1.9.2.

1.3 Source to Source Transforms

A source-to-source transform is simply a function mapping terms of one calculus to terms of another calculus. Since we work in typed lambda calculi, instead of mapping terms to terms, we study transforms that map typing sequents of one calculus to typing sequents of another calculus. However, we will often leave the typing implicit.

Even though, the languages of some of the calculi we study coincide, we will be precise in specifying the calculus of the domain and the calculus of the codomain of each of the transforms.

We study transforms which give slightly different interpretations of λ_{cp} in λ_{ml}, λ_c and $\lambda_{cp\forall}$. The main characteristic of these transforms is that they make the computational structure of the λ_{cp} terms explicit. For example, the CPS transform describes the computational structure of a term in terms of continuations which explicitly appear in the transformed term.

We study, the computational transform form λ_{cp} to λ_{ml} and its modification that maps λ_{cp} into λ_c. We also study the CPS transform and its modified counterpart as transforms form λ_{cp} into $\lambda_{cp\forall}$.

1.3.1 The computational transform

The computational transform, $\mathcal{T}_{ml}$, is a generalization of the CPS transform in which the notion of computations is generalized from continuation passing to an arbitrary one. Intuitively, the computational transform gives a functional interpretation of call-by-value programs with possible "impure" effects, in the calculus λ_{ml}. The computational transform was first presented in [Wad90] as an interpretation of programs by means of list comprehension. Wadler defines both, call-by-value and call-by-name variants of the transform. The same transform also appears in [Mog91] as the $\_^\circ$ translation connecting the "computational programming language" (i.e. λ_c) and the "monadic metalanguage" (i.e. λ_{ml}).[10] Furthermore, in [HD94], call-by-value and call-by-name variants of the computational transform appear as call-by-value and call-by-name encodings of programs in Moggi's monadic metalanguage. Hatcliff and Danvy use the transform to characterize the essential part of the CPS transform, thus breaking the CPS transform into a composition of the generic encoding, given by the transform $\mathcal{T}_{ml}$, and, what they call, the "continuation introduction".

$$
\begin{array}{rcl}
\overline{\overline{x}} & \stackrel{\mathrm{def}}{=} & [x]_T \\
\overline{\overline{\lambda x.M}} & \stackrel{\mathrm{def}}{=} & [\lambda x.\overline{\overline{M}}]_T \\
\overline{\overline{(M\ N)}} & \stackrel{\mathrm{def}}{=} & (\mathtt{let}_T\ x{\Leftarrow}\overline{\overline{M}}\ \mathtt{in}\ (\mathtt{let}_T\ y{\Leftarrow}\overline{\overline{N}}\ \mathtt{in}\ (x\ y))) \\
\overline{\overline{(\mathtt{let}\ x\mathtt{=}M\ \mathtt{in}\ N)}} & \stackrel{\mathrm{def}}{=} & (\mathtt{let}_T\ x{\Leftarrow}\overline{\overline{M}}\ \mathtt{in}\ \overline{\overline{N}})
\end{array}
$$

$$
\begin{array}{rcl}
\imath' & \stackrel{\mathrm{def}}{=} & \imath \\
(\sigma{\to}\tau)' & \stackrel{\mathrm{def}}{=} & \sigma'{\to}T(\tau')
\end{array}
\qquad\qquad
\begin{array}{rcl}
\{x_i\!:\sigma_i\}' & \stackrel{\mathrm{def}}{=} & \{x_i\!:\sigma_i'\} \\
\overline{\overline{\Gamma \triangleright_{cp} M\!:\sigma}} & \stackrel{\mathrm{def}}{=} & \Gamma' \triangleright_{ml} \overline{\overline{M}}\!:T(\sigma')
\end{array}
$$

Table 1.9: The computational transform $\mathcal{T}_{ml}$.

[10]In [Mog91], the $\_^\circ$ translation is defined only for his "simple language", which has no lambda abstractions.

We concentrate on the call-by-value variant of the computational transform as a transform mapping typing sequents of λ_{cp} into typing sequents of λ_{ml}. The definition of the transform is given in Table 1.9. In order to show that the transform is well defined, we need to show that it maps *derivable* typing sequents to *derivable* typing sequents.

Proposition 1.3.1. If $\Gamma \triangleright_{cp} M\colon \sigma$ then $\Gamma' \triangleright_{ml} \overline{\overline{M}}\colon T(\sigma')$.

Proof: Easy by induction on the derivation of $\Gamma \triangleright_{cp} M\colon \sigma$. ■

Another aspect of the correctness of the computational transform is the equational correspondence result, which says that the transform preserves equality in λ_{cp}, and moreover, nothing, up to λ_{cp}-equivalence, is lost in the transform.

Theorem 1.3.2 (Sabry-Wadler). $\lambda_{cp} \vdash M = N$ iff $\lambda_{ml} \vdash \overline{\overline{M}} = \overline{\overline{N}}$.

We point the reader to [HD94] and [SW96] for more detailed analysis of the transform $\mathcal{T}_{ml}$.

1.3.2 The modified computational transform

In an effort to define a transform, similar to the CT, but mapping λ_{cp} into a call-by-value calculus, we define the *modified computational transform*, $\mathcal{T}_{ml\sharp}$, to be a transform from λ_{cp} into λ_c. The transform $\mathcal{T}_{ml\sharp}$, is defined by composing the computational transform $\mathcal{T}_{ml}$ with the transform $\mathcal{T}_\sharp$ defined in Table 1.10. It is easy to see that the transform $\mathcal{T}_\sharp$ is well defined. Namely,

Proposition 1.3.3. If $\Gamma \triangleright_{ml} M\colon \sigma$ then $\Gamma \triangleright_{c} M^\sharp\colon \sigma$.

$$
\begin{aligned}
x^\sharp &\stackrel{\text{def}}{=} x \\
(\lambda x.M)^\sharp &\stackrel{\text{def}}{=} \lambda x.M^\sharp \\
(M\ N)^\sharp &\stackrel{\text{def}}{=} (M^\sharp\ N^\sharp) \\
{[M]_T}^\sharp &\stackrel{\text{def}}{=} \mathcal{E}(M^\sharp) \\
(\mathtt{let}_T\ x \Leftarrow M\ \mathtt{in}\ N)^\sharp &\stackrel{\text{def}}{=} (\mathtt{let}\ x\mathtt{=}\mathcal{R}(M^\sharp)\ \mathtt{in}\ N^\sharp) \\[1em]
(\Gamma \triangleright_{ml} M\colon \sigma)^\sharp &\stackrel{\text{def}}{=} \Gamma \triangleright_c M^\sharp\colon \sigma
\end{aligned}
$$

Table 1.10: Transform $\mathcal{T}_\sharp$ from λ_{ml} to λ_c.

In the previous section, we have seen that the transform $\mathcal{T}_{ml}$ preserves λ_{cp}-equivalence. However, transform $\mathcal{T}_\sharp$, does not preserve this result for all terms. We will, however, show that the equational correspondence result holds for the transform $\mathcal{T}_{ml\sharp}$, if we restrict our attention to pure closed terms.

To compare the transforms $\mathcal{T}_{ml}$ and $\mathcal{T}_{ml\sharp}$, we examine their effect on `let`-expressions. By definition we have:

$$\overline{\overline{(\mathtt{let}\ x\mathtt{=}M\ \mathtt{in}\ N)}} = (\mathtt{let}\ x \Leftarrow \overline{\overline{M}}\ \mathtt{in}\ \overline{\overline{N}}),$$

and

$$\overline{\overline{(\mathtt{let}\ x\mathtt{=}M\ \mathtt{in}\ N)}}^{\sharp} = (\mathtt{let}\ x\mathtt{=}\mathcal{R}(\overline{\overline{M}}^{\sharp})\ \mathtt{in}\ \overline{\overline{N}}^{\sharp}).$$

To see the difference, observe that the terms $(\mathtt{let}_T\ x{\Leftarrow}M\ \mathtt{in}\ N)$ and $(\mathtt{let}\ x\mathtt{=}\mathcal{R}(M)\ \mathtt{in}\ N)$, have similar, but different meanings. The difference is that if the computation denoted by M has any computational effects, they may be discarded by the function $\mathcal{R}$. For example, if M changed the store, the changes would not be seen in the evaluation of N. In the second terms, however, any possible effects in evaluation of M, will affect the evaluation of N. Therefore, if M changes the store, N will be evaluated in the changed store.

Not surprisingly, the equational correspondence result for $\mathcal{T}_{ml\sharp}$ fails for some terms. In particular we have

$$\lambda_{cp} \vdash (\mathtt{let}\ x\mathtt{=}M\ \mathtt{in}\ x) = M,$$

but

$$\lambda_c \vdash \overline{\overline{(\mathtt{let}\ x\mathtt{=}M\ \mathtt{in}\ x)}}^{\sharp} = \mathcal{E}(\mathcal{R}(\overline{\overline{M}}^{\sharp})),$$

and, in general, $\mathcal{E}(\mathcal{R}(\overline{\overline{M}}^{\sharp})) \neq \overline{\overline{M}}^{\sharp}$ unless $\overline{\overline{M}}^{\sharp}$ denotes an effect-free computation. However, if M is a pure closed term one has that $\overline{\overline{M}}^{\sharp}$ denotes an effect-free computation, and $\mathcal{E}(\mathcal{R}(\overline{\overline{M}}^{\sharp})) = \overline{\overline{M}}^{\sharp}$. We state this, somewhat informally, in the following lemma.

Lemma 1.3.4. Let ρ be a substitution mapping variables to "proper" terms. Then for any term M such that $\Gamma \triangleright_{cp} M{:}\,\sigma$,

$$\lambda_c \vdash \rho\,\overline{\overline{M}}^{\sharp} = \mathcal{E}(\mathcal{R}(\rho\,\overline{\overline{M}}^{\sharp})).$$

Proof Sketch: While it is possible to define "proper" terms and prove the above lemma in its own right, the lemma also follows as a corollary of Lemma 1.5.10, provided the "proper" terms are defined as in the hypothesis of the same lemma. ■

From the above lemma, we can conclude that, for the pure closed terms, $\mathcal{T}_{ml\sharp}$ preserves λ_{cp}-equivalence, and combined with the Retraction Theorem, we obtain the following equational correspondence result.

Theorem 1.3.5. Let M and N be closed λ_{cp} terms such that $\triangleright_{cp} M{:}\,\sigma$ and $\triangleright_{cp} N{:}\,\sigma$. Then,

$$\lambda_{cp} \vdash M = N \quad \text{iff} \quad \lambda_c \vdash \overline{\overline{M}}^{\sharp} = \overline{\overline{N}}^{\sharp}.$$

Proof Sketch: Assume $\lambda_{cp} \vdash M = N$. We need to show that $\mathcal{T}_{ml\sharp}$ preserves all axioms of λ_{cp}. The only questionable axiom is (unit), so let us assume that $M = (\mathtt{let}\ x\mathtt{=}N\ \mathtt{in}\ x)$. By the previous lemma, $\overline{\overline{N}}^{\sharp} = \mathcal{E}(\mathcal{R}(\overline{\overline{N}}^{\sharp})) = \overline{\overline{(\mathtt{let}\ x\mathtt{=}N\ \mathtt{in}\ x)}}^{\sharp}$ as desired.

The other implication in the theorem follows from the Retraction Theorem (Theorem 1.5.11). ■

1.3.3 The CPS transform

The Fischer-Reynolds [Fis72, Fis93, Rey72] continuation-passing style transform comes in several variations: call-by-name, call-by-value, left-to-right and right-to-left. We focus on a call-by-value variant based on the one defined by Plotkin [Plo75]. The definition is given in Table 1.11.

$$
\begin{array}{rcl}
\overline{x} & \stackrel{\text{def}}{=} & \lambda k.k\ x \\
\overline{(\lambda x.M)} & \stackrel{\text{def}}{=} & \lambda k.k\ \lambda x.\overline{M} \\
\overline{(M\ N)} & \stackrel{\text{def}}{=} & \lambda k.\overline{M}\ (\lambda m.\overline{N}\ (\lambda n.m\ n\ k)) \\
\overline{(\texttt{let}\ x\texttt{=}M\ \texttt{in}\ N)} & \stackrel{\text{def}}{=} & \lambda k.\overline{M}\ (\lambda x.\overline{N}\ k)
\end{array}
$$

$$
\begin{array}{rcl}
K(\sigma) & \stackrel{\text{def}}{=} & (\sigma \rightarrow \sigma) \rightarrow \sigma \\
\imath' & \stackrel{\text{def}}{=} & \imath \\
(\sigma \rightarrow \tau)' & \stackrel{\text{def}}{=} & \sigma' \rightarrow K(\tau')
\end{array}
\qquad
\begin{array}{rcl}
\{x_i\colon \sigma_i\}' & \stackrel{\text{def}}{=} & \{x_i\colon \sigma_i'\} \\
\overline{\Gamma \triangleright_{cp} M\colon \sigma} & \stackrel{\text{def}}{=} & \Gamma' \triangleright_n \overline{M}\colon K(\sigma)
\end{array}
$$

Table 1.11: The CPS transform.

A typed versions of the CPS transform has first been studied by Meyer and Wand [MW85] for the simple types, and later by Harper and Lillibridge [HL92] in a polymorphic setting. Our definition differs from the one in this and some other work (e.g. [Mog88, HD94]) in the definition of the type of computations. As discussed in Section 1.2.5, the type of computations, $K(\sigma)$ is defined to be $\forall s.(\sigma \rightarrow s) \rightarrow s$ instead of the more traditional $(\sigma \rightarrow o) \rightarrow o$, where o denotes a fixed type of answers. The following proposition asserts that the CPS transform is well defined on typing sequents.

Proposition 1.3.6. If $\Gamma \triangleright_{cp} M\colon \sigma$ then $\Gamma' \triangleright_{cp\forall} \overline{M}\colon K(\sigma')$.

As for the computational transform, the equational correspondence result holds for the CPS transform as well (see Theorem 1.1.2). In fact, this result is a corollary of the equational correspondence theorem for the computational transform (Theorem 1.3.2), when the CPS transform is viewed as a special case of the transform $\mathcal{T}_{ml}$ as discussed below.

One can define the CPS transform as a special case of the computational transform by considering the definition of the computational transform as a template, and using the terms corresponding to continuation monad K instead of the terms corresponding to a general monad T in λ_{ml}. Recall the definition of K, $[M]_K$ and $(\texttt{let}_K\ x \Leftarrow M\ \texttt{in}\ N)$ given in section 1.2.5. Substituting K, $[M]_K$ and $(\texttt{let}_K\ x \Leftarrow M\ \texttt{in}\ N)$ for T, $[M]_T$ and $(\texttt{let}_T\ x \Leftarrow M\ \texttt{in}\ N)$ in the definition of $\mathcal{T}_{ml}$, we obtain the CPS transform. For example,

$$
\begin{array}{rcl}
\overline{(M\ N)} & = & (\texttt{let}_K\ m \Leftarrow \overline{M}\ \texttt{in}\ (\texttt{let}_K\ n \Leftarrow \overline{N}\ \texttt{in}\ (m\ n))) \\
 & = & \lambda k.\overline{M}\ (\lambda m.((\lambda k.\overline{N}\ (\lambda n.(m\ n)\ k))\ k)) \\
 & = & \lambda k.\overline{M}\ (\lambda m.\overline{N}\ (\lambda n.m\ n\ k)).
\end{array}
$$

Another interesting property of the CPS terms is that they behave the same way under the call-by-name and call-by-value evaluation strategies. This indifference result was used by Plotkin to simulate call-by-value programs with a call-by-name evaluator. Equationally, this means that the call-by-name equational logic is conservative over the call-by-value logic when restricted to the range of the CPS transform (see Theorem 1.1.5). Because of this indifference result, without loss of generality, we can restrict the logic of the codomain of the CPS transform to the logic of λ_{cp}. Because of our definition of the types of computations $K(\sigma)$, the codomain of the CPS transform requires polymorphic types. Therefore we view the CPS transform as a transform from λ_{cp} to $\lambda_{cp\forall}$.

1.3.4 The modified CPS transform

Analogous to the modified computational transform we can define the modified CPS transform. We do so by using the terms $\mathcal{E}^K$ and $\mathcal{R}^K$ defined in Section 1.2.3 instead of the terms $\mathcal{E}$ and $\mathcal{R}$ in the definition of the transform $\mathcal{T}_\sharp$, thus obtaining the transform $\mathcal{T}_\sharp^K$ mapping $\lambda_{n\forall}$ to $\lambda_{cp\forall}$. Composing the CPS transform with the transform $\mathcal{T}_\sharp^K$, we get the modified CPS transform from λ_{cp} to $\lambda_{cp\forall}$. The complete definition of the modified CPS transform is given in the Table 1.12 below. By the axioms of λ_{cp}, we have that $\mathcal{R}^K \circ \mathcal{E}^K = \mathtt{id}$, and since

$$
\begin{aligned}
\overline{x}^\sharp &\stackrel{\text{def}}{=} (\mathcal{E}^K\ x) \\
\overline{(\lambda x.M)}^\sharp &\stackrel{\text{def}}{=} (\mathcal{E}^K\ \lambda x.\overline{M}^\sharp) \\
\overline{(M\ N)}^\sharp &\stackrel{\text{def}}{=} ((\mathcal{R}^K\ \overline{M}^\sharp)\ (\mathcal{R}^K\ \overline{N}^\sharp)) \\
\overline{(\mathtt{let}\ x\mathtt{=}M\ \mathtt{in}\ N)}^\sharp &\stackrel{\text{def}}{=} (\mathtt{let}\ x\mathtt{=}(\mathcal{R}^K\ \overline{M}^\sharp)\ \mathtt{in}\ \overline{N}^\sharp)
\end{aligned}
\qquad
\begin{aligned}
\mathcal{E}^K &\stackrel{\text{def}}{=} \lambda xk.k\ x \\
\mathcal{R}^K &\stackrel{\text{def}}{=} \lambda x.x\ (\lambda z.z)
\end{aligned}
$$

$$
\begin{aligned}
K(\sigma) &\stackrel{\text{def}}{=} \forall t.(\sigma\to t)\to t \\
\iota' &\stackrel{\text{def}}{=} \iota \\
(\sigma\to\tau)' &\stackrel{\text{def}}{=} \sigma'\to K(\tau')
\end{aligned}
\qquad
\begin{aligned}
\{x_i\colon\sigma_i\}' &\stackrel{\text{def}}{=} \{x_i\colon\sigma_i'\} \\
\overline{\Gamma \rhd_{cp} M\colon\sigma}^\sharp &\stackrel{\text{def}}{=} \Gamma' \rhd_{cp} \overline{M}^\sharp\colon K(\sigma')
\end{aligned}
$$

Table 1.12: The modified CPS transform.

this is the only property of the terms $\mathcal{E}$ and $\mathcal{R}$ we used in reasoning about the modified computational transform, the results of Section 1.3.2 apply to the modified CPS transform as well.

1.4 Retractions

In this section we will show that each type σ is a retract of the type $T(\sigma')$, and also that σ is a retract of the type $K(\sigma')$. In a lambda calculus, we have the following definition of a retract:

Definition 1.4.1 ([MW85]). Let λ be a lambda calculus. A λ type σ is a *retract* of a λ type τ, if there are closed λ terms $E_{\sigma,\tau}$ (called embedding) and $R_{\sigma,\tau}$ (called retraction), such that $\rhd E_{\sigma,\tau}\colon\sigma\to\tau$, $\rhd R_{\sigma,\tau}\colon\tau\to\sigma$, and $\lambda \vdash R_{\sigma,\tau} \circ E_{\sigma,\tau} = \mathtt{id}$.

In particular we want to show there is a retraction-embedding pair (R, E) such that R is a definable (in λ_c) inverse of the modified computational transform. Similarly, we want to find a retraction-embedding pair (R^K, E^K) such that R^K is a definable (in $\lambda_{cp\forall}$) inverse of the CPS transform. Therefore we need to show that any λ_{cp} type σ is a retract of the type $T(\sigma')$, and also that σ is a retract of the type $K(\sigma')$.

Recall that in Section 1.2.3, we have defined the terms $\mathcal{E}: \sigma \to T(\sigma)$ and $\mathcal{R}: T(\sigma) \to \sigma$ and showed that $\lambda_c \vdash \mathcal{R} \circ \mathcal{E} = \mathtt{id}$. Therefore, it follows immediately that any λ_c type σ is a retract of the type $T(\sigma)$.

It remains to show that each type σ is a retract of σ'. To do so we define the functions $i_\sigma : \sigma \to \sigma'$ and $j_\sigma : \sigma' \to \sigma$ inductively on the structure of σ:

$$\begin{array}{rcl@{\qquad}rcl} i_\imath & \stackrel{\text{def}}{=} & \mathtt{id} & j_\imath & \stackrel{\text{def}}{=} & \mathtt{id} \\ i_{\sigma\to\tau} & \stackrel{\text{def}}{=} & \lambda f.\mathcal{E} \circ i_\tau \circ f \circ j_\sigma & j_{\sigma\to\tau} & \stackrel{\text{def}}{=} & \lambda g.j_\tau \circ \mathcal{R} \circ g \circ i_\sigma \end{array}$$

It is easy to show that functions j_σ and i_σ indeed form retraction-embedding pairs.

Lemma 1.4.2. For all σ, $\lambda_c \vdash j_\sigma \circ i_\sigma = \mathtt{id}$.

Proof: For $\sigma = \imath$ the lemma follows trivially. For higher order types we have

$$\begin{array}{lll} ((j_{\sigma\to\tau} \circ i_{\sigma\to\tau})\ x) = (j_{\sigma\to\tau}\ (i_{\sigma\to\tau}\ x)) & & \\ \quad = \ ((\lambda g.j_\tau \circ \mathcal{R} \circ g \circ i_\sigma)\ ((\lambda f.\mathcal{E} \circ i_\tau \circ f \circ j_\sigma)\ x)) & & \text{by definitions} \\ \quad = \ j_\tau \circ \mathcal{R} \circ (\mathcal{E} \circ i_\tau \circ x \circ j_\sigma) \circ i_\sigma & & \text{by } (\beta_v) \\ \quad = \ x & & \text{by lemma 1.2.9, (r-e),} \\ & & \text{and induction hypothesis} \end{array}$$

■

Now define $E_{\sigma,T(\sigma')} = \mathcal{E} \circ i_\sigma$ and $R_{\sigma,T(\sigma')} = j_\sigma \circ \mathcal{R}$, and from the above lemma, it follows that σ is a retract of $T(\sigma')$. Moreover, in the next section we will show that $R_{\sigma,T(\sigma')}$ is the inverse of the modified computational transform at type σ, that is

$$\lambda_c \vdash (R_{\sigma,T(\sigma)}\ \overline{\overline{M}}^\sharp) = M$$

for closed M of type σ.

To obtain the retraction-embedding pair showing that σ is a retract of $K(\sigma')$ we simply use $\mathcal{E}^K = \lambda xk.k\ x$ and $\mathcal{R}^K = \lambda x.x\ \mathtt{id}$ instead of $\mathcal{E}$ and $\mathcal{R}$ in all the definitions above. It is immediate that the resulting functions, name them R^K and E^K, are a retraction-embedding pair, since in the proof of Lemma 1.4.2 we only used axioms of λ_{cp} and the equation $\mathcal{R} \circ \mathcal{E} = \mathtt{id}$, and we have seen that $\lambda_{cp} \vdash \mathcal{R}^K \circ \mathcal{E}^K = \mathtt{id}$. In the next section we will also show that R^K is a definable inverse of the modified CPS transform, as well as an inverse of the original CPS transform.

Remark. We have defined different pairs of functions i_σ and j_σ, for each type σ. However, when clear from context, we may omit writing the subscripts, keeping in mind that i (as well as j) may denote a different function depending on the particular context in which it is used. We should also point out, that whenever we talk about the modified computational transform, we refer to definitions of i and j using functions $\mathcal{E}$ and $\mathcal{R}$, and when we talk about the CPS transform, we will use the same notation to denote functions i and j defined using terms $\mathcal{E}^K$ and $\mathcal{R}^K$ instead.

1.5 The Retraction Theorem

The main result of this work is a stronger version of the Meyer-Wand Retraction Theorem. First we prove the Retraction Theorem for the modified computational transform (Theorem 1.5.11). We use the proof of this theorem as a template to prove the Retraction Theorem for particular transforms of interest, namely for the original CPS transform (Theorem 1.5.18), and for the *state passing style* transform (Theorem 1.5.19).

In our version, the conclusion of these theorems holds by convertibility in λ_c, $\lambda_{cp\forall}$, and $\lambda_{cp\times}$ respectively. Therefore, throughout this section, unless specified otherwise, whenever we write $M = N$, when considering the modified computational transform we assume the equality is provable in λ_c, when discussing the results for the CPS transform, we assume the equality is provable in $\lambda_{cp\forall}$, and when considering the state passing style transform we assume provability in $\lambda_{cp\times}$.

1.5.1 Interpretation of terms and types

To prove the Retraction Theorem, as discussed in the introduction, we define a framework similar to Mitchell's type inference model. We define interpretations of terms and types, as well as, a notion of validity of typing sequents with respect to these interpretations.

The key elements needed in the proof of the Retraction Theorem can be summarized as follows. For each of the transforms, $\mathcal{T}_{ml\sharp}$, $\mathcal{T}_{cps}$, etc., we define: *type sets*, interpretation of types as type sets, interpretation of terms using the transform, an ε-relation between meanings of terms and types, and a notion of *pseudo-application* for the terms in the codomain of the transform.

In this section we give the definitions suitable for the modified computational transform, but explain how to define analogous concepts for other transforms. By examining the differences in the definitions, we observe that the following conditions need to be re-proved for each of the transforms. Let $\textsc{App}(M, N)$ denote the pseudo-application, and let $[\![\sigma]\!]$ be the meaning of σ. Then for all terms M and N in the codomain of the transform we require, that if $M \,\varepsilon\, [\![\sigma \to \tau]\!]$ and $N \,\varepsilon\, [\![\sigma]\!]$, then

$$\textsc{App}(M, N) \,\varepsilon\, [\![\tau]\!] \tag{†}$$

$$j_\tau(\mathcal{R}(\textsc{App}(M, N))) = (j_{\sigma\to\tau}(\mathcal{R}(M))\; j_\sigma(\mathcal{R}(N))). \tag{‡}$$

We now define these key concepts for the modified computational transform and show that above conditions are satisfied.

Definition 1.5.1. Let ρ be a substitution. For each of the transforms, $\mathcal{T}_{ml\sharp}$, $\mathcal{T}_{cps\sharp}$ and $\mathcal{T}_{cps}$, we define an interpretation of a term M, relative to a particular transform, to be:

$$\begin{array}{rcl} \mathcal{T}_{ml\sharp}[\![M]\!]\rho & \stackrel{\mathrm{def}}{=} & \rho\,\overline{\overline{M}}^{\sharp} \\ \mathcal{T}_{cps\sharp}[\![M]\!]\rho & \stackrel{\mathrm{def}}{=} & \rho\,\overline{M}^{\sharp} \\ \mathcal{T}_{cps}[\![M]\!]\rho & \stackrel{\mathrm{def}}{=} & \rho\,\overline{M} \end{array}$$

When the transform used is clear from the context, we will simply write $[\![M]\!]\rho$.

Types are interpreted as certain sets of terms, the so called *type sets*.

Definition 1.5.2. Let $\mathcal{D}$ denote the set $\{M \mid M = \mathcal{E}(V)$ for some value $V\}$. A *type set* is any subset of $\mathcal{D}$.

Remark. We understand the above definition as parameterized by the term $\mathcal{E}$, so, for example, when we speak about the CPS transform, we will use term $\mathcal{E}^K$ instead.

Definition 1.5.3. Let M be a term and let S be a type set. Write $M \mathrel{\varepsilon} S$ iff there is a term $\mathcal{E}(V)$ in S such that

$$M = (\texttt{let } x_1\texttt{=}P_1, \ldots, x_k\texttt{=}P_k \texttt{ in } \mathcal{E}(V)),$$

where $x_1, \ldots, x_k$ is the sequence containing of all free variables in V, and P_i are arbitrary terms.

Remark. For simplicity, when $M \mathrel{\varepsilon} S$, we will assume that $M = (\texttt{let } x\texttt{=}P \texttt{ in } I(V))$ and that x is the only free variable in V.

The intuition behind the ε-relation is to extend the membership relation. In other words, M is ε-related to a set S, if a "value" of M is a member of S (note that a "value" of M may not be unique). A "value" represents, for our purposes, important part of a term. The definition of ε-relation will be different for each transform, based on the structure of the transformed terms.

Definition 1.5.4. The interpretation of type σ is a type set defined by induction on the structure of σ as follows:

- $[\![\imath]\!]$ is the set of all terms of the form $\mathcal{E}(V)$ for some value V,
- $[\![\sigma{\to}\tau]\!]$ is a set of all terms $\mathcal{E}(V)$ such that for all $\mathcal{E}(W) \in [\![\sigma]\!]$,

$$(V\ W) \mathrel{\varepsilon} [\![\tau]\!] \tag{P.1}$$

$$j_\tau(\mathcal{R}(V\ W)) = (j_{\sigma\to\tau}(V)\ j_\sigma(W)). \tag{P.2}$$

Remark. The preceding definition of the interpretation of types uses terms $\mathcal{E}$ and $\mathcal{R}$, as well as the ε-relation. As in the definition of type sets, we will treat terms $\mathcal{E}$ and $\mathcal{R}$, as well as the relation ε, as parameters, and when discussing, for example, the CPS transform, we will assume terms $\mathcal{E}^K$ and J^K, and relation ε^K were used instead.

In general, the pseudo-application, App, in the codomain of a particular transform, $\mathcal{T}$, will be defined so that

$$\mathcal{T}(M\ N) = \textsc{App}(\mathcal{T}(M), \mathcal{T}(N)).$$

In particular, from the definition of $\mathcal{T}_{ml\sharp}$, we know that

$$\overline{\overline{(M\ N)}}^{\sharp} = (\mathcal{R}(\overline{\overline{M}}^{\sharp})\ \mathcal{R}(\overline{\overline{N}}^{\sharp})),$$

and for arbitrary terms in the codomain of $\mathcal{T}_{ml\sharp}$, we have the following definition.

Definition 1.5.5. The *pseudo-application* in the codomain of $\mathcal{T}_{ml\sharp}$ is defined to be:

$$\textsc{App}(M, N) \stackrel{\text{def}}{=} (\mathcal{R}(M)\ \mathcal{R}(N)).$$

Definition 1.5.4 is motivated by the two conditions, (†) and (‡), which we now prove in lemmas 1.5.6 and 1.5.7 below. The property (P.1) ensures (†) holds, i.e., that type sets $[\![\sigma]\!]$ are "closed" under App. Namely we have,

Lemma 1.5.6. For any two terms M and N,

$$M \mathrel{\varepsilon} [\![\sigma\to\tau]\!] \text{ and } N \mathrel{\varepsilon} [\![\sigma]\!] \quad \text{implies} \quad \textsc{App}(M, N) \mathrel{\varepsilon} [\![\tau]\!].$$

Proof: Since $M \mathrel{\varepsilon} [\![\sigma\to\tau]\!]$ and $N \mathrel{\varepsilon} [\![\sigma]\!]$, for some $\mathcal{E}(V) \in [\![\sigma\to\tau]\!]$ and $\mathcal{E}(W) \in [\![\sigma]\!]$, $M = (\texttt{let } x\texttt{=}P \texttt{ in } \mathcal{E}(V))$ and $N = (\texttt{let } y\texttt{=}Q \texttt{ in } \mathcal{E}(W))$. Therefore,

$$\begin{array}{rcll}
\textsc{App}(M, N) & = & (\mathcal{R}(M)\ \mathcal{R}(N)) & \text{by defn.} \\
& = & (\mathcal{R}(\texttt{let } x\texttt{=}P \texttt{ in } \mathcal{E}(V))\ \mathcal{R}(\texttt{let } y\texttt{=}Q \texttt{ in } \mathcal{E}(W))) & \text{by assumption} \\
& = & ((\texttt{let } x\texttt{=}P \texttt{ in } \mathcal{R}(\mathcal{E}(V)))\ (\texttt{let } y\texttt{=}Q \texttt{ in } \mathcal{R}(\mathcal{E}(W)))) & \text{by (C-appl.1)} \\
& = & ((\texttt{let } x\texttt{=}P \texttt{ in } V)\ (\texttt{let } y\texttt{=}Q \texttt{ in } W)) & \text{by (r-e)} \\
& = & (\texttt{let } x\texttt{=}P \texttt{ in } (\texttt{let } y\texttt{=}Q \texttt{ in } (V\ W))) & \text{by (C-appl.2)} \\
& = & (\texttt{let } x\texttt{=}P \texttt{ in } (\texttt{let } y\texttt{=}Q \texttt{ in } (\texttt{let } z\texttt{=}R \texttt{ in } \mathcal{E}(U)))) & \text{by (P.1)}
\end{array}$$

where $\mathcal{E}(U) \in [\![\tau]\!]$ as desired. ∎

The property (P.2) ensures that (‡) holds, that is, on "proper" terms, $j \circ \mathcal{R}$ is a homomorphism with respect to pseudo-application in the codomain of $\mathcal{T}_{ml\sharp}$ and the application in the domain of $\mathcal{T}_{ml\sharp}$.

Lemma 1.5.7. Assume $M \mathrel{\varepsilon} [\![\sigma\to\tau]\!]$ and $N \mathrel{\varepsilon} [\![\sigma]\!]$. Then,

$$j_\tau(\mathcal{R}(\textsc{App}(M, N))) = (j_{\sigma\to\tau}(\mathcal{R}(M))\ j_\sigma(\mathcal{R}(N))).$$

Proof: Since $M \mathrel{\varepsilon} [\![\sigma\to\tau]\!]$ and $N \mathrel{\varepsilon} [\![\sigma]\!]$, for some $\mathcal{E}(V) \in [\![\sigma\to\tau]\!]$ and $\mathcal{E}(W) \in [\![\sigma]\!]$, $M =$

(let x=P in $\mathcal{E}(V)$) and $N =$ (let y=Q in $\mathcal{E}(W)$). Therefore,

$$\begin{array}{lll} j(\mathcal{R}(\textsc{App}(M,N))) = j(\mathcal{R}(\mathcal{R}(M)\ \mathcal{R}(N))) & & \text{by defn.} \\ \quad = j(\mathcal{R}(\texttt{let}\ x\texttt{=}P\ \texttt{in}\ (\texttt{let}\ y\texttt{=}Q\ \texttt{in}\ (V\ W)))) & & \text{as in Lemma 1.5.6} \\ \quad = (\texttt{let}\ x\texttt{=}P\ \texttt{in}\ (\texttt{let}\ y\texttt{=}Q\ \texttt{in}\ j(\mathcal{R}(V\ W)))) & & \text{by (C-appl.1)} \\ \quad = (\texttt{let}\ x\texttt{=}P\ \texttt{in}\ (\texttt{let}\ y\texttt{=}Q\ \texttt{in}\ (j(V)\ j(W)))) & & \text{by (P.2)} \\ \quad = ((\texttt{let}\ x\texttt{=}P\ \texttt{in}\ j(V))\ (\texttt{let}\ y\texttt{=}Q\ \texttt{in}\ j(W))) & & \text{by (C-appl.2)} \\ \quad = ((\texttt{let}\ x\texttt{=}P\ \texttt{in}\ j(\mathcal{R}(\mathcal{E}(V))))\ (\texttt{let}\ y\texttt{=}Q\ \texttt{in}\ j(\mathcal{R}(\mathcal{E}(W))))) & & \text{by (r-e)} \\ \quad = (j(\mathcal{R}(\texttt{let}\ x\texttt{=}P\ \texttt{in}\ \mathcal{E}(V)))\ j(\mathcal{R}(\texttt{let}\ y\texttt{=}Q\ \texttt{in}\ \mathcal{E}(W)))) & & \text{by (C-appl.1)} \\ \quad = (j(\mathcal{R}(M))\ j(\mathcal{R}(N))) & & \text{by assumption.} \end{array}$$

Note that we used the fact that $j(V)$ and $j(W)$ are values, which follows directly from the definition of j's. ■

Our goal is to prove validity of derivable typing sequents. To do so we define notions of truth and validity as follows.

Definition 1.5.8. Given a substitution ρ, a type assertion $M{:}\,\sigma$ is *true* if $[\![M]\!]\rho\ \varepsilon\ [\![\sigma]\!]$; $M{:}\,\sigma$ is *valid* if it is true for every substitution; and ρ is said to *satisfy* a typing hypothesis Γ, if every type assertion in Γ is true with respect to ρ. Finally, a typing sequent $\Gamma \rhd M{:}\,\sigma$ is *valid* if $M{:}\,\sigma$ is true for every substitution that satisfies Γ.

Remark. As with the other definitions of this section, we understand the definition of validity as parameterized by the ε-relation, and when speaking about other transforms, we assume the notion of validity is defined with respect to appropriate version of the ε-relation.

1.5.2 The Retraction Theorem for the modified computational transform

The main result of this paper, the Retraction Theorem, is a direct corollary of the Soundness Lemma below. The lemma says that every derivable typing sequent is valid. Moreover, the lemma includes an additional assumption, (S.2), which is the statement of the Retraction Theorem generalized to open terms. The structure of type sets interpreting types is such that it tightly couples the two conditions (S.1) and (S.2). Before we state and prove the Soundness Lemma, we prove an auxiliary result that is needed later in the proof.

Lemma 1.5.9. For each type σ, $\mathcal{E}(i_\sigma(V)) \in [\![\sigma]\!]$.

Proof: By induction on the structure of σ. For base type $\imath$, $\mathcal{E}(i_\imath(V)) = \mathcal{E}(V)$ is in $[\![\imath]\!]$ by definition. For higher order types we need to show (P.1) and (P.2). Let $\mathcal{E}(W) \in [\![\sigma]\!]$.

$$\begin{array}{rcll} (i_{\sigma\to\tau}(V)\ W) & = & \mathcal{E}(i_\tau(V\ j_\sigma(W))) & \text{by defn. of } i_{\sigma\to\tau} \\ & = & \mathcal{E}(\texttt{let}\ x\texttt{=}(V\ j_\sigma(W))\ \texttt{in}\ i_\tau(x)) & \text{by (let.2)} \\ & = & (\texttt{let}\ x\texttt{=}(V\ j_\sigma(W))\ \texttt{in}\ \mathcal{E}(i_\tau(x))) & \text{by (C-appl.1)} \end{array}$$

This shows (P.1), since $\mathcal{E}(i_\tau(x))$ is in $[\![\tau]\!]$ by the induction hypothesis. (P.2) can be shown by computing

$$\begin{array}{rcll} j_\tau(\mathcal{R}(i_{\sigma\to\tau}(V)\ W)) & = & j_\tau(\mathcal{R}(\mathcal{E}(i_\tau(V\ j_\sigma(W))))) & \text{by defn. of } i_{\sigma\to\tau} \\ & = & (V\ j_\sigma(W)) & \text{by (r-e) and Lemma 1.4.2} \\ & = & (j_{\sigma\to\tau}(i_{\sigma\to\tau}(V))\ j_\sigma(W)) & \text{by Lemma 1.4.2.} \end{array}$$

Thus, $\mathcal{E}(i_{\sigma\to\tau}(V))$ is in $[\![\sigma\to\tau]\!]$. ■

Lemma 1.5.10 (Soundness). Assume Γ is a typing hypothesis such that $\Gamma \triangleright_{cp} M\colon\sigma$. Let ρ be a substitution satisfying Γ, and let ρ' be a substitution such that for each $x_i\colon\sigma_i \in \Gamma$, $\rho'(x_i) = j_{\sigma_i}(\rho(x_i))$. Then,

$$\mathcal{T}_{ml\sharp}[\![M]\!]\rho \;\varepsilon\; [\![\sigma]\!] \tag{S.1}$$

$$j_\sigma(\mathcal{R}(\rho\overline{\overline{M}}^{\sharp})) = \rho' M \tag{S.2}$$

Proof: By induction on the type derivation of $\Gamma \triangleright_{cp} M\colon\sigma$. Note that (S.2) implies that

$$j(\mathcal{R}(\rho\,\overline{\overline{M}}^{\sharp})) = j(\mathcal{R}(\rho\{i(j(x))/x\}\overline{\overline{M}}^{\sharp})). \tag{S.3}$$

Case (var): Assume $\Gamma \triangleright_{cp} x\colon\sigma$. This implies that $x\colon\sigma \in \Gamma$. To prove (S.1) we need to show that $[\![x]\!]\rho \;\varepsilon\; [\![\sigma]\!]$, which is true for any ρ that satisfies Γ. To show (S.2) we compute

$$\begin{array}{rcll}
j_\sigma(\mathcal{R}(\rho(\overline{\overline{x}}^{\sharp}))) & = & j_\sigma(\mathcal{R}(\rho(\mathcal{E}(x)))) & \text{by defn. of } \overline{\overline{x}}^{\sharp} \\
& = & j_\sigma(\mathcal{R}(\mathcal{E}(\rho(x)))) & \text{since } \mathcal{E} \text{ is closed} \\
& = & j_\sigma(\rho(x)) & \text{by (r-e)} \\
& = & \rho'(x) & \text{by defn. of } \rho'.
\end{array}$$

Case (app): Assume $\Gamma \triangleright_{cp} M\colon\sigma\to\tau$ and $\Gamma \triangleright_{cp} N\colon\sigma$ so that $\Gamma \triangleright_{cp} (M\ N)\colon\tau$. Let ρ be a substitution that satisfies Γ. (S.1): By induction hypothesis $[\![M]\!]\rho \;\varepsilon\; [\![\sigma\to\tau]\!]$ and $[\![N]\!]\rho \;\varepsilon\; [\![\sigma]\!]$. By definition,

$$[\![(M\ N)]\!]\rho = (\mathcal{R}([\![M]\!]\rho)\ \mathcal{R}([\![N]\!]\rho)) = \text{App}([\![M]\!]\rho, [\![N]\!]\rho)$$

and, by Lemma 1.5.6, $\text{App}([\![M]\!]\rho, [\![N]\!]\rho) \;\varepsilon\; [\![\tau]\!]$. (S.2): By induction hypothesis,

$$j(\mathcal{R}([\![M]\!]\rho)) = \rho' M \quad \text{and} \quad j(\mathcal{R}([\![N]\!]\rho)) = \rho' N,$$

and by Lemma 1.5.7,

$$j(\mathcal{R}(\text{App}([\![M]\!]\rho, [\![N]\!]\rho))) = (j(\mathcal{R}([\![M]\!]\rho))\ j(\mathcal{R}([\![N]\!]\rho))) = (\rho' M\ \rho' N) = \rho'(M\ N).$$

Case (abs): Assume $\Gamma, x\colon\sigma \triangleright_{cp} M\colon\tau$ so that $\Gamma \triangleright_{cp} \lambda x.M\colon\sigma\to\tau$. To show (S.1), first observe that

$$\rho\,\overline{\overline{\lambda x.M}}^{\sharp} = \rho(\mathcal{E}(\lambda x.\overline{\overline{M}}^{\sharp})) = \mathcal{E}(\rho(\lambda x.\overline{\overline{M}}^{\sharp})),$$

so that $\rho\,\overline{\overline{\lambda x.M}}^{\sharp}$ is of the form $\mathcal{E}(V)$, and thus, we need to show that $\rho\,\overline{\overline{\lambda x.M}}^{\sharp} \in [\![\sigma\to\tau]\!]$. Let $\mathcal{E}(W) \in [\![\sigma]\!]$, and show (P.1) and (P.2). We have

$$\begin{array}{rcll}
(\rho(\lambda x.\overline{\overline{M}}^{\sharp})\ W) & = & ((\lambda x.\rho\{x/x\}\overline{\overline{M}}^{\sharp})\ W) & \text{by defn. of subst.} \\
& = & \rho\{W/x\}\overline{\overline{M}}^{\sharp} & \text{by } (\beta_v).
\end{array}$$

Since $\mathcal{E}(W) \in [\![\sigma]\!]$, $\rho\{W/x\}$ satisfies $\Gamma, x{:}\,\sigma$, so by the induction hypothesis (S.1), $\rho\{W/x\}\overline{\overline{M}}^{\sharp} \;\varepsilon\; [\![\tau]\!]$. This establishes (P.1). To show (P.2) compute

$$\begin{array}{rcll} j_\tau(\mathcal{R}(\rho(\lambda x.\overline{\overline{M}}^{\sharp})\; W)) & = & j_\tau(\mathcal{R}(\rho\{W/x\}\overline{\overline{M}}^{\sharp})) & \text{by } (\beta_v) \\ & = & j_\tau(\mathcal{R}(\rho\{i_\sigma(j_\sigma(W))/x\}\overline{\overline{M}}^{\sharp})) & \text{by ind. hyp. (S.2) and (S.3)} \\ & = & j_\tau(\mathcal{R}(\rho(\lambda x.\overline{\overline{M}}^{\sharp})\; i_\sigma(j_\sigma(W)))) & \text{by } (\beta_v) \\ & = & (j_{\sigma\to\tau}(\rho(\lambda x.\overline{\overline{M}}^{\sharp}))\; j_\sigma(W)) & \text{by defn. of } j_{\sigma\to\tau}. \end{array}$$

(S.2) follows by direct computation and Lemma 1.5.9.

$$\begin{array}{rcll} j(\mathcal{R}(\rho\,\overline{\overline{\lambda x.M}}^{\sharp})) & = & j(\lambda x.\rho\{x/x\}\overline{\overline{M}}^{\sharp}) & \text{by defn's and (r-e)} \\ & = & \lambda x.j(\mathcal{R}(\rho\{i(x)/x\}\overline{\overline{M}}^{\sharp})) & \text{by defn. of } j_{\sigma\to\tau} \\ & = & \lambda x.\rho'\{j(i(x))/x\}M & \text{by Lemma 1.5.9 and ind. hyp. (S.2)} \\ & = & \rho'(\lambda x.M) & \text{by Lemma 1.4.2 and defn. of subst.} \end{array}$$

■

Theorem 1.5.11 (Retraction, for $\mathcal{T}_{ml^\sharp}$). Let M be closed such that $\triangleright_{cp} M{:}\,\sigma$. Then,

$$\lambda_c \vdash M = j_\sigma(\mathcal{R}(\overline{\overline{M}}^{\sharp})).$$

Proof: Follows immediately from Lemma 1.5.10. ■

1.5.3 Back to CPS

We now focus on continuations and the modified, as well as original, CPS transform.

The modified transform

Observe that in the proof of the Soundness Lemma and the Retraction Theorem we only used abstract properties of functions $\mathcal{E}$ and $\mathcal{R}$, namely that $\mathcal{R} \circ \mathcal{E} = \mathtt{id}$. Therefore, if $\mathcal{E}$ and $\mathcal{R}$ are replaced by any functions $\mathcal{E}'$ and $\mathcal{R}'$ (of the correct type) for which $\mathcal{R}' \circ \mathcal{E}' = \mathtt{id}$, our results will still hold. In particular, recall that, in the definition of the modified CPS transform, $\mathcal{E}^K$ and $\mathcal{R}^K$ were defined as $\lambda xk.k\ x$ and $\lambda x.x\ \mathtt{id}$ respectively, and it follows that $\lambda_{cp\forall} \vdash \mathcal{R}^K \circ \mathcal{E}^K = \mathtt{id}$. This gives us the retraction result for the modified CPS transform as a direct corollary of Lemma 1.5.10.

Corollary 1.5.12. Let M be a λ_{cp} term, and let Γ be a typing hypothesis such that $\Gamma \triangleright_{cp} M{:}\,\sigma$. Let ρ be a substitution satisfying Γ and let $\rho'(x_i) = j_{\sigma_i}(\rho(x_i))$ for each $x_i{:}\,\sigma_i \in \Gamma$. Then,

$$\mathcal{T}_{cps^\sharp}[\![M]\!]\rho \;\varepsilon\; [\![\sigma]\!], \qquad (\mathrm{S.1}_K)$$

and

$$\lambda_{cp\forall} \vdash \rho' M = j_\sigma(\mathcal{R}^K(\rho\,\overline{M}^\sharp)). \qquad (\mathrm{S.2}_K)$$

In particular, if M is closed

$$\lambda_{cp\forall} \vdash M = j_\sigma(\mathcal{R}^K(\overline{M}^\sharp)).$$

The original transform

Unfortunately, the modified CPS transform differs slightly from the original CPS transform, and the above corollary does not apply. Nevertheless, we can salvage the retraction result by examining similarities and differences between the CPS transform and its modified counterpart, and adjusting the definitions of the ε-relation and pseudo-application appropriately. Looking at the transforms of an application one can see that if $\overline{M} = \mathcal{E}(\mathcal{R}(\overline{M}))$ and $\overline{N} = \mathcal{E}(\mathcal{R}(\overline{N}))$[11] we have

$$\overline{(M\ N)} = \lambda k.(\mathcal{R}(\overline{M})\ \mathcal{R}(\overline{N})\ k).$$

It will follow that for a proper substitution ρ, $\rho\,\overline{M} = \mathcal{E}(\mathcal{R}(\rho\,\overline{M}))$, and therefore that the the modified and original CPS terms only differ by (η) with respect to the continuation variable k, i.e. for a proper substitution, ρ, $\lambda k.(\rho\,\overline{M}^\sharp\ k) = \rho\overline{M}$. However, since $\rho\,\overline{M}^\sharp$ may not be a value, this equality may not hold in $\lambda_{cp\forall}$. The role of the ε-relation is to separate essential form non-essential (for our purposes) structure of a term. Thus we can modify the ε-relation to hide the extra structure introduced by the CPS transform. The new ε-relation is defined as follows.

Definition 1.5.13. For a term M and a type set S write

$$M\ \varepsilon^K\ S \quad \text{iff} \quad M = \lambda k.((\texttt{let } x_1\texttt{=}P_1,\ldots,x_k\texttt{=}P_k \texttt{ in } \mathcal{E}(V))\ k)$$

for some $\mathcal{E}(V) \in S$, and where k is a fresh variable and $x_1,\ldots,x_k$ include all free variables in V. As before we write $M = \lambda k.((\texttt{let } x\texttt{=}P \texttt{ in } I(V))\ k)$ whenever $M\ \varepsilon^K\ S$. Note that $M\ \varepsilon^K\ S$ iff $\lambda k.(M\ k)\ \varepsilon\ S$, assuming k is not free in M.

Remark. Recall that in the definition of the interpretation of types (definition 1.5.4) we used the ε-relation, as well as terms $\mathcal{E}$ and $\mathcal{R}$. In this section we assume $[\![\sigma]\!]$ to be defined using the ε^K relation and terms $\mathcal{E}^K$ and $\mathcal{R}^K$ instead. Similarly, we use the ε^K-relation in the definition of validity (definition 1.5.8).

The definition of the pseudo-application in the codomain of the CPS transform is derived from the transform of the application. Namely we have

$$\overline{(M\ N)} = \lambda k.\overline{M}\ (\lambda m.\overline{N}\ (\lambda n.m\ n\ k)),$$

and consequently we define $\textsc{App}^K$, so that $\overline{(M\ N)} = \textsc{App}^K(\overline{M},\overline{N})$.

[11] For the rest of this section we drop the superscripts from $\mathcal{E}^K$ and $\mathcal{R}^K$.

Definition 1.5.14. The pseudo-application in the codomain of the CPS transform is defined to be

$$\textsc{App}^K(M, N) \stackrel{\text{def}}{=} \lambda k.M\ (\lambda m.N\ (\lambda n.m\ n\ k)).$$

Having defined $\mathcal{T}_{cps}[\![M]\!]\rho$, $[\![\sigma]\!]$, $\textsc{App}^K$ and ε^K we need to show that these definitions fit together as before. Namely, it remains to prove that conditions (†) and (‡) hold. We do so in the two lemmas below.

Lemma 1.5.15. Let M and N be two terms such that $M\ \varepsilon^K\ [\![\sigma \to \tau]\!]$ and $N\ \varepsilon^K\ [\![\sigma]\!]$. Then,

$$\textsc{App}^K(M, N)\ \varepsilon^K\ [\![\tau]\!].$$

Proof: By assumption,

$$M = \lambda k.((\texttt{let}\ x_1\texttt{=}P_1\ \texttt{in}\ \mathcal{E}(V_1))\ k) \quad \text{and} \quad N = \lambda k.((\texttt{let}\ x_2\texttt{=}P_2\ \texttt{in}\ \mathcal{E}(V_2))\ k)$$

for some $\mathcal{E}(V_1) \in [\![\sigma \to \tau]\!]$ and $\mathcal{E}(V_2) \in [\![\sigma]\!]$. Thus we have

$$\begin{array}{lll}
\multicolumn{2}{l}{\lambda k.M\ (\lambda m.N\ (\lambda n.m\ n\ k)) =} & \\
= & \lambda k.(\lambda k.((\texttt{let}\ x_1\texttt{=}P_1\ \texttt{in}\ \mathcal{E}(V_1))\ k)) & \\
 & \quad (\lambda m.(\lambda k.((\texttt{let}\ x_2\texttt{=}P_2\ \texttt{in}\ \mathcal{E}(V_2))\ k))\ (\lambda n.m\ n\ k))) & \text{by assumption} \\
= & \lambda k.((\texttt{let}\ x_1\texttt{=}P_1\ \texttt{in}\ \mathcal{E}(V_1)) & \\
 & \quad (\lambda m.((\texttt{let}\ x_2\texttt{=}P_2\ \texttt{in}\ \mathcal{E}(V_2))\ (\lambda n.m\ n\ k))) & \text{by } (\beta_v) \\
= & \lambda k.(\mathcal{E}(\texttt{let}\ x_1\texttt{=}P_1\ \texttt{in}\ V_1) & \\
 & \quad (\lambda m.(\mathcal{E}(\texttt{let}\ x_2\texttt{=}P_2\ \texttt{in}\ V_2)\ (\lambda n.m\ n\ k))) & \text{by (C-appl.1)} \\
= & \lambda k.((\lambda x k'.k\ x)\ (\texttt{let}\ x_1\texttt{=}P_1\ \texttt{in}\ V_1) & \\
 & \quad (\lambda m.((\lambda x k'.k'\ x)\ (\texttt{let}\ x_2\texttt{=}P_2\ \texttt{in}\ V_2)\ (\lambda n.m\ n\ k))) & \text{by defn. of } \mathcal{E} \\
= & \lambda k.((\texttt{let}\ x_1\texttt{=}P_1\ \texttt{in}\ V_1)\ (\texttt{let}\ x_2\texttt{=}P_2\ \texttt{in}\ V_2)\ k) & \text{by (C-appl.3)} \\
 & & \text{and Lemma 1.2.8} \\
= & \lambda k.((\texttt{let}\ x_1\texttt{=}P_1\ \texttt{in}\ (\texttt{let}\ x_2\texttt{=}P_2\ \texttt{in}\ (V_1\ V_2)))\ k) & \text{by (C-appl.2)} \\
= & \lambda k.((\texttt{let}\ x_1\texttt{=}P_1\ \texttt{in}\ (\texttt{let}\ x_2\texttt{=}P_2\ \texttt{in} & \\
 & \quad\quad \lambda k'.((\texttt{let}\ x\texttt{=}Q\ \texttt{in}\ \mathcal{E}(W))\ k')))\ k) & \text{by (P.1)} \\
= & \lambda k.(\texttt{let}\ x_1\texttt{=}P_1\ \texttt{in}\ (\texttt{let}\ x_2\texttt{=}P_2\ \texttt{in}\ ((\texttt{let}\ x\texttt{=}Q\ \texttt{in}\ \mathcal{E}(W))\ k))) & \text{by (C-appl.2),} \\
 & & \text{and } (\beta_v) \\
= & \lambda k.((\texttt{let}\ x_1\texttt{=}P_1\ \texttt{in}\ (\texttt{let}\ x_2\texttt{=}P_2\ \texttt{in}\ (\texttt{let}\ x\texttt{=}Q\ \texttt{in}\ \mathcal{E}(W))))\ k) & \text{by (C-appl.2).}
\end{array}$$

■

Lemma 1.5.16. Assume $M\ \varepsilon^K\ [\![\sigma \to \tau]\!]$ and $N\ \varepsilon^K\ [\![\sigma]\!]$. Then,

$$j_\tau(\mathcal{R}(\textsc{App}^K(M, N))) = (j_{\sigma \to \tau}(\mathcal{R}(M))\ j_\sigma(\mathcal{R}(N))).$$

Proof: Assume that $M \; \varepsilon^K \; [\![\sigma\to\tau]\!]$ and $N \; \varepsilon^K \; [\![\sigma]\!]$ so that for $\mathcal{E}(V_1) \in [\![\sigma\to\tau]\!]$ and $\mathcal{E}(V_2) \in [\![\sigma]\!]$, $M = \lambda k.((\texttt{let } x_1\texttt{=}P_1 \texttt{ in } \mathcal{E}(V_1))\; k)$ and $N = \lambda k.((\texttt{let } x_2\texttt{=}P_2 \texttt{ in } \mathcal{E}(V_2))\; k)$.

$$
\begin{array}{lll}
j(\mathcal{R}(\textsc{App}^K(M,N))) = j(\mathcal{R}(\lambda k.M\ (\lambda m.N\ (\lambda n.m\ n\ k)))) & & \text{by defn.} \\
\quad = \ j(\mathcal{R}(\lambda k'.(\lambda k.(\texttt{let } x_1\texttt{=}P_1 \texttt{ in } \mathcal{E}(V_1))\ k) & & \\
\qquad\qquad (\lambda m.(\lambda k.(\texttt{let } x_2\texttt{=}P_2 \texttt{ in } \mathcal{E}(V_2))\ k)\ (\lambda n.m\ n\ k')))) & & \text{by assumption} \\
\quad = \ j(\mathcal{R}(\lambda k.(\texttt{let } x_1\texttt{=}P_1 \texttt{ in } (\texttt{let } x_2\texttt{=}P_2 \texttt{ in } (V_1\ V_2)))\ k)) & & \text{as in Lemma 1.5.15} \\
\quad = \ j(\mathcal{R}(\texttt{let } x_1\texttt{=}P_1 \texttt{ in } (\texttt{let } x_2\texttt{=}P_2 \texttt{ in } (V_1\ V_2)))) & & \text{by Lemma 1.2.8} \\
\quad = \ (\texttt{let } x_1\texttt{=}P_1 \texttt{ in } (\texttt{let } x_2\texttt{=}P_2 \texttt{ in } j(\mathcal{R}(V_1\ V_2)))) & & \text{by (C-appl.1)} \\
\quad = \ (\texttt{let } x_1\texttt{=}P_1 \texttt{ in } (\texttt{let } x_2\texttt{=}P_2 \texttt{ in } (j(V_1)\ j(V_2)))) & & \text{by (P.2)} \\
\quad = \ (j(\mathcal{R}(\texttt{let } x_1\texttt{=}P_1 \texttt{ in } \mathcal{E}(V_1)))\ j(\mathcal{R}(\texttt{let } x_2\texttt{=}P_2 \texttt{ in } \mathcal{E}(V_2)))) & & \text{by (r-e), and} \\
 & & \text{(C-appl.1)} \\
\quad = \ (j(\mathcal{R}(\lambda k.(\texttt{let } x_1\texttt{=}P_1 \texttt{ in } \mathcal{E}(V_1))\ k)) & & \\
\qquad\qquad j(\mathcal{R}(\lambda k.(\texttt{let } x_2\texttt{=}P_2 \texttt{ in } \mathcal{E}(V_2))\ k))) & & \text{by Lemma 1.2.8} \\
\quad = \ (j(\mathcal{R}(M))\ j(\mathcal{R}(N))) & & \text{by assumption}
\end{array}
$$

■

Using these results, we get a version of the Soundness Lemma (Lemma 1.5.10) for the original CPS transform.

Lemma 1.5.17. Assume Γ is a typing hypothesis such that $\Gamma \rhd_{cp} M\!:\sigma$. Let ρ be a substitution satisfying Γ, and let ρ' be the substitution such that, for each $x_i\!:\sigma_i \in \Gamma$, $\rho'(x_i) = j_{\sigma_i}(\rho(x_i))$. Then,

$$\mathcal{T}_{cps}[\![M]\!]\rho \; \varepsilon^K \; [\![\sigma]\!] \tag{S.1$'_K$}$$

$$j_\sigma(\mathcal{R}(\rho\,\overline{M})) = \rho' M \tag{S.2$_K$}$$

Proof: The (var) and (abs) cases are the same as in Lemma 1.5.10. The (app) case follows from lemmas 1.5.15 and 1.5.16. ■

Finally, as a direct corollary, we obtain the Retraction Theorem for the original CPS transform advertised in the introduction.

Theorem 1.5.18 (Retraction, for the CPS transform). Let M be a closed term such that $\rhd_{cp} M\!:\sigma$. Then,

$$\lambda_{cp\forall} \vdash M = j_\sigma(\mathcal{R}^K(\overline{M})).$$

Proof: Follows immediately from Lemma 1.5.17. ■

1.5.4 The *state passing style* transform

Analogous to the CPS transform, to give a functional interpretation of a language with mutable store, we define the *state passing style* (SPS) transform. In this section we will sketch how the Retraction Theorem can be proved for the SPS transform. As learned in the previous section, the key in translating the ideas from the abstract version of the Retraction Theorem for the modified computational transform is the definition of the proper ε-relation, as well as the definition of the pseudo-application such that conditions (†) and (‡) are satisfied.

The SPS transform, $\mathcal{T}_{sps}$, maps λ_{cp} to an extension of λ_{cp} with product types, denoted $\lambda_{cp\times}$. Assume S is a special type denoting the state of the store. For instance, S can be defined to be a function space $Loc \rightarrow Val$, where Loc denotes the type of locations and Val denotes the type of values that can be stored in the store. Note that the development we sketch below, does not depend on the actual implementation of a store, so we will think of S as representing an abstract mutable store.

The computations interpreting programs that may change the store are defined as terms of type $S \rightarrow (\sigma \times S)$. The SPS transform maps terms to computations, as given in Table 1.13. (Note: we write ($\texttt{let}\ \langle x, y\rangle\texttt{=}M\ \texttt{in}\ N$) to abbreviate ($\texttt{let}\ z\texttt{=}M, x\texttt{=}\pi_1(z), y\texttt{=}\pi_2(z)\ \texttt{in}\ N$).) To

$$\begin{aligned}
x^* &\stackrel{\text{def}}{=} \lambda s.\langle x, s\rangle \\
\lambda x.M^* &\stackrel{\text{def}}{=} \lambda s.\langle \lambda x.M^*, s\rangle \\
(M\ N)^* &\stackrel{\text{def}}{=} \lambda s.(\texttt{let}\ \langle x_1, s_1\rangle\texttt{=}(M^*\ s), \langle x_2, s_2\rangle\texttt{=}(N^*\ s_1)\ \texttt{in}\ (x_1\ x_2\ s_2)) \\
(\texttt{let}\ x\texttt{=}M\ \texttt{in}\ N)^* &\stackrel{\text{def}}{=} \lambda s.(\texttt{let}\ \langle x, s_1\rangle\texttt{=}(M^*\ s)\ \texttt{in}\ (N^*\ s_1))
\end{aligned}$$

$$\begin{aligned}
\imath' &\stackrel{\text{def}}{=} \imath & \{x_i\colon \sigma_i\}' &\stackrel{\text{def}}{=} \{x_i\colon \sigma_i'\} \\
(\sigma \rightarrow \tau)' &\stackrel{\text{def}}{=} \sigma' \rightarrow (S \rightarrow (\tau' \times S)) & (\Gamma \rhd_{cp} M\colon \sigma)^* &\stackrel{\text{def}}{=} \Gamma' \rhd_{cp\times} \overline{M}\colon S \rightarrow (\sigma' \times S)
\end{aligned}$$

Table 1.13: The SPS transform.

apply the proof of the Retraction Theorem for the modified computational transform to the SPS transform we define interpretations of terms and types, the pseudo-application, and the ε-relation. Before we do so, we need to define terms $\mathcal{E}^S$ and $\mathcal{R}^S$, the intuitive meaning of which is analogous to the meaning of $\mathcal{E}^K$ and $\mathcal{R}^K$ used in the Retraction Theorem for the CPS transform. We define

$$\mathcal{E}^S \stackrel{\text{def}}{=} \lambda x.\lambda s.\langle x, s\rangle \quad \text{and} \quad \mathcal{R}^S \stackrel{\text{def}}{=} \lambda x.(\pi_1(x\ \texttt{initial-state}),$$

where `initial-state` is a value of type S, denoting some initial state of the store. It follows easily, that $\mathcal{R}^S \circ \mathcal{E}^S = \texttt{id}$ is provable in $\lambda_{cp\times}$.

The interpretation of terms using the SPS transform is defined to be

$$\mathcal{T}_{sps}[\![M]\!]\rho \stackrel{\text{def}}{=} \rho(M^*).$$

The notion of type sets and interpretation of types, is the same as before, but using the definitions of $\mathcal{E}^S$, $\mathcal{R}^S$ and ε^S instead of $\mathcal{E}$, $\mathcal{R}$ and ε. Relation ε^S is defined as follows. If S is a type set and M is a term we write $M\ \varepsilon^S\ S$ iff there is a term P and a value V, such that $M = \lambda s.(\texttt{let}\ x\texttt{=}P\ \texttt{in}\ \langle V, s\rangle)$. Finally, the pseudo-application $\textsc{App}^S$ is defined to be

$$\textsc{App}^S(M, N) \stackrel{\text{def}}{=} \lambda s.(\texttt{let}\ \langle x_1, s_1\rangle\texttt{=}(M\ s), \langle x_2, s_2\rangle\texttt{=}(N\ s_1)\ \texttt{in}\ (x_1\ x_2\ s_2))).$$

It can be shown that, using the above definitions, conditions (†) and (‡) hold, and we obtain the Retraction Theorem for the SPS transform.

Theorem 1.5.19 (Retraction, for the SPS transform). For any type σ, and every closed term M, such that $\rhd_{cp\times} M : \sigma$,

$$\lambda_{cp\times} \vdash M = j_\sigma(\mathcal{R}(M^*)).$$

1.5.5 Extensions of the main results

Simply typed lambda calculus serves well as an elementary model of a functional programming language, but it is too simple to serve as a practical programming language. To make our results applicable to a real-life situation, we need to extend the Retraction Theorem to a more powerful language. There are two directions in which one can proceed. We can add additional language constructs to the language (such as numerals and primitive operations, if-then-else, recursion, etc.), or we can extend the type system of the language. We discuss some extensions using both approaches for which we have been able to prove the retraction theorem, as well as, those for which we only have partial results.

Adding new language constructs

Extending the Retraction Theorem to an extension of λ_{cp} with constants of a base type and primitive operators such as numerals is quite straightforward. However, adding arbitrary constants of higher order types may be more difficult. The difficulty lies in ensuring the closure conditions imposed on type sets by the addition of such constants are satisfied. For example if a constant $\mathtt{c}$ of type $\sigma \to \tau$ is added to λ_{cp}, we need to make sure that if $M\ \varepsilon\ \sigma$ then $\textsc{App}(\overline{\overline{\mathtt{c}}}^\sharp, M)\ \varepsilon\ \tau$. While such closure conditions are determined based on the type of new constants, the proof they are satisfied will, in general, depend on the functional behavior of the new constants.

Numerals and primitive operators: One of the simplest extensions of λ_{cp} is to add natural numbers together with primitive operators, such as successor and predecessor. This extension of λ_{cp} includes adding a new base type of natural numbers, as well as adding the constants to denote numerals and primitive operators. The Retraction Theorem, for any of the transforms we have discussed, can easily be extended to the extended calculus. The proof is given in Section 1.9.2.

Divergent element: The difference between call-by-name and call-by-value evaluation strategies becomes apparent only in presence of actual computational effects. So far we have only considered pure simply typed terms. In this setting every closed term is equivalent to a value in both logics of λ_{cp} and λ_n. Therefore, if we were to stop here, it would be unjustified to claim significant improvement over the original Meyer-Wand Retraction Theorem.

The simplest computational effect we can add to the language is divergence. In presence of divergence λ_n reasoning is no longer sound for call-by-value languages, so for any applications in λ_{cp} extended with divergence, we really need the stronger version of the Retraction Theorem provable in the weaker logic of λ_{cp}. While extension of the Retraction Theorem to a language with divergence, which we now present, is quite straightforward, it is important since it illustrates the practical difference between Meyer and Wand's and our formulation of the Retraction Theorem.

Divergence is represented by the divergent element, Ω, that is added to the language of λ_{cp} as a constant, but it is not considered a value. The axioms for Ω specify that an application diverges if either the operators or the operand diverges. Moreover, these axioms identify all divergent terms. The axioms are:

$$(\Omega\ M) = \Omega \tag{Ω.1}$$

$$(M\ \Omega) = \Omega. \tag{Ω.2}$$

One can verify that the resulting equational logic is consistent and that it cannot prove $\Omega = V$ for any value V. The type system is extended with the axiom

$$\Gamma \rhd_{cp} \Omega\colon \sigma \tag{Ω}$$

which says that Ω has every type. The modified computational transform is defined on Ω to be

$$\overline{\overline{\Omega}}^{\sharp} \stackrel{\text{def}}{=} \Omega.$$

Since Ω has every type, to prove the Retraction Theorem for $\lambda_{cp}{+}\Omega$ we need to extend Lemma 1.5.10 for the case of typing axiom (Ω). In other words we need to show (S.1): that $\overline{\overline{\Omega}}^{\sharp}\ \varepsilon\ [\![\sigma]\!]$, and (S.2): that $j_\sigma(\mathcal{R}(\overline{\overline{\Omega}}^{\sharp})) = \Omega$ for every type σ. The second condition follows trivially from the definition of $\overline{\overline{\Omega}}^{\sharp}$ and axiom (Ω.2). To prove the first condition observe that

$$\Omega = (\texttt{let}\ x\texttt{=}\Omega\ \texttt{in}\ \mathcal{E}(i(x))),$$

and by Lemma 1.5.9, $\mathcal{E}(i_\sigma(x)) \in [\![\sigma]\!]$ for every σ.

The very same reasoning can be applied to extend the Retraction Theorem for the CPS transform to $\lambda_{cp}{+}\Omega$. The CPS transform is defined on Ω to be

$$\overline{\Omega} \stackrel{\text{def}}{=} \lambda k.\Omega\ k.$$

Conditions (S.1$'_K$): that $\overline{\Omega}\ \varepsilon^K\ [\![\sigma]\!]$ for every type σ, and (S.2$_K$): that $j_\sigma(\mathcal{R}(\overline{\Omega})) = \Omega$, in Lemma 1.5.17 are proved as above.

Extending the type system

Many programming languages have more complex type systems than simple types, so to extend the applicability of our result we will discuss what is needed to extend the retraction result to a language with a richer type system.

A new type system will, in general, introduce new types, new typing rules, and sometimes new term constructors. First we need to extend the definitions of type sets $[\![\sigma]\!]$, as well as, functions j_σ and i_σ for each new type σ. This has to be done in such a way that

- $j_\sigma \circ i_\sigma = \mathtt{id}$,
- $j_\sigma(V)$ and $i_\sigma(V)$ are values, for any value V, and
- conditions (S.1) and (S.2) in Lemma 1.5.10 are preserved by the new type inference rules.

Moreover, if new term constructors are introduced, such as pairs and projections, one needs to extend the definition of the transform to include the new terms.

Products and sums: The retraction result can easily be extended to a language with products and/or sums. Let $\lambda_{cp\times}$ denote the extension of λ_{cp} with product types and similarly for λ_c and $\lambda_{cp\forall}$. Then we can obtain the following results.

Theorem 1.5.20. For every $\lambda_{cp\times}$ type σ, there is a term R_σ, such that for any closed M of type σ,

$$\lambda_{c\times} \vdash M = (R_\sigma\ \overline{\overline{M}}^{\sharp}).$$

Theorem 1.5.21. For every $\lambda_{cp\times}$ type σ, there is a term R_σ, such that for any closed M of type σ,

$$\lambda_{cp\forall\times} \vdash M = (R_\sigma\ \overline{M}).$$

The proof of the above extensions is quite straightforward, but lengthy and is given in Section 1.9.2. Analogous results can be obtained for an extension of λ_{cp} with sum types.

Recursive types: The recursive type discipline introduces types of the form $\mu t.\sigma$. We intend to interpret a type $\mu t.\sigma$ as a fixed point of the function $t \mapsto \sigma$. In order to extend our results to λ_{cp} extended with recursive types we need to define retraction-embedding pairs (j_σ, i_σ) at new types. In particular how does one define $i_{\mu t.\sigma}$ and $j_{\mu t.\sigma}$, or even i_t and j_t?

To motivate a solution, consider the following example. Let $\tau = \mu t.t{\rightarrow}t$. Then in the recursive type discipline one can type $\triangleright(\lambda x.x\ x){:}\,\tau{\rightarrow}\tau$. Assume we have defined terms i_τ and j_τ, and we try to compute

$$\begin{array}{rcl} j_{\tau\rightarrow\tau}(\mathcal{R}(\overline{\overline{\lambda x.x\ x}}^{\sharp})) & = & j_{\tau\rightarrow\tau}(\mathcal{R}(\mathcal{E}(\lambda x.x\ x))) \\ & = & j_{\tau\rightarrow\tau}(\lambda x.x\ x) \\ & = & \lambda x.j_\tau(\mathcal{R}(i_\tau(x)\ i_\tau(x))) \\ & = & ? \end{array}$$

To complete this derivation, one would like to have $i_\tau(x) = \mathcal{E} \circ i_\tau \circ x \circ j_\tau$, so we can continue

$$\begin{array}{rcl} \cdots & = & \lambda x.j_\tau(\mathcal{E}(i_\tau(x\ j_\tau(i_\tau(x))))) \\ & = & \lambda x.x\ x \end{array}$$

What we see from this example is that the two occurrences of x in $\lambda x.x\ x$ "act" as having types $\tau{\rightarrow}\tau$ and τ, respectively. Similarly, we would like the two occurrences of i_τ in $\lambda x.j_\tau(\mathcal{R}(i_\tau(x)\ i_\tau(x)))$ to "act" as $i_{\tau\rightarrow\tau}$ and i_τ. A solution to our problem is to find a uniform definition for i's and j's at all types. Namely, we want a retraction-embedding pair (j, i) that satisfies the following definition.

Definition 1.5.22. A term F is called a *total function* if F is a value and, for any value V, $(F\ V)$ is provably equal to a value.

A pair of total functions (j, i) is a *uniform* retraction-embedding pair if i and j satisfy system of equations

$$\begin{array}{rcl} j(f) & = & j \circ \mathcal{R} \circ f \circ i \\ i(g) & = & \mathcal{E} \circ i \circ g \circ j \\ j \circ i & = & \mathtt{id}. \end{array} \qquad (\star)$$

While it remains open whether there is a pair of terms satisfying the above conditions, we will assume we are given such a pair of functions and, under this assumption, show how the Retraction Theorem can be extended to recursive types. Moreover, since the recursive type system can type *all* terms, as a corollary we obtain the following theorem.

Theorem 1.5.23. Assume total functions i and j exist that satisfy equations $(\star)$. Then, for any closed lambda term M,

$$\lambda_c \vdash M = j(\mathcal{R}(\overline{\overline{M}}^{\sharp})).$$

Of course, the analogous theorems hold for the other transforms as well.

We now proceed with the proof of Theorem 1.5.23. First, since we need to define interpretation of all types, which includes types with free variables and, in particular, type variables alone, the interpretation will be defined relative to a *type environment*. A type environment is simply a function mapping type variables to type sets. Secondly, for technical reasons that will become apparent later, instead of dealing with untyped terms, we will work with derivable type sequents.[12] Hence, our definitions of Section 1.5.1 need to be modified slightly.

Definition 1.5.24. A *type set* is any set of derivable typing sequents of the form $\Gamma \triangleright \mathcal{E}(V)\colon \sigma$, for some value V, closed under all possible typings, i.e., if S is a type set and $\Gamma \triangleright \mathcal{E}(V)\colon \sigma$ is in S, then all derivable sequents $\Gamma_1 \triangleright \mathcal{E}(V)\colon \sigma_1$ are in S as well. Let $\mathcal{D}$ denote the set of all derivable typing sequents, and let $\mathcal{S}$ be the set of all type sets.

[12] Up to now, the typing of all terms was implicit, but in this section we need to refer to types of terms explicitly.

Definition 1.5.25. The relation ε is a subset of $\mathcal{D} \times \mathcal{S}$ defined by $\Gamma \triangleright M : \sigma \;\varepsilon\; \mathcal{S}$ iff there is $\Gamma_1 \triangleright \mathcal{E}(V) : \sigma_1 \in S$ such that $\Gamma \triangleright M = (\texttt{let}\ x_1\texttt{=}P_1, \ldots, x_k\texttt{=}P_k\ \texttt{in}\ \mathcal{E}(V)) : \sigma$, where x_i's contain all free variables in V.

The recursive type system extends the simple types by adding type variables and type expressions of the form $\mu t.\sigma$. The new inference rules are

$$\frac{\Gamma \triangleright M : \sigma\{\mu t.\sigma/t\}}{\Gamma \triangleright M : \mu t.\sigma} \qquad (\mu\ \mathrm{I})$$

$$\frac{\Gamma \triangleright M : \mu t.\sigma}{\Gamma \triangleright M : \sigma\{\mu t.\sigma/t\}} \qquad (\mu\ \mathrm{E})$$

One can understand these rules by considering the type $\mu t.\sigma$ as the type τ, satisfying equation $\tau = \sigma\{\tau/t\}$. Thus we need to define the interpretation, $[\![\tau]\!]\nu$, of τ such that it satisfies the equation

$$[\![\tau]\!]\nu = [\![\sigma\{\tau/t\}]\!]\nu.$$

In other words $[\![\tau]\!]\nu$ should be a fixed point of the function

$$\lambda S.[\![\sigma]\!]\nu\{S/t\}.$$

(We assume the interpretation will satisfy $[\![\sigma\{\tau/t\}]\!]\nu = [\![\sigma]\!]\nu\{[\![\tau]\!]\nu/t\}$.) The difficulty lies in showing that for any σ and ν, the function $\lambda S.[\![\sigma]\!]\nu\{S/t\}$ always has a fixed point. To do so, we define a metric on the space of all type sets so that the resulting metric space is complete. Then we show that each function $\lambda S.[\![\sigma]\!]\nu\{S/t\}$ is a contraction, and thus, by Banach's Fixed-point Theorem, has a unique fixed point. Mac Queen et al. [MPS86] have developed such a framework, of which our development can be viewed as a special case. Namely, our domain consists only of finite elements (typing sequents) ordered under discrete order, thus greatly simplifying general purpose structures used in [MPS86].

The metric on the space $\mathcal{S}$ of all type sets is defined using a rank function. In general, a rank function is any function assigning natural numbers to elements of the domain — in our case typing sequents.

Definition 1.5.26. For any two type sets S and R, the *closeness* of S and R is equal to

$$c(S,R) \stackrel{\mathrm{def}}{=} \min\{\mathrm{rank}(s) \mid s \in S \oplus R\},$$

where $S \oplus R$ denotes the symmetric difference between two sets. If the the sets S and R are the same, we define $c(S,R) = \infty$. The distance between S and R is defined

$$d(S,R) \stackrel{\mathrm{def}}{=} 2^{-c(S,R)}.$$

If $c(S,R) = \infty$, we define $d(S,R) = 0$.

The proof that d is a metric on $\mathcal{S}$ can be found in [MPS86]. Moreover, this construction is a general one for which it can be shown that $(\mathcal{S}, d)$ is a complete metric space (see [MPS86]).

Recall that $(\mathcal{S}, d)$ is a complete metric space iff every Cauchy sequence in $(\mathcal{S}, d)$ has a limit in $(\mathcal{S}, d)$.

Lemma 1.5.27. The metric space $(\mathcal{S}, d)$ is complete. In particular if $\langle S_i \rangle$ is a Cauchy sequence in $(\mathcal{S}, d)$, the set S of all typing sequents s, that occur in all but finitely many S_i is the limit of $\langle S_i \rangle$.

Recall the definition of a contraction and Banach's Fixed-point Theorem.

Definition 1.5.28. A function $f: \mathcal{S} \to \mathcal{S}$ is (uniformly) *contractive* if there is a real number $0 \leq \alpha < 1$, such that for all $S, R \in \mathcal{S}$,

$$d(f(S), f(R)) \leq \alpha \; d(S, R).$$

Function f is called *non-expansive* if the condition on α is relaxed to $0 \leq \alpha \leq 1$.

Theorem 1.5.29 (Banach). Let (X, d) be a complete metric space, and let $f: X \to X$ be a contraction. The, f has a unique fixed-point in X.

In particular, in metric space $(\mathcal{S}, d)$, the contractive (non-expansive) functions are characterized in terms of closeness.

Proposition 1.5.30 (Mac Queen et al.). A function $f: \mathcal{S}^n \to \mathcal{S}$ is non-expansive iff for all $S_1, \ldots, S_n, R_1, \ldots, R_n \in \mathcal{S}$ we have

$$c(f(S_1, \ldots, S_n), f(R_1, \ldots, R_n)) \geq \min_i \; c(S_i, R_i),$$

and is contractive iff for all $S_1, \ldots, S_n, R_1, \ldots, R_n \in \mathcal{S}$, with some $S_i \neq R_i$ we have

$$c(f(S_1, \ldots, S_n), f(R_1, \ldots, R_n)) > \min_i \; c(S_i, R_i).$$

Finally, we define a particular *rank* function based on the type of a sequent.

Definition 1.5.31. The *rank* of a type is defined as follows:

$$\begin{aligned}
\mathrm{rank}(\imath) &\stackrel{\mathrm{def}}{=} 0\\
\mathrm{rank}(t) &\stackrel{\mathrm{def}}{=} 0\\
\mathrm{rank}(T(\sigma)) &\stackrel{\mathrm{def}}{=} 1 + \mathrm{rank}(\sigma)\\
\mathrm{rank}(\sigma \to \tau) &\stackrel{\mathrm{def}}{=} 1 + \max\{\mathrm{rank}(\sigma), \mathrm{rank}(\tau)\}\\
\mathrm{rank}(\mu t.\sigma) &\stackrel{\mathrm{def}}{=} 1 + \mathrm{rank}(\sigma).
\end{aligned}$$

The *rank* of a derivable typing sequent $\Gamma \rhd M : \sigma$ is equal to $\mathrm{rank}(\sigma)$.

The important feature of the *rank* function is that is uniquely defined for every typing sequent, and that the rank of $\sigma \to \tau$ is strictly larger than rank of either σ or τ, intuitively saying that a function has higher rank than its argument or result. We use this fact in the definition of type sets intended to interpret function spaces.

Definition 1.5.32. Let S be a type set and r be an integer. Define

$$S_r \stackrel{\text{def}}{=} \{s \in S \mid \text{rank}(s) < r\}.$$

Let S and R be type sets. Define $S \rightarrow_{\mathcal{S}} R$ to be the set of all typing sequents $\Gamma \triangleright \mathcal{E}(V) : \sigma$ such that if $r = \text{rank}(\Gamma \triangleright \mathcal{E}(V) : \sigma)$, for all $\Gamma_1 \triangleright \mathcal{E}(W) : \sigma_1 \in S_r$

- for all Γ_2 and σ_2 of rank less than r, $\Gamma_2 \triangleright (V\ W) : \sigma_2\ \varepsilon\ R_r$, and
- $\Gamma_3 \triangleright j(\mathcal{R}(V\ W)) {=} (j(V)\ j(W)) : \sigma_3$, for some Γ_3 and σ_3.

Lemma 1.5.33. $f : S, R \mapsto (S \rightarrow_{\mathcal{S}} R)$ is a contraction.

Proof: Fix type sets S_1, S_2, R_1 and R_2. Using Proposition 1.5.30, it suffices to show that if s is a witness that $S_1 \rightarrow_{\mathcal{S}} R_1$ is different from $S_2 \rightarrow_{\mathcal{S}} R_2$, then there is, either a witness that S_1 is different from S_2 or that R_1 is different form R_2, of rank lesser than rank(s). Assume $S_1 \rightarrow_{\mathcal{S}} R_1 \neq S_2 \rightarrow_{\mathcal{S}} R_1$. Then there is $\Gamma \triangleright \mathcal{E}(V) : \sigma$ that is in one set but not the other. Without loss of generality we can assume $\Gamma \triangleright \mathcal{E}(V) : \sigma \in S_1 \rightarrow_{\mathcal{S}} R_1$. Let $r = \text{rank}(\Gamma \triangleright \mathcal{E}(V) : \sigma)$. Since $\Gamma \triangleright \mathcal{E}(V) : \sigma \notin S_2 \rightarrow_{\mathcal{S}} R_2$, by the definition of $\rightarrow_{\mathcal{S}}$, there is $\Gamma_1 \triangleright \mathcal{E}(W) : \sigma_1 \in (S_2)_r$, such that either

- $\Gamma_2 \triangleright (V\ W) : \sigma_2 \not\varepsilon\ (R_2)_r$, for some Γ_2 and σ_2 of rank less than r, or
- $\Gamma_3 \triangleright j(\mathcal{R}(V\ W)) \neq (j(V)\ j(W)) : \sigma_3$, for all Γ_3 and σ_3.

We have two cases: either $\Gamma_1 \triangleright \mathcal{E}(W) : \sigma_1$ is in $(S_1)_r$, or not.

Case $\Gamma_1 \triangleright \mathcal{E}(W) : \sigma_1 \in (S_1)_r$: Then, by definition of $\rightarrow_{\mathcal{S}}$, and since $\Gamma \triangleright \mathcal{E}(V) : \sigma$ is in $S_1 \rightarrow_{\mathcal{S}} R_1$, we must have $\Gamma_3 \triangleright j(\mathcal{R}(V\ W)) = (j(V)\ j(W)) : \sigma_3$, and therefore it must be the case that $\Gamma_2 \triangleright (V\ W) : \sigma_2 \not\varepsilon\ (R_2)_r$, but, by definition of $\rightarrow_{\mathcal{S}}$, $\Gamma_2 \triangleright (V\ W) : \sigma_2\ \varepsilon\ (R_1)_r$. This, by definition of ε, implies there is $\Gamma_4 \triangleright \mathcal{E}(U) : \sigma_4$, of rank less than r, in R_1, but not in R_2. Therefore, $c(R_1, R_2) < r$.

Case $\Gamma_1 \triangleright \mathcal{E}(W) : \sigma_1 \notin (S_1)_r$: Since $\Gamma_1 \triangleright \mathcal{E}(W) : \sigma_1$ is in $(S_2)_r$, it has rank less than r and is in S_2, but not in S_1 and hence, $c(S_1, S_2) < r$.

In either case, $c(S_1 \rightarrow_{\mathcal{S}} R_1, S_2 \rightarrow_{\mathcal{S}} R_2) > \min(c(S_1, S_2), c(R_1, R_2))$. ■

We can now define interpretation of types.

Definition 1.5.34. Let ν be a type environment mapping type variables to type sets. The interpretation of types $[\![\cdot]\!]$ is defined as follows:

$$\begin{array}{rcl} [\![\imath]\!]\nu & \stackrel{\text{def}}{=} & \bigcup \mathcal{S} \\ [\![t]\!]\nu & \stackrel{\text{def}}{=} & \nu(t) \\ [\![\sigma \rightarrow \tau]\!]\nu & \stackrel{\text{def}}{=} & [\![\sigma]\!]\nu \rightarrow_{\mathcal{S}} [\![\tau]\!]\nu \\ [\![\mu t.\sigma]\!]\nu & \stackrel{\text{def}}{=} & \text{fixed point of } \lambda S.[\![\sigma]\!]\nu\{S/t\} \end{array}$$

To show that this definition is a valid one we need to show that every function $\lambda S.[\![\sigma]\!]\nu\{S/t\}$ is a contraction, and therefore, by Banach's theorem has a unique fixed point. Obviously, the function $\lambda S.[\![t]\!]\nu\{S/t\} = \lambda S.S$ is not contractive. As discussed in [MPS86] (section 5), this is the only problematic case, and for all types σ that do not contain $\mu t.t$ as a subexpression, $[\![\sigma]\!]\nu$ is well defined. To remedy this situation, we can either syntactically restrict the set of types, as done in [MPS86], to exclude the ones containing $\mu t.t$ as a subexpression, or we can define the meaning of $\mu t.t$ to be, say, the type set $\bigcup \mathcal{S}$. Since every type set S is a fixed point of the identity function, this definition is consistent with our specification that $[\![\mu t.t]\!]\nu$ is a fixed point of $\lambda S.[\![t]\!]\nu\{S/t\}$.

It remains to show how to modify Lemma 1.5.10 to extend the Retraction Theorem to recursive types. We only sketch the proof.

Lemma 1.5.35. Assume $\Gamma \triangleright M\colon\sigma$ is derivable, and let ρ be a substitution, and ν be a type environment such that for each $x_i\colon\sigma_i \in \Gamma$, $\Gamma' \triangleright \rho(\mathcal{E}(x))\colon T(\sigma_i') \in [\![\sigma_i]\!]\nu$. Let ρ' be substitution such that $\Gamma \triangleright \rho'(x_i) = j(\rho(x_i)) : \sigma_i$ for each $x_i\colon\sigma_i \in \Gamma$. Then

- $\Gamma' \triangleright \rho\,\overline{\overline{M}}^{\sharp} : T(\sigma')\;\varepsilon\;[\![\sigma]\!]\nu$, and
- $\Gamma \triangleright M = j(\mathcal{R}(\rho\overline{\overline{M}}^{\sharp})) : \sigma$.

Proof Sketch: All cases, (var), (app), (abs) and (let) follow by almost the same argument. The only difference in (app) and (abs) cases comes form the modified definition of $[\![\sigma{\rightarrow}\tau]\!]\nu$, where one need to pay attention to ranks of typing sequents. The recursive types introduction and elimination rules, (μ I) and (μ E), preserve validity, since by definition, $[\![\mu t.\sigma]\!]\nu$ is a fixed point of $\lambda S.[\![\sigma]\!]\nu\{S/t\}$, and therefore we have

$$[\![\mu t.\sigma]\!]\nu = [\![\sigma]\!]\nu\{[\![\mu t.\sigma]\!]\nu/t\} = [\![\sigma\{\mu t.\sigma/t\}]\!]\nu.$$

The second condition of the lemma for these rules follows trivially. ∎

We conclude with the proof of Retraction Theorem.

Proof: [of Theorem 1.5.23] If M is closed then the typing sequent $\emptyset \triangleright M\colon \mu t.t{\rightarrow}t$ is derivable. By the previous lemma we have

$$\emptyset \triangleright M = j(\mathcal{R}(\overline{\overline{M}}^{\sharp})) : \mu t.t{\rightarrow}t.$$

Erasing the type information from the proof of the above equation in context, we obtain the proof of the untyped equation $M = j(\mathcal{R}(\overline{\overline{M}}^{\sharp}))$. ∎

By same arguments as presented in Section 1.5.3, the above proof can be modified for the CPS transform.

1.6 Applications

The CPS transform has found applications in studying semantics of programs, as well as, in devising better compilation techniques. We point out that the retraction approach only attempts to explain the relation between equivalence classes of direct style and CPS terms. In CPS based compilers, one is also interested in other features of CPS terms, such as their structure and suitability for various optimizations (see [App92]). In comparison with the results in [FSDF93, SF92, SW96], which provide more precise relation between terms and their images under the CPS transform in terms of reduction relation, we feel that our result will be of greater importance in studying the semantics of programs. We elaborate on an interesting application of the Retraction Theorem.

Riecke and Viswanathan [RV95] have found a particular use for the retraction-embedding pairs $(j_\sigma \circ \mathcal{R}, \mathcal{E} \circ i_\sigma)$ defined in Section 1.4. Their main result shows how to define an operator, called `encap`, that enables a safe use of functional code in an extension of a programming language with imperative features such as assignments or jumps. Informally, their results can be summarized by saying that a term (`encap` M) behaves the same in the extended language as the term M does in the pure language.

The development in [RV95] depends on fully-abstract least fixed-point models of call-by-value PCF (VPCF). We believe that their results can be shown to hold in a larger class of models by using the stronger equational version of the Retraction Theorem.

However, we must point out, that we have not been able to extend our version of the Retraction Theorem, to a programming language like the VPCF. Namely, our results do not cover a language with recursion.

To summarize Riecke-Viswanathan's results in equational framework, let us focus on the CPS transform and an extension of the basic programming language with control.

Let $E^K_\sigma \stackrel{\text{def}}{=} \mathcal{E}^K \circ i_\sigma$ and $R^K_\sigma \stackrel{\text{def}}{=} j_\sigma \circ \mathcal{R}^K$. For each type σ define the operator $\mathtt{encap}_\sigma$ so that the transform of (`encap` M) is

$$\overline{(\mathtt{encap}_\sigma\ M)} = E^K_\sigma(R^K_\sigma(\overline{M})).$$

Let λ_{cp} (assume λ_{cp} includes numerals) be the calculus representing the pure programming language, and let λ_{cp+K} denote the extended calculus. In particular, λ_{cp+K} is extended with the control operators `callcc` and `abort`, and `encap`. Let $C\{\![\cdot]\!\}$ be a context in λ_{cp}, and let $C_+\{\![\cdot]\!\}$ be a context in λ_{cp+K}. Then the results analogous to those in [RV95] can be stated as follows.

Theorem 1.6.1. For any (typable) closed terms M and N in λ_{cp} we have,

- adequacy: $\lambda_{cp} \vdash M = \mathtt{n}$ iff $\lambda_{cp+K} \vdash (\mathtt{encap}_{\mathtt{nat}}\ M) = \mathtt{n}$,
- preservation: $\forall C\{\![\cdot]\!\}\ .\ \lambda_{cp+K} \vdash C\{\![M]\!\} = \mathtt{n}$ iff $\lambda_{cp+K} \vdash C\{\![(\mathtt{encap}\ M)]\!\} = \mathtt{n}$, and
- full abstraction:

$$\forall C\{\![\cdot]\!\}\ .\ \lambda_{cp} \vdash C\{\![M]\!\} = C\{\![N]\!\}$$
$$\text{iff}$$
$$\forall C_+\{\![\cdot]\!\}\ .\ \lambda_{cp+K} \vdash C_+\{\![(\mathtt{encap}\ M)]\!\} = C_+\{\![(\mathtt{encap}\ N)]\!\}.$$

The proof relies on the fact that the CPS transform is a transform from λ_{cp+K} to λ_{cp}, equational correspondence result, and the Retraction Theorem (see [RV95] for details).

Analogous result holds for the SPS transform and an extension of the basic programming language with assignment.

Remark. Since the equational correspondence result holds for all (untyped) terms, and by Theorem 1.5.23, the retraction result holds in all models of untyped λ_{cp} that contain total functions i and j satisfying equations ($\star$), the above theorem holds in all such models as well. Is remains to be investigated how these models relate to those used in [RV95].

1.7 Related Work

In this work, we have established a relation between direct style and CPS terms using definable retraction functions. The Retraction Theorem shows that a term can be recovered, up to λ_{cp}-equivalence, from its image under the CPS transform. Therefore, the retraction approach, in fact, only provides a relation between equivalence classes of terms. On the other hand, in contrast to other results, the relation is established via retraction functions that are *definable* in the target language of the transform, and therefore can be used in applications like the one described in Section 1.6.

The retraction approach was first introduced by Meyer and Wand [MW85], and was further explored by Meyer and Riecke in [MR], where the authors point out some of the limitations of the original Meyer-Wand Retraction Theorem. Most notably, the original Retraction Theorem was proved in $\beta\eta$-equational logic which is not sound for call-by-value languages. In [MR], the provable equality is replaced by call-by-value observational congruence. However, authors fail to prove the Retraction Theorem in this setting and, in fact, show that the original retraction pairs defined in [MW85] no longer serve when $\beta\eta$-equational reasoning is replaced with call-by-value observational congruence.

A semantic variation of the retraction approach has been studied by Riecke [Rie93], and later refined by Riecke and Viswanathan [RV95]. The motivation for their work was to isolate side-effects in sequential programs. To apply the reasoning to a call-by-value language, the authors focus on call-by-value PCF. Rather than working with provable equality, Riecke and Viswanathan construct a relation between two semantics of VPCF in least fixed-point models: the *direct* semantics, and *monadic* semantics (generalizing the CPS semantics) utilizing the additional structure of a *strict monad.* While their version of the Retraction Theorem holds only at base type, this suffices for deriving results by observational congruence as those discussed in Section 1.6.

One generalization of the Meyer-Wand Retraction Theorem is given by Filinski [Fil94]. He extends the CPS transform to the monadic metalanguage, λ_{ml}, and shows that for closed simply typed λ_{ml} term M,

$$\lambda_{ml} \vdash M = \beta_\sigma(\overline{M}).$$

While this result enables embedding a purely functional language with a monadic feature in a call-by-value language with continuations (provided certain conditions are met), it still prohibits generalizations to a language with divergence as does the original Meyer-Wand theorem.

In applications of the CPS transform to compiler technology, definability of the retraction functions may not be relevant, but a more detailed relation between a term and its CPS form, as well as the knowledge about the structure of CPS terms is desirable.

The proof of Sabry Felleisen's [SF92] equational correspondence result (Theorem 1.1.2) yields an inverse to CPS transform, but again, only up to λ_c-equivalence. Similar, but more extensive study of the several variants of the CPS transform has been done by Hatcliff and Danvy [HD94]. Besides the equational correspondence, they also study the structure of the transform in greater detail (e.g. separating the CPS transform into its generic part and "continuation introduction", overhead of administrative reductions, etc.).

A different view of the CPS transform was presented by Flanagan et al. [FSDF93]. By observing implementations of typical CPS based compilers they suggest an alternative transformation that combines several phases done by the compiler.

Sabry and Wadler [SW96] exhibit an even stronger relation between direct style terms and their CPS forms. In particular, they construct an inverse of the CPS transform, as well as the computational transform, so that the transform and its inverse form a *reflection*. A transform $\mathcal{T}$ and its inverse $\mathcal{T}^{-1}$ are said to form a reflection if $\mathcal{T}(M) \twoheadrightarrow^* N$ iff $M \twoheadrightarrow^* \mathcal{T}^{-1}(N)$, and M is identical to $\mathcal{T}^{-1}(\mathcal{T}(M))$.

Above mentioned results provide more information on the structure of CPS terms which can be exploited in the design of CPS based compilers, but seem to have fewer implications for studying semantics of programs in presence of different computational effects.

1.8 Some Open Questions

In all practical applications, functional programming languages are equipped with some form of recursion. Therefore, to make the retraction approach applicable in practice, we need to extend our results to a language with recursion. This can be done in two ways: By extending the type system so that the fixed-point operator is definable in the pure language, or by adding a language construct such as constant Y, `letrec`, etc. The first approach has been discussed in Section 1.5.5.

One difficulty in adding fixed-point operator Y, or a similar language construct, is that additional closure conditions are needed in the definition of type sets, and we haven't been able to construct type sets satisfying these conditions. The other difficulty is determining the correct axiomatization of a fixed-point operator. It appears that the axiom

$$(Y\ V) = (V\ (Y\ V)) \tag{fix}$$

does not suffice. In models of λ_c, fixed-point operator can be defined using the so called *fixpoint object*. Crole and Pitts [CP92] define such an object in models of λ_c, and discuss a logical system for reasoning about fixpoint computations, which may hold the answer to above questions.

Another class of extensions is motivated by the application of the Retraction Theorem described in Section 1.6, where we have seen how one can isolate effects of an extension of a language with assignment or control from interfering with pure functional code. A natural question arises, whether it is possible to extend this approach to isolate one computational effect from interfering with code possibly containing a different computational effect. For instance, if M is a program in, say VPCF with assignment, can we define an operator similar to `encap`, so that, in an extension of VPCF with both assignment *and* control, (`encap` M) will behave the same as M behaves in the extension of VPCF with assignment. We believe that an appropriate extension of the Retraction Theorem to a programming language with imperative features will give us such results.

1.9 Appendix

1.9.1 Monads

In section 1.2.2 we gave an intuitive explanation of monads in the category of sets. We now summarize the definitions in a more general setting. The interested reader is also referred to [Mog88, Mog91, LS86, Cro93, ML71] for a more detailed discussion.

Standard categorical interpretation of functional programs assigns objects to types and morphism to terms. The Cartesian closed structure of a category ensures that (β) and (η) axioms are sound. However, in a call-by-value language, these axioms take restricted forms and the Cartesian close structure is no longer appropriate.

To capture *computational aspects* of programs, such as call-by-name and call-by-value evaluation strategies, Moggi [Mog88] introduced a framework based on monads. The basic idea behind using monads in a category to interpret programs, is that values are separated from computations. If an object A, in a category $\mathcal{C}$, represents values (of type A) then the object $T(A)$, where T is the functor of the monad, represents computations of type A. The reason for this distinction is that computations can have possible *computational effects*. For example, a computation can diverge, alter the store, or branch nondeterministically. The separation between values and computations allows an interpretation to differentiate between call-by-value and call-by-name languages. For instance, in a call-by-value language, a function form A to B would be interpreted as a morphism from A to $T(B)$, modeling the fact that arguments are evaluated before they are passed to a function. On the other hand the same function in a call-by-name interpretation would be interpreted as a morphism from $T(A)$ to $T(B)$, since arguments are not evaluated before they are passed to a function. The functor T represents a particular notion of computation, e.g. continuation passing, partiality, nondeterminism, etc.

The formal definition of a monad in a category $\mathcal{C}$ is equivalent to one of a *Kleisli triple* in a sense that each Kleisli triple over $\mathcal{C}$ uniquely determines a monad in $\mathcal{C}$ and vice versa (see [Mog91]). We find the Kleisli triples more convenient to use.

Definition 1.9.1 ([Man76]). A *Kleisli triple* over a category $\mathcal{C}$ is a triple $(T, \eta, \_^*)$, where T is a function mapping objects of $\mathcal{C}$ to objects of $\mathcal{C}$, η is a family of morphisms indexed by objects of $\mathcal{C}$, such that for each object A, $\eta_A: A \to T(A)$, and for each morphism $f: A \to T(B)$, f^* is a morphism $f^*: T(A) \to T(B)$ such that

- $\eta_A^* = \mathrm{id}_A$,
- $\eta_A; f^* = f$, for all $f: A \to T(B)$, and
- $f^*; g^* = (f; g^*)^*$, for all $f: A \to T(B)$ and $g: B \to T(C)$.

Given a Kleisli triple $(T, \eta, \_^*)$, the corresponding monad is defined by the triple (T, η, μ), where T is an extension of the function T to an endofunctor on $\mathcal{C}$, by defining $T(f) = (f; \eta_B)^*$ for $f: A \to B$; $\eta: \mathrm{Id} \dot{\to} T$ is a natural transformation defined by η of the Kleisli triple, and $\mu: T^2 \dot{\to} T$ is a natural transformation defined by $\mu_A = \mathrm{id}^*_{T(A)}$.

Note that our discussion of a monad in **Set** is a special case of this definition. Given a Kleisli triple over $\mathcal{C}$ one can construct the *Kleisli category*, $\mathcal{C}_T$, which is a syntax-free representation of the computational lambda calculus.

Definition 1.9.2 ([ML71, Mog91]). Let $(T, \eta, \_^*)$ be a Kleisli triple. The *Kleisli category*, $\mathcal{C}_T$, is defined as follows:

- the objects of $\mathcal{C}_T$ are objects of $\mathcal{C}$,
- a morphisms $f: A \to B$ in $\mathcal{C}_T$ is a morphism $f: A \to T(B)$ in $\mathcal{C}$,
- the identity on A in $\mathcal{C}_T$ is the morphism $\eta_A: A \to T(A)$ in $\mathcal{C}$, and
- the composition of $f: A \to B$ followed by $g: B \to C$ in $\mathcal{C}_T$ is defined by the morphism $f; g^*: A \to T(C)$ in $\mathcal{C}$.

It is easy to verify that $\mathcal{C}_T$ is a category. $\mathcal{C}_T$ provides a model for the core of the computational lambda calculus, but to express products and functions additional structure is needed. Namely,

Definition 1.9.3. A *strong monad* in $\mathcal{C}$ is a monad (T, η, μ) together with a natural transformation $t_{A,B}: A \times T(B) \to T(A \times B)$, satisfying certain identities (see [Mog91]). Natural transformation t is also called *tensorial strength.*

Definition 1.9.4 ([Mog91]). A λ_c*-model* is a category $\mathcal{C}$ with finite products, a strong monad (T, η, μ, t) satisfying the mono requirement (i.e. η_A is mono for all $A \in \mathcal{C}$) and T-exponential $T(B)^A$ for every $A, B \in \mathcal{C}$.

A Kleisli category (with tensorial strength and T-exponentials), is a syntax free representation of the computational lambda calculus, λ_c. There are notable differences between a CCC and a Kleisli category. For example, the composition in the Kleisli category does not represent substitution, as it does in a CCC, but rather it represents the `let`-constructor. Substitution in the computational lambda calculus corresponds to composition in the underlying category of the Kleisli category. Another difference is the interpretation of function spaces, which are interpreted as T-exponentials in the Kleisli category as opposed to ordinary exponentials in a CCC.

Example 1.9.5. Let $\mathcal{C}$ be a CCC, and let R be an object in $\mathcal{C}$. The *continuation monad*, K, in $\mathcal{C}$ is given by the Kleisli triple $(K, \eta, \_^*)$ with tensorial strength t defined as follows (using lambda notation):

$$\begin{aligned}
K(A) &\stackrel{\mathrm{def}}{=} (A \to R) \to R \\
\eta_A: A \to K(A) &\stackrel{\mathrm{def}}{=} \lambda a{:}\, A.\lambda k{:}\, (A \to R).k\ a \\
f^*: K(A) \to K(B) &\stackrel{\mathrm{def}}{=} \lambda c{:}\, K(A).\lambda k{:}\, (B \to R).c\ (\lambda a{:}\, A.(f\ a)\ k) \\
t_{A,B}: (A \times K(B)) \to K(A \times B) &\stackrel{\mathrm{def}}{=} \lambda \langle a, c\rangle{:}\, A \times K(B).\lambda k{:}\, ((A \times B) \to R).c\ (\lambda b{:}\, B.k\ \langle a, b\rangle)
\end{aligned}$$

Other notions of computation, such as partiality, side-effects, exceptions and interactive I/O can also be expressed in terms of monads (see [Mog91]).

Every category $\mathcal{C}$ (with enough structure) can be viewed as a lambda theory, $Th(\mathcal{C})$, called the *internal language* of the category, and conversely, for every lambda theory, Th, there is a category, $Cl(Th)$, the *classifying category* of Th, such that the categories $Cl(Th(\mathcal{C}))$ and $\mathcal{C}$ are equivalent (see [Cro93] and [LS86] for more details).

In this sense, the computational lambda calculus, λ_c is the internal language of the *initial* Kleisli category. Similarly, λ_{ml} is the internal language of the initial category in the category of CCC's equipped with strong monads.

1.9.2 Extensions

Adding Numerals

The calculus λ_{cp}+*numerals* represents the following extension of λ_{cp}.

- The language of λ_{cp}+*numerals* is extended with constructs:

$$\textit{values} \qquad V \quad ::= \quad \cdots \mid \mathtt{n} \mid \mathtt{succ} \mid \mathtt{pred}$$

  where `n` stands for numerals `0`, `1`, `2`, etc. We write `n+1` to denote the successor numeral of `n`.

- The logic of λ_{cp}+*numerals* is extended with axioms:

$$(\mathtt{succ}\ \mathtt{n}) = \mathtt{n+1} \tag{succ}$$

$$(\mathtt{pred}\ \mathtt{0}) = \mathtt{0} \tag{pred.0}$$

$$(\mathtt{pred}\ \mathtt{n+1}) = \mathtt{n} \tag{pred.n}$$

- The type system of λ_{cp}+*numerals* is extended with the base type `nat` and the following typing axioms:

$$\Gamma \rhd_{cp} \mathtt{n}\colon \mathtt{nat} \tag{num}$$

$$\Gamma \rhd_{cp} \mathtt{succ}\colon \mathtt{nat} \to \mathtt{nat} \tag{succ}$$

$$\Gamma \rhd_{cp} \mathtt{pred}\colon \mathtt{nat} \to \mathtt{nat} \tag{pred}$$

The modified computational transform must be extended appropriately. In particular we have

$$\overline{\overline{\mathtt{n}}}^{\sharp} \quad \stackrel{\text{def}}{=} \quad \mathcal{E}(\mathtt{n})$$

$$\begin{array}{rcl} \overline{\overline{\mathtt{succ}}}^{\sharp} & \stackrel{\text{def}}{=} & \mathcal{E}(\lambda x.\mathcal{E}(\mathtt{succ}\ x)) \\ \overline{\overline{\mathtt{pred}}}^{\sharp} & \stackrel{\text{def}}{=} & \mathcal{E}(\lambda x.\mathcal{E}(\mathtt{pred}\ x)) \end{array}$$

To prove the Retraction Theorem for the extended calculus we need to extend the proof of Lemma 1.5.10 with the three cases for the new type inference axioms. Before we do so, since `nat` is treated as a base type, we define

$$i_{\mathtt{nat}} \stackrel{\text{def}}{=} \mathtt{id} \quad \text{and} \quad j_{\mathtt{nat}} \stackrel{\text{def}}{=} \mathtt{id},$$

and

$$[\![\mathtt{nat}]\!] \stackrel{\text{def}}{=} \{M \mid M = \mathcal{E}(V) \text{ for some value } V\}.$$

We are now ready to extend the proof of Lemma 1.5.10.

Proof: (Extending the proof of Lemma 1.5.10.) Let Γ be a typing hypothesis and assume that ρ satisfies Γ.

Case (num): (S.1) follows immediately from definitions. For (S.2) compute

$$j_{\mathtt{nat}}(\mathcal{R}(\rho\,\overline{\overline{\mathtt{n}}}^{\sharp})) = \mathtt{id}(\mathcal{R}(\mathcal{E}(\mathtt{n}))) = \mathtt{n}.$$

Case (succ): To show (S.1) first observe that $\rho'\,\overline{\overline{\mathtt{succ}}}^{\sharp}$ is of the form $\mathcal{E}(V)$, so we need to show that $\rho\,\overline{\overline{\mathtt{succ}}}^{\sharp} \in [\![\mathtt{nat}\to\mathtt{nat}]\!]$. Let $\mathcal{E}(W) \in [\![\mathtt{nat}]\!]$.

$$\rho\,\overline{\overline{\mathtt{succ}}}^{\sharp} = \mathcal{E}(\lambda x.\mathcal{E}(\mathtt{succ}\ x)),$$

so

$$\begin{array}{rcl} ((\lambda x.\mathcal{E}(\mathtt{succ}\ x))\ W) & = & \mathcal{E}(\mathtt{succ}\ W) \\ & = & (\mathtt{let}\ x\mathtt{=}(\mathtt{succ}\ W)\ \mathtt{in}\ \mathcal{E}(x)), \end{array}$$

and, by definition of $[\![\mathtt{nat}]\!]$, $\mathcal{E}(x) \in [\![\mathtt{nat}]\!]$. Therefore,

$$((\lambda x.\mathcal{E}(\mathtt{succ}\ x))\ W)\ \varepsilon\ [\![\mathtt{nat}]\!]$$

as desired. To show (P.2) we compute

$$\begin{array}{rcl} j_{\mathtt{nat}}(\mathcal{R}((\lambda x.\mathcal{E}(\mathtt{succ}\ x))\ W)) & = & j_{\mathtt{nat}}(\mathcal{R}((\lambda x.\mathcal{E}(\mathtt{succ}\ x))\ i_{\mathtt{nat}}(j_{\mathtt{nat}}(W)))) \\ & = & (j_{\mathtt{nat}\to\mathtt{nat}}(\lambda x.\mathcal{E}(\mathtt{succ}\ x))\ j_{\mathtt{nat}}(W)), \end{array}$$

using the fact that $i_{\mathtt{nat}} = j_{\mathtt{nat}} = \mathtt{id}$. Finally, to show (S.2) compute

$$\begin{array}{lll} j_{\mathtt{nat}\to\mathtt{nat}}(\mathcal{R}(\rho'\,\overline{\overline{\mathtt{succ}}}^{\sharp})) & & \\ \quad = \ j_{\mathtt{nat}\to\mathtt{nat}}(\mathcal{R}(\mathcal{E}(\lambda x.\mathcal{E}(\mathtt{succ}\ x)))) & & \text{by defn.} \\ \quad = \ j_{\mathtt{nat}\to\mathtt{nat}}(\lambda x.\mathcal{E}(\mathtt{succ}\ x)) & & \text{by (r-e)} \\ \quad = \ \lambda x.j_{\mathtt{nat}}(\mathcal{R}(\mathcal{E}(\mathtt{succ}\ i_{\mathtt{nat}}(x)))) & & \text{by defn. of } j_{\mathtt{nat}\to\mathtt{nat}} \\ \quad = \ \lambda x.\mathtt{succ}\ x & & \text{by (r-e) and defn. of } i_{\mathtt{nat}} \text{ and } j_{\mathtt{nat}} \\ \quad = \ \mathtt{succ} & & \text{by } (\eta_v). \end{array}$$

Case (pred): This case is identical to the (succ) case. ■

The Retraction Theorem for the CPS transform can be extended in a similar manner by defining

$$\begin{array}{rcl} \overline{\mathtt{n}} & \stackrel{\text{def}}{=} & \lambda k.k\ \mathtt{n} \\ \overline{\mathtt{succ}} & \stackrel{\text{def}}{=} & \lambda k.k\ \lambda x.(\lambda k'.k'\ (\mathtt{succ}\ x)) \\ \overline{\mathtt{pred}} & \stackrel{\text{def}}{=} & \lambda k.k\ \lambda x.(\lambda k'.k'\ (\mathtt{pred}\ x)), \end{array}$$

and extending the proof of Lemma 1.5.17 the same way we have done above.

Adding Products

The calculus $\lambda_{cp\times}$ is an extension of λ_{cp} with products. The language of $\lambda_{cp\times}$ is extended with the following term constructs:

$$\begin{array}{lcll} \textit{values} & V & ::= & \cdots \mid \pi_1(V) \mid \pi_2(V) \mid \langle V, V\rangle \\ \textit{terms} & M & ::= & \cdots \mid \pi_1(M) \mid \pi_2(M) \mid \langle M, M\rangle. \end{array}$$

The types are enriched with the binary type constructor $\times$ and the type inference rules given in Table 1.14. The logic of $\lambda_{cp\times}$ contains the following axioms (taken from [Mog89])

Types:

$$\sigma ::= \cdots \mid \sigma \times \sigma$$

Inference Rules:

$$\frac{\Gamma \rhd_\times M{:}\,\sigma, \quad \Gamma \rhd_\times N{:}\,\tau}{\Gamma \rhd_\times \langle M, N\rangle{:}\,\sigma \times \tau} \tag{$\times$I}$$

$$\frac{\Gamma \rhd_\times M{:}\,\sigma_1 \times \sigma_2}{\Gamma \rhd_\times \pi_i(M){:}\,\sigma_i}, \qquad \text{i=1,2} \tag{$\times$E}$$

Table 1.14: Extension of a type system with products.

$$\langle M_1, M_2\rangle = (\mathtt{let}\ x_1\mathtt{=}M_1\ \mathtt{in}\ (\mathtt{let}\ x_2\mathtt{=}M_2\ \mathtt{in}\ \langle x_1, x_2\rangle)) \tag{let.$\times$}$$

$$\pi_i\langle V_1, V_2\rangle = V_i \tag{$\times.\beta$}$$

$$\langle \pi_1(V), \pi_2(V)\rangle = V \tag{$\times.\eta$}$$

These axioms specify that the pairs are strict and that the first component is always evaluated before the second. Consequently, the modified computational transform is extended to the new language constructs as follows:

$$\overline{\overline{\pi_i(M)}}^{\sharp} \stackrel{\text{def}}{=} \mathcal{E}(\pi_i(\mathcal{R}(\overline{\overline{M}}^{\sharp})))$$

$$\overline{\overline{\langle M_1, M_2\rangle}}^\sharp \stackrel{\text{def}}{=} (\texttt{let } x_1\texttt{=}\mathcal{R}(\overline{\overline{M}}^\sharp) \texttt{ in } (\texttt{let } x_2\texttt{=}\mathcal{R}(\overline{\overline{M_2}}^\sharp) \texttt{ in } \mathcal{E}\langle x_1, x_2\rangle))$$

A product type, $\sigma \times \tau$ is interpreted as the following type set

$$[\![\sigma \times \tau]\!] \stackrel{\text{def}}{=} \{\mathcal{E}\langle V, W\rangle \mid \mathcal{E}(V) \in [\![\sigma]\!] \text{ and } \mathcal{E}(W) \in [\![\tau]\!]\}.$$

The definition of the retraction functions i and j is extended to product types to be

$$j_{\sigma\times\tau} \stackrel{\text{def}}{=} \lambda x.\langle j_\sigma(\pi_1(x)), j_\tau(\pi_2(x))\rangle,$$

and

$$i_{\sigma\times\tau} \stackrel{\text{def}}{=} \lambda x.\langle i_\sigma(\pi_1(x)), i_\tau(\pi_2(x))\rangle.$$

It is easy to see that $i_{\sigma\times\tau}(V)$ and $j_{\sigma\times\tau}(V)$ are values for any value V. Moreover, we can compute

$$\begin{array}{lll}
\multicolumn{2}{l}{j_{\sigma\times\tau} \circ i_{\sigma\times\tau} =} & \\
= & \lambda x.\langle j_\sigma(\pi_1(x)), j_\tau(\pi_2(x))\rangle \circ \lambda x.\langle i_\sigma(\pi_1(x)), i_\tau(\pi_2(x))\rangle & \text{by defn.} \\
= & \lambda x.\langle j_\sigma(\pi_1\langle i_\sigma(\pi_1(x)), i_\tau(\pi_2(x))\rangle), j_\tau(\pi_2\langle i_\sigma(\pi_1(x)), i_\tau(\pi_2(x))\rangle)\rangle & \text{by } (\beta_v) \\
= & \lambda x.\langle j_\sigma(i_\sigma(\pi_1(x))), j_\tau(i_\tau(\pi_2(x)))\rangle & \text{by } (\times.\beta) \\
= & \lambda x.\langle \pi_1(x), \pi_2(x)\rangle & \text{by Lemma 1.4.2} \\
= & \lambda x.x & \text{by } (\times.\eta)
\end{array}$$

It remains to show that the rules ($\times$ E) and ($\times$ I) preserve properties (S.1) and (S.2) of Lemma 1.5.10. To do so we first prove the following results.

Lemma 1.9.6. Let M and N be terms such that $M \mathrel{\varepsilon} [\![\sigma]\!]$ and $N \mathrel{\varepsilon} [\![\tau]\!]$. Then,

$$(\texttt{let } x\texttt{=}\mathcal{R}(M) \texttt{ in } (\texttt{let } y\texttt{=}\mathcal{R}(N) \texttt{ in } \mathcal{E}\langle x, y\rangle)) \mathrel{\varepsilon} [\![\sigma \times \tau]\!].$$

Proof: By assumption $M = (\texttt{let } x_1\texttt{=}P_1 \texttt{ in } \mathcal{E}(V_1))$ and $N = (\texttt{let } x_2\texttt{=}P_2 \texttt{ in } \mathcal{E}(V_2))$ for some $\mathcal{E}(V_1) \in [\![\sigma]\!]$ and $\mathcal{E}(V_2) \in [\![\tau]\!]$. Therefore,

$$\begin{array}{lll}
\multicolumn{2}{l}{(\texttt{let } x\texttt{=}\mathcal{R}(M) \texttt{ in } (\texttt{let } y\texttt{=}\mathcal{R}(N) \texttt{ in } \mathcal{E}\langle x, y\rangle)) =} & \\
= & (\texttt{let } x\texttt{=}\mathcal{R}(\texttt{let } x_1\texttt{=}P_1 \texttt{ in } \mathcal{E}(V_1)) \texttt{ in} & \\
 & \quad (\texttt{let } y\texttt{=}\mathcal{R}(\texttt{let } x_2\texttt{=}P_2 \texttt{ in } \mathcal{E}(V_2)) \texttt{ in } \mathcal{E}\langle x, y\rangle)) & \text{by assumption} \\
= & (\texttt{let } x\texttt{=}(\texttt{let } x_1\texttt{=}P_1 \texttt{ in } V_1) \texttt{ in} & \\
 & \quad (\texttt{let } y\texttt{=}(\texttt{let } x_2\texttt{=}P_2 \texttt{ in } V_2) \texttt{ in } \mathcal{E}\langle x, y\rangle)) & \text{by (C-appl.1) and (r-e)} \\
= & (\texttt{let } x_1\texttt{=}P_1 \texttt{ in } (\texttt{let } x\texttt{=}V_1 \texttt{ in} & \\
 & \quad (\texttt{let } x_2\texttt{=}P_2 \texttt{ in } (\texttt{let } y\texttt{=}V_2 \texttt{ in } \mathcal{E}\langle x, y\rangle)))) & \text{by (assoc)} \\
= & (\texttt{let } x_1\texttt{=}P_1 \texttt{ in } (\texttt{let } x_2\texttt{=}P_2 \texttt{ in } \mathcal{E}\langle V_1, V_2\rangle)) & \text{by (C-let.1),}
\end{array}$$

and by the definition of $[\![\sigma \times \tau]\!]$, $\mathcal{E}\langle V_1, V_2\rangle \in [\![\sigma \times \tau]\!]$. ■

Lemma 1.9.7. If M is such that $M \mathrel{\varepsilon} [\![\sigma_1 \times \sigma_2]\!]$ then $\mathcal{E}(\pi_i(\mathcal{R}(M))) \mathrel{\varepsilon} [\![\sigma_i]\!]$.

Proof: By assumption $M \; \varepsilon \; [\![\sigma_1 \times \sigma_2]\!]$, so for some $\mathcal{E}(V_1) \in [\![\sigma_1]\!]$ and $\mathcal{E}(V_2) \in [\![\sigma_2]\!]$, $M = (\texttt{let}\ x\texttt{=}P\ \texttt{in}\ \mathcal{E}\langle V_1, V_2\rangle)$. Therefore,

$$\begin{array}{rcll}
\mathcal{E}(\pi_i(\mathcal{R}(M))) & = & \mathcal{E}(\pi_i(\mathcal{R}(\texttt{let}\ x\texttt{=}P\ \texttt{in}\ \mathcal{E}\langle V_1, V_2\rangle))) & \text{by assumption} \\
& = & (\texttt{let}\ x\texttt{=}P\ \texttt{in}\ \mathcal{E}(\pi_i(\mathcal{R}(\mathcal{E}\langle V_1, V_2\rangle)))) & \text{by (C-appl.1)} \\
& = & (\texttt{let}\ x\texttt{=}P\ \texttt{in}\ \mathcal{E}(\pi_i\langle V_1, V_2\rangle)) & \text{by (r-e)} \\
& = & (\texttt{let}\ x\texttt{=}P\ \texttt{in}\ \mathcal{E}(V_i)) & \text{by } (\times.\beta),
\end{array}$$

and $\mathcal{E}(V_i)$ is in $[\![\sigma_i]\!]$ by assumption. ∎

Now we are ready to extend the proof of Lemma 1.5.10 to product types.

Proof: (Extending the proof of Lemma 1.5.10.)

Case $(\times\ \mathrm{E})$: Assume ρ satisfies Γ and $\Gamma \triangleright_{cp\times} M{:}\, \sigma_1 \times \sigma_2$, so that $\Gamma \triangleright_{cp\times} \pi_i(M){:}\, \sigma_i$. Then by induction hypothesis, $\rho\overline{\overline{M}}^{\sharp} \; \varepsilon \; [\![\sigma_1 \times \sigma_2]\!]$. By definition $\rho\,\overline{\overline{\pi_i(M)}}^{\sharp} = \rho(\mathcal{E}(\pi_i(\mathcal{R}(\overline{\overline{M}}^{\sharp}))))$, so by Lemma 1.9.7, $\rho\,\overline{\overline{\pi_i(M)}}^{\sharp} \; \varepsilon \; [\![\sigma_i]\!]$. This proves (S.1). To show (S.2) compute

$$\begin{array}{rll}
\multicolumn{2}{l}{j_{\sigma_i}(\mathcal{R}(\rho\,\overline{\overline{\pi_i(M)}}^{\sharp})) = j_{\sigma_i}(\mathcal{R}(\rho(\mathcal{E}(\pi_i(\mathcal{R}(\overline{\overline{M}}^{\sharp}))))))} & \text{by defn.} \\
= & j_{\sigma_i}(\mathcal{R}(\mathcal{E}(\pi_i(\mathcal{R}(\rho\overline{\overline{M}}^{\sharp}))))) & \text{by defn. of subst.} \\
= & j_{\sigma_i}(\mathcal{R}(\mathcal{E}(\pi_i(\mathcal{R}(\texttt{let}\ x\texttt{=}P\ \texttt{in}\ \mathcal{E}\langle V_1, V_2\rangle))))) & \text{by assumption} \\
= & (\texttt{let}\ x\texttt{=}P\ \texttt{in}\ j_{\sigma_i}(\mathcal{R}(\mathcal{E}(\pi_i(\mathcal{R}(\mathcal{E}\langle V_1, V_2\rangle)))))) & \text{by (C-appl.1)} \\
= & (\texttt{let}\ x\texttt{=}P\ \texttt{in}\ j_{\sigma_i}(V_i)) & \text{by (r-e) and } (\times.\beta) \\
= & (\texttt{let}\ x\texttt{=}P\ \texttt{in}\ \pi_i\langle j_{\sigma_1}(V_1), j_{\sigma_2}(V_2)\rangle) & \text{by } (\times.\beta) \\
= & \pi_i(\texttt{let}\ x\texttt{=}P\ \texttt{in}\ \langle j_{\sigma_1}(V_1), j_{\sigma_2}(V_2)\rangle) & \text{by (C-appl.1)} \\
= & \pi_i(\texttt{let}\ x\texttt{=}P\ \texttt{in}\ j_{\sigma_1\times\sigma_2}\langle V_1, V_2\rangle) & \text{by defn. of } j_{\sigma_1\times\sigma_2} \\
= & \pi_i(j_{\sigma_1\times\sigma_2}(\mathcal{R}(\texttt{let}\ x\texttt{=}P\ \texttt{in}\ \mathcal{E}(\langle V_1, V_2\rangle)))) & \text{by (C-appl.1) and (r-e)} \\
= & \pi_i(j_{\sigma_1\times\sigma_2}(\mathcal{R}(\rho\overline{\overline{M}}^{\sharp}))) & \text{by assumption} \\
= & \pi_i(\rho' M) & \text{by ind. hyp.} \\
= & \rho'\,\pi_i(M) & \text{by defn. of subst.,}
\end{array}$$

where $\rho'(x) = j_\sigma(\rho(x))$ for each $x{:}\,\sigma \in \Gamma$.

Case $(\times\ \mathrm{I})$: Assume $\Gamma \triangleright_{cp\times} M{:}\, \sigma$ and $\Gamma \triangleright_{cp\times} N{:}\, \tau$ so that $\Gamma \triangleright_{cp\times} \langle M, N\rangle{:}\, \sigma \times \tau$. By definition $\rho\,\overline{\overline{\langle M, N\rangle}}^{\sharp} = (\texttt{let}\ x\texttt{=}\mathcal{R}(\rho\overline{\overline{M}}^{\sharp})\ \texttt{in}\ (\texttt{let}\ y\texttt{=}\mathcal{R}(\rho\overline{\overline{N}}^{\sharp})\ \texttt{in}\ \mathcal{E}\langle x, y\rangle))$, and by induction hypothesis, for ρ that satisfies Γ, $\rho\overline{\overline{M}}^{\sharp} \; \varepsilon \; [\![\sigma]\!]$ and $\rho\overline{\overline{N}}^{\sharp} \; \varepsilon \; [\![\tau]\!]$, so by Lemma 1.9.6, $\rho\,\overline{\overline{\langle M, N\rangle}}^{\sharp} \; \varepsilon \; [\![\sigma \times \tau]\!]$.

This establishes (S.1). Again, (S.2) follows by computation.

$$
\begin{array}{lll}
\multicolumn{3}{l}{j_{\sigma\times\tau}(\mathcal{R}(\rho\,\overline{\overline{\langle M,N\rangle}}^{\sharp})) =} \\
= & j_{\sigma\times\tau}(\mathcal{R}(\texttt{let } x\texttt{=}\mathcal{R}(\rho\overline{\overline{M}}^{\sharp})\texttt{ in (let } y\texttt{=}\mathcal{R}(\rho\overline{\overline{N}}^{\sharp})\texttt{ in }\mathcal{E}\langle x,y\rangle))) & \text{by defn.} \\
= & j_{\sigma\times\tau}(\mathcal{R}(\texttt{let } x\texttt{=}\mathcal{R}(\texttt{let } x_1\texttt{=}P_1\texttt{ in }\mathcal{E}(V_1))\texttt{ in} & \\
 & \quad (\texttt{let } y\texttt{=}\mathcal{R}(\texttt{let } x_2\texttt{=}P_2\texttt{ in }\mathcal{E}(V_2))\texttt{ in }\mathcal{E}\langle x,y\rangle))) & \text{by ind. hyp. (S.1)} \\
= & j_{\sigma\times\tau}(\mathcal{R}(\texttt{let } x\texttt{=(let } x_1\texttt{=}P_1\texttt{ in } V_1)\texttt{ in} & \\
 & \quad (\texttt{let } y\texttt{=(let } x_2\texttt{=}P_2\texttt{ in } V_2)\texttt{ in }\mathcal{E}\langle x,y\rangle))) & \text{by (C-appl.1) \& (r-e)} \\
= & j_{\sigma\times\tau}(\mathcal{R}(\texttt{let } x_1\texttt{=}P_1\texttt{ in (let } x_2\texttt{=}P_2\texttt{ in }\mathcal{E}\langle V_1,V_2\rangle))) & \text{by (assoc)} \\
 & & \text{and (C-let.1)} \\
= & (\texttt{let } x_1\texttt{=}P_1\texttt{ in (let } x_2\texttt{=}P_2\texttt{ in } j_{\sigma\times\tau}\langle V_1,V_2\rangle)) & \text{by (C-appl.1) \& (r-e)} \\
= & (\texttt{let } x_1\texttt{=}P_1\texttt{ in (let } x_2\texttt{=}P_2\texttt{ in }\langle j_\sigma(V_1), j_\tau(V_2)\rangle)) & \text{by defn of } j_{\sigma\times\tau},\ (\beta_v) \\
 & & \&\ (\times.\beta) \\
= & \langle(\texttt{let } x_1\texttt{=}P_1\texttt{ in } j_\sigma(V_1)), (\texttt{let } x_2\texttt{=}P_2\texttt{ in } j_\tau(V_2))\rangle & \text{by (C-let.1)} \\
 & & \&\ (\text{let}.\times) \\
= & \langle j_\sigma(\mathcal{R}(\texttt{let } x_1\texttt{=}P_1\texttt{ in }\mathcal{E}(V_1))), j_\tau(\mathcal{R}(\texttt{let } x_2\texttt{=}P_2\texttt{ in }\mathcal{E}(V_2)))\rangle & \text{by (r-e) \& (C-appl.1)} \\
= & \langle j_\sigma(\mathcal{R}(\rho\overline{\overline{M}}^{\sharp})), j_\tau(\mathcal{R}(\rho\overline{\overline{N}}^{\sharp}))\rangle & \text{by ind. hyp. (S.1)} \\
= & \langle \rho' M, \rho' N\rangle & \text{by ind. hyp. (S.2)} \\
= & \rho'\langle M,N\rangle & \text{by defn. of subst.}
\end{array}
$$

■

The extension of the Retraction Theorem for the CPS transform can be done in a similar way. The CPS transform of pairs and projections is defined to be

$$
\begin{array}{rcl}
\overline{\pi_i(M)} & \stackrel{\text{def}}{=} & \lambda k.\overline{M}\ (\lambda m.k\ \pi_i(m)) \\
\overline{\langle M,N\rangle} & \stackrel{\text{def}}{=} & \lambda k.\overline{M}\ (\lambda m.\overline{N}\ (\lambda n.k\ \langle m,n\rangle)).
\end{array}
$$

Using relation ε^K instead of ε and terms $\mathcal{E}^K$ and $\mathcal{R}^K$ instead of $\mathcal{E}$ and $\mathcal{R}$, one can extend Lemma 1.5.17 and the Retraction Theorem for the CPS transform to $\lambda_{cp\times}$.

1.10 Acknowledgments

I'd like to thank to my advisor Albert Meyer for lengthy discussions about this work, and to Prof. Mitch Wand for helpful comments. I'd also like to thank Jens Palsberg and Jon Riecke for the discussions and comments on some issues in the early stages of this work.

Chapter 2

Another Look at "Theorems for Free!"

Abstract: We reexamine Wadler's "free theorems" — theorems about polymorphic functions derivable from the type of the function — in an extension of Mairson's syntactic treatment of logical relations. Logical relations are defined as formulas in second (and higher) order logic over untyped lambda terms, from which we can instantiate equations involving a given term. As a consequence we can formulate a free theorem as an equation schema, each instance of which is provable by convertibility, given typing conditions on the terms involved are met. In our framework datatypes are represented as free algebras generated by constants, and we define sufficient conditions which a representation of a datatype must satisfy to preserve free theorems. As a special case we prove examples of free theorems, as equations by convertibility, using Böhm-Berarducci representation of datatypes.

2.1 Introduction

What are "free theorems"? The purpose of using types in programming languages is to avoid certain kinds of errors. For example, in a well typed program a function cannot be applied to an argument of a wrong type. Thus, types give us certain guarantees about the behavior of well typed programs. What Wadler, in his "Theorems for Free!" paper [Wad89], points out is that not only we know that well type programs will not have any run-time type errors, but we can also infer, from the type of a program, certain identities that the program must satisfy. To quote Wadler:

> *Write down a definition of a polymorphic function on a piece of paper. Tell me its type, but be careful not to let me see the function's definition. I will tell you a theorem that the function satisfies.*

Such "free theorems" are all instances of parametric behavior of polymorphic functions in a model of polymorphic lambda calculus. In this paper, we propose a syntactic treatment of free theorems inspired by the types-as-propositions paradigm, extending the one developed by Mairson [Mai91]. We develop a framework, in which parametric behavior of polymorphic lambda terms is captured by formulae of type theory. This setting allows us to present the free theorems as metatheorems about convertibility. In other words, a free theorem is an equation schema that is provable by convertibility whenever certain typing assumptions about the terms involved hold. For example, the famous "map theorem" can be restated as follows. Let $\underline{\texttt{List}(t)}$ abbreviate the type of lists $\forall s.(t{\rightarrow}s{\rightarrow}s){\rightarrow}s{\rightarrow}s$, and write $\underline{\texttt{map}}$ be the *map* function on lists: $\lambda flcn.(l\ (\lambda xycn.(c\ (f\ x)\ (y\ c\ n)))\ n)$.[1]

Proposition 2.1.1. Assume M is a closed pure λ-term of type

$$\forall st.(s{\rightarrow}t){\rightarrow}(\underline{\texttt{List}(s)}){\rightarrow}(\underline{\texttt{List}(t)}),$$

and let L be a pure λ-term such that

$$\{x_1\!:r, \ldots x_k\!:r\} \triangleright L\!:(\underline{\texttt{List}(r)}).$$

Then equation,

$$(\underline{\texttt{map}}\ f\ (M\ (\lambda x.x)\ L)) = (M\ f\ L).$$

is provable by convertibility in $\beta\eta$-theory.

This theorem, as well as other examples in [Wad89], are all considered manifestations of *parametric polymorphism.* Intuitively, a polymorphic function is called parametric if it can be described as having a uniform algorithm that works for all types [Str67]. There have been numerous attempts to formulate this notion, for example [Rey83, BFSS90, MR91, Mai91], and in each of these formulations, the parametric functions have the expected properties. Even though there doesn't seem to be a general agreement on the "right" definition of parametricity, it is agreed that the polymorphic λ-terms denote parametric functions.

To capture the parametric behavior of lambda terms, we use a form of logical relations defined as formulae of type theory. In a way, our formulation of logical relation is an extension of types-as-propositions paradigm. For instance, the proposition (in the type theory) $\forall ST.S \supset T$, corresponding to type $\sigma \equiv \forall st.s{\rightarrow}t$, can be considered a *null-ary* logical relation. Unary logical relation, $R^\sigma(x)$, is defined to be the proposition $\forall ST.\forall y.T(y) \supset S(x\ y)$ of the type theory over lambda terms. Similarly, one can define logical relations of any arity. For this definition of logical relations we prove the The Fundamental Theorem of Logical Relations which, informally, states that if a term M has type σ, then the proposition $R^\sigma(M)$ is provable in type theory. Moreover, by proper instantiations of the universally quantified variables in the logical relation, $R^\sigma(M)$, we derive, in type theory, an equation involving the term M. By soundness of type theory and completeness of a lambda theory we conclude that the type theory is conservative over equational logic of the lambda theory, so the derived equation about M, in fact, holds by convertibility.

An interesting point in the syntactic version of free theorems is that we obtain an equation, about a term M, that holds by convertibility without actually knowing what M is. Of course, any derivation (by convertibility) of the resulting equality would depend on the structure of the term involved. However, in the types-as-propositions paradigm a term

[1] The type of lists and the *map* function are derived from the Böhm-Berarducci representation of lists discussed in more detail in Section 2.5.

corresponds to a proof of the proposition corresponding to its type, and the derivation by convertibility corresponds to the normalization of the proof. Thus, it is not surprising, that from the proposition corresponding to the type of a term, one can extract the *computational content* corresponding to the derivation of an equation by convertibility.

In proving free theorems where elements of an inductive data type, such as lists, are used, as Mairson points out, the apparent absence of induction looks suspicious. Wadler defines the logical relation corresponding to lists to say that two lists are related if they are of the same length and the corresponding elements of the lists are related. This definition relies on our intuitive understanding what lists are, and assumes that, in the model in which he studies the free theorems, lists behave according to our intuition. On the other hand, in the syntactic formulation of free theorems, we need to be precise how we represent datatypes such as lists. For instance, we can use the Böhm-Berarducci [BB85] encoding of lists in polymorphic lambda calculus similar to the encoding of natural numbers as Church numerals. Then, the type of lists yields a logical relation quite different from the induction axiom corresponding to Wadler's semantic interpretation of the logical relation on lists. It seems that the free theorems require the induction axiom, and the question is whether the free theorems hold for Böhm-Berarducci lists, without using induction. To answer this question, rather than asking if Böhm-Berarducci lists are "inductive", we reformulate the question and ask what is a correct way to represent inductive datatypes such as lists.

To answer the question we develop a framework where datatypes are represented as free algebras generated by a set of constants. For example, lists are generated by constants $\mathtt{nil}{:}\,\mathtt{List}(t)$ and $\mathtt{cons}{:}\, t{\to}\mathtt{List}(t){\to}\mathtt{List}(t)$. In this framework logical relations corresponding to inductive datatypes are the induction axioms, from which free theorems follow easily. Then we define a general notion of a representation (of generators of a datatype), and define sufficient conditions that allows us to translate the free theorems. In particular, we show that the Böhm-Berarducci representation satisfies the conditions and the free theorems hold when Böhm-Berarducci encodings of lists are used. This also answers Mairson's question whether induction is necessary in proving the free theorem when Böhm-Berarducci lists are used. To be precise we should say that the explicit use of induction in not necessary. Implicitly, induction is "contained" in the type of Böhm-Berarducci lists.

2.2 Definitions

In this section we define the main notions used in the paper. We assume the reader is familiar with untyped λ-calculus and elementary concepts such as free and bound variables, substitution, parallel substitution, as well as, (α), (β) and (η) axiom schemes. For precise treatment of these notions one is referred to [Bar84].

2.2.1 Untyped λ-calculus and λ-theories

We work with standard definition of the untyped λ-calculus with constants. Let $\mathcal{C}_\Lambda$ be a set of constants and write $\Lambda(\mathcal{C}_\Lambda)$ to denote the set of λ-terms over constants in $\mathcal{C}_\Lambda$ and an infinite set of λ-variables. We do not distinguish between α-equivalent λ-terms. We use lower-case letters towards the end of the alphabet to denote λ-variables, and upper-case letters, M, N, etc. to denote arbitrary λ-terms. Following [Bar84] we write $\Lambda^0(\mathcal{C}_\Lambda)$ for the set of all closed λ-terms over constants in $\mathcal{C}_\Lambda$. We will use parentheses freely to make expressions more readable, and in doing so we assume application of terms associates to the left.

A set, $\mathfrak{Th}$, of equations between λ-terms is called λ-*congruence* if it contains all instances of axiom (ref) and is closed under the rules given in Table 2.1. If a λ-congruence also contains

$$M = M \qquad \text{(ref)}$$

$$\frac{M = N}{N = M} \qquad \text{(sym)}$$

$$\frac{M = P, \quad P = N}{M = N} \qquad \text{(trans)}$$

$$\frac{M = P, \quad N = Q}{(M\ N) = (P\ Q)} \qquad \text{(cong)}$$

$$\frac{M = N}{\lambda x.M = \lambda x.N} \qquad (\xi)$$

Table 2.1: Inference rules of a λ-congruence.

all instances of (β), it is a λ-*theory*. If a theory, $\mathfrak{Th}$, contains an equation $M = N$, we say $M = N$ is provable by convertibility and write $\mathfrak{Th} \vdash_\lambda M = N$. In particular if $\mathfrak{Th}$ is the minimal λ-theory containing (η), and $\mathfrak{Th} \vdash_\lambda M = N$, we write $M =_{\beta\eta} N$.

2.2.2 Type Inference System F_ω+$\mathfrak{Th}$

Let $\mathfrak{Th}$ be a λ-congruence. The type inference system F_ω+$\mathfrak{Th}$ is a version of Girard's system F_ω [Gir72]. In F_ω+$\mathfrak{Th}$ we use implicit typing in the style of Curry, and add an extra typing rule (term-cong) [Mit88]. Moreover, we enrich the set of type constructors with a set of type constructor constants.

The set of type constructors is defined as *simply kinded* λ-calculus, over a set of constant type constructors, with base kind $\star$. More precisely, we have that $\star$ is a kind, and if κ' and κ are kinds, then so is $\kappa' \Rightarrow \kappa$. For each kind κ there are infinitely many variables t^κ of that kind. Let $\mathcal{C}_\mathcal{T}$ be a set of constant type constructors, each assigned a kind. We write c^κ to denote a constant type constructor of kind κ. The set $\mathcal{T}(\mathcal{C}_\mathcal{T})$, of type constructors over constants in $\mathcal{C}_\mathcal{T}$ is defined inductively.

- A constant c^κ in $\mathcal{C}_\mathcal{T}$ is a type constructor of kind κ.
- A type constructor variable t^κ is a type constructor of kind κ.
- If σ and τ are type constructors of kind $\star$, then $\sigma \rightarrow \tau$ is a type constructor of kind $\star$.
- If σ is a type constructor of kind $\star$, then $\forall t^\kappa.\sigma$ is a type constructor of kind $\star$.
- If σ is a type constructor of kind $\kappa' \Rightarrow \kappa$ and τ is a type constructor of kind κ', then $(\sigma\ \tau)$ is a type constructor of kind κ.
- If σ is a type constructor of kind κ, then $\lambda t^{\kappa'}.\sigma$ is a type constructor of kind $\kappa' \Rightarrow \kappa$.

We define equality between type constructors of the same kind by the usual $\beta\eta$-equational rules for simply typed lambda calculus. We write $\sigma =_\lambda \sigma'$ if σ and σ' can be proven equal using these equational rules. Type constructors of kind $\star$ are called *λ-types*, or simply types.

As with the λ-terms, we will use parentheses freely to make type constructors more readable. In doing so, we assume the application associates to the left and the exponentiation operator $\rightarrow$ associates to the right, so that $\sigma\rightarrow(\tau\rightarrow\upsilon)$ can be written as $\sigma\rightarrow\tau\rightarrow\upsilon$. We will also omit the kind information form type constructors, when it is clear from the context or irrelevant.

An expression of the form $M{:}\,\sigma$, where M is a λ-term and σ is a type is called a *type assertion.* A set of type assertions, $\Gamma = \{x_1{:}\,\sigma_1, \ldots, x_k{:}\,\sigma_k\}$ where no variable occurs twice in Γ is called a *typing hypothesis.* We write $\Gamma, x{:}\,\sigma$ for the set $\Gamma \cup \{x{:}\,\sigma\}$, assuming x does not occur in Γ. A *typing sequent* is a formula of the form $\Gamma \rhd M{:}\,\sigma$, where Γ is a typing hypothesis, M is a λ-term and σ is a type.

Let Type be a partial function, assigning types to constants in $\mathcal{C}_\Lambda$. Write $\mathsf{Type}(\mathsf{g}) = \sigma_\mathsf{g}$ to say that $\mathsf{Type}(\mathsf{g})$ is defined and equal to σ_g. Then, the inference rules for proving typing sequents in $F_\omega{+}\mathfrak{Th}$ are given in Table 2.2 below. If a sequent $\Gamma \rhd M{:}\,\sigma$ is derivable using

$$\Gamma, x{:}\,\sigma \rhd x{:}\,\sigma \qquad \text{(var)} \qquad\qquad \frac{\Gamma \rhd M{:}\,\forall t.\tau}{\Gamma \rhd M{:}\,((\lambda t.\tau)\ \sigma)} \qquad \text{(inst)}$$

$$\frac{\Gamma \rhd M{:}\,\sigma\rightarrow\tau, \quad \Gamma \rhd N{:}\,\sigma}{\Gamma \rhd (M\ N){:}\,\tau} \qquad \text{(app)} \qquad\qquad \frac{\Gamma \rhd M{:}\,\sigma}{\Gamma \rhd M{:}\,\forall t.\sigma}, \quad \text{if } t \notin \mathrm{FV}(\Gamma) \qquad \text{(gen)}$$

$$\frac{\Gamma, x{:}\,\sigma \rhd M{:}\,\tau}{\Gamma \rhd (\lambda x.M){:}\,\sigma\rightarrow\tau} \qquad \text{(abs)} \qquad\qquad \frac{\Gamma \rhd M{:}\,\sigma}{\Gamma \rhd M{:}\,\tau}, \quad \text{if } \sigma =_\lambda \tau \qquad \text{(type-cong)}$$

$$\Gamma \rhd \mathsf{g}{:}\,\sigma_\mathsf{g}, \qquad \text{if } \mathsf{g} \in \mathcal{C}_\Lambda \text{ and } \mathsf{Type}(\mathsf{g}) = \sigma_\mathsf{g} \qquad \text{(const)}$$

$$\frac{\Gamma \rhd N{:}\,\sigma}{\Gamma \rhd M{:}\,\sigma}, \qquad \text{if } N = M \in \mathfrak{Th} \qquad \text{(term-cong)}$$

Table 2.2: Type inference system $F_\omega{+}\mathfrak{Th}$.

these rules, we write $\mathfrak{Th} \vdash \Gamma \rhd M{:}\,\sigma$. If the rule (term-cong) was not used in the derivation, we write $\vdash \Gamma \rhd M{:}\,\sigma$, or simply $\Gamma \rhd M{:}\,\sigma$. We will also abbreviate $\mathfrak{Th} \vdash \emptyset \rhd \mathsf{g}{:}\,\sigma_\mathsf{g}$ by $\mathsf{g}{:}\,\sigma_\mathsf{g}$ when g is in $\mathcal{C}_\Lambda$ and $\mathsf{Type}(\mathsf{g}) = \sigma_\mathsf{g}$.

Note that if σ_g is not closed, since $\vdash \emptyset \rhd \mathsf{g}{:}\,\sigma_\mathsf{g}$, then also $\vdash \emptyset \rhd \mathsf{g}{:}\,\forall \vec{t}.\sigma_\mathsf{g}$, where $\vec{t}$ is a sequence of all free type constructor variables in σ_g. Thus, without loss of generality, we may assume that each σ_g is closed. If c is a constant type constructor and $t_1, \ldots t_k$ are type constructor variables of appropriate kinds so that $(\mathsf{c}\, t_1 \cdots t_k)$ is of kind $\star$, type $(\mathsf{c}\, t_1 \cdots t_k)$ is called a *base type.*

Given a set $\mathcal{C}_\Lambda$ of constant λ-terms, a set $\mathcal{C}_\mathcal{T}$ of constant type constructors, a partial function Type typing constant λ-terms and a λ-congruence $\mathfrak{Th}$, the tuple $\mathfrak{C} = (\mathcal{C}_\Lambda, \mathcal{C}_\mathcal{T}, \mathsf{Type}, \mathfrak{Th})$ will simply be called a *calculus.*

2.2.3 Data Structures

In this paper a *data structure* is a free algebra over lambda terms defined by a signature $(\{\mathtt{c}_1,\ldots,\mathtt{c}_k\},\{\mathtt{g}_1\colon\phi_1,\ldots,\mathtt{g}_l\colon\phi_l\})$ where $\mathtt{c}_1,\ldots,\mathtt{c}_k$ are type constructor constants in $\mathcal{C}_{\mathcal{T}}$, and $\mathtt{g}_1,\ldots,\mathtt{g}_l$ are constants in $\mathcal{C}_\Lambda$, called the *generators* of the data structure. We assume the function Type is defined on all $\mathtt{g}_i$'s and $\mathsf{Type}(\mathtt{g}_i)=\phi_i$ for all i. We also assume that $\mathtt{c}_1,\ldots,\mathtt{c}_k$ are the only constant type constructors occurring in ϕ_i's.

Example 2.2.1. The data structure representing lists is given by the signature

$$\mathcal{L}\stackrel{\text{def}}{=}(\{\mathtt{List}^{\star\Rightarrow\star}\},\{\mathtt{cons}\colon\forall t.t\to(\mathtt{List}\,t)\to(\mathtt{List}\,t),\mathtt{nil}\colon\forall t.(\mathtt{List}\,t)\}).$$

The lists of type $(\mathtt{List}\ \sigma)$ are terms of the form $(\mathtt{cons}\ N_1\ (\cdots(\mathtt{cons}\ N_k\ \mathtt{nil})\cdots))$, where N_i's are lambda terms of type σ.

Example 2.2.2. Natural numbers can be defined with the signature

$$\mathcal{N}\stackrel{\text{def}}{=}(\{\mathtt{Nat}^\star\},\{\mathtt{succ}\colon\mathtt{Nat}\to\mathtt{Nat},\mathtt{zero}\colon\mathtt{Nat}\}).$$

Natural number k corresponds to the k-fold application of $\mathtt{succ}$ to $\mathtt{zero}$.

Let $(\{\mathtt{c}_1,\ldots\mathtt{c}_m\},\{\mathtt{g}_1\colon\phi_1,\ldots\mathtt{g}_n\colon\phi_n\})$ be a signature of a data structure such that each $\mathtt{c}_i$ is of kind $\star\Rightarrow\cdots\Rightarrow\star$ and each ϕ_i is of the form $\forall\vec{t}.\sigma_1\to\cdots\to\sigma_r\to(\mathtt{c}\ t_1\cdots t_k)$ with σ_i's either type variables or base types. Call such a data structure *simple*. This class of data structures is sufficient to express a large set of commonly used datatypes, such as list, trees, natural numbers, etc.

If $(\{\mathtt{c}_1,\ldots,\mathtt{c}_k\},\{\mathtt{g}_1,\ldots,\mathtt{g}_l\})$ and $(\{\mathtt{c}'_1,\ldots,\mathtt{c}'_m\},\{\mathtt{g}'_1,\ldots,\mathtt{g}'_n\})$ are two signatures of different data structures, we always assume that $\{\mathtt{c}_1,\ldots,\mathtt{c}_k\}\cap\{\mathtt{c}'_1,\ldots,\mathtt{c}'_m\}=\emptyset=\{\mathtt{g}_1,\ldots,\mathtt{g}_l\}\cap\{\mathtt{g}'_1,\ldots,\mathtt{g}'_n\}$.

2.2.4 Programs over λ-terms

Let $\mathtt{f}$ be a constant in $\mathcal{C}_\Lambda$, that is not a generator of any data structure. A *program* for $\mathtt{f}$ is a set, $\mathfrak{P}_\mathtt{f}$, of equations between λ-terms. Intuitively, equations in $\mathfrak{P}_\mathtt{f}$ define a function denoted by $\mathtt{f}$. For example, if $\mathtt{f}$ denotes a function f, from natural numbers to natural numbers, the equations in $\mathfrak{P}_\mathtt{f}$ will correspond to the inductive definition of f.

We say that the calculus contains a program $\mathfrak{P}_\mathtt{f}$ for $\mathtt{f}$, if $\mathfrak{P}_\mathtt{f}\subseteq\mathfrak{Th}$ and $\mathtt{f}\in\mathcal{C}_\Lambda$. Such a constant $\mathtt{f}$ is called a *function symbol*. In general we will assume that constants in $\mathcal{C}_\Lambda$ are either generators of a data structure or function symbols.

Example 2.2.3. Given the constructors $\mathtt{cons}$ and $\mathtt{nil}$, as in the above example, the program $\mathfrak{P}_\mathtt{map}$ for $\mathtt{map}$ is given by the following set of equations:

$$\mathfrak{P}_\mathtt{map}\stackrel{\text{def}}{=}\{(\mathtt{map}\ f\ \mathtt{nil})=\mathtt{nil},(\mathtt{map}\ f\ (\mathtt{cons}\ x\ t))=(\mathtt{cons}\ (f\ x)\ (\mathtt{map}\ f\ t))\}.$$

Example 2.2.4. Given the data structure $\mathcal{N}$ of natural numbers, one can define addition by the following program:

$$\mathfrak{P}_{\texttt{add}} \stackrel{\text{def}}{=} \{(\texttt{add}\ \texttt{zero}\ x) = x, (\texttt{add}\ (\texttt{succ}\ x)\ y) = (\texttt{succ}\ (\texttt{add}\ x\ y))\}$$

2.2.5 Type Theory over λ-terms

Type theory is a generalization of the minimal second order logic over lambda terms[2] to include higher-order predicates. We define the language of type theory with base logical types $\imath$ and o. Therefore, a *logical type* is either $\imath$, o, or $\varkappa_1 \Rightarrow \varkappa_2$, whenever $\varkappa_1$ and $\varkappa_2$ are logical types. The terms of logical type $\imath$ will be λ-terms and the logical type o will represent truth values. We have an infinite supply of variables of each logical type. Variables of type $\varkappa$ are denoted $X^{\varkappa}$. We do not distinguish λ-variables from variables of type $\imath$, and use lowercase letters to range over either. The terms of type theory are defined inductively.

- Every variable $X^{\varkappa}$ is a term of type $\varkappa$.
- If M is a λ-term then M is a term of type $\imath$.
- If M and N are terms of type $\imath$ then $M = N$ is a term of type o.
- If P is a term of type $\varkappa' \Rightarrow \varkappa$ and Q is a term of type $\varkappa'$ then $(P\ Q)$ is a term of type $\varkappa$.
- If P is a term of type $\varkappa$ then $\Lambda X^{\varkappa'}.P$ is a term of type $\varkappa' \Rightarrow \varkappa$.
- If P and Q are terms of type o then $P \supset Q$ is a term of type o.
- If P is a term of type o then $\forall X^{\varkappa}.P$ is a term of type o.

We assume the usual $\beta\eta$-equality between terms of type theory and write $P =_{\Lambda} Q$ if terms P and Q of the same type can be proven equal with respect to "$=_{\Lambda}$". This equality should not be confused with equality "$=$" between λ-terms. The equality "$=_{\Lambda}$" is defined in terms of (β), (η), (ξ), etc. defined w.r.t. Λ and *not* λ. Moreover, the equality "$=_{\Lambda}$" is used to denote equality of terms of type theory, and is not a part of the language of the type theory, while the equality "$=$" is used to form the terms of the type theory of the form $M = N$.

Notation. We reserve the letters M, N, L and K to range over lambda terms, and letters P and Q to range over terms of type theory. Letters X, Y and Z, as well as S and T, will range over variables of type theory of any logical type, and specifically for the variables of logical type $\imath$, we will use lower case x, y, z, etc. When writing formulas of type theory, we use parentheses freely, and assume application associates to the left and implication associates to the right. If P is a term of type theory of logical type $\imath \Rightarrow o$, and

[2] By the *minimal* logic, we mean the fragment of the logic that contains only implication and universal quantification.

M is a lambda term, we will write $P(M)$ for the application $(P\ M)$. Similarly, if P is of logical type $\imath \Rightarrow \imath \Rightarrow \cdots \Rightarrow \imath \Rightarrow o$, and M_i are lambda terms, we write $P(M_1, M_2, \ldots, M_k)$ for the application $(P\ M_1\ M_2 \cdots M_k)$.

Let Γ be a set of terms of type o. For a term P of type o we write $\Gamma \vdash_{TT} P$, to say that P is provable in type theory from assumptions in Γ, if the sequent $\Gamma \vdash P$ can be derived using the rules of inference given in Table 2.3. Note that P and Q range over terms of type theory of type o, M, N, L and K range over lambda terms, and X ranges over variables of any logical type, while x ranges over λ-variables (i.e. variables of logical type $\imath$). The

$\Gamma, P \vdash P$	(Id)	$\dfrac{\Gamma \vdash P[M/x], \quad \Gamma \vdash M = N}{\Gamma \vdash P[N/x]}$	(Cong)
$\dfrac{\Gamma, P \vdash Q}{\Gamma \vdash P \supset Q}$	($\supset$ I)	$\Gamma \vdash M = M$	(ref)
$\dfrac{\Gamma \vdash P \supset Q, \quad \Gamma \vdash P}{\Gamma \vdash Q}$	($\supset$ E)	$\dfrac{\Gamma \vdash M = N}{\Gamma \vdash N = M}$	(sym)
$\dfrac{\Gamma \vdash P}{\Gamma \vdash \forall X.P}$, if $X \notin \mathrm{FV}(\Gamma)$	($\forall$ I)	$\dfrac{\Gamma \vdash M = N, \quad \Gamma \vdash N = L}{\Gamma \vdash M = L}$	(trans)
$\dfrac{\Gamma \vdash \forall X.P}{\Gamma \vdash P[Q/X]}$	($\forall$ E)	$\dfrac{\Gamma \vdash M = N, \quad \Gamma \vdash K = L}{\Gamma \vdash (M\ K) = (N\ L)}$	(cong)
$\dfrac{\Gamma \vdash P}{\Gamma \vdash Q}$, if $P =_\Lambda Q$	($=_\Lambda$)	$\dfrac{\Gamma \vdash M = N}{\Gamma \vdash \lambda x.M = \lambda x.N}$, if $x \notin \mathrm{FV}(\Gamma)$	(ξ)

Table 2.3: Inference rules of type theory.

set of equations provable from assumptions in Γ using the above rules is a λ-congruence. Moreover, the rules of type theory only prove equations between lambda terms, that are derivable by convertibility form the equations in Γ. We prove this conservativity result in the next section.

Remark. We have identified variables of type theory of logical type $\imath$ and lambda variables. While, from the perspective of the interpretation described in the next section, it is correct to treat the two kinds of variables the same way, we need to clarify a few points to avoid possible confusion. Lambda variables are treated differently in the context of type theory than in the context of equational logic. It helps one's intuition to think of lambda variables in the context of type theory as *metavariables.* For example, consider the sequent $x = \lambda y.y \vdash (x\ z) = z$. In the context of a lambda theory, this sequent conveys little information, since it says that in a theory in which $x = \lambda y.y$, which is necessarily inconsistent, equation $(x\ z) = z$ is provable. On the other hand the same sequent, in the context of type theory makes perfect sense, and it says that $x = \lambda y.y$ *implies* $(x\ z) = z$. This apparent paradox comes from the fact that in equational theory, there is no way to express implication.

2.2.6 Semantics of Type Theory

In this paper, we only consider a special class of models of the type theory. Observe that the type theory is essentially a simply typed lambda calculus. Thus, we can use the same construction as in a set-theoretic model of simply typed lambda calculus to construct a model of type theory. More precisely, let the set $\mathcal{I}$ be an environment model of untyped λ-calculus (as defined in [Mey82]) and let $\mathcal{O}$ be the two element set $\{\mathsf{True}, \mathsf{False}\}$. Logical types are given the following interpretation:

$$\begin{aligned} [\![o]\!] &\stackrel{\text{def}}{=} \mathcal{O} \\ [\![\imath]\!] &\stackrel{\text{def}}{=} \mathcal{I} \\ [\![\varkappa_1 \Rightarrow \varkappa_2]\!] &\stackrel{\text{def}}{=} [\![\varkappa_2]\!]^{[\![\varkappa_1]\!]} \end{aligned}$$

To give an interpretation of terms we define an environment to be a function mapping variables of type $\varkappa$ to $[\![\varkappa]\!]$. Given an environment, ρ, the interpretation function $[\![\cdot]\!]$ assigning meaning to the terms of type theory is defined inductively:

$$\begin{aligned} [\![X^\varkappa]\!]\rho &\stackrel{\text{def}}{=} \rho(X^\varkappa) \\ [\![M]\!]\rho &\stackrel{\text{def}}{=} [\![M]\!]_{\mathcal{I}}\rho, \text{ if } M \text{ is a } \lambda\text{-term and } [\![M]\!]_{\mathcal{I}}\rho \text{ is} \\ & \qquad \text{the interpretation of } M \text{ in } \mathcal{I} \\ [\![M = N]\!]\rho &\stackrel{\text{def}}{=} \mathsf{True} \text{ if } [\![M]\!]_{\mathcal{I}}\rho = [\![N]\!]_{\mathcal{I}}\rho \text{ and } \mathsf{False} \text{ otherwise} \\ [\![(P\ Q)]\!]\rho &\stackrel{\text{def}}{=} [\![P]\!]\rho([\![Q]\!]\rho) \\ [\![\Lambda X^\varkappa.P]\!]\rho &\stackrel{\text{def}}{=} \text{function } f \text{ such that for all } e \in [\![\varkappa]\!],\ f(e) = [\![P]\!]\rho[e/X^\varkappa] \\ [\![P \supset Q]\!]\rho &\stackrel{\text{def}}{=} \mathsf{True} \text{ iff } [\![P]\!]\rho = \mathsf{False} \text{ or } [\![Q]\!]\rho = \mathsf{True} \\ [\![\forall X^\varkappa.P]\!]\rho &\stackrel{\text{def}}{=} \mathsf{True} \text{ iff for all } e \in [\![\varkappa]\!],\ [\![P]\!]\rho[e/X^\varkappa] = \mathsf{True} \end{aligned}$$

In order for this interpretation to be well defined, one must have the meaning of all λ-terms in $\Lambda(\mathcal{C}_\Lambda)$ defined in $\mathcal{I}$. We will always assume this is so.

Any model of type theory is completely determined by the underlining model of the untyped lambda calculus $\mathcal{I}$. Therefore we will speak of the meaning of a term P in $\mathcal{I}$, but with the understanding that P is interpreted in the model of type theory defined by the model $\mathcal{I}$.

Let $\mathcal{I}$ be model of the untyped lambda calculus, and let ρ be an environment. If P is a term of type o, ρ is said to *satisfy* P, if $[\![P]\!]\rho = \mathsf{True}$. A sequent $\Gamma \vdash P$ is said to be *valid* in $\mathcal{I}$, written $\Gamma \models_{\mathcal{I}} P$, if for every ρ that satisfies all terms in Γ, ρ satisfies P.

The inference rules of type theory are sound for the class of models we are considering. This result follows easily by showing that each inference rule in Table 2.3 preserves validity.

Lemma 2.2.5 (Soundness). Let $\mathcal{I}$ be a model of the untyped lambda calculus. Then,

$$\text{if} \quad \Gamma \vdash_{TT} P \quad \text{then} \quad \Gamma \models_{\mathcal{I}} P.$$

Even though we lack the completeness result for the type theory, one can use semantic reasoning in type theory to prove that an equation between lambda terms holds by convertibility.

Lemma 2.2.6. Let $\mathfrak{Th}$ be a λ-theory. If, for every model $\mathcal{I}$, $\mathfrak{Th} \models_{\mathcal{I}} M = N$ then $\mathfrak{Th} \vdash_{\lambda} M = N$. In particular, if $\mathcal{I}$ is the term model, consisting of $\mathfrak{Th}$-equivalence classes of lambda terms, and $\mathfrak{Th} \models_{\mathcal{I}} M = N$, then $M = N$ is provable by convertibility from equations in $\mathfrak{Th}$.

Proof: To say that $\mathfrak{Th} \models_{\mathcal{I}} M = N$ in every model $\mathcal{I}$, is to say that $M = N$ is true in every interpretation satisfying equations in $\mathfrak{Th}$. Therefore, by the completeness of λ-convertibility [Mey82], $M = N$ is provable by convertibility, i.e. $\mathfrak{Th} \vdash_{\lambda} M = N$. ■

Theorem 2.2.7. Type theory is conservative over λ-convertibility, that is

$$\mathfrak{Th} \vdash_{TT} M = N \quad \text{iff} \quad \mathfrak{Th} \vdash_{\lambda} M = N.$$

Proof: Since $\mathfrak{Th} \vdash_{TT} M = N$, by soundness of the inference rules of type theory, for any model $\mathcal{I}$, $\mathfrak{Th} \models_{\mathcal{I}} M = N$, and by the above lemma, $\mathfrak{Th} \vdash_{\lambda} M = N$. ■

2.2.7 Models of Type Inference

To define a model of type inference, one needs to define how to interpret terms and types, as well as define a notion of *type sets*, relating the two interpretations, so that if M is a term of type σ, the meaning of M is an element of the type set corresponding to σ. The type inference models of the second order polymorphic lambda calculus have been studied in detail by Mitchell [Mit88], and his definitions easily generalize to system $F_\omega + \mathfrak{Th}$. We summarize the main definitions.

A *type structure*, $\mathfrak{T}$, is a set theoretic model of simply typed lambda calculus, containing special elements $\rightarrow^{\mathfrak{T}}$ and $\forall^{\mathfrak{T}}$. More precisely, $\mathfrak{T}$ consists of a family of sets, $\{\mathcal{U}_\kappa\}$, indexed by kinds. We interpret type constructors of kind κ as elements of set $\mathcal{U}_\kappa$. We have $\rightarrow^{\mathfrak{T}} \in \mathcal{U}_{\star \Rightarrow \star \Rightarrow \star}$ and $\forall^{\mathfrak{T}} \in \mathcal{U}_{(\star \Rightarrow \star) \Rightarrow \star}$. Given a mapping, η, from type constructor variables of kind κ to $\mathcal{U}_\kappa$, define the interpretation of functional and universal types to be

$$[\![\sigma \rightarrow \tau]\!]\eta = \rightarrow^{\mathfrak{T}}([\![\sigma]\!]\eta)([\![\tau]\!]\eta) \quad \text{and} \quad [\![\forall t.\sigma]\!]\eta = \forall^{\mathfrak{T}}([\![\lambda t.\sigma]\!]\eta).$$

Let $\mathcal{D}$ be a model of the untyped lambda calculus. Then, we define type sets using the *subset* function that, to each element, a, in $\mathcal{U}_\star$ assigns a subset, D_a, of $\mathcal{D}$. In addition, we require that the type sets satisfy the following conditions:[3]

$$D_{a \rightarrow b} = \{d \mid d \cdot D_a \subseteq D_b\} \tag{Arrow}$$

$$D_{\forall f} = \bigcap_{a \in \mathcal{U}_\star} D_{f(a)}. \tag{ForAll}$$

We say that a type structure $\mathfrak{T}$, a model of untyped lambda calculus $\mathcal{D}$, and a *subset* function define a model of type inference.

[3] We assume $\mathcal{D}$ is a model of $\beta\eta$ calculus, and thus (Arrow) simplifies conditions (Arrow.1) and (Arrow.2) in [Mit88]

2.2.8 Logical Relations

Curry-Howard isomorphism gives a bijective correspondence between typed lambda calculus and propositional logic. In this isomorphism, polymorphic types correspond to propositions of higher order logic, and terms correspond to proofs. For example, the type $\forall t.t \rightarrow t$ correspond to proposition $\forall T.T \supset T$, and the identity function $\lambda x.x$ corresponds to a proof of that proposition. The syntactic version of logical relation represents, in a way, a generalization of this correspondence. In our example, the proposition $\forall T.T \supset T$ is a null-ary logical relation corresponding to type $\forall t.t \rightarrow t$. We define unary logical relation, $R^{\forall t.t \rightarrow t}(f)$, to be $\forall T.\forall x.T(x) \supset T(f(x))$, and a binary version $R^{\forall t.t \rightarrow t}(f_1, f_2)$, to be $\forall T.\forall x_1 x_2.T(x_1, x_2) \supset T(f_1(x_1), f_2(x_2))$, etc.

Our definition of logical relations extends the one used by Mairson [Mai91] to type theory in order to accommodate higher order, as well as constant, type constructors of the type inference system $F_\omega + \mathfrak{T}\mathfrak{h}$. For each type constructor σ, of kind κ, we define a term R^σ of type theory of logical type $\varkappa(\kappa)$, where $\varkappa$ is an injection from kinds to logical types defined inductively so that: $\varkappa(\star) \stackrel{\text{def}}{=} \imath \Rightarrow o$ and $\varkappa(\kappa' \Rightarrow \kappa) \stackrel{\text{def}}{=} \varkappa(\kappa') \Rightarrow \varkappa(\kappa)$.

We also assume there is a bijection between type constructor variables and of kind κ and propositional variables of logical type $\varkappa(\kappa)$ that maps a variable t^κ to a propositional variable $T^{\varkappa(\kappa)}$. We will write type constructors variables in lowercase, and use the same letters in uppercase to denote variables of type theory, that are uniquely determined by the above mentioned bijection. Given a sequence of type constructor variables $t_1, \ldots, t_k$, we say the variables $T_1, \ldots, T_k$ correspond to variables $t_1, \ldots, t_k$, and vice versa.

Using these conventions, for each type σ, we define the *unary logical relation*, R^σ, inductively as follows:

$$\begin{aligned}
R^{t^\kappa} &\stackrel{\text{def}}{=} T^{\varkappa(\kappa)} \\
R^{\mathsf{c}} &\stackrel{\text{def}}{=} D_{\mathsf{c}}, \quad \text{if } \mathsf{c} \text{ is in } \mathcal{C}_{\mathcal{T}} \\
R^{(\sigma\ \tau)} &\stackrel{\text{def}}{=} (R^\sigma\ R^\tau) \\
R^{(\lambda t.\tau)} &\stackrel{\text{def}}{=} \Lambda T.R^\tau \\
R^{\sigma \rightarrow \tau} &\stackrel{\text{def}}{=} \Lambda f.\forall x.R^\sigma(x) \supset R^\tau(f\ x) \\
R^{\forall t.\sigma} &\stackrel{\text{def}}{=} \Lambda x.\forall T.R^\sigma(x)
\end{aligned}$$

In the above definition, for each $\mathsf{c} \in \mathcal{C}_{\mathcal{T}}$, the term D_{c} is defined as in [Lei90]. Namely, let $(\{\mathsf{c}_1, \ldots, \mathsf{c}_m\}, \{\mathsf{g}_1 \colon \phi_i, \ldots, \mathsf{g}_n \colon \phi_n\})$ be a signature of a data structure. Assume for $j = 1, \ldots, m$, c_j is of kind $\kappa_{j1} \Rightarrow \cdots \kappa_{jl_j} \Rightarrow \star$. Let c_j be a type variable of the same kind as c_j, and let $\hat{\phi}$ denote a type constructor obtained from ϕ by replacing all c_j's by c_j's. Let $\vec{C}$ be the sequence of variables of type theory corresponding to $c_1, \ldots, c_m$. Let $t_{j1}, \ldots, t_{jl_j}$ be a sequence of fresh type constructor variables of kinds $\kappa_{j1}, \ldots, \kappa_{jl_j}$ and let $\vec{T_j}$ be the corresponding sequence of variables of type theory. Then define

$$D_{\mathsf{c}_j} \stackrel{\text{def}}{=} \Lambda \vec{T_j}.\Lambda x.\forall \vec{C}.R^{\hat{\phi}_1}(g_1) \supset \cdots \supset R^{\hat{\phi}_n}(g_n) \supset (C_j\ T_{j1} \cdots T_{jl_j})(x).$$

The definition of logical relations, in particular of D_{c_j}, is well defined, since all types $\hat{\phi}_i$, in the definition of D_{c_j} above, are pure.

Example 2.2.8. Let data structure $\mathcal{B} = (\{\texttt{Bool}^\star\}, \{\texttt{true}\colon\texttt{Bool}, \texttt{false}\colon\texttt{Bool}\})$. The logical relation corresponding to `Bool` is

$$R^{\texttt{Bool}} \stackrel{\text{def}}{=} \Lambda x.\forall P^{\imath\Rightarrow o}.P(\texttt{true}) \supset P(\texttt{false}) \supset P(x).$$

Example 2.2.9. Recall the data structure $\mathcal{N}$ defining the natural numbers. The logical relation $R^{\texttt{Nat}}$ is nothing but the Peano's induction axiom:

$$\Lambda x.\forall P^{\imath\Rightarrow o}.(\forall y.P(y) \supset P(\texttt{succ}\ y)) \supset P(\texttt{zero}) \supset P(x).$$

Example 2.2.10. Consider the data structure $\mathcal{L}$ of lists defined in example 2.2.1. The logical relation $R^{\texttt{List}}$ corresponding to the type constructor `List` is defined to be

$$\begin{aligned}\Lambda T^{\imath\Rightarrow o}.\Lambda z.\forall P^{(\imath\Rightarrow o)\Rightarrow\imath\Rightarrow o}.\forall S^{\imath\Rightarrow o}(\forall x.S(x) \supset \forall y.(P\ S)(y) \supset (P\ S)(\texttt{cons}\ x\ y)))\\ \supset \forall S^{\imath\Rightarrow o}.(P\ S)(\texttt{nil})\\ \supset (P\ T)(z).\end{aligned}$$

The relation $R^{(\texttt{List}\ t)}(z)$ is provably equivalent to a more familiar induction axiom

$$\forall P.(\forall xy.T(x) \supset P(y) \supset P(\texttt{cons}\ x\ y)) \supset P(\texttt{nil}) \supset P(z)$$

used in [Mai91].

The definition of unary logical relations, can be easily modified to obtain a definition of logical relations of any arity. Fix arity $k \geq 0$, and let $\varkappa(\star) \stackrel{\text{def}}{=} \imath\Rightarrow\cdots\Rightarrow\imath\Rightarrow o$ (k $\imath$'s). For higher kinds κ, let $\varkappa(\kappa)$ be defined as before. Then the k-ary logical relations are defined as follows:

$$\begin{aligned}
R^{t^\kappa} &\stackrel{\text{def}}{=} T^{\varkappa(\kappa)}\\
R^{\mathsf{c}} &\stackrel{\text{def}}{=} D_{\mathsf{c}}, \text{ for } \mathsf{c} \in \mathcal{C}_{\mathcal{T}}\\
R^{(\sigma\ \tau)} &\stackrel{\text{def}}{=} (R^\sigma\ R^\tau)\\
R^{(\lambda t.\tau)} &\stackrel{\text{def}}{=} \Lambda T.R^\tau\\
R^{\sigma\to\tau} &\stackrel{\text{def}}{=} \Lambda f^{(1)}\cdots f^{(k)}.\forall x^{(1)}\cdots x^{(k)}.\\
&\qquad R^\sigma(x^{(1)},\cdots,x^{(k)}) \supset R^\tau(f^{(1)}(x^{(1)}),\cdots,f^{(k)}(x^{(k)}))\\
R^{\forall t.\sigma} &\stackrel{\text{def}}{=} \Lambda x^{(1)}\cdots x^{(k)}.\forall T.R^\sigma(x^{(1)},\cdots,x^{(k)})
\end{aligned}$$

In the above definition D_{c} is defined using k-ary versions of $R^{(\cdot)}$, and $x^{(i)}$ and $f^{(i)}$ denote distinct variables of type $\imath$.

2.3 The Fundamental Theorem of Logical Relations

The Fundamental Theorem we prove in this section is a key tool in proving free theorems. As we explain in Section 2.4, free theorems about a term M, are instances of the logical relation corresponding to the type of M, and the Fundamental Theorem states that, whenever M

is typable, the logical relation corresponding to the type M is provable.

We prove two versions of the Fundamental Theorem. The first theorem applies to all F_ω types, and states that if a term is typable, then the corresponding logical relation is provable. However, we need to place certain restrictions on the types of constants. For example, it would be wrong to assign type $\forall t.t$ to a constant g, since the corresponding logical relation, $\forall T.T(\mathsf{g})$, is not provable. The constraints we place on types of constants, help us avoid such problems.

The second version of the Fundamental Theorem applies only to a subset of types similar to *ML* types, and says that a term is typable *if and only if* the corresponding logical relation is provable.

We first state a lemma which is essentially the Fundamental Theorem for the pure calculus, that is we will assume the types of constants are such that the corresponding logical relations are provable.

Lemma 2.3.1. Let $\mathfrak{Th}$ be a λ-theory, and suppose for all constants g in $\mathcal{C}_\Lambda$, if $\mathsf{Type}(\mathsf{g}) = \phi$ then $\mathfrak{Th} \vdash_{TT} R^\phi(\mathsf{g})$. Then, for any λ-term M

$$\text{if} \quad \mathfrak{Th} \vdash \{x_i\colon \sigma_i\} \triangleright M\colon \sigma \quad \text{then} \quad \mathfrak{Th}, \{R^{\sigma_i}(x_i)\} \vdash_{TT} R^\sigma(M).$$

Proof: The proof of this lemma is a straight-forward generalization of the proof of Lemma 2.5 in [Mai91] to type theory. ■

We would like to reformulate the assumptions of the above lemma in terms of typing constraints. In other words, how can one characterize a type ϕ, such that if $\mathsf{g}\colon\phi$, $R^\phi(\mathsf{g})$ is provable in type theory?

For example, the generators for the natural numbers, `zero` and `succ` have types `Nat` and `Nat`$\rightarrow$`Nat` respectively, so we need to show that $\vdash_{TT} R^{\mathtt{Nat}}(\mathtt{zero})$ and $\vdash_{TT} R^{\mathtt{Nat}\rightarrow\mathtt{Nat}}(\mathtt{succ})$. These relations are, indeed, provable and the Church numeral $\lambda sz.z$ and Church successor function $\lambda nsz.(s\ (n\ s\ z))$ represent exactly the proofs of these statements (see [Lei90])! Similarly, if the function symbol `add` is assigned type `Nat`$\rightarrow$`Nat`, we need to show that $\mathfrak{P}_{\mathtt{add}} \vdash_{TT} R^{\mathtt{Nat}\rightarrow\mathtt{Nat}}(\mathtt{add})$. Again, the lambda representation of the addition represents the proof of this relation.

We define a class of *sound* types, which can safely be assigned to generators of a data structure, so that the corresponding logical relations are provable.

Definition 2.3.2. We say a type ϕ is *sound* if

$$\phi = \forall \vec{t}.\sigma_1 \rightarrow \cdots \rightarrow \sigma_k \rightarrow (\mathsf{c}\ \rho_1 \cdots \rho_l),$$

where each ρ_i is pure and each $\sigma_i \in \mathcal{T}^+$. The the set of types $\mathcal{T}^+$ is defined by the following grammar:

$$\begin{array}{rcl} \mathcal{T}^+ & ::= & (\mathsf{c}\ \mathcal{T}\cdots\mathcal{T}) \mid (t\ \mathcal{T}\cdots\mathcal{T}) \mid \forall t.\mathcal{T}^+ \mid \mathcal{T}^- \rightarrow \mathcal{T}^+ \\ \mathcal{T}^- & ::= & (t\ \mathcal{T}\cdots\mathcal{T}) \mid \forall t.\mathcal{T}^- \mid \mathcal{T}^+ \rightarrow \mathcal{T}^-, \end{array}$$

where c ranges over a set of type constructor constants, and $\mathcal{T}$ is the set of pure type constructors. We may write $\mathcal{T}^+(\mathcal{C})$ to emphasize that constant type constructors in $\mathcal{T}^+$ are taken from a set $\mathcal{C}$.

Lemma 2.3.3. Let $\mathsf{g} \in \mathcal{C}_\Lambda$ be a generator of a data structure such that $\mathsf{Type}(\mathsf{g})$ is sound. Then,

$$\vdash_{TT} R^{\phi}(\mathsf{g}).$$

Proof: Let $(\{\mathsf{c}_1, \ldots, \mathsf{c}_m\}, \{\mathsf{g}_1{:}\,\phi_1, \ldots, \mathsf{g}_n{:}\,\phi_n\})$ be the signature of a data structure such that $\mathsf{g} = \mathsf{g}_i$ for some i. Recall that we write $\hat{\phi}$ to for the type obtained from a type ϕ, be substituting a fresh variable c_i for each constant c_i in ϕ. To prove the lemma, we first show the following. Let $\Gamma = \{R^{\phi_1}(\mathsf{g}_1), \ldots, R^{\phi_n}(\mathsf{g}_n)\}$ and let $\widehat{\Gamma} = \left\{R^{\hat{\phi}_1}(\mathsf{g}_1), \ldots, R^{\hat{\phi}_n}(\mathsf{g}_n)\right\}$. Then for every σ,

(1) if $\sigma \in \mathcal{T}^{+}(\{\mathsf{c}_1, \ldots, \mathsf{c}_m\})$, then $\widehat{\Gamma}, R^{\sigma}(x) \vdash R^{\hat{\sigma}}(x)$, and

(2) if $\sigma \in \mathcal{T}^{-}(\{\mathsf{c}_1, \ldots, \mathsf{c}_m\})$, then $\widehat{\Gamma}, R^{\hat{\sigma}}(x) \vdash R^{\sigma}(x)$.

We have four cases:

Case $\sigma \equiv (t\, \rho_1 \cdots \rho_k)$: Trivial, since $\sigma \equiv \hat{\sigma}$.

Case $\sigma \equiv (\mathsf{c}_j\, \rho_1 \cdots \rho_k)$: (1) follows from the definition of $R^{(\mathsf{c}_j\ \rho_1 \cdots \rho_k)}(x)$, and (2) is vacuously true.

Case $\sigma \equiv \tau_1 \to \tau_2$: Recall that $R^{\tau_1 \to \tau_2}(x) \equiv \forall y. R^{\tau_1}(x) \supset R^{\tau_2}(x\, y)$. Therefore, $\widehat{\Gamma}, R^{\tau_1 \to \tau_2}(x) \vdash R^{\tau_1}(y) \supset R^{\tau_2}(xy)$ and $\widehat{\Gamma}, R^{\tau_1 \to \tau_2}(x), R^{\tau_1}(y) \vdash R^{\tau_2}(xy)$. If $\tau_1 \to \tau_2$ is in $\mathcal{T}^{+}$, we must have that τ_1 is in $\mathcal{T}^{-}$ and τ_2 is in $\mathcal{T}^{+}$. Then, by the induction hypothesis $\widehat{\Gamma}, R^{\hat{\tau}_1}(y) \vdash R^{\tau_1}(y)$ and $\widehat{\Gamma}, R^{\tau_2}(x\, y) \vdash R^{\hat{\tau}_2}(x\, y)$. Therefore, $\widehat{\Gamma}, R^{\tau_1 \to \tau_2}(x) \vdash R^{\hat{\tau}_1 \to \hat{\tau}_2}(x)$. This establishes (1), and (2) follows by a similar argument.

Case $\sigma \equiv \forall t.\tau$: We have $R^{\forall t.\tau}(x) \equiv \forall T. R^{\tau}(x)$, and (1) and (2) follow easily by the induction hypothesis.

To complete the proof, assume $\phi \equiv \forall \vec{t}. \sigma_1 \to \cdots \sigma_k \to (\mathsf{c}\ \rho_1 \cdots \rho_l)$, and let

$$\Delta = \{R^{\sigma_1}(x_1), \ldots, R^{\sigma_k}(x_k)\} \quad \text{and} \quad \widehat{\Delta} = \{R^{\hat{\sigma}_1}(x_1), \ldots, R^{\hat{\sigma}_k}(x_k)\}.$$

By assumption we have that each σ_j is in $\mathcal{T}^{+}$ and therefore, by (1) above, $\widehat{\Gamma}, \Delta \vdash R^{\hat{\sigma}_j}(x_j)$ for all j. Moreover, since $\widehat{\Gamma}, \widehat{\Delta} \vdash (C\ R^{\rho_1} \cdots R^{\rho_l})(\mathsf{g}\,\vec{x}))$, we also have $\widehat{\Gamma}, \Delta \vdash (C\ R^{\rho_1} \cdots R^{\rho_l})(\mathsf{g}\,\vec{x}))$ from which we conclude $\vdash R^{\phi}(\mathsf{g})$ by ($\supset$ I) and ($\forall$ I). ■

Remark. Lemma 2.3.3 is an extension of Theorem 3.2 in [Lei90] to a more general class of types.

Lemma 2.3.3 covers the generators of a data structure. To say that, for a function symbol $\mathtt{f}$, of type $\sigma \to \tau$, logical relation $R^{\sigma \to \tau}(\mathtt{f})$ is provable, assuming equations $\mathfrak{P}_{\mathtt{f}}$, is to say that $\mathtt{f}$ denotes a provably total function from σ to τ [Lei90]. We will satisfy ourselves with this criterion for the function symbols included in the calculus, and note that this covers quite a large set of functions used in practice. For example, every *iterative function* on simple data structure is provably total, and in fact, its Böhm-Berarducci representation, which must exist [BB85], represents the proof of its totality [Lei90].

Theorem 2.3.4 (Fundamental Theorem of Logical Relations). Let $\mathfrak{Th}$ be a λ-theory, and assume that for every generator of a data structure, g, $\mathsf{Type}(\mathsf{g})$ is sound, and all function symbols denote provably total functions. Then, for any λ-term M,

$$\text{if} \quad \mathfrak{Th} \vdash \{x_i\colon \sigma_i\} \triangleright M\colon \sigma \quad \text{then} \quad \mathfrak{Th}, \{R^{\sigma_i}(x_i)\} \vdash_{TT} R^{\sigma}(M).$$

Proof: Immediate form lemmas 2.3.1 and 2.3.3. ■

To prove the converse of the Fundamental Theorem for *ML* types, we compare a complete type inference model with a model of type theory. Recall that a type inference model is a triple $(\mathfrak{T}, \mathcal{D}, \mathit{subset})$, where $\mathfrak{T}$ is a type structure, $\mathcal{D}$ is a model of untyped lambda calculus, and *subset* is a function mapping interpretations of types into subsets of $\mathcal{D}$. In what follows we will define a complete model of type inference based on the term model of untyped lambda calculus, and compare it with the model of type type theory defined by the same term model of the untyped lambda calculus.

The types we consider are essentially *ML* types. Namely, we restrict our attention to the simple types over type variables and base types. In addition assume that no type constructor constant has a kind of order more than one.

Definition 2.3.5. A type constructor σ is said to be *ML* type constructor if it is of kind $\star\Rightarrow\cdots\Rightarrow\star$ (or kind $\star$ as a special case), it is $\forall$-free, and contains no subexpression of kind greater than one (i.e. all subexpressions are of kind $\star$ or $\star\Rightarrow\cdots\Rightarrow\star$). A type of the form $\forall\vec{t}.\sigma$ is called *quantified ML* type, if σ is *ML* type.

Fix a λ-theory, $\mathfrak{Th}$, and let $\mathcal{D}$ be a term model consisting of $\mathfrak{Th}$-equivalence classes of λ-terms. If M is a λ-term, let $[M]_{\mathfrak{Th}}$ be its equivalence class.

We construct the type structure $\mathfrak{T} = (\{\mathcal{U}_\kappa\}, \rightarrow, \forall)$ to be the term model of type constructors. By abuse of notation, we write σ to denote the equivalence class of a type constructor σ.

The *subset* function assigns to each type σ a set D_σ defined as follows. Let Σ to be an infinite set of type assignments to λ-variables such that: Σ only assigns pure *ML* types, each variable is assigned a type exactly once by Σ, and each pure *ML* type is assigned to infinitely many variables by Σ. Then D_σ is defined to be the set

$$D_\sigma \stackrel{\text{def}}{=} \{[M]_{\mathfrak{Th}} \mid \text{for some finite } \Gamma \subseteq \Sigma, \mathfrak{Th} \vdash \Gamma \triangleright M\colon \sigma\},$$

It is easy to see that the *subset* function can be extended to type constructors of higher kind. Let $\sigma \in \mathcal{U}_{\star\Rightarrow\cdots\Rightarrow\star}$ and define D_σ to be the function such that for all $\tau_1, \ldots, \tau_k$, $(D_\sigma\ D_{\tau_1} \cdots D_{\tau_k}) = D_{(\sigma\ \tau_1 \cdots \tau_k)}$.

Assuming that no $D_{(\mathsf{c}\ t_1 \cdots t_k)}$ is empty, it is easy to see that this gives us a type inference model. Therefore, assuming $\mathfrak{Th}$ contains (η), we can characterize the sets D_σ as follows (see [Mit88]):

$$\begin{array}{rcl} D_{\sigma\rightarrow\tau} & = & \{d \in \mathcal{D} \mid d \cdot D_\sigma \subseteq D_\tau\}, \text{ and} \\ D_{\forall t.\sigma} & = & \bigcap_\tau D_{\sigma[\tau/t]}. \end{array}$$

Now consider sets $D_{(\mathsf{c}\, t_1 \cdots t_k)}$, where $(\mathsf{c}\, t_1 \cdots t_k)$ is a base type. For simple data structures, we get a clean characterization of the sets $D_{(\mathsf{c}\ t_1 \cdots t_k)}$. Namely we have,

Lemma 2.3.6. Let $(\{\mathsf{c}_1, \ldots \mathsf{c}_m\}, \{\mathsf{g}_1\colon \phi_1, \ldots \mathsf{g}_n\colon \phi_n\})$ be a signature of a simple data structure. For each c_i, the set $D_{(\mathsf{c}_i\ t_1 \cdots t_k)}$ consists exactly of equivalence classes $[M]_{\mathfrak{Th}}$, where M is an applicative term over g_i's and terms $N \in \bigcup_t D_t$.

Proof: Observe that M cannot be a variable, since Σ only assigns pure types to variables. M cannot be an abstraction, for it would have a functional type. Therefore, M is an application $(N_1\ N_2 \cdots N_n)$. Now, by the Normalization Theorem, assume M is in normal form, and therefore N_1 cannot be an abstraction. Similarly, for the same reason M cannot be a variable, N_1 cannot be a variable either, so it must be one of the g_i's. From the restriction on the type of g_i we conclude that each N_j (for $j = 2, \ldots, n$) must either be in D_t for some t, or by induction, is an applicative term over g_i's and terms in $\bigcup_t D_t$. ■

On the other hand the meaning, $\llbracket R^{(\mathsf{c}\, t_1 \cdots t_k)} \rrbracket \rho$, of the unary logical relation $R^{(\mathsf{c}\, t_1 \cdots t_k)}$ in $\mathcal{D}$ is exactly the set of equivalence classes of terms which are applicative terms over generators g_i and terms in $\rho(T_i)$. Thus, if ν is a type substitution, and $\widetilde{\nu}$ is an environment mapping such that $\widetilde{\nu}(T) = D_{\nu(t)}$, we have that $\llbracket R^{(\mathsf{c}\ t_1 \cdots t_k)} \rrbracket \widetilde{\nu} = D_{\nu(\mathsf{c}\ t_1 \cdots t_k)}$. In fact we can generalize this correspondence beyond base types.

Lemma 2.3.7. Assume $\mathfrak{Th}$ is a λ-theory containing (η), and that all data structures are simple. Let ν be a type substitution. Write $\widetilde{\nu}$ for an environment mapping, induced by ν, mapping variables of type theory to the model of type theory defined by $\mathcal{D}$. Assume moreover that $\widetilde{\nu}(T^{\varkappa(\star)}) = D_{\nu(t^\star)}$ and for all τ_i, $(\widetilde{\nu}(T^{\varkappa(\star\Rightarrow \cdots \Rightarrow\star)})\ D_{\tau_1} \cdots D_{\tau_k}) = (D_{\nu(t^{\star\Rightarrow\cdots\Rightarrow\star})}\ D_{\tau_1} \cdots D_{\tau_k})$. Let σ be a *ML* type constructor. Then,

- if σ is of kind $\star$ then $\llbracket R^\sigma \rrbracket \widetilde{\nu} = D_{\nu(\sigma)}$, and
- if σ is of kind $\star\Rightarrow \cdots \Rightarrow\star$ then for all τ_i, $(\llbracket R^\sigma \rrbracket \widetilde{\nu}\ D_{\tau_1} \cdots D_{\tau_k}) = (D_{\nu(\sigma)}\ D_{\tau_1} \cdots D_{\tau_k})$.

In particular for any closed quantified *ML* type τ, $\llbracket R^\tau \rrbracket \widetilde{\nu} \subseteq D_{\nu(\tau)}$.

Proof: By induction of structure of σ.

Case $\sigma = t$: This case follows from assumption.

Case $\sigma = \mathsf{c}$: Given $\tau_1, \ldots, \tau_k$ for $k \geq 0$, $(\llbracket R^{\mathsf{c}} \rrbracket \widetilde{\nu}\ D_{\tau_1} \cdots D_{\tau_k}) = \llbracket R^{(\mathsf{c}\, t_1 \cdots t_k)} \rrbracket \widetilde{\nu}[D_{\tau_i}/T_i]$ which, by Lemma 2.3.6 and the discussion following the lemma, is the same as $D_{(\mathsf{c}\, \tau_1 \cdots \tau_k)} = (D_{\mathsf{c}}\ D_{\tau_1} \cdots D_{\tau_n})$.

Case $\sigma = (\tau\ \tau_1 \cdots \tau_k)$: It suffices to consider the case where τ is of kind $\star\Rightarrow \cdots \Rightarrow\star$ and τ_i's are of kind $\star$. By definition $R^{(\tau\ \tau_1 \cdots \tau_k)} = (R^\tau\ R^{\tau_1} \cdots R^{\tau_k})$, so $\llbracket R^{(\tau\ \tau_1 \cdots \tau_k)} \rrbracket \widetilde{\nu} = (\llbracket R^\tau \rrbracket \widetilde{\nu}\ \llbracket R^{\tau_1} \rrbracket \widetilde{\nu} \cdots \llbracket R^{\tau_k} \rrbracket \widetilde{\nu})$. Moreover, by induction hypotheses $\llbracket R^{\tau_i} \rrbracket \widetilde{\nu} = D_{\nu(\tau_i)}$, so $(\llbracket R^\tau \rrbracket \widetilde{\nu}\ \llbracket R^{\tau_1} \rrbracket \widetilde{\nu} \cdots \llbracket R^{\tau_k} \rrbracket \widetilde{\nu}) = (D_{\nu(\tau)}\ D_{\nu(\tau_1)} \cdots D_{\nu(\tau_k)}) = D_{\nu(\tau\ \tau_1 \cdots \tau_k)}$.

Case $\sigma = (\lambda \vec{t}.\tau)$: Again, we only need to consider the case where t's and τ are of kind $\star$ so that σ is of kind $\star\Rightarrow \cdots \Rightarrow\star$. By definition $\left\llbracket R^{(\lambda \vec{t}.\tau)} \right\rrbracket \widetilde{\nu}$ is a function f such that for all $d_i \subseteq \mathcal{D}$, $(f\ d_1 \cdots d_k) = \llbracket R^\tau \rrbracket \widetilde{\nu}[\vec{d}/\vec{T}]$. Since for $d = D_{\tau'}$, $\widetilde{\nu}[d/T]$ still satisfies the assumption of the lemma, by induction hypothesis, for $D_{\tau_i'}$, $\llbracket R^\tau \rrbracket \widetilde{\nu}[D_{\tau_i'}/T_i] = D_{\nu[\tau_i'/t_i](\tau)}$. Therefore, f coincides with $D_{\nu(\lambda \vec{t}.\tau)}$ on elements of the form D_τ as required.

Case $\sigma = \tau_1 \rightarrow \tau_2$: By definition, $[\![R^{\tau_1\rightarrow\tau_2}]\!]\tilde{\nu} = \{d \in \mathcal{D} \mid \forall e \in [\![R^{\tau_1}]\!]\tilde{\nu} \,.\, d \cdot e \in [\![R^{\tau_2}]\!]\tilde{\nu}\}$. By induction hypothesis, $[\![R^{\tau_i}]\!]\tilde{\nu} = D_{\nu(\tau_i)}$, $i = 1, 2$, so $[\![R^{\tau_1\rightarrow\tau_2}]\!]\tilde{\nu} = D_{\nu(\tau_1\rightarrow\tau_2)}$.

Finally, for a type $\forall t.\sigma$ we have that $D_{\forall t.\sigma} = \bigcap_\tau D_{\sigma[\tau/t]}$, so that

$$[\![R^{\forall t.\sigma}]\!]\tilde{\nu} = \bigcap_d [\![R^\sigma]\!]\tilde{\nu}[d/t] \subseteq D_{\nu(\forall t.\sigma)}.$$

■

We are now ready to state and prove the Fundamental Theorem for the *ML* types. However, in Section 2.5 we need a special case of the Fundamental Theorem, which we state in the following lemma.

Lemma 2.3.8. Let $\mathfrak{Th}$ be a theory containing (η), and assume all data structures are simple, types of generators are sound, and function symbols denote total functions. Then if σ is a quantified *ML* type, and σ_i are pure *ML* types we have,

$$\mathfrak{Th}, \{R^{\sigma_i}(x_i)\} \models_{\mathcal{D}} R^\sigma(M) \quad \text{implies} \quad \mathfrak{Th} \vdash \{x_i\colon \sigma_i\} \triangleright M\colon \sigma.$$

Proof: Let Σ be an assignment of types to variables as defined earlier in this section on page 79. Assume $\mathfrak{Th}, \{R^{\sigma_i}(x_i)\} \models_{\mathcal{D}} R^\sigma(M))$. For each σ_i let x_i' be such that $x_i'\colon \sigma_i \in \Sigma$ and let $\nu(x_i) = x_i'$ and identity on other variables as well as on type variables. Let $\tilde{\nu}$ be the environment map induced by ν (as in Lemma 2.3.7). Then $[x_i']_{\mathfrak{Th}} \in D_{\sigma_i} = [\![R^{\sigma_i}]\!]\tilde{\nu}$ and by assumption $[\nu(M)]_{\mathfrak{Th}} \in [\![R^\sigma]\!]\tilde{\nu}$. By Lemma 2.3.7, $[\nu(M)]_{\mathfrak{Th}} \in D_\sigma$. Therefore $\mathfrak{Th} \vdash \Gamma \triangleright M[x_i'/x_i]\colon \sigma$, for some finite $\Gamma \subseteq \Sigma$. Moreover, Γ must contain all free variables in $M[x_i'/x_i]$, i.e. all the x_i''s, and we can conclude that $\mathfrak{Th} \vdash \{x_i\colon \sigma_i\} \triangleright M\colon \sigma$. ■

Theorem 2.3.9 (Fundamental Theorem of Logical Relations for *ML* types). Under assumptions of Lemma 2.3.8, we have,

$$\mathfrak{Th} \vdash \{x_i\colon \sigma_i\} \triangleright M\colon \sigma \quad \text{iff} \quad \mathfrak{Th}, \{R^{\sigma_i}(x_i)\} \vdash_{TT} R^\sigma(M).$$

Proof: Immediate from Theorem 2.3.4, soundness of type theory and Lemma 2.3.8. ■

Remark. The Fundamental Theorem can be easily modified to logical relation of any arity. Let $R^{(k)\sigma}$ denote the k-ary version of R^σ, and let $M^{(j)} = M[x_1^{(j)}/x_1, \ldots, x_n^{(j)}/x_n]$, where $x^{(j)}$ denote distinct variables. Then the conclusion of the Fundamental Theorem reads

$$\text{if} \quad \mathfrak{Th} \vdash \{x_i\colon \sigma_i\} \triangleright M\colon \sigma \quad \text{then} \quad \mathfrak{Th}, \left\{R^{(k)\sigma_i}(x_i^{(1)}, \ldots, x_i^{(k)})\right\} \vdash_{TT} R^{(k)\sigma}(M^{(1)}, \cdots, M^{(k)}).$$

The proof of the k-ary version of the theorem can be obtained form the proof of the unary version by replacing each unary version of a logical relation by the corresponding k-ary version.

2.4 Deriving the Free Theorems

Having proven the Fundamental Theorem we are ready to show the proofs of the free theorems. In general, our free theorems are equations by convertibility that follow from typing assumptions. We now try to describe the general heuristics involved in the deduction of free theorems from the instances of the Fundamental Theorem. Let $A[M, N_1, \ldots, N_r]$ and $B[M, N_1, \ldots, N_r]$ denote lambda terms (in $\Lambda(\mathcal{C}_\Lambda)$) with possible occurrences of M and N_i as subterms. We can state a generic free theorem as follows:

"Let M and $N_1, \ldots, N_r$ be terms such that $\emptyset \triangleright M{:}\,\sigma$ and $\Gamma_i \triangleright N_i{:}\,\tau_i$. Then

$$\mathfrak{Th} \vdash_\lambda A[M, N_1, \ldots, N_r] = B[M, N_1, \ldots, N_r]."$$

The term M represents the polymorphic function that the theorem describes, and N_i are other terms for which the typing hypotheses are necessary. All the examples from [Wad89] can be expressed in this form. The result of a free theorem is an equation that holds by convertibility form equations in $\mathfrak{Th}$. In most of the examples, $\mathfrak{Th}$ contains only (β), (η), as well as δ-axioms defining certain functions (e.g. program for `map`). However, it may contain additional assumptions as in the example about sort-like functions (see [Wad89] and Proposition 2.4.3).

All the examples of free theorems are deduced from the Fundamental Theorem using only rules ($\forall$ E), ($\supset$ E) and (Cong). This suggests the following heuristics for deducing the target equation $A[M, N_1, \ldots, N_r] = B[M, N_1, \ldots, N_r]$. Without loss of generality, we may assume that $R^\sigma(M)$ is of the form $\forall\vec{T}.\forall\vec{x}.\Phi_1 \supset \cdots \supset \Phi_n \supset (T_i(M\ x_1 \cdots x_n))$.

- Start with k-ary version of $R^\sigma(M)$ where k is the number of occurrences of M in the target equation.
- Instantiate T_i to Ψ_i, and $x_j^{(1)}, \ldots, x_j^{(k)}$, for $j = 1, \ldots, n$, to appropriate lambda terms $L_j^{(i)}$ so that $\Psi_i((M\ L_1^{(1)} \cdots L_n^{(1)}), \ldots, (M\ L_1^{(k)} \cdots L_n^{(k)}))$ equals, or is equivalent, to the target equation.
- Find instantiations $\vec{\Psi}$ and $\vec{L}$ for universally quantified variables $\vec{T}$ and $\vec{x}$ (other than those instantiated in the previous step) such that $\mathfrak{Th} \vdash_{TT} \Phi_i[\vec{\Psi}/\vec{T}, \vec{L}/\vec{x}], i = 1, \ldots, n$. In proving this, one will, in general, use the typing assumptions about N_i's and the Fundamental Theorem, as well as equations in $\mathfrak{Th}$.
- Using ($\supset$ E) conclude the target equation is provable, i.e. $\mathfrak{Th} \vdash_{TT} A[M, N_1, \ldots, N_r] = B[M, N_1, \ldots, N_r]$.
- Finally, by Theorem 2.2.7 we conclude $\mathfrak{Th} \vdash_\lambda A[M, N_1, \ldots, N_r] = B[M, N_1, \ldots, N_r]$.

This is an entirely mechanical procedure, where only nontrivial step is finding formulae $\vec{\Psi}$ and terms $\vec{L}$, so we can prove all clauses $\Phi_i[\vec{\Psi}/\vec{T}, \vec{L}/\vec{x}]$. The following examples illustrate how the above heuristics is used to prove several free theorems.

Proposition 2.4.1. Assume the calculus contains data structures defining lists and natural numbers as defined in Section 2.2.3. Let $M \in \Lambda(\mathcal{C}_\Lambda)$ be such that $\emptyset \triangleright M{:}\,\forall t.(\mathtt{List}\ t){\to}\mathtt{Nat}$, and let $L \in \Lambda(\mathcal{C}_\Lambda)$ be such that $\{x_1{:}\,r, \ldots x_k{:}\,r\} \triangleright L{:}\,(\mathtt{List}\ r)$. Then,

$$(M\ L) =_{\beta\eta} (M\ ((\lambda x_1 \cdots x_k.L)\ y_1 \cdots y_k)).$$

More informally, this proposition says, that a polymorphic function from lists to natural numbers can only depend on the number of elements in the list.

Proof: We start with the binary version (since M occurs twice in the target equation) of the logical relation corresponding to the type of M. This relation is provable by the Fundamental Theorem, so we have $\mathfrak{Th} \vdash_{TT} R^{\forall t.(\mathtt{List}\ t)\rightarrow \mathtt{Nat}}(M, M)$. We expand this relation into

$$\forall T.\forall z' z'' \,.\, R^{(\mathtt{List}\ t)}(z', z'') \supset R^{\mathtt{Nat}}((M\ z'), (M\ z'')),$$

which is provably equivalent to

$$\begin{aligned}\forall T P.\forall z' z'' \,.\, & R^{(\mathtt{List}\ t)}(z', z'') \\ & \supset (\forall x' x''. P(x', x'') \supset P((\mathtt{succ}\, x'), (\mathtt{succ}\, x''))) \\ & \quad \supset P(\mathtt{zero}, \mathtt{zero}) \supset P((M\ z'), (M\ z'')).\end{aligned}$$

We first instantiate P to $\Lambda uv.u = v$, and z' and z'' to L and $((\lambda x_1 \cdots x_k.L)\, y_1 \cdots y_k)$, so that $P((M\ z'), (M\ z''))$ converts to the target equation $((M\ L) = (M\ ((\lambda x_1 \cdots x_k.L)\, y_1 \cdots y_k)))$. Instantiating T to an always true relation, say $\Lambda xy.Q \supset Q$, and using the Fundamental Theorem we can prove

$$R^{(\mathtt{List}\ t)}(z'\ z'')[L/z', ((\lambda x_1 \cdots x_k.L)\, y_1 \cdots y_k)/z'', \Lambda xy.Q \supset Q/T].$$

The remaining two clauses are obviously provable with the above instantiation of P, so by $(\supset \mathrm{E})$ we derive

$$\mathfrak{Th} \vdash_{TT} (M\ L) = (M\ ((\lambda x_1 \cdots x_k.L)\, y_1 \cdots y_k)).$$

Finally by Theorem 2.2.7, we conclude

$$(M\ L) =_{\beta\eta} (M\ ((\lambda x_1 \cdots x_k.L)\, y_1 \cdots y_k)).$$

■

Remark. In the theorem above, one can include $\mathfrak{P}_{\mathtt{map}}$ so that the function M is allowed to use function symbol `map`. Then, of course, the conclusion of the theorem would read:

$$\mathfrak{Th}, \mathfrak{P}_{\mathtt{map}} \vdash_{\lambda} (M\ L) = (M\ ((\lambda x_1 \cdots x_k.L)\, y_1 \cdots y_k)).$$

The following proposition is proved similarly.

Proposition 2.4.2. Let S be a pure λ-term, such that $\emptyset \triangleright S : \forall stu.(s \rightarrow t \rightarrow u) \rightarrow (s \rightarrow t) \rightarrow s \rightarrow u$. Then,

$$S =_{\beta\eta} (\lambda xyz.(x\ z)\,(y\ z)).$$

Proof: (Proposition 2.4.2) The target equation is $S = \lambda xyz.(x\ y)\,(x\ z)$, so we start with the unary version of the logical relation $R^{\forall ptu.(p\rightarrow t\rightarrow u)\rightarrow(p\rightarrow t)\rightarrow p\rightarrow u}(S)$ which is provable by the Fundamental Theorem. This relation is provably equivalent to

$$\begin{aligned}\forall PTU.\forall xyz \,.\, & (\forall vw.P(V) \supset T(w) \supset U(x\ v\ w))) \\ & \supset (\forall v.P(v) \supset T(y\ v))) \supset P(z) \supset U(S\ x\ y\ z)).\end{aligned}$$

We first instantiate U to $\Lambda v.v = (x\ z)\,(y\ z)$, so that $U(S\ x\ y\ z))$ converts to $((S\ x\ y\ z) = ((x\ z)\,(y\ z)))$. We then instantiate P to $\Lambda v.v = z$, and T to $\Lambda v.v = (y\ z)$. Then the first three clauses are easily provable and we derive $\beta, \eta \vdash_{TT} (S\ x\ y\ z) = ((x\ z)\,(y\ z))$. Using (η) and Theorem 2.2.7, we get $S =_{\beta\eta} \lambda xyz.(x\ z)\,(y\ z)$. ■

For illustration we show how the result about sort-like functions is proved.

Proposition 2.4.3. Assume the calculus contains data structures for booleans and lists, and $\mathfrak{Th}$ contains the program for `map`. Let σ be arbitrary but fixed type. Let M, C and L be terms such that $\emptyset \triangleright M\colon \forall t.(t \to t \to \mathtt{Bool}) \to (\mathtt{List}\,t) \to (\mathtt{List}\,t)$, $\emptyset \triangleright C\colon (\sigma \to \sigma \to \mathtt{Bool})$ and $\{x_i\colon r\} \triangleright L\colon (\mathtt{List}\ r)$. Let F and C' be arbitrary lambda terms. Then,

$$\mathfrak{Th}, (C\ x\ y) = (C'\ (F\ x)\,(F\ y)) \vdash_\lambda (\mathtt{map}\ F\ (M\ C\ L)) = (M\ C'\ (\mathtt{map}\ F\ L)).$$

Proof: As before, we start with the binary version of the Fundamental Theorem which gives us

$$\mathfrak{Th} \vdash_{TT} R^{\forall t.(t \to t \to \mathtt{Bool}) \to (\mathtt{List}\ t) \to (\mathtt{List}\ t)}(M, M),$$

which can be expanded into

$$\begin{aligned}\mathfrak{Th} \vdash_{TT} \forall T.\forall zz'uu'.\,&(\forall xyx'y'.T(x,x') \supset T(y,y') \supset R^{\mathtt{Bool}}((z\ z\ z),(z'\ x'\ y')))\\ &\supset R^{(\mathtt{List}\ t)}(u,u')\\ &\supset \forall P.P(\mathtt{nil},\mathtt{nil})\\ &\qquad \supset (\forall xyx'y'.T(x,x') \supset P(y,y')\\ &\qquad\qquad\qquad \supset P((\mathtt{cons}\ x\ y),(\mathtt{cons}\ x'\ y')))\\ &\qquad \supset P(M\ z\ u),(M\ z'\ u')).\end{aligned}$$

We instantiate P to $\Lambda xy.((\mathtt{map}\ F\ x) = y)$, z to C, z' to C', u to L, u' to $(\mathtt{map}\ F\ L)$ and T to $\Lambda xy.(Y = (F\ x)) \wedge (R^\sigma\ x\ y)$. Then, the last clause implies our target equation $(\mathtt{map}\ F\ (M\ C\ L)) = (M\ C'\ (\mathtt{map}\ F\ L))$, and it remains to show the first two clauses are derivable with given instantiations. Using the Fundamental Theorem, we can prove

$$(\forall xyx'y'.R^\sigma(x,x') \supset R^\sigma(y,y') \supset R^{\mathtt{Bool}}((C\ x\ y),(C\ x'\ y'))),$$

which implies

$$(\forall xyx'y'.T(x,x') \supset T(y,y') \supset R^{\mathtt{Bool}}((C\ x\ y),(C\ x'\ y'))),$$

where T stands for above instantiation. With the additional assumption that $(C\ x\ y) = (C'\ (F\ x)\,(F\ y))$ we derive

$$(\forall xyx'y'.T(x,x') \supset T(y,y') \supset R^{\mathtt{Bool}}((C\ x\ y),(C'\ x'\ y'))).$$

$R^{(\mathtt{List}\ t)}(L,(\mathtt{map}\,F\,L))$ can also be derived using the Fundamental Theorem and equations for `map`. Finally, we use Theorem 2.2.7 to conclude the target equation holds by convertibility. ■

In our formulation, the conclusions of the free theorems are equations between untyped lambda terms provable by convertibility. In contrast to this, Wadler's free theorems are semantic equations between polymorphic lambda terms. We claim that our formulation is more general. First observe that, by Church-Rosser and subject reduction properties, typed lambda theory and untyped lambda theory prove exactly the same equations between terms typable with the same type.

Secondly, by soundness, every provable equation is valid in all models of polymorphic lambda calculus, so a conclusion of a free theorem in our formulation implies the one in Wadler's formulation.

However, a little care is needed to give a proper interpretation of a "conditional" theorem such as Wadler's result for the sort-like functions. In Wadler's formulation the theorem reads as follows: If M is a closed term of type $\forall t.(t \rightarrow t \rightarrow \texttt{Bool}) \rightarrow \texttt{List}(t) \rightarrow \texttt{List}(t)$, then if the equation

$$(C\ x\ y) = (C'\ (F\ x)\ (F\ y))$$

is valid then the equation

$$(\texttt{map}\ F\ (M\ C\ l)) = (M\ C'\ (\texttt{map}\ F\ l))$$

is valid as well. Note that x, y and l are variables, and C, C', and F are arbitrary terms. We also assume that all terms (and variables) are appropriately typed. Since validity in particular model may not imply provable equality, the conclusion of Proposition 2.4.3 does not (directly) imply the above statement. To remedy the situation, we extend the calculus with constants $\texttt{c}$, $\texttt{c}'$, and $\texttt{f}$, and assume theory $\mathfrak{Th}$ contains equations equations $\texttt{c} = C$, $\texttt{c}' = C'$, and $\texttt{f} = F$. Then we conclude (as in Proposition 2.4.3) that

$$\mathfrak{Th}, (\texttt{c}\ x\ y) = (\texttt{c}'\ (\texttt{f}\ x)\ (\texttt{f}\ y) \vdash_\lambda (\texttt{map}\ \texttt{f}\ (M\ \texttt{c}\ L)) = (M\ c'\ (\texttt{map}\ \texttt{f}\ L)),$$

where L is a term is in the assumption of Proposition 2.4.3. Now, any model of $\mathfrak{Th}$ in which $(C\ x\ y) = (C'\ (F\ x)\ (F\ y))$ is valid is a model of $\mathfrak{Th}, (\texttt{c}\ x\ y) = (\texttt{c}'\ (\texttt{f}\ x)\ (\texttt{f}\ y))$, so $(\texttt{map}\ \texttt{f}\ (M\ \texttt{c}\ L)) = (M\ c'\ (\texttt{map}\ \texttt{f}\ L))$ is true in the model. Moreover, the meanings of C and $\texttt{c}$ coincide in the model (and similarly for C' and F), and any interpretation of l (because of the type of l) must be a list and thus a meaning of some term L satisfying the typing hypothesis. Thus, we conclude that $(\texttt{map}\ F\ (M\ C\ l)) = (M\ C'\ (\texttt{map}\ F\ l))$ is valid in the model.

2.5 Representation of Constants

The examples of free theorems given in the previous section were derived from logical relations defined using a particular way of describing data structures. Naturally, one could ask whether the free theorems hold independently of the way we defined data structures. For example, datatypes such as lists, natural numbers, trees, etc., can be represented in the pure polymorphic lambda calculus using a generalization of Church numerals [BB85]. Do the free theorems hold if we choose to work with these representations? The answer is "yes", but it requires some explaining. Let us illustrate the problem with the example of natural numbers. If we choose to represent natural numbers using the data structure

$(\{\mathtt{Nat}{:}\star\}, \{\mathtt{succ}{:}\,\mathtt{Nat}{\to}\mathtt{Nat}, \mathtt{zero}{:}\,\mathtt{Nat}\})$, as we did in our development so far, the logical relation corresponding to type Nat is the Peano's induction axiom

$$Ind(n) \stackrel{\text{def}}{=} \forall P.(\forall x.P(x) \supset P(\mathtt{succ}\ x)) \supset P(\mathtt{zero}) \supset P(n).$$

On the other hand, if we choose to represent the natural numbers as Church numerals, the type of natural numbers is defined to be

$$\underline{\mathtt{Nat}} \stackrel{\text{def}}{=} \forall t.(t{\to}t){\to}t{\to}t,$$

and the corresponding logical relation is

$$Nat(n) \stackrel{\text{def}}{=} \forall P.\forall sz.(\forall x.P(x) \supset P(s\ z)) \supset P(z) \supset P(n\ s\ z).$$

Since the free theorems are, in fact, instances of logical relations, it is not obvious that one can derive the same theorems, regardless whether *Ind* or *Nat* was used in the definition of a logical relation. It is not hard to see, and we show this in general in Lemma 2.5.2, that if one substitutes Church representations of the successor and zero, $\lambda xsz.(s\ (x\ s\ z))$ and $\lambda sz.z$, for constants succ and zero, that

$$\forall n.Ind(n) \supset Nat(n)$$

is provable in type theory. The other implication seems difficult to prove, but we can show it is valid when interpreted in a term model (of appropriate theory) of lambda calculus. Thus, the formulae $Ind(n)$ and $Nat(n)$ are semantically equivalent in the term model of lambda calculus, and if an equation is derivable from $Ind(n)$, it holds by convertibility whenever $Nat(n)$ is valid, and vice versa.

Rather than restricting ourselves to Böhm-Berarducci representation, we discuss a general notion of a *representation* of data structures. A representation translates terms and types, so it preserves typings and equations, thus allowing us to translate assumptions as well as conclusions of free theorems. We also discuss sufficient conditions that a representation needs to satisfy, so we can translate the proofs of the free theorems as well.

Definition 2.5.1. Let $\mathfrak{C} = (\mathcal{C}_\Lambda, \mathcal{C}_\mathcal{T}, \mathsf{Type}, \mathfrak{Th})$ and $\mathfrak{C}' = (\mathcal{C}_\Lambda{}', \mathcal{C}_\mathcal{T}{}', \mathsf{Type}', \mathfrak{Th}')$ be two calculi. Suppose we want to translate data structures and programs from $\mathfrak{C}$ into $\mathfrak{C}'$. Consider a function $\textsc{Rep}_\Lambda$ mapping constants in $\mathcal{C}_\Lambda$ to $\Lambda^0(\mathcal{C}_\Lambda{}')$, and a function $\textsc{Rep}_\mathcal{T}$ mapping constants in $\mathcal{C}_\mathcal{T}$ into $\mathcal{T}^0(\mathcal{C}_\mathcal{T}{}')$. Extend functions $\textsc{Rep}_\Lambda$ and $\textsc{Rep}_\mathcal{T}$ to be identity on variables and constants in $\mathcal{C}_\Lambda{}'$ and $\mathcal{C}_\mathcal{T}{}'$ respectively, and finally consider the unique homomorphic extensions of $\textsc{Rep}_\Lambda$ and $\textsc{Rep}_\mathcal{T}$ to all of $\Lambda(\mathcal{C}_\Lambda + \mathcal{C}_\Lambda{}')$ and $\mathcal{T}(\mathcal{C}_\mathcal{T} + \mathcal{C}_\mathcal{T}{}')$ respectively. By abuse of notation, we write $\textsc{Rep}_\Lambda$ and $\textsc{Rep}_\mathcal{T}$ to denote these extensions. The pair $(\textsc{Rep}_\Lambda, \textsc{Rep}_\mathcal{T})$ is called a *representation* of $\mathfrak{C}$ in $\mathfrak{C}'$ if it satisfies the following:

- for all $\mathsf{g} \in \mathcal{C}_\Lambda$, if $\sigma_\mathsf{g} = \mathsf{Type}(\mathsf{g})$ then $\emptyset \rhd \textsc{Rep}_\Lambda(\mathsf{g}){:}\ \textsc{Rep}_\mathcal{T}(\sigma_\mathsf{g})$, and
- if $M = N \in \mathfrak{Th}$ then $\textsc{Rep}_\Lambda(M) = \textsc{Rep}_\Lambda(N) \in \mathfrak{Th}'$.

In other words, a representation preserves typings and equations. For simplicity we may write $\textsc{Rep}$ to denote either $\textsc{Rep}_\Lambda$ or $\textsc{Rep}_\mathcal{T}$, or both.

Now let σ be type in $\mathcal{T}(\mathcal{C}_\mathcal{T})$. In general, if σ and $\textsc{Rep}(\sigma)$ are different types, the corresponding logical relations, R^σ and $R^{\textsc{Rep}(\sigma)}$ will be different formulae. Nevertheless if for all types σ, the sequent

$$\mathfrak{Th}' \cup \{\mathsf{g} = \textsc{Rep}(\mathsf{g}) \mid \mathsf{g} \in \mathcal{C}_\Lambda\} \vdash \forall x. R^{\textsc{Rep}(\sigma)}(x) \supset R^\sigma(x) \qquad (\dagger)$$

is valid in the model of type theory defined by the the term model of $\mathfrak{Th} \cup \{\mathsf{g} = \textsc{Rep}(\mathsf{g}) \mid \mathsf{g} \in \mathcal{C}_\Lambda\}$, we will be able able to translate any free theorem using this representation, and the proof will still hold! This is formalized in Theorem 2.5.4 below.

However, checking sequent ($\dagger$) at *every* type may be quite difficult, making it unjustified to call the translated theorems "free". Because of Lemma 2.5.2, it is not necessary to do so, and it suffices to prove sequent ($\dagger$) only when σ is a base type. (Lemma 2.5.2 is a generalization of Theorem 3.3 in [Mai91].)

Lemma 2.5.2. $\mathfrak{Th}', \{\mathsf{g} = \textsc{Rep}(\mathsf{g}) \mid \mathsf{g} \in \mathcal{C}_\Lambda\} \vdash_{TT} \forall x. R^{(\mathsf{c}_j\, t_1 \cdots t_k)}(x) \supset R^{\textsc{Rep}((\mathsf{c}_j\, t_1 \cdots t_k))}(x)$

Proof Sketch: Let c_j's be type constants and g_i's be generators in the signature of the data structure where $\mathsf{g} = \mathsf{g}_i$ for some i. Instantiate each C_j in R^{c_j} with $R^{\textsc{Rep}(\mathsf{c}_j)}$. Then using $\mathsf{g}_i = \textsc{Rep}(\mathsf{g}_i)$ and the Fundamental Theorem we can deduce $R^{\textsc{Rep}(\phi_i)}(\textsc{Rep}(\mathsf{g}_i))$ for each i. ■

Lemma 2.5.3. If sequent ($\dagger$) is valid (derivable) at all base types, then it is valid (derivable) at all types.

Proof Sketch: The proof is by induction on the structure of type σ using the assumption and Lemma 2.5.2 for the base case. ■

Theorem 2.5.4 (Translation). Let $\mathfrak{C}$ and $\mathfrak{C}'$ be two calculi. Let $\textsc{Rep}$ be a representation of $\mathfrak{C}$ in $\mathfrak{C}'$ such that for each base type sequent ($\dagger$) is valid in the model of type theory defined by the term model of $\mathfrak{Th} \cup \{\mathsf{g} = \textsc{Rep}(\mathsf{g}) \mid \mathsf{g} \in \mathcal{C}_\Lambda\}$. Suppose also that a free theorem

> *"Let M and $N_1, \ldots, N_r$ be terms in $\Lambda(\mathcal{C}_\Lambda)$ such that $\emptyset \rhd M{:}\,\sigma$ and $\Gamma_i \rhd N_i{:}\,\tau_i$. Then*
> $$\mathfrak{Th} \vdash_\lambda A[M, N_1, \ldots, N_r] = B[M, N_1, \ldots, N_r]."$$

is provable as described in Section 2.4. Then, its translation,

> *"Let M' and $N_1', \ldots, N_r'$ be terms in $\Lambda(\mathcal{C}_\Lambda{}')$ such that $\emptyset \rhd M'{:}\,\textsc{Rep}(\sigma)$ and $\textsc{Rep}(\Gamma_i) \rhd N_i'{:}\,\textsc{Rep}(\tau_i)$. Furthermore, let $A'[\cdot, \ldots, \cdot] = \textsc{Rep}(A[\cdot, \ldots, \cdot])$ and $B'[\cdot, \ldots, \cdot] = \textsc{Rep}(B[\cdot, \ldots, \cdot])$. Then*
> $$\mathfrak{Th}' \vdash_\lambda A'[M', N_1', \ldots, N_r'] = B'[M', N_1', \ldots, N_r']."$$

is provable as well.

Proof: Let $\mathcal{D}$ be the term model of $\mathfrak{Th} \cup \{\mathsf{g} = \textsc{Rep}(\mathsf{g}) \mid \mathsf{g} \in \mathcal{C}_\Lambda\}$. By the Fundamental Theorem we have $\mathfrak{Th}' \vdash_{TT} R^{\textsc{Rep}(\sigma)}(M')$, and by the assumption and Lemma 2.5.3, it follows that

$$\mathfrak{Th}', \{\mathsf{g} = \textsc{Rep}(\mathsf{g}) \mid \mathsf{g} \in \mathcal{C}_\Lambda\} \models_\mathcal{D} R^\sigma(M').$$

Using the same derivation as in the original theorem, we derive

$$\mathfrak{Th}, \mathfrak{Th}', \{\mathsf{g} = \textsc{Rep}(\mathsf{g}) \mid \mathsf{g} \in \mathcal{C}_\Lambda\} \models_{\mathcal{D}} A[M', N_1', \ldots, N_r'] = B[M', N_1', \ldots, N_r'].$$

Moreover, since representation preserves equations, we conclude that

$$\begin{aligned}\mathfrak{Th}, \mathfrak{Th}', \{\mathsf{g} = \textsc{Rep}(\mathsf{g}) \mid \mathsf{g} \in \mathcal{C}_\Lambda\} \quad \models_{\mathcal{D}} \quad & A'[\textsc{Rep}(M'), \textsc{Rep}(N_1'), \ldots, \textsc{Rep}(N_r')] = \\ & B'[\textsc{Rep}(M'), \textsc{Rep}(N_1'), \ldots, \textsc{Rep}(N_r')].\end{aligned}$$

Notice that, since M' and N_j' are in $\Lambda({\mathcal{C}_\Lambda}')$, $\textsc{Rep}(M') = M'$ and $\textsc{Rep}(N_j') = N_j'$, so we have

$$\mathfrak{Th}, \mathfrak{Th}', \{\mathsf{g} = \textsc{Rep}(\mathsf{g}) \mid \mathsf{g} \in \mathcal{C}_\Lambda\} \models_{\mathcal{D}} A'[M', N_1', \ldots, N_r'] = B'[M', N_1', \ldots, N_r'].$$

Now observe that, since representation preserves equations, every equation provable from $\mathfrak{Th} \cup \{\mathsf{g} = \textsc{Rep}(\mathsf{g}) \mid \mathsf{g} \in \mathcal{C}_\Lambda\}$ is also provable from $\mathfrak{Th}' \cup \{\mathsf{g} = \textsc{Rep}(\mathsf{g}) \mid \mathsf{g} \in \mathcal{C}_\Lambda\}$, and that $\mathfrak{Th}' \cup \{\mathsf{g} = \textsc{Rep}(\mathsf{g}) \mid \mathsf{g} \in \mathcal{C}_\Lambda\}$ is conservative over $\mathfrak{Th}'$. Since $\mathcal{D}$ is the term model of $\mathfrak{Th} \cup \{\mathsf{g} = \textsc{Rep}(\mathsf{g}) \mid \mathsf{g} \in \mathcal{C}_\Lambda\}$, any equation between terms in $\Lambda({\mathcal{C}_\Lambda}')$ that is valid in $\mathcal{D}$ is provable by convertibility from equations in $\mathfrak{Th}'$. Hence, we conclude

$$\mathfrak{Th}' \vdash_\lambda A'[M', N_1', \ldots, N_r'] = B'[M', N_1', \ldots, N_r'].$$

■

Going back to Böhm-Berarducci representation of datatypes in the pure lambda calculus, we now show that sequent (†) is valid at all base types in a term model of lambda calculus that identifies constants with their Böhm-Berarducci representations. Following [BB85] we write $\underline{\sigma}$ for the representation of type σ, and likewise, $\underline{M}$ for the representation of term M.

For example, consider the Church numerals. The type of church numerals is

$$\underline{\mathtt{Nat}} \stackrel{\text{def}}{=} \forall t.(t \to t) \to t \to t.$$

The corresponding logical relation is

$$R^{\underline{\mathtt{Nat}}}(x) \stackrel{\text{def}}{=} \forall P.\forall sz.(\forall y.P(y) \supset P(s\ y)) \supset P(z) \supset P(x\ s\ z).$$

Let $\mathcal{D}$ be a term model of a $\beta\eta$-theory which also identifies $\mathtt{zero} = \lambda sz.z$ and $\mathtt{succ} = \lambda xsz.(s\ (x\ s\ z))$, and let us examine the interpretation of the above formula in this model. Fix variables s and z, and define $\mathcal{N}$ to be the set of elements in the model of the form $s(s(\cdots(s(z))\cdots))$. (Note: when speaking of terms as elements of $\mathcal{D}$, we identify terms with their equivalence classes.) Then if $n \in [\![R^{\underline{\mathtt{Nat}}}]\!]\rho$ it follows, by interpreting P, in $R^{\underline{\mathtt{Nat}}}$, as the set $\mathcal{N}$, that $(n\ z\ s) = s(s(\cdots(s(z))\cdots))$, and therefore n is the meaning, in $\mathcal{D}$, of a Church numeral. It is easy to see that meanings of Church numerals satisfy the induction axiom

$$\forall P.P(\mathtt{zero}) \supset (\forall y.P(y) \supset P(\mathtt{succ}\ y)) \supset P(x),$$

under the assumption that $\mathtt{zero} = \lambda sz.z$ and $\mathtt{succ} = \lambda nzs.(s\ (n\ s\ z))$. Combining this observation with Lemma 2.5.2, it follows that, in $\mathcal{D}$, the induction axiom is semantically equivalent to the logical relation corresponding to the type of Church numerals. This, as we now show, can be generalized to any data structure and its Böhm-Berarducci representation.

Let us first recall the definition of the Böhm-Berarducci representation.

Definition 2.5.5. Let $(\{\mathtt{c}_1, \ldots, \mathtt{c}_n\}, \{\mathtt{g}_1 : \phi_1, \ldots, \mathtt{g}_m : \phi_m\})$ be a signature of a data structure. The Böhm-Berarducci representation of a base type $(\mathtt{c}_j\ t_1 \cdots t_k)$ is defined to be

$$\underline{(\mathtt{c}_j\ t_1 \cdots t_k)} \stackrel{\mathrm{def}}{=} \forall c_1 \cdots c_n . \hat{\phi}_1 \to \cdots \hat{\phi}_m \to (c_j\ t_1 \cdots t_k),$$

where we write $\hat{\phi}$ for the type ϕ with constants $\mathtt{c}_j$ replaced by the fresh variables c_j. The representation of a generator $\mathtt{g}_i$ is defined as follows. Assume

$$\phi_i = \forall \vec{t}. \alpha_1 \to \cdots \to \alpha_k \to \beta_1 \to \cdots \to \beta_l,$$

where α's are type variables, and β's are base types. Then define

$$\underline{\mathtt{g}_i} \stackrel{\mathrm{def}}{=} \lambda x_1 \cdots x_k . \lambda y_1 \cdots y_l . \lambda g_1 \cdots g_m . (g_i\ x_1 \cdots x_k\ (y_1\ g_1 \cdots g_m) \cdots (y_l\ g_1 \cdots g_m)).$$

Assume data structures in the calculus correspond to the class of free algebras for which the Completeness Theorem in [BB85] holds.

Theorem 2.5.6. Let $(\{\mathtt{c}_1, \ldots, \mathtt{c}_n\}, \{\mathtt{g}_1 : \phi_1, \ldots, \mathtt{g}_m : \phi_m\})$ be a signature of a data structure satisfying the above criterion. Let $\mathcal{D}$ be the term model of the $\beta\eta$-theory that also identifies $\mathtt{g}_i$'s with their representation. Then,

$$(\beta), (\eta), \left\{ g_i = \underline{g_i} \mid 1 \le i \le m \right\} \models_{\mathcal{D}} \forall x. R^{\underline{(\mathtt{c}_j\ t_1 \cdots t_k)}}(x) \supset R^{(\mathtt{c}_j\ t_1 \cdots t_k)}(x).$$

Proof Sketch: Rather than giving a proof of the completely general case, we make up a representative example and prove the theorem for this example. Let our data structure be defined by the signature

$\{\mathtt{OList} : \star \Rightarrow \star, \mathtt{EList} : \star \Rightarrow \star\},$
$\{\mathtt{enil} : \forall s. \mathtt{EList}(s),\ \mathtt{onil} : \forall s. \mathtt{OList}(s),$
$\mathtt{econs} : \forall s. s \to \mathtt{OList}(s) \to \mathtt{EList}(s),\ \mathtt{ocons} : \forall s. s \to \mathtt{EList}(s) \to \mathtt{OList}(s)\}$

We call terms generated by above signature "even lists" if they are of type $\mathtt{EList}(s)$, and "odd lists" if they are of type $\mathtt{OList}(s)$. The Böhm-Berarducci representation of the above constants is defined as follows:

$$\begin{array}{rcl}
\underline{\mathtt{EList}(t)} & \stackrel{\mathrm{def}}{=} & \forall oe. o \to e \to (t \to e \to o) \to (t \to o \to e) \to e \\
\underline{\mathtt{OList}(t)} & \stackrel{\mathrm{def}}{=} & \forall oe. o \to e \to (t \to e \to o) \to (t \to o \to e) \to o \\
\underline{\mathtt{enil}} & \stackrel{\mathrm{def}}{=} & \lambda n_e n_o c_e c_o . n_e \\
\underline{\mathtt{onil}} & \stackrel{\mathrm{def}}{=} & \lambda n_e n_o c_e c_o . n_o \\
\underline{\mathtt{econs}} & \stackrel{\mathrm{def}}{=} & \lambda x y n_e n_o c_e c_o . (c_e\ x\ (y\ n_e\ n_o\ c_e\ c_o)) \\
\underline{\mathtt{ocons}} & \stackrel{\mathrm{def}}{=} & \lambda x y n_e n_o c_e c_o . (c_o\ x\ (y\ n_e\ n_o\ c_e\ c_o))
\end{array}$$

Using the definition of logical relations we obtain logical relations, $R^{\mathtt{EList}(s)}(x)$ and

$R^{\underline{\texttt{EList}(s)}}(x)$, corresponding to the constants type constructor, and its representation.

$$\begin{array}{rcl} R^{\texttt{EList}(s)}(x) & \stackrel{\text{def}}{=} & \forall EO.E(\texttt{enil}) \supset O(\texttt{onil}) \supset \\ & & (\forall yz.S(y) \supset O(z) \supset E(\texttt{econs}\ y\ z)) \supset \\ & & (\forall yz.S(z) \supset E(z) \supset O(\texttt{ocons}\ y\ z)) \supset \\ & & E(x) \end{array}$$

$$\begin{array}{rcl} R^{\underline{\texttt{EList}(s)}} & \stackrel{\text{def}}{=} & \forall EO.\forall n_e n_o c_e c_o.E(n_e) \supset O(n_o) \\ & & (\forall yz.S(z) \supset O(z) \supset E(c_e\ y\ z)) \supset \\ & & (\forall yz.S(z) \supset E(z) \supset E(c_o\ y\ z)) \supset \\ & & E(x\ n_e\ n_o\ c_e\ c_o) \end{array}$$

Let $\mathcal{D}$ be the term model of the $\beta\eta$-theory including axioms that identify constants of our data structure with their representations. (Note: when we speak of a λ-term M as an element of $\mathcal{D}$ we are referring to the equivalence class of M.) Fix λ-variables n_e, n_o, c_e and c_o. Let the set $\mathcal{S}$ to be the set of all lambda terms M with variables n_e, n_o, c_e and c_o not free in M. Define sets $\mathcal{E}$ and $\mathcal{O}$ of elements in $\mathcal{D}$ by the following grammar:

$$\begin{array}{rcl} \mathcal{E} & ::= & n_e \mid (c_e\ \mathcal{S}\ \mathcal{O}) \\ \mathcal{O} & ::= & n_o \mid (c_o\ \mathcal{S}\ \mathcal{E}), \end{array}$$

Interpret variable S in $R^{\underline{\texttt{EList}(s)}}$ as the set $\mathcal{S}$, and assume $R^{\underline{\texttt{EList}(s)}}(x)$ is true. Interpret universally quantified variables E and O as the sets $\mathcal{E}$ and $\mathcal{O}$ defined above. Interpret universally quantified variables n_e, n_o, c_e and c_o as (equivalence classes of) variables with the same names. By definition of the sets $\mathcal{E}$ and $\mathcal{O}$, we know that $n_e \in \mathcal{E}$, and $n_o \in \mathcal{O}$. Moreover, if N is a term in $\mathcal{O}$, and M is a term in $\mathcal{S}$, then $(c_e\ M\ N)$ is in $\mathcal{E}$, and likewise, if N is a term in $\mathcal{E}$, the term $(c_o\ M\ N)$ is in $\mathcal{O}$. Therefore we can conclude that (the interpretation of) $(x\ n_e\ n_o\ c_e\ c_o)$ is in $\mathcal{E}$. Since the choice of variables n_e, n_o, c_e and c_o was arbitrary and these variables do not appear anywhere in the assumptions, we can lambda abstract to conclude that (the interpretation of) x must be a representation of an "even-list", and therefore $R^{\texttt{EList}(s)}(x)$ must be true under the assumptions that the constants `enil`, `onil`, `econs` and `ocons` are identified with their representations. ■

Combining this theorem with the Translation Theorem, we conclude that the above representation preserves free theorems. Therefore, for example, Proposition 2.5.7 directly follows from Proposition 2.4.1.

Proposition 2.5.7. Let M be a closed pure λ-term, such that $\emptyset \triangleright M\colon \forall t.(\underline{\texttt{List}}\,t) \rightarrow \underline{\texttt{Nat}}$, and let L be a pure λ-term such that $\{x_1\colon r, \ldots x_k\colon r\} \triangleright L\colon (\underline{\texttt{List}}\ r)$. Then,

$$(M\ L) =_{\beta\eta} (M\ ((\lambda x_1 \cdots x_k.L)\, y_1 \cdots y_k)).$$

Similarly, Proposition 2.1.1 follows from derivation of the corresponding result in [Mai91].

Validity of sequent (†) at base types is a sufficient condition for a representation to preserve the free theorems. As wee have seen in the preceding theorem, the proof of the validity of this sequent requires detailed understanding of the particular representation. We would like to give a more practical and, perhaps more intuitive, condition that will imply the validity of sequent (†), and thus, guarantee the free theorems are preserved.

Consider the set, $D^0_\sigma = \{M \mid \ \triangleright M{:}\,\sigma\}$, of all closed terms of type σ and the restriction $\textsc{Rep}|_{D^0_\sigma}$ as a function from D^0_σ to $D^0_{\textsc{Rep}(\sigma)}$. Then showing that this function is surjective suffices to preserve the free theorems. More formally, we have,

Theorem 2.5.8. For any quantified *ML* type σ, if function $\textsc{Rep}|_{D^0_\sigma}$ is surjective then, at σ, sequent (†) is valid in the term model of $\mathfrak{Th}' \cup \{\mathsf{g} = \textsc{Rep}(g)\}$.

Proof: Let $\mathcal{D}$ be the term model of $\mathfrak{Th}' \cup \{\mathsf{g} = \textsc{Rep}(g)\}$, and assume that $R^{\textsc{Rep}(\sigma)}(M)$ is true, so that

$$\mathfrak{Th}', \{\mathsf{g} = \textsc{Rep}(\mathsf{g})\} \models_{\mathcal{D}} R^{\textsc{Rep}(\sigma)}(M).$$

Then by Lemma 2.3.8, we conclude that

$$\mathfrak{Th}', \{\mathsf{g} = \textsc{Rep}(\mathsf{g})\} \vdash \emptyset \triangleright M{:}\,\textsc{Rep}(\sigma),$$

so that M is in $D^0_{\textsc{Rep}\sigma}$. By assumption, $\textsc{Rep}|_{D^0_\sigma}$ is surjective so there is a lambda term N such that $M = \textsc{Rep}(N)$, and $\mathfrak{Th} \vdash \emptyset \triangleright N{:}\,\sigma$. By the Fundamental Theorem, $\mathfrak{Th} \vdash_{TT} R^\sigma(N)$ and therefore $\mathfrak{Th}', \{\mathsf{g} = \textsc{Rep}(\mathsf{g})\} \vdash_{TT} R^\sigma(M)$, since $\{\mathsf{g} = \textsc{Rep}(\mathsf{g})\} \vdash_{TT} M = N$ and representation preserves equations in $\mathfrak{Th}$. Hence, by soundness, $R^\sigma(M)$ is true in $\mathcal{D}$. ■

By the above theorem, it suffices to check that, for each base type σ, the set $D^0_{\textsc{Rep}(\sigma)}$ contains only terms representing elements in D^0_σ. For example the Completeness Theorem [BB85] for the Böhm-Berarducci representation would suffice to conclude that this representation preserves the free theorems.

While the *surjectivness* condition in Theorem 2.5.8 is (arguably) more intuitive, we must say it is somewhat stronger than the validity of sequent (†). For instance, the surjectivness condition may be true at base types, but fail at higher types, while, as seen in Lemma 2.5.3, if sequent (†) is valid at base types, it is also valid at higher types.

2.6 Conclusions

We have developed a syntactic framework in which parametric behavior of lambda terms is captured by logical relations as formulae of type theory. In this framework we have been able to formulate free theorems — identities about polymorphic programs that follow from the types of the programs — as equations schemas that are derivable by convertibility whenever typing assumptions about the terms involved are met.

Furthermore, we have defined a general notion of a representation of data structures. For a representation of a data structure we have defined, in two ways, a sufficient condition that guarantees that the representation preserves free theorems. In particular, we have shown that the Böhm-Berarducci representation of inductive data structures in pure lambda calculus does preserve the free theorems. This also answers the question raised by Mairson [Mai91] asking whether induction is necessary to derive free theorems.

2.7 Acknowledgments

The credit for the right formulation of the main problem addressed in this chapter should go Albert Meyer. Thanks, also to Harry Mairson for explaining the issues dealt with in this work. Thanks to both for the help in solving the problems.

Bibliography

[App92] Anrew W. Appel. *Compiling with Continuations.* Cambridge Universisty Press, 1992.

[Bar84] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic.* North-Holland Publishing Co., Amsterdam, second edition, 1984.

[BB85] Corrado Böhm and Alessandro Berarducci. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, 39(2,3):135–154, August 1985.

[BFSS90] E. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(1):35–64, January 1990.

[CP92] Roy L. Crole and Andrew M. Pitts. New foundations for fixpoint computations: FIX-hyperdoctrines and the FIX-logic. *Information and Computation*, 98(2):171–210, jun 1992.

[Cro93] Roy L. Crole. *Categories for Types.* Cambridge Mathematical Textbooks. Cambridge University Press, Cambridge, UK, 1993.

[Fil94] Andrzej Filinski. Representing monads. In *Symposium on Principles of Programming Languages*, pages 446–457, Portland, Oregon, January 1994.

[Fis72] Michael J. Fischer. Lambda calculus schemata. In *Proceedings of An ACM Conference on Proving Assertions About Programs*, pages 104–109, Las Cruces, NM, 1972. ACM.

[Fis93] Michael Fischer. Lambda-calculus schemata. *Lisp and Symbolic Computation*, 6(3/4):259–288, November 1993.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. *SIGPLAN Notices*, 28(6):237–247, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation.*

[Gir71] Jean-Yves Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, 2^{nd} *Scandinavian Logic Symp.*, pages 63–92. North-Holland Publishing Co., 1971.

[Gir72] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieure.* PhD thesis, Université Paris VII, 1972.

[HD94] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In *Proceedings of the 21st Symposium on Principles of Programming Languages*, pages 458–471, Portland, Oregon, January 1994. ACM.

[HL92] Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. Technical Report CMU-CS-92-210, Carnegie Mellon University, October 1992. This is an expanded version of a paper presented at the ACM SIGPLAN/SIGACT Symposium on Principles of Programming Languages, Charleston, SC, January, 1993.

[Lei90] Daniel Leivant. Contracting proof to programs. In P. Odifreddi, editor, *Logic and Computer Science*, volume 31 of *APIC Studies in Data Processing*, pages 279–327. Academic Press, 1990.

[LS86] L. Lambek and P. J. Scott. *Introduction to Higher Order Categorical Logic.* Cambridge University Press, 1986.

[Mai91] Harry G. Mairson. Outline of a proof theory of parametricity. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *LNCS*, pages 313–327. Springer-Verlag, 1991.

[Man76] Ernest G. Manes. *Algebraic Theories*, volume 26 of *Graduate Texts in Mathematics.* Springer-Verlag, 1976.

[Mey82] Albert R. Meyer. What is a model of the lambda calculus? *Information and Control*, 52(1):87–122, January 1982.

[Mit88] John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76:211–249, 1988.

[MKO95] D. McAllester, J. Kučan, and D. F. Otth. A proof of strong normalization of F_2, F_ω and beyond. *Information and Computation*, 121(2):193–200, September 1995.

[ML71] Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics.* Springer-Verlag, 1971.

[Mog88] Eugenio Moggi. Computational lambda-caluclus and monads. Technical Report ECS-LFCS-88-86, University of Edinburgh, October 1988.

[Mog89] Eugenio Moggi. Computational lambda-caluclus and monads. In *LICS*, 1989.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.

[MPS86] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1/2):95–130, October/November 1986.

[MR] Albert R. Meyer and Jon G. Riecke. Continuations may be unreasonable. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 63–71.

[MR91] QingMing Ma and John C. Reynolds. Types, abstraction, and parametric polymorphism, part 2. In *Mathematical Foundations of Programming Semantics*, 1991.

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

[Plo75] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.

[Rey72] John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM National Meeting (Boston, 1972)*, pages 717–740, 1972.

[Rey83] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523. IFIP, North-Holland Publishing Co., 1983.

[Rey93] John C. Reynolds. The discoveries of continutions. *Lisp and Symbolic Computation*, 6(3/4), November 1993.

[Rie93] Jon G. Riecke. Delimiting scope of side effects. In *Proceedings of the 1993 Conference on Functional Programming and Computer Architecture*, pages 146–158. ACM, 1993.

[RV95] Jon G. Riecke and Ramesh Viswanathan. Isolating side effects in sequential languages. In *In Proceedings of Symposium on Principles of Programming Languages*, 1995.

[SF92] Amr Sabry and Mathias Felleisen. Reasoning about programs in continuation-passing style. Technical Report COMP TR 92-180, Rice University, 1992.

[SF93] Amr Sabry and Matthias Felleisen. Reasoning with continuations III: A complete calculus of control. Unpublished manuscript, 1993.

[Ste78] Guy L. Steele. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, MIT Artificial Intelligence Laboratory, 1978.

[Sto77] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge, MA, 1977.

[Str67] C. Strachey. Fundamental concepts in programming languages. Lecture Notes, Int'l. Summer School in Computer Programming, Copenhagen, 1967., 1967.

[SW74] C. Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Report PRG–11, Oxford University Computing Laboratory, 1974.

[SW96] Amr Sabry and Philip Wadler. A reflection on call-by-value. In *International Conference on Functional Programing*, Philadelphia, May 1996. ACM Press.

[Wad89] Philip Wadler. Theorems for free! In *4th International Symposium on Functional Programming Languages and Computer Architecture*, London, September 1989.

[Wad90] Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, December 1990.

[Wad95] Philip Wadler. Monads for functional programming. In J. Jeuring and E Meijer, editors, *Advanced Functional Programming.* Proceedings of the Båstard Spring School, May 1995. Springer Verlag Lecture Notes in Computer Science 925.