FIXED POINTS AND EXTENSIONALITY IN TYPED FUNCTIONAL PROGRAMMING LANGUAGES

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Brian T. Howard
August 1992

© Copyright 1992 by Brian T. Howard
All Rights Reserved

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

John C. Mitchell
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Vaughan Pratt

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Luca Cardelli
(DEC-SRC)

Approved for the University Committee on Graduate Studies:

Abstract

We consider the interaction of recursion with extensional data types in several typed functional programming languages based on the simply-typed lambda calculus. Our main results concern the relation between the equational proof systems for reasoning about terms and the operational semantics for evaluating programs. We also present several results about the expressivity of the languages, compared according to both the functions and the data types definable in them. The methods used are those of classical lambda calculus and category theory.

The first language discussed is a variant of Scott and Plotkin's PCF, which adds to the simply-typed lambda calculus products, fixed points of functions, and algebraic data types specified by a signature and a set of equations. PCF is able to express all partial computable functions over the given basic types, but the corresponding reduction system is not confluent if we include the usual surjective pairing rule, which expresses the extensionality of products. Extensionality is necessary in the proof system for establishing many useful isomorphisms between types, but it does not seem to have an intuitive "computational content." We show that a smaller reduction system without extensional rules is sufficient for computing the result of program execution, and that this smaller system is confluent whenever the algebraic rules are confluent and left-linear. If the algebraic rules are also terminating and left-normal then a leftmost reduction strategy is complete for finding normal forms.

We then consider a pair of languages, $\lambda^{\mu\nu}$ and $\lambda^{\perp\rho}$, which support the definition of structured data types through categorical means rather than via multi-sorted algebras. The first language, $\lambda^{\mu\nu}$, extends the types of the simply-typed lambda calculus with extensional products and sums, and least and greatest fixed points of positive

recursive type expressions. By dropping the extensional rules as for PCF, we obtain a confluent and strongly normalizing reduction system, adequate for obtaining results of programs. It is easy to represent many common data types in this language, such as booleans, natural numbers, lists, trees, and (computable) streams, as well as many of the total functions over such structures. Indeed, we may define more functions over the natural numbers than are provably total in Peano arithmetic, hence the language is more expressive than Gödel's system **T**. It is no more expressive than the Girard/Reynolds system **F** in terms of definable functions; however, we are able to define *algorithms* that are not expressible in **F**, such as a constant-time predecessor function on the naturals.

The final language, $\lambda^{\perp\rho}$, extends $\lambda^{\mu\nu}$ by introducing lifted types, which contain an element $\perp$, called "bottom", signifying that evaluation of a term of such a type may not terminate. Lifted types allow us to find fixed points of mixed-variant recursive type expressions; for example, the solution of $X = X \to X_\perp$ gives a type for expressions of an eager untyped lambda calculus, while the solution of $X = (X \to X)_\perp$ is suitable as a type for expressions of a lazy untyped calculus. We again have a confluent operational semantics for the language, although of course it is not strongly normalizing. However, we show that a lazy reduction strategy will find normal forms for terms which have them. We also examine the relations among the three kinds of recursive type in $\lambda^{\perp\rho}$, which we refer to as *inductive* (least), *projective* (greatest), and *retractive* (mixed-variant); in the natural cpo model of the language, we give conditions under which the different constructions will coincide.

Acknowledgements

The material in Chapter 2 is a revised and expanded version of a joint paper with John Mitchell which was presented at the 1990 ACM Conference on Lisp and Functional Programming [26]. Among the many people whom I wish to thank for helpful discussions and insightful suggestions are Michael Barr, Val Breazu-Tannen, Kim Bruce, Luca Cardelli, Pierre-Louis Curien, John Greiner, Bob Harper, Furio Honsell, Pat Lincoln, Eugenio Moggi, Andy Pitts, Vaughan Pratt, Andre Scedrov, Rick Statman, Ramesh Viswanathan, Roel de Vrijer, and Phil Wadler. In particular, this work would not have been possible without the guidance and inspiration of my advisor, John Mitchell, and the love, support, and wisdom of my wife, Eleanor.

This material is based upon work supported under a National Science Foundation Graduate Fellowship. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author and do not necessarily reflect the views of the National Science Foundation.

Contents

Chapter 1

Introduction

The subject of this thesis is the interaction of recursion with extensionality in typed functional programming languages. The particular languages we will be considering are all extensions of the simply-typed lambda calculus, λ^τ, as described for example in Appendix A of Barendregt [1]. The system λ^τ itself is not particularly expressive; since every term has a normal form, only terminating functions are definable.[1] In Chapter 2 we add an explicit fixed point operator on values, while in Chapters 3 and 4 we extend the type system with solutions to recursive type equations; the net effect in either case is that all partial recursive functions over the natural numbers become expressible. By Church's Thesis this means we can represent any computable function, using an appropriate coding of the function's domain into the natural numbers.

For real programming applications, of course, we would not be satisfied with a language that required coding everything into natural numbers before execution. Instead we want types whose values have more structure. In Chapter 2 the types which are added are products, whose values are pairs, and algebraic data types, whose values are built up from a user-specified set of constructors. In Chapters 3 and 4 a more unified approach is taken, where the user may define types from products and sums by taking fixed points of type expressions. In both cases we are interested in proving that various types are in some sense equivalent; for example, we would like

[1] In fact, the only functions definable over the Church numerals are the *extended polynomials*, which include addition, multiplication, and zero-test, but not exponentiation [49, 52].

to be able to show that $(A \times B) \to C$ and $A \to (B \to C)$ are isomorphic, since the latter is the type of "curried" versions of the two-argument functions of the former type. To prove such equivalences we need to know that the basic type constructors such as product and function space are *extensional*, in the sense that the whole of any term of such a type is no greater than the sum of its parts. In the case of product, for example, this is what is usually referred to as a *surjective* pairing — every element of a product type is equal to the pair composed of the first and second projections of the element. We draw considerable inspiration from category theory in developing the systems in this thesis; in categorical terms, a type constructor is extensional when it is *universal*, which means that it is unique in a particular way among some class of constructors.

Extensionality causes several problems when naively combined with recursion. In Chapter 2, the problem arises in trying to associate an operational semantics with a given equational description of the language PCF. In the presence of recursively formed terms, a reduction rule based on surjective pairing will cause confluence of the system to fail; that is, there may be two distinct orders in which to perform reductions on a single term which will never be able to produce a common descendent term. Even without recursion, the standard η-reduction rule, which provides extensionality for functions, may cause a failure of confluence through interaction with algebraic rules. We show that this dependence on reduction order is not an essential difficulty, since there is a confluent reduction system without the extensional rules which is able to compute the normal forms of all programs (technically, a *program* is a term of observable type with no free variables) that have normal forms in the full, non-confluent system. This supplies a theoretical basis for the folkloric belief that the extensional rules do not have any "computational content." In Chapter 2 we also show that this smaller reduction system remains confluent when combined with any confluent set of linear algebraic rewrite rules, thus the resulting reduction system serves as a computationally adequate operational semantics corresponding to the axiomatic semantics given by the equations (including extensionality) for the language.

The problem considered in Chapters 3 and 4 is that a type system with extensional sum and product types and solutions to all recursive type equations must be

inconsistent — all the types will be isomorphic, and all the terms will be equal. The standard solution is to drop the extensionality condition from either sums or products. We prefer to keep extensionality, for the sake of the equational proof system, and instead restrict the type equations which may be solved. A benefit of this approach is that the resulting system, $\lambda^{\perp\rho}$, contains a very expressive terminating subsystem (this is the subject of Chapter 3), which permits us to analyze quite finely the possible sources of non-termination in programs; in theory, an optimizing compiler could use this information to perform extensive partial evaluation of a program, with no worry of going into an infinite loop (although this would be unacceptably inefficient if applied naively). Another possible application of this terminating sublanguage is in proving properties of programs, where knowing that a function is total may permit stronger proof techniques. We show that the smaller, strongly normalizing calculus, $\lambda^{\mu\nu}$, is more expressive than Gödel's system **T**, because it is possible to define all the functions that are provably total in Peano arithmetic as well as some that are not.

All the reduction systems we have mentioned so far are non-deterministic. By showing that they are confluent we are able to abstract away considerations of reduction order and examine the properties of an "ideal" implementation. Since the reduction system for $\lambda^{\mu\nu}$ is strongly normalizing, we may choose any deterministic strategy for evaluating programs and be guaranteed of reaching the same results. For PCF and $\lambda^{\perp\rho}$, however, the choice of strategy can affect whether reduction will reach a normal form or diverge. We show for both of these languages that a leftmost reduction strategy is normalizing. The result for PCF depends on the form of the algebraic rules; they must be terminating and *left-normal*, a syntactic condition which intuitively says that a leftmost strategy is normalizing for the algebraic rewrite system alone.

Chapter 2

Extensionality in PCF with Algebraic Types

Most systems of lambda calculus have three parts: an equational proof system, a set of reduction rules, and a model theory. These correspond to the standard programming language notions of axiomatic, operational, and denotational semantics. To a first approximation, the connections between axiomatic and operational semantics are straightforward in basic systems such as the simply-typed lambda calculus. The reduction rules may be derived by orienting each equational axiom in a computationally reasonable way and the resulting system is confluent. As a result, the reduction rules serve simultaneously as a useful characterization of equational provability and as a natural model of execution. When we add recursion to simply-typed lambda calculus with cartesian products, this straightforward correspondence breaks down. Since confluence fails [38], the reduction rules do not give a good picture of equational provability. Moreover, upon examining the reduction rules more carefully, many investigators have come to the conclusion that neither η-reduction nor surjective pairing is computationally compelling. In fact, it seems to be a common view that η-reduction and surjective pairing do not have any "computational content." Therefore, for both technical and intuitive reasons, we are led to define evaluation without these reduction rules.

In this chapter, we study a simply-typed lambda calculus with functions, pairing,

fixed-point operators and arbitrary algebraic data types. We will refer to this language as PCF, since it is based on the calculus considered in Plotkin's seminal paper [43], with pairing and algebraic data types added.[1] If we include algebraic data types of natural numbers and booleans, as in [43, 50], then it is easy to program any partial recursive function on the natural numbers. With algebraic data types of trees, lists, stacks, and so on, we may write common functional programs in the style of Miranda or Lazy ML, for example [53].

While most sequential implementations of lazy languages are based on a deterministic (typically "leftmost") evaluation order, there are several reasons to study arbitrary order. One motivation is parallel execution. If the result of evaluation does not rely on evaluation order, then many subexpressions may safely be evaluated in parallel (see [25, 40], for example, for related discussion). Another reason to consider arbitrary evaluation order is to identify desirable properties of a particular implementation. For example, if a set of reduction (or evaluation) rules is confluent, then the result of nondeterministic evaluation is well-defined and we may regard this as the "ideal" implementation. We may then show that a particular evaluation order is satisfactory by comparison with nondeterministic evaluation. We will do this in Section 2.8, where we show that a leftmost reduction strategy is complete for finding normal forms, provided the algebraic rules satisfy certain conditions.

For any variant of PCF with equationally-axiomatized algebraic data types, there is a traditional and accepted equational proof system. The axioms of this proof system include η-equivalence (extensionality) for functions

$$(\eta)_{eq} \qquad \lambda x{:}\sigma.\, Mx = M, \quad x \text{ not free in } M,$$

and the *surjective pairing* axiom

$$(sp)_{eq} \qquad \langle \pi_1 P, \pi_2 P \rangle = P,$$

[1] The language itself seems attributable to Scott, since the basic ideas are presented in the manuscript [50] and Plotkin's name for the calculus is clearly derived from Scott's LCF (*Logic for Computable Functions).*

where π_1 and π_2 are the first and second projection functions. These seem essential for proving common facts about functions and pairs. For example, both are needed to establish that *currying* and *uncurrying* are inverses (or, equivalently, that types $\sigma\to(\tau\to\rho)$ and $(\sigma \times \tau)\to\rho$ are isomorphic). However, neither seems to be used in standard implementations. If we direct these equational axioms from left to right, we obtain evaluation rules, (η) and (sp), that could be used in program execution but which typically do not have any counterpart in practical implementation. One reason surjective pairing is difficult to implement is that it is *non-linear*: the meta-variable P occurs twice on the left-hand side. In order to apply the reduction, we must therefore test two potentially large subexpressions for syntactic equality. This seems inefficient, in general, and it follows from our results that such a test is unnecessary for evaluation of complete programs.

In addition to the folkloric view that (η) and (sp) are not needed in computation, which we justify in Corollary 2.5.6, there are technical problems with these rules. While pure typed lambda calculus with these rules but *without* fixed-point operators is confluent [45], the situation is substantially different in the presence of recursion. Even without any algebraic data types, (sp) and recursion cause confluence to fail. This may be demonstrated by adapting Klop's well-known counterexample[2] to confluence in the untyped lambda calculus with surjective pairing [38]. Similar reasoning also shows that in a language with recursion, confluence may fail when confluent but non-linear algebraic rules are added. Although not as immediately problematic, (η) also interferes with confluence of algebraic rules. For example, the simple rewrite rule $\mathit{zero}\, x \longrightarrow 0$, combined with (η), is not confluent; the term $\lambda x\colon \mathit{nat}.\, \mathit{zero}\, x$ reduces to both $\lambda x\colon \mathit{nat}.\, 0$ and zero, which are distinct normal forms. Therefore, considering only the technical property of confluence, both (η) and (sp) seem problematic.

Our first major theorem is that, without (η) and (sp), PCF reduction is confluent over any algebraic data types, provided that the algebraic rewrite rules are all linear and confluent when considered apart from PCF. This result, described in Section 2.4, extends a similar theorem of [6] to pairing and fixed-point operators. Our proof

[2] A slick presentation is given in [1]. However, the exact counterexample given there cannot be typed. Instead, one must consider a version of the counterexample given in Klop's thesis [28].

technique combines labelled reduction [1] with the method of [6], which relies on strong normalization. In combining lambda calculus with additional reduction rules, our theorem is also similar in spirit to the delta reduction theorem of Mitschke (see [1]), although our result neither subsumes nor is subsumed by his — Mitschke deals with the untyped calculus, so the delta rules may contain arbitrary lambda terms instead of just algebraic terms, but the set of delta rules has to be disjoint, while we only require that the algebraic rules be confluent. Given the apparent suitability of PCF reduction without (η) and (sp), we proceed to study connections between this limited reduction system and the natural equational axioms.

Since we have dropped two equational rules, it is not immediately clear whether the reduction system is computationally adequate. In other words, do we still have "enough" evaluation rules? While there are provably equal expressions, such as $\lambda x\colon nat.zero\, x$ and $zero$, which do not reduce to a common form, we show that this is not the case for "full programs." In more detail, the programs we execute in practice are closed expressions of basic types such as nat, $bool$ and $list$, or perhaps products of basic types. We show that for any closed expression M of "observable" type, and possible result N of complete execution, M reduces to N using all reduction rules iff M reduces to N without (η) and (sp). Thus we have not lost anything by dropping these rules. Moreover, and perhaps more importantly, such expressions M and N are provably equal, possibly using $(\eta)_{eq}$ and $(sp)_{eq}$, iff M reduces to N by reduction without (η) and (sp). In standard computational terms, this demonstrates the computational adequacy of our operational semantics with respect to the axiomatic semantics (*cf.* [23]). The first of these theorems follows from a postponement property of (η) and (sp), while the second uses a refinement of the postponement proof. Typing is essential here, since (sp) postponement fails for untyped terms.[3]

Our last connection between axiomatic and operational semantics is soundness of the axioms with respect to computation. In general terms, if we begin with an intuitively appealing axiomatic semantics, then it seems fair to base evaluation on any equational principles which follow from the axioms. However, once we have selected an

[3]The untyped term $\langle \pi_1(\lambda x.\, x), \pi_2(\lambda x.\, x)\rangle(\lambda x.\, x)$ provides a simple counterexample.

operational semantics, we must also ask whether this semantics justifies the equational principles we began with. Put simply, are the equational axioms sound statements about the result of program execution? We answer this question (affirmatively) using the standard notion of *observational congruence.* This is the natural equivalence relation generated by execution of full programs, and as the name implies it is a congruence relation. Briefly, two expressions are observationally congruent iff they are interchangeable in all programs. Put another way, expressions M and N, which may be higher-order functions or other "non-programs," are observationally congruent iff we may replace any occurrence of M in a full program by N without affecting the result of program execution. In Section 2.6, we show that the axioms $(\eta)_{eq}$ and $(sp)_{eq}$, and hence all the equational rules, are sound for observational congruence. This shows that although we have eliminated some equational principles from execution, we have not invalidated our axiomatic semantics. The proof uses postponement of (η) and (sp).

In Section 2.7 we consider an alternate method of avoiding the confluence problems of the extensional rules. By using (η) and (sp) as *expansion* instead of contraction rules, we obtain a confluent reduction system for PCF. It is an immediate corollary of the confluence theorem for this system that every term which is provably equal to a normal form will reduce to a unique syntactic variant of that normal form known as a *long* normal form; thus this system is useful for reasoning about equality of arbitrary terms with normal forms. For reasoning about equality of programs, of course, the computational adequacy theorem shows that the expansion rules add nothing.

Finally, we show that, under certain conditions on the algebraic rules, a leftmost reduction strategy is normalizing. That is, if there is any reduction from a term M to a normal form N, then there is a reduction from M to N in which the leftmost redex is contracted at each step. This is similar to a proof by Klop [28] of a standardization theorem for the untyped lambda calculus plus a left-normal, regular term rewriting system, which generalizes our algebraic rules at the expense of requiring that the rules be non-overlapping, a stronger condition than that they merely be confluent.

2.1 Signatures and terms

The language PCF may be defined over any collection of base types (sorts) and constant symbols. The base types might include natural numbers and booleans or atoms, lists, trees and so on. Since PCF is a typed language, each constant symbol must have a type. Typical constants include the number 3, + for natural number addition, and the list operation *cons*. A difference between PCF and the pure typed lambda calculus is that we assume a *fixed-point constant* $fix_\sigma : (\sigma \to \sigma) \to \sigma$ for each type σ.

Using b to stand for any base type, the type expressions of PCF are defined by the grammar

$$\sigma ::= b \mid \sigma_1 \to \sigma_2 \mid \sigma_1 \times \sigma_2.$$

For example, if *nat* is a base type, then $nat \to (nat \times nat)$ is a type. We may avoid writing too many parentheses by adopting the convention that $\to$ associates to the right and $\times$ has a higher precedence than $\to$. Thus $nat \times nat \to nat \to nat$ is the type of functions which, given a pair of natural numbers, return a numeric function.

A PCF *signature* $\Sigma = \langle B, C \rangle$ consists of

- a set B whose elements are called *base types* or *type constants*, and
- a collection C of pairs $\langle c, \sigma \rangle$, where c is called a *term constant* and σ is an *algebraic* type expression over B, of the form $b_1 \to \ldots \to b_n \to b$ for base types $b_1, \ldots, b_n, b$, with $n \geq 0$.

We require that no term constant appear with more than one type, and that the term constants be disjoint from type constants and other syntactic classes of the language. If $\langle c, \sigma \rangle \in C$, then c is said to be a *constant symbol of type* σ, and we sometimes write c^σ when convenient. Note that the base types and term constants must be consistent, in that the type of each constant may only contain the given base types. For example, it only makes sense to have a natural number constant 3^{nat} when we have *nat* as a base type. Note that constants have the curried type $b_1 \to \ldots \to b_n \to b$ instead of the more usual type $b_1 \times \ldots \times b_n \to b$; this is chiefly a matter of preference, since all of the results in this paper hold in either case. The

choice of curried functions simplifies some proofs since we do not have to worry about *sp*-redexes in the arguments of functions.

Before defining the syntax of terms over a given signature, we choose some infinite set $\mathcal{V}$ of variables. We will give the well-formed terms and their types using an inference system for typing assertions

$$\Gamma \triangleright M\colon \tau,$$

where Γ is a *type assignment* of the form

$$\Gamma = \{x_1\colon \sigma_1, \ldots, x_k\colon \sigma_k\},$$

with no x_i occurring twice. Intuitively, the assertion $\Gamma \triangleright M\colon \tau$ means that if variables $x_1, \ldots, x_k$ have types $\sigma_1, \ldots, \sigma_k$ (respectively), then M is a well-formed term of type τ. If Γ is any type assignment, we will write $\Gamma, x\colon \sigma$ for the type assignment

$$\Gamma, x\colon \sigma \;=\; \Gamma \cup \{x\colon \sigma\}.$$

In doing so, we always assume that x does not appear in Γ.

The atomic expressions of PCF over the signature $\Sigma = \langle B, C\rangle$ are given by typing axioms. The typing axiom

$$(cst) \qquad \emptyset \triangleright c\colon \sigma, \quad \text{provided } \langle c, \sigma\rangle \in C, \text{ or } c \text{ is } \mathit{fix}_\tau \text{ and } \sigma = (\tau\to\tau)\to\tau$$

says that each constant symbol c^σ is a term of type σ. Variables are given by the axiom

$$(var) \qquad x\colon \sigma \triangleright x\colon \sigma,$$

which says that a variable x has whatever type it is declared to have. The compound expressions and their types are defined by the following inference rules.

$$(\times \textit{ Intro}) \qquad \frac{\Gamma \rhd M\colon \sigma, \quad \Gamma \rhd N\colon \tau}{\Gamma \rhd \langle M, N\rangle\colon \sigma \times \tau}$$

$$(\times \textit{ Elim}) \qquad \frac{\Gamma \rhd M\colon \sigma \times \tau}{\Gamma \rhd \pi_1 M\colon \sigma, \quad \Gamma \rhd \pi_2 M\colon \tau}$$

$$(\to \textit{ Intro}) \qquad \frac{\Gamma, x\colon \sigma \rhd M\colon \tau}{\Gamma \rhd (\lambda x\colon \sigma.\, M)\colon \sigma \to \tau}$$

$$(\to \textit{ Elim}) \qquad \frac{\Gamma \rhd M\colon \sigma \to \tau, \quad \Gamma \rhd N\colon \sigma}{\Gamma \rhd MN\colon \tau}$$

$$(\textit{add var}) \qquad \frac{\Gamma \rhd M\colon \sigma}{\Gamma, x\colon \tau \rhd M\colon \sigma}$$

The final rule allows us to add variables to the type assignment.

We say *M is a PCF term over signature Σ with type τ in context Γ* if $\Gamma \rhd M\colon \tau$ is either a typing axiom for Σ, or follows from axioms by the typing rules. We often write $\Gamma \rhd M\colon \tau$ to mean that "$\Gamma \rhd M\colon \tau$ is derivable," in much the same way as one often writes a formula $\forall x.P(x)$ in logic, as a way of saying "$\forall x.P(x)$ is true." A term M is *algebraic* if it is of base type and is formed from only algebraic constants and variables of base type, using only ($\to$ Elim), *i.e.*, application.

The free and bound occurrences of a variable x in term M have the usual inductive definition. In particular, a variable x occurs free unless it is within the scope of λx, in which case it becomes bound. Since the name of a bound variable is not important, we will generally identify terms that differ only in the names of bound variables. We will write $\{N/x\}M$ for the result of substituting N for free occurrences of x in M,

with renaming of bound variables as usual to avoid capture. We use the notion of a *context* for substitution without renaming: a context $\mathcal{C}[\ ]$ for a type σ and variable assignment Γ is a term with a "hole", such that if $\Gamma \triangleright M{:}\,\sigma$, then $\mathcal{C}[M]$ will be a well-typed term, formed by "plugging" M into the hole.

The following basic lemmas about terms are easily proved; see [35], *e.g.*, for more details.

Lemma 2.1.1 *If $\Gamma \triangleright M{:}\,\sigma$, then every free variable of M appears in Γ.*

Lemma 2.1.2 *If $\Gamma \triangleright M{:}\,\sigma$ and $\Gamma' \subseteq \Gamma$ contains all the free variables of M, then $\Gamma' \triangleright M{:}\,\sigma$.*

Lemma 2.1.3 *If $\Gamma, x{:}\,\sigma \triangleright M{:}\,\tau$ and $\Gamma \triangleright N{:}\,\sigma$ are well-typed terms, then so is the substitution instance $\Gamma \triangleright \{N/x\}M{:}\,\tau$.*

2.2 Equations and reduction rules

Typed equations have the form

$$\Gamma \triangleright M = N : \tau,$$

where we assume that M and N have type τ in context Γ. Intuitively, the equation

$$\{x_1{:}\,\sigma_1, \ldots, x_k{:}\,\sigma_k\} \triangleright M = N : \tau$$

means that for all type-correct values of the variables $x_1{:}\,\sigma_1$ through $x_k{:}\,\sigma_k$, expressions M and N denote the same element of type τ.

The equational proof system for PCF is standard, with axiom

$$(\mathit{fix})_{eq} \qquad \Gamma \triangleright \mathit{fix}_\sigma = \lambda f{:}\,\sigma{\to}\sigma.\, f(\mathit{fix}_\sigma\, f) : (\sigma{\to}\sigma){\to}\sigma$$

for each fixed-point operator. We choose this form instead of the more common $\mathit{fix}_\sigma M = M(\mathit{fix}_\sigma M)$ simply to avoid an awkward clash with the (η) reduction rule,

since $\lambda x{:}\,\sigma{\to}\sigma.\, \mathit{fix}_\sigma x$ could reduce to either $\lambda x{:}\,\sigma{\to}\sigma.\, x(\mathit{fix}_\sigma x)$ or fix_σ, which would have no common reduct. This is not a serious problem, since the (η) rule will be dropped, but using the (fix) axiom above makes things easier. The remaining axioms, $(\beta)_{eq}$, $(\eta)_{eq}$, $(\pi)_{eq}$, and $(sp)_{eq}$, resemble the corresponding reduction rules given below. Since we include type assignments in equations, we have an equational version

$$(\mathit{add\ var}) \qquad \frac{\Gamma \triangleright M = N : \sigma}{\Gamma, x{:}\,\tau \triangleright M = N : \sigma}$$

of the structural rule which lets us add an additional typing hypothesis.

Reduction is a "directed" form of equational reasoning that we will adopt as a form of symbolic evaluation. Technically, reduction is a relation on α-equivalence classes of terms. While we are only interested in reducing typed terms, we will define reduction without mentioning types. Since reduction models program execution, this is a way of emphasizing that execution does not depend on the types of terms. We will formulate reduction so that the type of a term does not change as it is reduced.

The "logical" axioms of reduction are

$$(\mathit{fix}) \qquad \mathit{fix}_\sigma \longrightarrow \lambda f{:}\,\sigma{\to}\sigma.\, f(\mathit{fix}_\sigma\, f)$$

for fixed-point operators, and the following standard reduction rules for functions and pairs:

$$(\beta) \qquad (\lambda x{:}\,\sigma.M)N \longrightarrow \{N/x\}M$$

$$(\eta) \qquad \lambda x{:}\,\sigma.Mx \longrightarrow M, \text{ provided } x \text{ not free in } M$$

$$(\pi) \qquad \pi_i\langle M_1,\, M_2\rangle \longrightarrow M_i, \text{ for } i = 1, 2$$

$$(sp) \qquad \langle \pi_1 P,\, \pi_2 P\rangle \longrightarrow P.$$

We also allow any set $\mathcal{R}$ of algebraic rewrite rules of the form $M \to N$, where M and N are algebraic terms over Σ with $\Gamma \triangleright M{:}\,b$ and $\Gamma \triangleright N{:}\,b$ for some typing context Γ and basic type b. Additional restrictions on M and N are that M may not be a

variable and all the variables in N must occur in M. If no variable occurs twice in M, the rule $M \to N$ is said to be *left-linear*, or simply *linear.* We write $\mathcal{E}_{\mathcal{R}}$ for the set of all well-typed equations $\Gamma \triangleright M = N : b$ with $(M \to N) \in \mathcal{R}$.

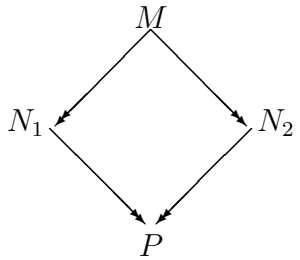
A term of the form $(\lambda x{:}\sigma.M)N$ is a *β-redex*, $\lambda x{:}\sigma.Mx$ is an *η-redex*, and similarly for (π) and (sp). We say M *reduces to N in one step,* written $M \to N$, if N can be obtained by applying a single reduction rule to some subterm of M. To emphasize that a rule r is used, we write $M \xrightarrow{r} N$. As usual, $\twoheadrightarrow$ is the reflexive and transitive closure of one-step reduction. A term M is in *normal form* if there is no N with $M \to N$.

Using Lemma 2.1.3, inspection of the logical rules, and the constraints on variables in algebraic rules, it is easy to show that one-step reduction preserves type.

Lemma 2.2.1 *If $\Gamma \triangleright M{:}\sigma$, and $M \to N$, then $\Gamma \triangleright N{:}\sigma$.*

It follows by an easy induction that $\twoheadrightarrow$ also preserves type.

A critical property of reduction systems is *confluence,* which may be drawn graphically as follows.



In this picture, the top two arrows are universally quantified, and the bottom two existentially, so the picture "says" that whenever $M \twoheadrightarrow N_1$ and $M \twoheadrightarrow N_2$, there exists a term P such that $N_1 \twoheadrightarrow P$ and $N_2 \twoheadrightarrow P$. In lambda calculus, it is traditional to say that a confluent notion of reduction is *Church-Rosser,* since confluence for untyped lambda calculus was first proved by Church and Rosser [9].

The convertibility relation $\leftrightarrow$ on typed terms is the least type-respecting equivalence relation containing reduction. This can be visualized by saying that $\Gamma \triangleright M \leftrightarrow N : \sigma$ iff there is a sequence of terms $M_0, \ldots, M_k$ with $\Gamma \triangleright M_i{:}\sigma$ such that

$$M \equiv M_0 \twoheadrightarrow M_1 \twoheadleftarrow \ldots \twoheadrightarrow M_k \equiv N.$$

In this picture, the directions of $\twoheadrightarrow$ and $\twoheadleftarrow$ should not be regarded as significant. (However, by reflexivity and transitivity of $\twoheadrightarrow$, the order of reduction and "backward reduction" is completely general.) A few words are in order regarding the assumption that $\Gamma \triangleright M_i{:}\sigma$ for each i. For pure typed lambda calculus, this assumption is not necessary; if $\Gamma \triangleright M \leftrightarrow N : \sigma$ and $\Gamma \cap \Gamma'$ mentions all free variables of M and N, then $\Gamma' \triangleright M \leftrightarrow N : \sigma$. However, with algebraic rewrite rules this fails.[4] For conversion as defined here, we have $\Gamma \triangleright M \leftrightarrow N : \sigma$ using rules from $\mathcal{R}$ iff $\mathcal{E}_{\mathcal{R}} \vdash \Gamma \triangleright M = N : \sigma$, regardless of confluence. Therefore, we only mention convertibility in the sequel.

We will write pcf_0 for the fixed-point and lambda calculus reduction rules (fix), (β) and (π), and write pcf for $pcf_0 + \mathcal{R}$, where $\mathcal{R}$ is our chosen set of algebraic rules. We will write $pcf_{\eta,sp}$ for $pcf + (\eta) + (sp)$. Various labelled versions of these reductions will be introduced for technical purposes in the following sections.

2.3 Example: natural numbers and booleans

An example signature for PCF provides booleans and natural numbers. The basic boolean expressions are the constants *true* and *false*, and boolean-valued conditional

$$\texttt{if } \langle bool \rangle \texttt{ then } \langle bool \rangle \texttt{ else } \langle bool \rangle.$$

The basic natural number expressions include *numerals*

$$0, 1, 2, 3, \ldots,$$

the usual symbols for natural numbers, and addition, written $+$. Thus if M and N are natural number expressions, so is $M + N$.

We can also compute natural numbers using conditional tests,

$$\texttt{if } \langle bool \rangle \texttt{ then } \langle nat \rangle \texttt{ else } \langle nat \rangle,$$

[4] Consider the algebraic rules $fx \longrightarrow c$ and $fx \longrightarrow d$, for $f{:}\,a{\to}b$. In this case, we do not want $\emptyset \triangleright c \leftrightarrow d : b$, since the equation $c = d$ is not provable without a variable x of type a.

and compare natural numbers for equality. For example, $Eq?\,3\,0$ has the boolean value *false*, since 3 is different from 0, but $Eq?\,5\,5 = \mathit{true}$. To summarize, the basic natural number and boolean expressions may be characterized by the following productions.

$$\begin{array}{rcl} \langle \mathit{bool} \rangle & ::= & \mathit{true} \mid \mathit{false} \mid \mathit{Eq?}\,\langle \mathit{nat} \rangle\,\langle \mathit{nat} \rangle \mid \\ & & \texttt{if}\ \langle \mathit{bool} \rangle\ \texttt{then}\ \langle \mathit{bool} \rangle\ \texttt{else}\ \langle \mathit{bool} \rangle \end{array}$$

$$\begin{array}{rcl} \langle \mathit{nat} \rangle & ::= & 0 \mid 1 \mid 2 \mid \ldots \mid \langle \mathit{nat} \rangle + \langle \mathit{nat} \rangle \mid \\ & & \texttt{if}\ \langle \mathit{bool} \rangle\ \texttt{then}\ \langle \mathit{nat} \rangle\ \texttt{else}\ \langle \mathit{nat} \rangle \end{array}$$

The equational axioms for natural number and boolean expressions are straightforward. We have an infinite collection of basic axioms

$$0+0=0,\ 0+1=1,\ \ldots,\ 1+0=1,\ 1+1=2,\ \ldots$$

for addition, and two axiom schemes for each type of conditional:

$$\begin{array}{rcl} \texttt{if}\ \mathit{true}\ \texttt{then}\ M\ \texttt{else}\ N & = & M, \\ \texttt{if}\ \mathit{false}\ \texttt{then}\ M\ \texttt{else}\ N & = & N. \end{array}$$

There are infinitely many axioms for equality test, determined according to the scheme

$$\begin{array}{rcll} \mathit{Eq?}\,n\,n & = & \mathit{true}, & \text{each numeral } n, \\ \mathit{Eq?}\,m\,n & = & \mathit{false}, & m, n \text{ distinct numerals.} \end{array}$$

Each of these axioms determines a reduction rule, read from left to right. Note that we do *not* have the equational axiom $\mathit{Eq?}\,M\,M = \mathit{true}$, for arbitrary natural number expression M. The reason is that the more general reduction rule $\mathit{Eq?}\,M\,M \to \mathit{true}$ is non-linear, and confluence fails.

To see how the reduction rules allow us to evaluate basic natural number and boolean expressions, consider the expression

$$\texttt{if}\ \mathit{Eq?}\,(6+5)\,17\ \texttt{then}\ (1+1)\ \texttt{else}\ 27.$$

We cannot simplify the conditional without first producing a boolean constant *true* or *false*. This in turn requires numerals for both arguments to $Eq?$, so we begin by applying the reduction rule $6 + 5 \to 11$. This gives us the expression

$$\texttt{if } Eq?\, 11\ 17 \texttt{ then } (1+1) \texttt{ else } 27,$$

which is simplified using a reduction rule for $Eq?$ to

$$\texttt{if } \mathit{false} \texttt{ then } (1+1) \texttt{ else } 27.$$

Finally, one of the rules for conditional applies, and we produce the numeral 27. In order to simplify this expression, we needed to evaluate the test before simplifying the conditional. However, it was not necessary to simplify the number expression $1 + 1$ since this is discarded by the conditional. Since we may choose to reduce any subterm at any point, we could have simplified $1 + 1 \to 2$ between any two of the reduction steps given. With the added flexibility of "nondeterministic choice," the steps involved mimic the action of any ordinary interpreter fairly closely.

2.4 Confluence of the reduction system *pcf*

We will now show that *pcf*-reduction is confluent. Standard techniques for showing confluence in the typed lambda calculus will not work directly, because the fixed-point operator allows terms that have no normal form. Conversely, we can not use results in the untyped lambda calculus such as Mitschke's delta reduction theorem because we are only assuming that the algebraic rules are confluent and linear. We will proceed by first considering a related system, $pcf^{\mathcal{N}}$, in which recursion is bounded, restoring strong normalization at the cost of diminished computational power.

The system $pcf^{\mathcal{N}}$ is formed by replacing the (*fix*) rule with a family of labelled rules, one for each type σ and natural number $n > 0$:

$$(\mathit{fix}^n) \qquad \mathit{fix}^n_\sigma \longrightarrow (\lambda f{:}\,\sigma{\to}\sigma.\, f(\mathit{fix}^{n-1}_\sigma\ f)).$$

In the absence of algebraic rules, the effect of this is to limit the number of times each fixed-point operator may reduce. The reduction system with no algebraic rules will be denoted $pcf_0^{\mathcal{N}}$. Our first task will be to prove that $pcf_0^{\mathcal{N}}$ is confluent. An argument due to Breazu-Tannen [6] will then be used to show that $pcf^{\mathcal{N}}$ is confluent, from which we may prove that *pcf* itself is confluent.

We use the method of logical relations to prove that $pcf_0^{\mathcal{N}}$ is confluent and strongly normalizing. Following [35], if we can show that a property $\mathcal{S}$ of terms (*i.e.*, a type-indexed family of predicates S^σ over terms of type σ) is *type-closed*, then that property holds for all well-formed terms. The appropriate definition of type-closed depends on the types available; for PCF we will need clauses for function and product types. For convenience, we introduce the concept of an *elimination context*, $\mathcal{E}[\ ]$, which in the case of PCF is a context with a single hole at the head of some sequence of applications and projections (no abstractions or pairs are allowed). We write $\mathcal{S}(\mathcal{E}[\ ])$ to mean that $\mathcal{S}$ holds for each application argument in $\mathcal{E}[\ ]$, *e.g.*, if $\mathcal{E}[\ ] \equiv (\pi_1(\cdot\ N_1))N_2$, then $\mathcal{S}(\mathcal{E}[\ ])$ is short for $S^{\sigma_1}(N_1) \wedge S^{\sigma_2}(N_2)$. Then a property $\mathcal{S}$ is type-closed for PCF if

- $\mathcal{S}(\mathcal{E}[\ ])$ implies $S^\rho(\mathcal{E}[X])$, where X is any variable or constant of appropriate type
- if $S^\tau(Mx)$ for any variable x of type σ, then $S^{\sigma\to\tau}(M)$
- if $S^\sigma(\pi_1 M)$ and $S^\tau(\pi_2 M)$, then $S^{\sigma\times\tau}(M)$
- if $S^\rho(\mathcal{E}[\{N/x\}M])$ and $S^\sigma(N)$, then $S^\rho(\mathcal{E}[(\lambda x{:}\sigma.\, M)N])$
- if $S^\rho(\mathcal{E}[M])$ and $S^\sigma(N)$, then $S^\rho(\mathcal{E}[\pi_1\langle M, N\rangle])$ and $S^\rho(\mathcal{E}[\pi_2\langle N, M\rangle])$.

Lemma 2.4.1 *If a property $\mathcal{S}$ of terms is type-closed, then $S^\sigma(M)$ holds for every well-formed term M of type σ.*

Proof. We may construct another property $\mathcal{P}$ from $\mathcal{S}$ such that $\mathcal{P}$ implies $\mathcal{S}$ and $\mathcal{P}$ is an admissible logical relation; the type-closed conditions on $\mathcal{S}$ are precisely those needed to show this. Then by the Basic Lemma for logical relations, $P^\sigma(M)$, and hence $S^\sigma(M)$, holds for every M; see [35] for details. ∎

Theorem 2.4.2 *The reduction system $pcf_0^{\mathcal{N}}$ is confluent and strongly normalizing.*

Proof. We need to show that the properties $\mathcal{CR}$ and $\mathcal{SN}$, which assert that reduction from a term is respectively confluent (Church-Rosser) and strongly normalizing, are type-closed. Most of the conditions are easy to establish. The hardest part is to show that each property satisfies the first condition when X is a labelled fixed-point constant; in each case an induction on the label is required. ∎

Next we prove that the system $pcf^{\mathcal{N}}$, obtained by adding a set $\mathcal{R}$ of confluent left-linear algebraic rules to $pcf_0^{\mathcal{N}}$, is also confluent. In [6], it is shown that the pure typed lambda calculus (with β-reduction only) remains confluent when any confluent (not necessarily linear) $\mathcal{R}$ is added. While the original proof of one lemma has a subtle but reparable bug when non-linear rules are considered [7], we observe that the proof is correct if all the rules in $\mathcal{R}$ are linear. Therefore, our confluence proof for $pcf^{\mathcal{N}}$ will be based on the original development of [6]. We then use the linearity of $\mathcal{R}$ again to prove that *pcf* itself is confluent; for this argument, linearity is essential.

The following lemma allows us to consider algebraic reductions only on terms in $pcf_0^{\mathcal{N}}$ normal form. By Theorem 2.4.2, every labelled term M has a unique $pcf_0^{\mathcal{N}}$ normal form, which we refer to as $pcf_0^{\mathcal{N}}(M)$.

Lemma 2.4.3

1. *If $M \xrightarrow{r} N$ for $r \in \mathcal{R}$, then $pcf_0^{\mathcal{N}}(M) \xrightarrow{r}\!\!\!\!\rightarrow pcf_0^{\mathcal{N}}(N)$;*
2. *If $M \xrightarrow{pcf^{\mathcal{N}}}\!\!\!\!\rightarrow N$, then $pcf_0^{\mathcal{N}}(M) \xrightarrow{\mathcal{R}}\!\!\!\!\rightarrow pcf_0^{\mathcal{N}}(N)$.*

Proof.

1. If the rule r is $s \rightarrow t$ and the variables in s are $\bar{x} \equiv x_1 \ldots x_n$, then there must be a context $\mathcal{C}[\ ]$ and terms $Q_1 \ldots Q_n$ such that $M \equiv \mathcal{C}[\{\bar{Q}/\bar{x}\}s]$ and $N \equiv \mathcal{C}[\{\bar{Q}/\bar{x}\}t]$. Let z be a new variable of the same type as $(\lambda\bar{x}\colon\bar{\sigma}.\, s)$, and define $P \equiv pcf_0^{\mathcal{N}}(\mathcal{C}[z\bar{Q}])$. Then

$$M \xleftarrow{pcf_0^{\mathcal{N}}}\!\!\!\!\leftarrow \{(\lambda\bar{x}\colon\bar{\sigma}.\, s)/z\}\mathcal{C}[z\bar{Q}] \xrightarrow{pcf_0^{\mathcal{N}}}\!\!\!\!\rightarrow \{(\lambda\bar{x}\colon\bar{\sigma}.\, s)/z\}P,$$

and similarly $N \overset{pcf_0^{\mathcal{N}}}{\longleftrightarrow} \{(\lambda\bar{x}\colon\bar{\sigma}.\, t)/z\}P$. Observe that, since $z\bar{Q}$ is of base type, every occurrence of z in P must be at the head of a subterm of base type; it is then easy to show by induction on the structure of such terms that

$$pcf_0^{\mathcal{N}}(\{(\lambda\bar{x}\colon\bar{\sigma}.\, s)/z\}P) \xrightarrow{r}\!\!\!\!\rightarrow pcf_0^{\mathcal{N}}(\{(\lambda\bar{x}\colon\bar{\sigma}.\, t)/z\}P).$$

Since $pcf_0^{\mathcal{N}}(M) \equiv pcf_0^{\mathcal{N}}(\{(\lambda\bar{x}\colon\bar{\sigma}.\, s)/z\}P)$ and similarly for $pcf_0^{\mathcal{N}}(N)$, we are done.

2. Follows directly from 1 by induction on the length of the reduction.

■

Lemma 2.4.4 *$\mathcal{R}$-reduction is confluent on labelled terms in $pcf_0^{\mathcal{N}}$ normal form.*

Proof. By induction on the length of a normal form M we show that $\mathcal{R}$ is confluent from M. In the base case, M is a variable or fix_σ^0, whence no $\mathcal{R}$-reductions apply, or M is an algebraic constant, for which we know $\mathcal{R}$ is confluent.

Now assume that confluence holds from all $pcf_0^{\mathcal{N}}$ normal forms strictly shorter than M. If M is an abstraction or a pair, then $\mathcal{R}$-reduction can only affect a proper subterm of M, so confluence holds by the induction hypothesis. The remaining case is that M consists of some sequence of applications and projections of a head variable or constant h, *i.e.*, $M \equiv \mathcal{E}[h]$ for some elimination context $\mathcal{E}[\,]$, where all the application arguments $P_1 \ldots P_n$ are $pcf_0^{\mathcal{N}}$ normal forms strictly shorter than M.

If h is a variable or fix_τ^0, then $\mathcal{R}$-reduction will only take place within the arguments $P_1 \ldots P_n$, so confluence holds by the induction hypothesis. If h is an algebraic constant, then because of its type $\mathcal{E}[\,]$ must consist entirely of applications, so $M \equiv h\bar{P}$. If M is not of base type then we are in the same situation as if h were a variable, so confluence holds.

Finally, if $M \equiv h\bar{P}$ is of base type, then it may be decomposed as $\{\bar{Q}/\bar{x}\}s$, where s is algebraic, $x_1 \ldots x_m$ are new variables of base type, and $Q_1 \ldots Q_m$ are $pcf_0^{\mathcal{N}}$ normal forms of base type, each with a variable or fix_σ^0 in head position. Breazu-Tannen calls this an *algebraic trunk decomposition* in [7]. If M is algebraic then confluence holds

by assumption about $\mathcal{R}$. Otherwise, we notice that $Q_1 \ldots Q_m$ and s are all strictly shorter $pcf_0^{\mathcal{N}}$ normal forms, and that $M \xrightarrow{\mathcal{R}}\!\!\!\!\rightarrow N$ only if $N \equiv \{\bar{R}/\bar{x}\}t$, where $s \xrightarrow{\mathcal{R}}\!\!\!\!\rightarrow t$ and $Q_i \xrightarrow{\mathcal{R}}\!\!\!\!\rightarrow R_i$, for $i = 1 \ldots m$. We may prove this by induction on the length of the reduction $M \xrightarrow{\mathcal{R}}\!\!\!\!\rightarrow N$, noting that because we assume that $\mathcal{R}$ is left-linear we avoid the problem with this step which was pointed out in [7]. Thus confluence holds from M by the induction hypothesis, and the induction is complete. ∎

These two lemmas now let us prove that $pcf^{\mathcal{N}}$ is confluent. As a corollary, we use the linearity of the rules in $\mathcal{R}$ to show that *pcf* itself is confluent.

Theorem 2.4.5 *$pcf^{\mathcal{N}}$-reduction is confluent on all labelled PCF terms.*

Proof. For any terms M, N, and P, if $N \xleftarrow{pcf^{\mathcal{N}}}\!\!\!\!\!\!\leftarrow M \xrightarrow{pcf^{\mathcal{N}}}\!\!\!\!\rightarrow P$, then by Lemma 2.4.3 we know that

$$pcf_0^{\mathcal{N}}(N) \xleftarrow{\mathcal{R}}\!\!\!\!\!\!\leftarrow pcf_0^{\mathcal{N}}(M) \xrightarrow{\mathcal{R}}\!\!\!\!\rightarrow pcf_0^{\mathcal{N}}(P).$$

Lemma 2.4.4 then asserts that there is a Q such that

$$pcf_0^{\mathcal{N}}(N) \xrightarrow{\mathcal{R}}\!\!\!\!\rightarrow Q \xleftarrow{\mathcal{R}}\!\!\!\!\!\!\leftarrow pcf_0^{\mathcal{N}}(P).$$

Since $N \xrightarrow{pcf^{\mathcal{N}}}\!\!\!\!\rightarrow pcf_0^{\mathcal{N}}(N)$ and $P \xrightarrow{pcf^{\mathcal{N}}}\!\!\!\!\rightarrow pcf_0^{\mathcal{N}}(P)$, we thus have that $N \xrightarrow{pcf^{\mathcal{N}}}\!\!\!\!\rightarrow Q \xleftarrow{pcf^{\mathcal{N}}}\!\!\!\!\!\!\leftarrow P$. ∎

Corollary 2.4.6 *pcf-reduction is confluent on all PCF terms.*

Proof. For any terms M, N, and P, if $N \xleftarrow{pcf}\!\!\!\!\!\!\leftarrow M \xrightarrow{pcf}\!\!\!\!\rightarrow P$, then there are corresponding labelled terms M^*, N^*, and P^* such that $N^* \xleftarrow{pcf^{\mathcal{N}}}\!\!\!\!\!\!\leftarrow M^* \xrightarrow{pcf^{\mathcal{N}}}\!\!\!\!\rightarrow P^*$. To see this, let m be the length of the longer of the two *pcf*-reduction sequences from M, and form M^* by labelling each fix_σ in M with m; then mimic the two reductions from M by replacing (*fix*) steps with (fix^n) steps, for appropriate choices of n. Now, since $pcf^{\mathcal{N}}$ is confluent, N^* and P^* must have a common reduct Q^*; erasing the labels in Q^* gives a term Q such that $N \xrightarrow{pcf}\!\!\!\!\rightarrow Q \xleftarrow{pcf}\!\!\!\!\!\!\leftarrow P$, hence *pcf* is confluent. ∎

The proof of this corollary uses linearity in asserting that a single labelled term M^* will permit both reductions from M to be mimicked. This property fails in the presence of non-linear rules because a reduction step may then require that two subterms have identical labels. An example of the problem, using (*sp*), is the term

$M^* \equiv \langle \pi_1(\mathit{fix}^m_\sigma\, I), \pi_2(\mathit{fix}^n_\sigma\, I)\rangle$, where $I \equiv (\lambda x{:}\,\sigma.\, x)$, and σ is a product type. Consider the following two reduction sequences:

$$M \xrightarrow{sp} \mathit{fix}^m_\sigma\, I$$

and

$$\begin{array}{rcl} M & \xrightarrow{\mathit{fix}^n} & \langle \pi_1(\mathit{fix}^m_\sigma\, I), \pi_2((\lambda f{:}\,\sigma{\to}\sigma.\, f(\mathit{fix}^{n-1}_\sigma\ f))I)\rangle \\ & \xrightarrow{\beta}\!\!\!\!\rightarrow & \langle \pi_1(\mathit{fix}^m_\sigma\, I), \pi_2(\mathit{fix}^{n-1}_\sigma\, I)\rangle \\ & \xrightarrow{sp} & \mathit{fix}^m_\sigma\, I. \end{array}$$

In the first case, the (sp) step requires that $m = n$, while in the second case we must have $m = n - 1$, hence there is no such M^*.

2.5 Postponement

In this section, we analyze the reduction system $\mathit{pcf}_{\eta,sp}$. Our main technical tool is a postponement theorem for (η) and (sp), which is proved following the pattern for postponement of (η) in the untyped lambda calculus [1]. While (sp) postponement fails for untyped lambda terms (as noted in the introduction), we are able to prove postponement for typed PCF terms. The main "trick" in the proof is to find the correct analogy between sp-reduction and η-reduction. The postponement theorem will be used in the next section to show that pcf is sufficient to compute the $\mathit{pcf}_{\eta,sp}$ normal form of any *program*, *i.e.*, any closed term of base type (in fact, programs may have free variables as long as they are of algebraic type, since they act like algebraic constants with no associated reductions).

To show postponement of (η) and (sp) for $\mathit{pcf}_{\eta,sp}$, we will use the related system pcf^{lab}, in which η- and sp-redexes are represented as labels. We add term formation rules to allow M^η whenever M is a term of function type, and M^{sp} whenever M is of product type; substitution is then extended in the natural way. We also define two functions from labelled to unlabelled terms: $|\cdot|$ and φ. The action of $|\cdot|$ is simply to erase all the labels, while φ replaces labelled subterms with the corresponding redexes: $\varphi(M^\eta) = (\lambda x{:}\,\sigma.\, \varphi(M)x)$ if $M{:}\,\sigma{\to}\tau$, and $\varphi(M^{sp}) = \langle \pi_1\varphi(M), \pi_2\varphi(M)\rangle$. Finally, the

reduction system pcf^{lab} is defined by lifting pcf to labelled terms and adding the following contractions:

$$(act\ \eta) \qquad M^{\eta}N \longrightarrow MN$$

$$(act\ sp) \qquad \pi_i M^{sp} \longrightarrow \pi_i M, \text{ for } i = 1, 2$$

$$(int\ \eta) \qquad (\lambda x{:}\sigma.\, M)^{\eta} \longrightarrow (\lambda x{:}\sigma.\, M)$$

$$(int\ sp) \qquad \langle M, N\rangle^{sp} \longrightarrow \langle M, N\rangle.$$

The effect of the first two rules is to simulate the reduction of "active" η- and sp-redexes, *i.e.*, those that are also top-level constituents of β- or π-redexes. The other two rules mimic the situation where an η- or sp-redex is reduced internally by (β) or (π), *e.g.*,

$$\lambda x{:}\sigma.\, (\lambda x{:}\sigma.\, M)x \xrightarrow{\beta} \lambda x{:}\sigma.\, M;$$

these two rules are not necessary for the postponement proof, but they will be needed in the next section and there is no harm in adding them here.

We will now prove a series of lemmas relating pcf^{lab} to the unlabelled systems. The first three are easy inductions, either on the length of a reduction or on the structure of a term.

Lemma 2.5.1 *If* $P \xrightarrow{pcf^{lab}\!\!\!\!\!\rightarrow} P'$, *then* $\varphi(P) \xrightarrow{pcf}\!\!\!\!\!\rightarrow \varphi(P')$.

Proof. This follows by an easy induction on the length of the pcf^{lab} reduction. The $(act\ \eta)$ steps may be simulated in the unlabelled reduction by (β) steps, since the η-redexes involved are active; similarly, the $(act\ sp)$ steps become (π_i) steps. The two internal rules may also be simulated, because β- and π_i-redexes are created by φ. Any other reduction step corresponds directly to a similar reduction on the unlabelled term. ∎

Lemma 2.5.2 $\varphi(P') \xrightarrow{\eta, sp}\!\!\!\!\!\rightarrow |P'|$.

Proof. Another easy induction, this time on the structure of P'. Each η- or sp-redex introduced by φ can obviously be reduced to get to $|P'|$. ∎

Lemma 2.5.3 *If* $|P| \xrightarrow{pcf}\!\!\!\!\!\rightarrow N$, *then there exists a term* P' *such that* $P \xrightarrow{pcf^{lab}}\!\!\!\!\!\rightarrow P'$ *and* $|P'| \equiv N$.

Proof. The *pcf* reduction to N can be simulated on P because whenever a label is in the way it must indicate an active redex, so it can be erased by one of the extra rules in pcf^{lab}. ∎

Lemma 2.5.4 *If* $M \xrightarrow{\eta,sp} M' \xrightarrow{pcf}\!\!\!\!\!\rightarrow N$, *then there exists a term* Q *such that* $M \xrightarrow{pcf}\!\!\!\!\!\rightarrow Q \xrightarrow{\eta,sp}\!\!\!\!\!\rightarrow N$.

Proof. Either $M \equiv \mathcal{C}[(\lambda x{:}\sigma.\, Lx)]$ and $M' \equiv \mathcal{C}[L]$, or $M \equiv \mathcal{C}[\langle \pi_1 L, \pi_2 L\rangle]$ and $M' \equiv \mathcal{C}[L]$, for some context $\mathcal{C}[\ ]$ and term L. In the first case, take $P \equiv \mathcal{C}[L^{\eta}]$; in the second case, take $P \equiv \mathcal{C}[L^{sp}]$. Then $M \equiv \varphi(P)$ and $M' \equiv |P|$. By Lemma 2.5.3, there is a term P' such that $P \xrightarrow{pcf^{lab}}\!\!\!\!\!\rightarrow P'$ and $|P'| \equiv N$. Then by Lemma 2.5.1 we find that $M \equiv \varphi(P) \xrightarrow{pcf}\!\!\!\!\!\rightarrow \varphi(P')$. Since by Lemma 2.5.2, $\varphi(P') \xrightarrow{\eta,sp}\!\!\!\!\!\rightarrow |P'| \equiv N$, we may take $Q \equiv \varphi(P')$. ∎

From this lemma we may now prove the postponement of (η) and (sp) in $pcf_{\eta,sp}$.

Theorem 2.5.5 *If* $L \xrightarrow{pcf_{\eta,sp}}\!\!\!\!\!\rightarrow N$, *then there is a term* M *such that* $L \xrightarrow{pcf}\!\!\!\!\!\rightarrow M$ *and* $M \xrightarrow{\eta,sp}\!\!\!\!\!\rightarrow N$.

Proof. Given a reduction sequence from L to N, we may use the previous lemma to push all the (η) and (sp) steps to the end. ∎

In the special case that N is a program in $pcf_{\eta,sp}$ normal form, which we refer to as a *result*, we find that the reductions (η) and (sp) are unnecessary:

Corollary 2.5.6 *If* $L \xrightarrow{pcf_{\eta,sp}}\!\!\!\!\!\rightarrow R$ *and* R *is a result, then* $L \xrightarrow{pcf}\!\!\!\!\!\rightarrow R$.

Proof. By Theorem 2.5.5 there is an M such that $L \xrightarrow{pcf}\!\!\!\!\!\rightarrow M \xrightarrow{\eta,sp}\!\!\!\!\!\rightarrow R$; we will proceed by induction on the length of the reduction from M to R. If the last step in this reduction is $P \xrightarrow{\eta,sp} R$, with Q the redex in P and Q' its contractum in R, then assume first that Q is passive. Thus Q' is of non-base type and it cannot be the head of an application or the object of a projection; if it is in the body of an abstraction or a pair, then there is a larger subterm of R that is of non-base type.

Consider the largest such enclosing subterm of non-base type (possibly Q' itself). Since R is a normal form of base type, this subterm must be in the argument of an application. But algebraic constants only take base type arguments, and the presence of a *fix* contradicts R being in normal form, so the head of the application must be a variable. This is also impossible, because there are no free variables in R (except perhaps of algebraic type), and a surrounding abstraction would give yet a larger subterm of non-base type. Hence Q must be active, so $P \xrightarrow{pcf} R$ and we may use Lemma 2.5.4 to push this step in front of the (η) and (sp) steps; it is easy to see by the form of Q that this will not increase the number of remaining (η) and (sp) steps. ■

2.6 The result property

Although $pcf_{\eta,sp}$ is not confluent, there are other properties of reduction that might be considered as plausible substitutes. In this section, we will show that $pcf_{\eta,sp}$ has a weaker *result property.* To put this property in perspective, we might consider three general properties of reduction, in order of decreasing strength. These are confluence, the so-called *normal form property* of [29], which says that if $\Gamma \rhd M \leftrightarrow N : \sigma$ and N is a normal form, then $M \twoheadrightarrow N$, and the uniqueness of normal forms. It is not hard to see that confluence implies the normal form property, and the normal form property implies that each term has at most one normal form. While Klop and de Vrijer have shown that the normal form property fails in untyped lambda calculus with surjective pairing [29], we will show that typed PCF has a modified version of this property, which we call the *result property.*

The result property, proved in Theorem 2.6.4 below, states that if M is $pcf_{\eta,sp}$ convertible to a result N (see Section 2.5), then M is $pcf_{\eta,sp}$ reducible to N. Since conversion is equivalent to provable equality in the full PCF proof system, it follows by Corollary 2.5.6 that if a term M is provably equal to a result N, then $M \xrightarrow{pcf} N$. Furthermore, it follows from the result property and postponement that (η) and (sp) are sound equational rules for reasoning about pcf observational congruence. Thus, when combined with other properties of $pcf_{\eta,sp}$ and pcf reduction, the result property

not only gives a certain coherence to $pcf_{\eta,sp}$ reduction, but relates provable equality using (η) and (sp) to program execution without these "non-computational" rules.

We will prove the result property using a series of lemmas similar to those used to show postponement in Section 2.5. The same labelled system pcf^{lab} will be used, and here the internal rules will be important.

Lemma 2.6.1 *If $P \xrightarrow{pcf^{lab}}\!\!\!\!\rightarrow P'$, then $|P| \xrightarrow{pcf}\!\!\!\!\rightarrow |P'|$.*

Proof. This is almost trivial; any of the extra reductions in pcf^{lab} not in pcf may be ignored because all of the labels have been erased. ∎

Lemma 2.6.2 *If $\varphi(P) \xrightarrow{pcf}\!\!\!\!\rightarrow R$ and R is a result, then there exists a term P' such that $P \xrightarrow{pcf^{lab}}\!\!\!\!\rightarrow P'$ and $\varphi(P') \equiv |P'| \equiv R$.*

Proof. Since there are no η- or sp-redexes in R, there will be no labels left in P'. Therefore, we will have $\varphi(P') \equiv |P'|$, and every descendent of a redex in $\varphi(P)$ corresponding to a label in P must eventually either pcf-reduce or be erased. The pcf-reductions will correspond to cases where a redex either is active or reduces internally; they may thus be simulated by the labelled reduction. One complication comes in simulating the reduction step $\langle \pi_1 M, \pi_2 M\rangle \rightarrow \langle \pi_1 M', \pi_2 M\rangle$; there is no reduction on the corresponding labelled term M^{sp} that matches this, but since pcf is confluent and R is in normal form, we know that the two components of the pair will eventually reduce to the same normal form (or the entire pair will be erased), so we may arbitrarily choose to follow only reductions to the first component.

The other situation where we must be careful is when there are rules in $\mathcal{R}$ of the form $(f M_1 \ldots M_{k-1} x) \rightarrow N$; *i.e.*, rules that accept an arbitrary rightmost argument. If $P \equiv (f M_1 \ldots M_{k-1})^{\eta}$, then $\varphi(P) \rightarrow (\lambda x{:}\, b.\, N)$, but P does not pcf^{lab}-reduce. This case is ruled out by the condition that R be a result, however, because reasoning similar to that in the proof of Corollary 2.5.6 shows that algebraic constants in R must always be at the head of subterms of base type. Hence we may simulate the pcf reduction in the labelled system. ∎

Lemma 2.6.3 *If $M \xrightarrow{\eta,sp} M'$ and $M \xrightarrow{pcf}\!\!\!\!\rightarrow R$, where R is a result, then $M' \xrightarrow{pcf}\!\!\!\!\rightarrow R$.*

Proof. Take P as in Lemma 2.5.4, so that $M \equiv \varphi(P)$ and $M' \equiv |P|$. By Lemma 2.6.2, there is a term P' such that $P \overset{pcf^{lab}}{\longrightarrow\!\!\!\!\!\rightarrow} P'$ and $\varphi(P') \equiv |P'| \equiv R$. By Lemma 2.6.1 we find that $M' \equiv |P| \overset{pcf}{\longrightarrow\!\!\!\!\!\rightarrow} |P'|$, so we are done. ∎

Now we may prove the result property for *pcf*.

Theorem 2.6.4 *If* $\Gamma \rhd N \overset{pcf_{\eta,sp}}{\longleftrightarrow} R : \sigma$ *and* R *is a result, then* $N \overset{pcf}{\longrightarrow\!\!\!\!\!\rightarrow} R$.

Proof. We will prove that if $L \overset{pcf_{\eta,sp}}{\longrightarrow\!\!\!\!\!\rightarrow} N$ and $L \overset{pcf_{\eta,sp}}{\longrightarrow\!\!\!\!\!\rightarrow} R$, where R is a result, then $N \overset{pcf}{\longrightarrow\!\!\!\!\!\rightarrow} R$; this is easily seen to be equivalent. By the postponement theorem there is a term M such that $L \overset{pcf}{\longrightarrow\!\!\!\!\!\rightarrow} M \overset{\eta,sp}{\longrightarrow\!\!\!\!\!\rightarrow} N$; by the corollary to postponement, $L \overset{pcf}{\longrightarrow\!\!\!\!\!\rightarrow} R$. Now, since *pcf* is confluent and R is in normal form, we know that $M \overset{pcf}{\longrightarrow\!\!\!\!\!\rightarrow} R$. Using Lemma 2.6.3 once for each reduction step from M to N, we find that $N \overset{pcf}{\longrightarrow\!\!\!\!\!\rightarrow} R$. ∎

As a corollary to this theorem, we will show that the equational forms of (η) and (sp) are sound for reasoning about observational congruence. First we define a *program context* for a given type σ and variable assignment Γ to be a context $\mathcal{P}[\ ]$ such that $\mathcal{P}[M]$ is a program whenever $\Gamma \rhd M{:}\sigma$. We say that two terms M and N of the same type σ and variable assignment Γ are *observationally congruent*, written $\Gamma \rhd M \simeq N{:}\sigma$, if for every program context $\mathcal{P}[\ ]$ for σ and Γ, $\mathcal{P}[M]$ *pcf*-reduces to a result R iff $\mathcal{P}[N] \overset{pcf}{\longrightarrow\!\!\!\!\!\rightarrow} R$. In other words, M and N are completely interchangeable when computing the result of a program.

Corollary 2.6.5 *The equational axioms* $(\eta)_{eq}$ *and* $(sp)_{eq}$ *are sound for* $\simeq$.

Proof. To show that (η) is sound, we need to show that for any term M of type $\sigma{\to}\tau$ over a variable assignment Γ, with x a variable not free in M, we have $\Gamma \rhd (\lambda x{:}\sigma.\, Mx) \simeq M{:}\sigma{\to}\tau$. If $\mathcal{P}[\ ]$ is a program context for $\sigma{\to}\tau$ and Γ, then obviously $\Gamma \rhd \mathcal{P}[(\lambda x{:}\sigma.\, Mx)] \overset{pcf_{\eta,sp}}{\longleftrightarrow} \mathcal{P}[M] : \sigma$; if $\mathcal{P}[(\lambda x{:}\sigma.\, Mx)]$ *pcf*-reduces to a result R, then we also have that $\Gamma \rhd \mathcal{P}[M] \overset{pcf_{\eta,sp}}{\longleftrightarrow} R : \sigma$, and by the previous theorem we find that $\mathcal{P}[M] \overset{pcf}{\longrightarrow\!\!\!\!\!\rightarrow} R$. The same argument in reverse shows that if $\mathcal{P}[M] \overset{pcf}{\longrightarrow\!\!\!\!\!\rightarrow} R$, then $\mathcal{P}[(\lambda x{:}\sigma.\, Mx)] \overset{pcf}{\longrightarrow\!\!\!\!\!\rightarrow} R$. Similar reasoning shows that (sp) is sound. ∎

2.7 Expansion

An alternate approach to avoiding the non-confluence of the full reduction system was suggested to the author after the rest of this chapter was originally written. If, instead of directing the extensional axioms from left to right to obtain the contraction rules (η) and (sp), one orients them from right to left, then the resulting *expansion* rules are considerably better-behaved. We will refer to the reduction system consisting of *pcf* plus the two expansion rules

$$(\eta)_{exp} \qquad M \longrightarrow \lambda x{:}\sigma.\, Mx, \text{ for } x \text{ not free in } M$$

$$(sp)_{exp} \qquad P \longrightarrow \langle \pi_1 P, \pi_2 P \rangle$$

as pcf_{exp}. Naturally, for the right-hand sides of these rules to be well-typed, M must be a term of function type (with argument type σ) and P must be a term of product type.

At first glance, using the expansion forms of these rules might seem worse than the previous contraction rules, since they allow infinite reduction sequences such as

$$f \longrightarrow \lambda x{:}\sigma.\, fx \longrightarrow \lambda x{:}\sigma.\, (\lambda y{:}\sigma.\, fy)x \longrightarrow \ldots.$$

However, in the third term of this sequence we have created a β-redex which, if reduced, will return the sequence to $\lambda x{:}\sigma.\, fx$. In general this will be true — after some point in a sequence of expansion steps any further expansions will be reversible. If we restrict application of the expansion steps to cases where no β- or π-redexes are created, then such infinite reductions will be avoided. More formally, we will take an η-expansion redex to be a subterm M of function type which is neither an abstraction nor the head of an application, and an *sp*-expansion redex to be a subterm P of product type which is neither a pair nor the object of a projection.

It is not difficult to prove that pcf_{exp} is confluent, following the procedure used in Section 2.4. First we may show that $pcf^{\mathcal{N}}_{exp,0}$, the system with labelled fixed-point constants and no algebraic rules, is confluent and strongly normalizing, either by adapting the logical relations proof discussed previously, or by modifying the

proof for a similar system (with iteration instead of labelled *fix*-reduction) given by Jay [27], which is based on Girard's "candidats de reducibilité." The normal forms in this strongly normalizing system are known as *long normal forms*; for example, the long normal form of a variable $f\colon \rho \to \sigma \times \tau$, where σ and τ are base types, is $lnf(f) \equiv \lambda x\colon \rho.\, \langle \pi_1(fx), \pi_2(fx)\rangle$.

Using long normal forms instead of $pcf_0^{\mathcal{N}}$ normal forms, Breazu-Tannen's proof goes through as before to establish that $pcf_{exp}^{\mathcal{N}}$ is confluent, provided $\mathcal{R}$ is a confluent set of linear algebraic rules. Since the expansion rules are linear, there is then no problem extending this result as above to show that $pcf_{\,exp}$ itself is confluent.

Theorem 2.7.1 *The reduction system $pcf_{\,exp}$ is confluent.*

Since $pcf_{\,exp}$ is confluent, we immediately find that it satisfies the normal form property.

Corollary 2.7.2 *If $\Gamma \rhd M \overset{pcf_{\,exp}}{\longleftrightarrow} L : \sigma$ (or, equivalently, $\Gamma \rhd M \overset{pcf_{\,\eta,sp}}{\longleftrightarrow} L : \sigma$), where L is a long normal form, then $M \overset{pcf_{\,exp}}{\longrightarrow} L$.*

Postponement does not hold in general for the expansion rules. For example, consider the reduction sequence

$$Ip \overset{sp_{exp}}{\longrightarrow} \langle \pi_1(Ip), \pi_2(Ip)\rangle \overset{\beta}{\longrightarrow} \langle \pi_1 p, \pi_2(Ip)\rangle,$$

where $I \equiv \lambda x\colon \sigma \times \tau.\, x$. The *sp*-expansion must happen first since the β-contraction only affects one component of the pair. It is not enough to require that the reduction sequence end in a normal form, either. Consider the system with the algebraic rule $r\colon zero\, x \longrightarrow 0$; in the sequence

$$zero \overset{\eta_{exp}}{\longrightarrow} \lambda x\colon nat.\, zero\, x \overset{r}{\longrightarrow} \lambda x\colon nat.\, 0,$$

the final term is in normal form, but the η-expansion step must precede the algebraic step because it creates the algebraic redex (note that we only disallow η-expansions which create β-redexes).

Postponement of the expansion rules does hold for reduction sequences ending in

results. However, just as in Corollary 2.5.6, when the final term is a result, all of the extensional steps disappear:

Theorem 2.7.3 *If $M \overset{pcf_{exp}}{\longrightarrow\!\!\!\!\rightarrow} R$, where R is a result, then $M \overset{pcf}{\longrightarrow\!\!\!\!\rightarrow} R$.*

Proof. This is just a special case of the result property for *pcf*. ∎

Thus we find again that *pcf* is the proper reduction system for evaluating programs; also, by adding the extensional rules as expansions, we may find normal forms for arbitrary terms which have them.

2.8 Completeness of leftmost reduction

If the set $\mathcal{R}$ of algebraic rules satisfies some additional constraints, then we may prove that the leftmost reduction strategy is complete with respect to finding *pcf* (or pcf_{exp}) normal forms. The extra constraints are that $\mathcal{R}$ be terminating and *left-normal* (which will be defined below). This result is similar to one obtained by Klop [28], who proves that leftmost reduction is normalizing for the untyped lambda calculus plus any left-normal, regular term rewriting system. For a term rewriting system to be regular the rules must be linear and non-overlapping; we take the alternate approach of accepting any confluent set of linear algebraic rules, as long as they are terminating. Our proof is made simpler by this assumption of termination, as well as the isolation of non-termination in the base language to the fixed-point constants; also, we do not pass through the intermediate step of a general standardization theorem.

An algebraic rule is left-normal if all the variables in the left-hand side of the rule appear to the right of all the constants. For example, all of the rules given in Section 2.3 for natural numbers and booleans are left-normal. If we had chosen to permute the arguments of the conditional so that the test was at the end, *i.e.*, so that the reduction rules would be

$$\begin{aligned} cond\ M\ N\ true &\longrightarrow M \\ cond\ M\ N\ false &\longrightarrow N, \end{aligned}$$

then we would not have had a left-normal set of rules. Intuitively, if we have left-normal rules, then we only need to evaluate the arguments of an algebraic term from left to right; it will never be the case that skipping an argument and evaluating one to its right will create an algebraic redex involving the skipped term. In many cases, obtaining left-normal rules is simply a matter of permuting arguments; in some other cases, we may obtain left-normal rules by introducing auxiliary function symbols. For example, consider the following set of rules, where the value of the first argument determines which of the other two arguments is to be evaluated:

$$\begin{aligned} f\ true\ true\ N &\longrightarrow true \\ f\ true\ false\ N &\longrightarrow false \\ f\ false\ M\ true &\longrightarrow false \\ f\ false\ M\ false &\longrightarrow true. \end{aligned}$$

An equivalent left-normal system is

$$\begin{aligned} f\ true\ M\ N &\longrightarrow M \\ f\ false\ M\ N &\longrightarrow not\ N \\ not\ true &\longrightarrow false \\ not\ false &\longrightarrow true. \end{aligned}$$

This is just the process of determining the *sequentiality index* of the function defined by the algebraic rules (see [3]) and rearranging arguments so that the index is always the leftmost unevaluated argument. If a function is inherently non-sequential then we will not be able to find a left-normal set of rules; the classic example is the *parallel or* function, *por*:

$$\begin{aligned} por\ true\ N &\longrightarrow true \\ por\ M\ true &\longrightarrow true \\ por\ false\ false &\longrightarrow false \end{aligned}$$

The first lemma we will need formalizes a special case of our intuition about the creation of $pcf^{\mathcal{N}}$ redexes when the algebraic rules are left-normal.

Lemma 2.8.1 *If the rules $\mathcal{R}$ are left-normal, then whenever the leftmost redex R in a labelled term M is contracted by a rule of $pcf^{\mathcal{N}}$ to obtain a term N, all of the non-algebraic subterms of M to the left of R will also be to the left of the leftmost redex of N.*

Proof. Suppose that P is a non-algebraic subterm of M to the left of R and that contraction of R creates a redex S in N which is to the left of P. No (β), (π) or (fix) rule could be created so far to the left of R, so S must be an algebraic redex which contains the reduct of R, and hence also contains P. The only way an algebraic redex can have a non-algebraic subterm P is if the rule contains a variable covering that position; since the rules are left-normal, this means that the rule also contains a variable covering the position of the reduct of R, which contradicts the fact that S was created by the contraction of R. Therefore P must be to the left of the leftmost redex in N. ∎

We will also need the fact that $pcf^{\mathcal{N}}$ is strongly normalizing whenever the set of rules $\mathcal{R}$ is terminating. This can be seen as a special case of a theorem of Breazu-Tannen and Gallier [7], where they consider adding a terminating set of algebraic rules to the polymorphic lambda calculus, including the (η) rule. The proof we give is adapted from a simplification of theirs.

Lemma 2.8.2 *If the rules of $\mathcal{R}$ are terminating, then $pcf^{\mathcal{N}}$ is strongly normalizing.*

Proof. We will extend the logical relations proof of strong normalization for $pcf_0^{\mathcal{N}}$ given in Section 2.4. The only significant case we need to consider is showing that $\mathcal{SN}(\mathcal{E}[\ ])$ implies $SN^{\rho}(\mathcal{E}[f])$, where f is an algebraic constant. That is, if strong normalization in $pcf^{\mathcal{N}}$ holds from $N_1, \ldots, N_k$, then we need to show that it holds from $M \equiv fN_1 \ldots N_k$. If the arity of f is greater than k then we are done, so assume that $fN_1 \ldots N_k$ is of base type. Since each term N_i is strongly normalizing, and the rules in $\mathcal{R}$ are linear, the only way to have an infinite reduction from M is if there is an infinite reduction from $fP_1 \ldots P_k$, where each P_i is the normal form of the corresponding N_i.

Now, an algebraic reduction cannot create new $pcf_0^{\mathcal{N}}$ redexes, so this infinite reduction must be all algebraic, contradicting the fact that $\mathcal{R}$ is terminating. Therefore M is strongly normalizing. ∎

Using these two lemmas we may now prove the main theorem of this section:

Theorem 2.8.3 *If $M \xrightarrow{pcf}\!\!\!\!\rightarrow N$, where N is a normal form and the rules in $\mathcal{R}$ are confluent, terminating, left-linear, and left-normal, then there is a pcf reduction from M to N that reduces the leftmost redex at each step.*

Proof. If $M \xrightarrow{pcf}\!\!\!\!\rightarrow N$ then using the linearity of the rules in $\mathcal{R}$ we may find a labelled reduction $M^* \xrightarrow{pcf^{\mathcal{N}}}\!\!\!\!\rightarrow N^*$ such that M^* and N^* erase to M and N, respectively. Since $pcf^{\mathcal{N}}$ is confluent and strongly normalizing, we may find an equivalent leftmost reduction from M^* to N^*. Now, the only way the erasure of this leftmost reduction will not be a leftmost *pcf* reduction is if there is some subterm fix^0 to the left of the leftmost redex at some step in the labelled reduction. But by Lemma 2.8.1, this subterm must also be present in N^*, which contradicts the fact that N is a normal form. Therefore we have a leftmost reduction from M to N. ∎

Corollary 2.8.4 *If a program P is provably equal to a result R, and the rules in $\mathcal{R}$ are confluent, terminating, left-linear, and left-normal, then the leftmost pcf reduction from P will reach R.*

Proof. Obvious from the theorem and the result property. ∎

In particular, the algebraic rules for natural numbers and booleans given in Section 2.3 satisfy all the restrictions, hence the leftmost reduction strategy will find results of programs over *nat* and *bool*. This gives a correspondence between nondeterministic reduction and the deterministic evaluation strategy used in the semantic study of [43], providing a full correspondence between axiomatic, operational and denotational semantics of PCF.

All the results of this section hold for the system with expansion rules. In particular, if any term is provably equal to a long normal form in the full proof system with an appropriate set of algebraic rules, then leftmost pcf_{exp} reduction will find that normal form.

2.9 Conclusion

Since the full reduction system $pcf_{\eta,sp}$ with (η) and (sp) is not confluent, we consider the more limited system *pcf* to be more appropriate for PCF execution. The system *pcf* includes fixed-point rules (*fix*), lambda calculus rules (β) for function calls and (π) for pairs, and any set $\mathcal{R}$ of left-linear, confluent algebraic rules. We have shown that *pcf* reduction is confluent and demonstrated several connections between *pcf* and $pcf_{\eta,sp}$. The first is that (η) and (sp) rules may be postponed in $pcf_{\eta,sp}$ reduction, and so whenever a closed term of base type $pcf_{\eta,sp}$ reduces to normal form, this reduction may be accomplished without (η) or (sp). Thus, if we consider programs to be closed terms of base type (or any product of such types), *pcf* is equivalent for the purpose of program execution to the apparently stronger but non-confluent $pcf_{\eta,sp}$. We also prove a *result property* of $pcf_{\eta,sp}$ in Section 2.6, from which it follows that *(i)* whenever a term is provably equal (in the full system) to a result, the term reduces to this result by *pcf* reduction, and *(ii)* all equational rules, including $(\eta)_{eq}$ and $(sp)_{eq}$, are sound for *pcf* observational congruence. We consider adding expansion rules for (η) and (sp), showing that the resulting system is confluent and adequate for finding long normal forms of arbitrary terms. We also show that a leftmost reduction strategy is complete for *pcf* reduction if the algebraic rules satisfy the additional constraints that they are terminating and left-normal. In summary, these technical results suggest that while the full equational proof system is a natural "axiomatic semantics" for PCF, a more limited reduction system has more desirable technical properties and seems suitable as a corresponding operational semantics.

One open problem is to extend our confluence theorem to include reduction rules for non-algebraic terms. For example, we might like to give reduction rules for the evaluation of some higher-order function, and be sure that the resulting extension of *pcf* is confluent. Presumably our current proof already applies to some cases of this form, but we have not yet done a careful analysis.

Chapter 3

Inductive and Projective Types

In this and the next chapter we consider two related typed λ-calculi which allow the construction of types as fixed points of recursive type expressions. The smaller system, $\lambda^{\mu\nu}$, restricts the recursion so that all functions definable in the system are total. It is still quite powerful, however; it is more expressive than Gödel's system **T**, since not only are all the functions that are provably total in Peano arithmetic definable, but also many that are not. The larger calculus, $\lambda^{\perp\rho}$, contains all of the types and terms of $\lambda^{\mu\nu}$, plus it allows more general recursive types by including a lifting constructor to explicitly mark the types where non-termination is allowed; as a result, all computable partial functions are definable. By separating out the recursive types which only allow bounded computations, we obtain a finer resolution of where non-termination is possible than that provided by other type systems. In particular, we compare our work to Plotkin's versions of PCF [43, 44] and Hagino's Categorical Data Types [21, 22]. Indeed, our work may be viewed as an extension of Hagino's language, which is roughly equivalent in expressive power to $\lambda^{\mu\nu}$, into a system equivalent in power to PCF. The existence of a clean category theoretic semantics for the language has been the motivating force behind a number of design decisions; we will also discuss these choices and some of the alternatives.

When trying to give a categorical semantics to a functional language with recursion as well as product and sum types, it is well-known that something must be given up. Specifically, if a cartesian closed category has an initial object 0 (the identity for the

sum) and if there is a solution to the equation $X \cong X \to 0$, then every object will be isomorphic and all arrows will be equal.[1] The standard solution is to drop the requirement of full categorical sums. For example, the category of complete pointed partial orders and total continuous functions (**CPPO**) has a categorical product and solutions of all recursive domain equations (RDEs), but only has the "coalesced" sum, which identifies the bottom elements of the summands — this loss of information leads to the sum not being "extensional," in the sense of every element of the sum type being uniquely expressible as the injection of an element of one of the summands. On the other hand, Plotkin's category $\mathbf{CPO}^{\rightharpoonup}$ of complete (not necessarily pointed) partial orders and partial continuous functions has extensional sums and solutions to all RDEs, but the cartesian product is not the categorical product; there *is* a categorical product, but it is not adjoint to the function space functor. A third alternative, which we adopt, is to drop the requirement that every RDE have a solution; an example of this approach is the category **CPO** of complete partial orders and total continuous functions, which is cartesian closed and has full categorical sums, but only a restricted class of RDEs has solutions. It is this last category which provides the motivation for the calculi we study here.

In the approach we take to giving a categorical model for a typed functional language, the objects of the category represent the types and the arrows represent terms, where the source gives the type of the free variables and the target gives the resulting type of the term. A type expression with a free type variable then becomes an endofunctor, with the action of the functor on an object being given by substitution of the corresponding type for the free variable (we will describe the action on an arrow later). Given a type expression σ with a free type variable t, corresponding to a functor F, a solution to the RDE $t = \sigma$ is a type τ such that τ is isomorphic to σ with τ substituted for t (written $\tau \cong \{\tau/t\}\sigma$). That is, a solution

[1] Using terms in the language we develop below, if we have a type X such that $X \cong X \to 0$, then the closed term $\omega \equiv \lambda x{:}X.\ \mathit{unfold}_X\, xx$ has type $X \to 0$. Therefore, the term $\Omega \equiv \omega(\mathit{fold}_X\, \omega)$ is a closed term of type 0 (which is essentially a typed version of the classical combinator Ω, the simplest term with no normal form). It is then easy to see that the terms $\lambda x{:}1.\,\Omega$ and $\lambda y{:}0.\,\diamond$ are inverses, hence $0 \cong 1$; it follows that every type A is isomorphic to 0, since $A \cong A \times 1 \cong A \times 0 \cong 0$. For another approach to the proof, see for example [31], section I.8.

is an object X such that $X \cong FX$, *i.e.*, a fixed point of F. Common notation for the type τ so constructed is $\mu t.\,\sigma$; we will often write μF when we are particularly thinking of σ as a functor.

In a category with an initial object 0 and colimits of ω-chains, a fixed point of an endofunctor F which preserves such colimits (*i.e.*, an *ω-cocontinuous* functor) may be found as the colimit of the chain

$$0 \xrightarrow{\Box} F0 \xrightarrow{F\Box} F^2 0 \xrightarrow{F^2\Box} \ldots,$$

where $\Box$ is the unique arrow from 0 to $F0$. The fixed point μF found by this method is the least one, in the sense that it is the initial object in the category of F-algebras (whose objects are arrows $f\colon FA \to A$ in the original category; an arrow between f and $g\colon FB \to B$ is an arrow $h\colon A \to B$ such that $f; h = Fh; g$). Since a colimit is also known as an inductive limit, we refer to the types constructed in this manner as *inductive* types. Dually, in a category with a terminal object 1 and limits of ω-chains, we find the greatest fixed point (which is terminal in the category of F-coalgebras) of an ω-continuous endofunctor by taking the (projective) limit of

$$1 \xleftarrow{\Diamond} F1 \xleftarrow{F\Diamond} F^2 1 \xleftarrow{F^2\Diamond} \ldots,$$

where $\Diamond$ is the unique arrow from $F1$ to 1. Thus, the *projective* types are the greatest solutions to their corresponding RDEs; we write $\nu t.\,\sigma$ or νF for these.

The language $\lambda^{\mu\nu}$ studied in this chapter contains type constructors for the inductive and projective types as well as function spaces and finite sums and products. The class of functors for which we will be able to find fixed points is those which can be built from these constructors and which are strictly covariant in their argument (that is, the argument can only be used to the right of any function space constructors; the functor $FX = ((X \to A) \to B)$ is not allowed). All the functors in this restricted class are ω-(co)continuous over **Set**, the category of sets and total functions, so finding a model for $\lambda^{\mu\nu}$ will not require the machinery of cpo's that will be used in the next chapter.

We have already mentioned the work of Hagino above; his language CPL [21]

and the related typed λ-calculus with categorical type constructors [22] start from the very general concept of F, G-dialgebra and, through various computationally-imposed restrictions, result in a language with a type system equivalent to that of $\lambda^{\mu\nu}$, although with a more elegant (but less practical) notation. More recently, a language almost identical to $\lambda^{\mu\nu}$ has been studied by Greiner [19]; his primary concern is to show examples of types and programs that may be written in the language. Greiner also discusses the problem that arises when inductive and projective (which he calls *co-inductive*) types are simulated in a language such as Girard's system **F**, namely that the types are no longer extensional and so certain algorithms (most famously, the constant-time predecessor operation on the Church numerals) are not expressible. He makes the same observation which we make (independently) in Section 3.4, that a simple addition to our language allows us to express these algorithms with the desired efficiency.

3.1 Syntax of the language $\lambda^{\mu\nu}$

In this section we will describe the syntax for the types and terms of $\lambda^{\mu\nu}$. We will continue to use σ, τ, and υ as metavariables for type expressions, s and t for type variables, M and N for arbitrary terms, and x for (regular) variables. Type expressions may be any of the following:

$$t \mid 0 \mid 1 \mid \sigma + \tau \mid \sigma \times \tau \mid \sigma \rightarrow \tau \mid \mu t.\, \sigma \mid \nu t.\, \sigma.$$

The types of the language will be the closed type expressions, *i.e.*, those with no free type variables. Thus, the expression t is not a type on its own, but may be used only as a bound variable in the body of a μt or νt constructor. The type 0 is to be thought of as the empty type; it is the identity for the sum, that is, $0 + \sigma \cong \sigma + 0 \cong \sigma$ for any type σ. The type 1 has just one element; it is the identity for the product, $\times$. The unit type also serves as a left identity for the function space constructor, since $1 \rightarrow \sigma \cong \sigma$. We will follow the same grouping conventions as in the previous chapter, adding the convention that $+$ associates to the left and has a precedence between

that of $\times$ and that of $\rightarrow$; thus, the fully parenthesized form of $1 + s + t \times t \rightarrow s$ is $((1 + s) + (t \times t)) \rightarrow s$.

Not every type expression σ may be the body of an inductive (μ) or projective (ν) type — the bound type variable t may only occur *strictly positively* in σ, that is, it may not appear in any subterm on the left of an arrow. This is stronger than saying that σ must be covariant in t, which only requires that t always occur positively, *i.e.*, to the left of an even number of arrows. We will relax this restriction somewhat in the next chapter, but the relaxed condition adds nothing to the types we can define in $\lambda^{\mu\nu}$. What we are avoiding by demanding strict positivity are types such as $\mu t.\, ((t \rightarrow bool) \rightarrow bool)$, where $bool \equiv 1 + 1$; since our interpretation of a type in $\lambda^{\mu\nu}$ is a discrete cpo, *i.e.*, a set, there can be no solutions to such equations by cardinality considerations (the cardinality would be some κ such that $\kappa = 2^{2^\kappa}$, which is impossible in standard set theory).

As for PCF in the previous chapter, we give an inference system for typing assertions $\Gamma \triangleright M : \sigma$ for well-formed terms of $\lambda^{\mu\nu}$. We start with the rules for variables, which are the same as for PCF:

$$(var) \qquad x : \sigma \triangleright x : \sigma$$

$$(add\ var) \qquad \frac{\Gamma \triangleright M : \sigma}{\Gamma, x : \tau \triangleright M : \sigma.}$$

The categorical interpretation of a term $\Gamma \triangleright M : \sigma$ is an arrow from γ to σ, where γ is (the object corresponding to) the type $\sigma_1 \times \ldots \times \sigma_n$ if Γ is the type assignment $\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$. According to this interpretation, the (var) rule gives the identity arrow for σ. Composition of arrows is given by substitution — if $f : \gamma \rightarrow \sigma$ is the arrow represented by $\Gamma \triangleright M : \sigma$ and $g : \sigma \rightarrow \tau$ is the arrow represented by $x : \sigma \triangleright N : \tau$, then their composition $g \circ f : \gamma \rightarrow \tau$ is given by the term $\Gamma \triangleright \{M/x\}N : \tau$. The situation is slightly complicated because we may wish to have more than one free variable in N; thus, if we have $g' : \gamma' \times \sigma \rightarrow \tau$ corresponding to $\Gamma', x : \sigma \triangleright N : \tau$, then the substitution $\Gamma', \Gamma \triangleright \{M/x\}N : \tau$ represents the composite arrow $g' \circ (\gamma' \times f) : \gamma' \times \gamma \rightarrow \tau$. The $(add\ var)$ rule allows us to form the appropriate projections for this scheme of multiple free

variables to work, since if $\Gamma \triangleright M\colon \sigma$ is the arrow $f\colon \gamma \to \sigma$, then $\Gamma, x\colon \tau \triangleright M\colon \sigma$ is the composite $f \circ \pi_\gamma\colon \gamma \times \tau \to \sigma$, where $\pi_\gamma\colon \gamma \times \tau \to \gamma$ is the projection which drops the last component of type τ.

The constants and term constructors corresponding to the given type constructors also come from the categorical interpretation of the types. The type 0 is meant to represent an initial object, so there must be an arrow from 0 to any other type υ:

$$(0\ \textit{Elim}) \qquad \emptyset \triangleright \Box^{\upsilon}\colon 0 \to \upsilon.$$

The sum type $\sigma + \tau$ requires two injection arrows,

$$(+\ \textit{Intro}_1) \qquad \emptyset \triangleright \iota_1^{\sigma+\tau}\colon \sigma \to \sigma + \tau$$

$$(+\ \textit{Intro}_2) \qquad \emptyset \triangleright \iota_2^{\sigma+\tau}\colon \tau \to \sigma + \tau,$$

as well as, for every pair of arrows with a common target, an arrow from the sum of their sources into that target:

$$(+\ \textit{Elim}) \qquad \frac{\Gamma \triangleright M\colon \sigma \to \upsilon, \quad \Gamma \triangleright N\colon \tau \to \upsilon}{\Gamma \triangleright [M, N]\colon \sigma + \tau \to \upsilon.}$$

Dually, the type 1 represents a terminal object, and $\sigma \times \tau$ is a product, so we get the following axioms and rules:

$$(1\ \textit{Intro}) \qquad \Gamma \triangleright \Diamond\colon 1$$

$$(\times\ \textit{Elim}_1) \qquad \emptyset \triangleright \pi_1^{\sigma\times\tau}\colon \sigma \times \tau \to \sigma$$

$$(\times\ \textit{Elim}_2) \qquad \emptyset \triangleright \pi_2^{\sigma\times\tau}\colon \sigma \times \tau \to \tau$$

$$(\times\ \textit{Intro}) \qquad \frac{\Gamma \triangleright M\colon \sigma, \quad \Gamma \triangleright N\colon \tau}{\Gamma \triangleright \langle M, N\rangle\colon \sigma \times \tau.}$$

Note that in the introduction rules for 1 and $\times$ the source of the arrows is Γ, whereas the other constants and the elimination rules for 0 and $+$ are given in a

"curried" form, *i.e.*, the term created has a functional type corresponding to the desired arrow type. This is a purely stylistic decision; we could have written the $+$ elimination rule, for example, as

$$(+\ \mathit{Elim}') \qquad \frac{\Gamma, x{:}\,\sigma \triangleright M{:}\,\upsilon, \quad \Gamma, y{:}\,\tau \triangleright N{:}\,\upsilon}{\Gamma, z{:}\,\sigma+\tau \triangleright (\mathtt{case}\ \ z\ \ \mathtt{of}\ \ \iota_1^{\sigma+\tau} x.\, M,\ \iota_2^{\sigma+\tau} y.\, N){:}\,\upsilon;}$$

the disadvantage of this approach is that we would have two basic constructions which bind variables, which adds to the complexity of the system. Conversely, we could have curried the 1 introduction rule as

$$(1\ \mathit{Intro}') \qquad \emptyset \triangleright \mathtt{unit}^{\upsilon}{:}\,\upsilon{\to}1,$$

but then we would have no simple expression for a closed term of type 1 (the simplest would be something like $\mathtt{unit}^{1\to 1}\,\mathtt{unit}^{1}$). Finally, we write the other constants with empty type assignments, which makes them arrows from 1, that is, *elements* of the objects corresponding to their types; by using the (*add var*) rule we may construct constant terms for any desired type assignment Γ. We could have taken this approach for $\Diamond$, but then its status as the unique arrow from a given type to the terminal object would have been obscured. All of these design decisions are for the most part non-issues because we are intending to work in a system where all of the extensional rules hold, hence all of the necessary isomorphisms exist.

Continuing with the syntax of the language, we have the usual λ-abstraction and application rules:

$$(\to\ \mathit{Intro}) \qquad \frac{\Gamma, x{:}\,\sigma \triangleright M{:}\,\tau}{\Gamma \triangleright (\lambda x{:}\,\sigma.\, M){:}\,\sigma{\to}\tau}$$

$$(\to\ \mathit{Elim}) \qquad \frac{\Gamma \triangleright M{:}\,\sigma{\to}\tau, \quad \Gamma \triangleright N{:}\,\sigma}{\Gamma \triangleright MN{:}\,\tau.}$$

In light of the above discussion of the categorical basis for the syntax, we observe that abstraction is the process by which an arrow may be "curried" on one of its

free variables to obtain an *internal* arrow, *i.e.*, an element of a functional type. Application then provides the means of composing these internal arrows with other terms; also, a combination of the variable rules and application has the effect of "uncurrying" a function, *e.g.*, $x{:}\,0 \triangleright \Box^{\upsilon} x{:}\,\upsilon$ is the uncurried form of the (0 *Elim*) axiom.

Finally, we have the terms associated with the inductive and projective types. The categorical interpretation of an inductive type μF ($\equiv \mu t.\,\sigma$) is an initial F-algebra, *i.e.*, an arrow φ from $F(\mu F)$ to μF such that for any other F-algebra $f{:}\,F(X){\rightarrow}X$ there is a unique F-algebra morphism $h_f{:}\,\varphi{\rightarrow}f$, that is, an arrow $h_f{:}\,\mu F{\rightarrow}X$ such that the following diagram commutes:

$$\begin{array}{ccc} F(\mu F) & \xrightarrow{F(h_f)} & F(X) \\ \varphi \downarrow & & \downarrow f \\ \mu F & \xrightarrow[h_f]{} & X \end{array} \quad .$$

In terms of their actions on values of a recursive type, the function φ "folds" up the outermost level of a recursively defined value, while the function h_f takes a value of the type and iteratively applies f to each of its levels; this will become clearer when we discuss the equations and give some examples. For now, we simply introduce the following constants corresponding to φ and h_f (actually, $it^{\tau}_{\mu F}$ corresponds to the function h which takes an arbitrary F-algebra f and produces the appropriate morphism h_f):

(μ *Intro*) $\qquad\qquad \emptyset \triangleright \mathit{fold}_{\mu F}{:}\,F(\mu F){\rightarrow}\mu F$

(μ *Elim*) $\qquad\qquad \emptyset \triangleright it^{\tau}_{\mu F}{:}\,(F(\tau){\rightarrow}\tau){\rightarrow}\mu F{\rightarrow}\tau.$

Recall that we are using $F(\tau)$ as an abbreviation for $\{\tau/t\}\sigma$, since we are treating the type expression σ with free type variable t as an endofunctor F; the unabbreviated

form of the (μ *Elim*) axiom would be

$$\emptyset \triangleright it^{\tau}_{\mu t.\,\sigma}\colon (\{\tau/t\}\sigma \to \tau) \to \mu t.\,\sigma \to \tau,$$

which is harder to read.

We have the dual situation for projective types. The interpretation of νF is a terminal F-coalgebra, *i.e.*, an arrow $\psi\colon \nu F \to F(\nu F)$ such that for every F-coalgebra $g\colon Y \to F(Y)$ there is a unique arrow $k_g\colon Y \to \nu F$ such that

$$\begin{array}{ccc} Y & \xrightarrow{k_g} & \nu F \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle \psi} \\ F(Y) & \xrightarrow[F(k_g)]{} & F(\nu F) \end{array}$$

commutes. In this case, the action of the function ψ is to "unfold" a value of a recursive type by one level, and the function k_g creates a new value of the recursive type by bundling together the function g with a starting value from Y; again, we will give examples after discussing the equations. The constants corresponding to ψ and k are

(ν *Elim*) $$\emptyset \triangleright unfold_{\nu F}\colon \nu F \to F(\nu F)$$

(ν *Intro*) $$\emptyset \triangleright new^{\tau}_{\nu F}\colon (\tau \to F(\tau)) \to \tau \to \nu F.$$

There are a number of abbreviations, or *syntactic sugar*, which we will use in the following sections. The first is simply that we will omit sub- and superscripted type information whenever it can reasonably be inferred. Likewise, we will often leave implicit a description of the typing context Γ, and assume that one may be found that allows the given term to be proven to have the desired type. For example, when we say that we will write $M \circ N\colon \sigma \to \upsilon$ as an abbreviation for $(\lambda x\colon \sigma.\, M(Nx))$, where M has type $\tau \to \upsilon$, N has type $\sigma \to \tau$, and x does not occur free in either M or N, this assumes that M and N have the given types with respect to some common type

assignment. If M has type $\sigma \to \sigma$, then we may write $M^{(n)}$ for the n-fold composition $(\ldots(M \circ M) \circ \ldots \circ M)$; when $n = 0$, this becomes the identity $id^\sigma \equiv (\lambda x{:}\sigma.\, x)$.

We have already been using the notation $\{M/x\}N$ to indicate the result of substituting the term M for the free variable x in N, and similarly $\{\sigma/t\}\tau$ for substitution in type expressions. We will not formalize this more here, except to note that λ is the only variable binding operator for terms, and μ and ν are the only binding operators for type expressions. All variables not in the body of an eponymous binding operation are free; the set of free variables in a term M is denoted $\mathrm{FV}(M)$. As for PCF, we have the following useful lemmas:

Lemma 3.1.1 *If $\Gamma \triangleright M{:}\sigma$ is well-formed, then every $x \in \mathrm{FV}(M)$ appears in Γ.*

Lemma 3.1.2 *If $\Gamma \triangleright M{:}\sigma$ is well-formed, then so is $\Gamma' \triangleright M{:}\sigma$ for every $\Gamma' \subset \Gamma$ which contains all the free variables in M.*

Lemma 3.1.3 *If $\Gamma \triangleright M{:}\sigma$ and $\Gamma, x{:}\sigma \triangleright N{:}\tau$ are well-formed, then so is $\Gamma \triangleright \{M/x\}N{:}\tau$.*

We may form arbitrary finite products and sums as follows. The nullary product is $\Diamond{:}\,1$ and the nullary sum is $\Box^\upsilon{:}\,0 \to \upsilon$. Given terms $M_1{:}\,\sigma_1, \ldots, M_n{:}\,\sigma_n$, for $n > 1$, we may form the n-ary product $\langle M_1, \ldots, M_n\rangle{:}\,\sigma_1 \times \ldots \times \sigma_n$ as an abbreviation for $\langle\ldots\langle M_1, M_2\rangle, \ldots, M_n\rangle$ (remembering that $\times$ associates to the left); in the special case $n = 1$ we may write $\langle M_1\rangle{:}\,\sigma_1$ as an "abbreviation" for M_1. The mth projection of an n-ary product, $1 \le m \le n$, may be found by applying the constant π^n_m defined as follows:

$$\begin{aligned} \pi^1_1 &\equiv id \\ \pi^n_{m+1} &\equiv \pi^n_m \circ \pi_1 \quad (n \le m) \\ \pi^{m+1}_{m+1} &\equiv \pi_2. \end{aligned}$$

Similar definitions hold for the n-ary sum, if each of the types σ_k is of the form $\tau_k \to \upsilon$ for some fixed type υ. That is, we get the term $[M_1, \ldots, M_n]{:}\,\tau_1 + \ldots + \tau_n \to \upsilon$ as an abbreviation for $[\ldots[M_1, M_2], \ldots, M_n]$, with the special case $[M_1] \equiv M_1$. The mth

injection ι^n_m into an n-ary sum is given by

$$\begin{aligned}
&\iota^1_1 \equiv id \\
&\iota^n_{m+1} \equiv \iota_1 \circ \iota^n_m \quad (n \leq m) \\
&\iota^{m+1}_{m+1} \equiv \iota_2.
\end{aligned}$$

The product constructors allow us to define a pattern-matching version of λ-abstraction. Given n variables $x_1, \ldots, x_n$ and n types $\sigma_1, \ldots, \sigma_n$, for $n \geq 0$, we write $(\lambda\langle x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n\rangle.\, M)$ as an abbreviation for

$$(\lambda p\colon \sigma_1 \times \ldots \times \sigma_n.\, \{\pi^n_1 p/x_1\} \ldots \{\pi^n_n p/x_n\} M),$$

where p is not free in M. We will add other kinds of patterns eventually, so let us speak generally of binding a pattern $\mathcal{P}$ in M with the syntax $(\lambda\mathcal{P}.\, M)$. Note that we may generalize the product pattern to have an arbitrary pattern for each component, instead of just the basic pattern $x\colon \sigma$.

Another useful piece of syntactic sugar is the `let` statement. We will write `let` $\mathcal{P} = M$ `in` N for the compound term $(\lambda\mathcal{P}.\, N)M$, where $\mathcal{P}$ is a pattern as above. In the case that M is of the form $(\lambda\mathcal{Q}.\, M')$, in which case $\mathcal{P}$ must look like $f\colon \sigma \to \tau$, we will also write this as `let` $f(\mathcal{Q})\colon \tau = M'$ `in` N. As an example, the term `let` $car\langle x\colon nat, y\colon bool\rangle\colon nat = x$ `in` $\ldots$ is an abbreviation for

$$\texttt{let}\ \ car\colon nat \times bool \to nat = (\lambda\langle x\colon nat, y\colon bool\rangle.\, x)\ \ \texttt{in}\ \ \ldots,$$

which is itself an abbreviation for (after several substitutions)

$$(\lambda car\colon nat \times bool \to nat.\ \ldots)(\lambda p\colon nat \times bool.\, \pi_1 p).$$

We may use a similar "in-line" form for the n-ary sum, namely the `case` statement mentioned above in a slightly different form (which we did not want to add as a basic piece of syntax, but which is quite useful as auxiliary syntax). Thus, the term

$[(\lambda\mathcal{P}_1.\, N_1), \ldots, (\lambda\mathcal{P}_n.\, N_n)]M$ may be written out as

$$\texttt{case}\ M\ \texttt{of}\ \iota_1^n\mathcal{P}_1.\, N_1, \ldots, \iota_n^n\mathcal{P}_n.\, N_n.$$

As an example of the use of several of these abbreviations, we introduce the type of booleans, defined as $\mathit{bool} \equiv 1 + 1$. There are two values of this type, namely $\iota_1\Diamond$ and $\iota_2\Diamond$; we will refer to them respectively as *true* and *false*. A conditional similar to the term $\texttt{if}\ B\ \texttt{then}\ M\ \texttt{else}\ N$ of the previous chapter is then expressible as $\texttt{case}\ B\ \texttt{of}\ \mathit{true}.\, M,\ \mathit{false}.\, N$, which is certainly more readable than the "unsugared" version, $[(\lambda x{:}\,1.\, M), (\lambda x{:}\,1.\, N)]B$.

We have mentioned several times the interpretation of substitution in a type expression as being the application of a functor to an object; we now describe the effect of applying a functor to an arrow. For our purposes, it will be more useful to apply a functor to an internal arrow, that is, a term of function type; because our intended model is a cartesian closed category, *i.e.*, we have extensional finite products and function spaces, this is entirely equivalent to defining functor application on arbitrary terms and their typing contexts. Thus, given a functor F and a term $M{:}\,\sigma{\to}\tau$, we will produce a term $F(M){:}\,F(\sigma){\to}F(\tau)$. In addition, after the equations are introduced in the next section, we will be able to prove that $F(\mathit{id}) = \mathit{id}$ and $F(M \circ N) = F(M) \circ F(N)$, completely justifying our referring to F as a functor. The basic idea for this comes from Hagino [21, 22], although his presentation is somewhat more difficult to read and treats only strictly positive functors. The definition of $F(M)$ will proceed by cases on the structure of $F(t)$, where t is a fresh type variable:

- if $F(t) = \upsilon$, where t does not occur free in υ, then $F(M) = \mathit{id}^{\upsilon}$;
- if $F(t) = t$, then $F(M) = M$;
- if $F(t) = G(t) + H(t)$, then $F(M) = [\iota_1 \circ G(M), \iota_2 \circ H(M)]$;
- if $F(t) = G(t) \times H(t)$, then $F(M) = \lambda\langle x{:}\,G(\sigma), y{:}\,H(\sigma)\rangle.\, \langle G(M)x, H(M)y\rangle$;
- if $F(t) = G(t){\to}H(t)$, then $F(M) = \lambda f{:}\,G(\sigma){\to}H(\sigma).\, H(M) \circ f \circ G(M^{op})$ (see below);

- if $F(t) = \mu s.\, G(s,t)$, then $F(M) = it^{F(\tau)}_{\mu s.G(s,\sigma)}(\mathit{fold}_{\mu s.G(s,\tau)} \circ G(F(\tau), M))$;
- if $F(t) = \nu s.\, G(s,t)$, then $F(M) = \mathit{new}^{F(\sigma)}_{\nu s.G(s,\tau)}(G(F(\sigma), M) \circ \mathit{unfold}_{\nu s.G(s,\sigma)})$.

For the system as defined in this chapter, F will always be strictly positive, so the type expression $G(t)$ will not depend on t in the case $F(t) = G(t) \to H(t)$ above. For purposes of the next chapter, however, we include the mechanism for dealing with arbitrary covariant functors F. The notation M^{op} is meant to indicate a term of type $\tau \to \sigma$, the *opposite* of $M\colon \sigma \to \tau$. Since in general we have no way of forming an opposite term, the only rules we have are that opposite is an *anti-involution*, *i.e.*, the opposite of the opposite is the original term: $(M^{op})^{op} \equiv M$, and opposite is contravariant with respect to composition: $(M \circ N)^{op} \equiv N^{op} \circ M^{op}$ (from which it is easy to prove that also $\mathit{id}^{op} \equiv \mathit{id}$). In forming the subterm $G(M^{op})$, if G is contravariant then by using this rule we will only ever need M; if M^{op} appears in the fully expanded term $F(M)$, then F must not have been covariant.

Note that in the definition of $F(M)$ when F is a recursive type, we treat the body of F as a functor G with *two* arguments, reflecting the fact that we are keeping track of two free type variables — the variable t for which we are "substituting" M, and the variable s which is bound by the recursive constructor in F. In fact, we may treat a type expression σ with n free type variables as an n-argument functor F; the application $F(\tau_1, \ldots, \tau_n)$ represents the simultaneous substitution of the n type expressions for the corresponding variables, and the application to terms is defined essentially as above. We will not go through the details of this extension here, as they will only be used once in the sequel (in the proof that F preserves composition, for the case when F is a recursive type).

Recall that the intended meaning of μF and νF is that they are recursive types, that is, that $\mu F \cong F(\mu F)$ and $\nu F \cong F(\nu F)$, but so far we have only seen arrows giving one direction for each of these isomorphisms. With the aid of the equational proof system given in the next section, we may see that an inverse to $\mathit{fold}_{\mu F}$ is provided by the term $it^{F(\mu F)}_{\mu F}\, F(\mathit{fold}_{\mu F})\colon \mu F \to F(\mu F)$. We will refer to this term as $\mathit{unfold}_{\mu F}$. Similarly, an inverse to $\mathit{unfold}_{\nu F}$ is given by the term $\mathit{new}^{F(\nu F)}_{\nu F}\, F(\mathit{unfold}_{\nu F})\colon F(\nu F) \to \nu F$, which we call $\mathit{fold}_{\nu F}$. With these inverse terms we may define another kind of pattern

matching — if we write $(\lambda\, fold_{\mu F}(x\colon F(\mu F)).\, M)$, then that will be understood as syntactic sugar for $(\lambda y\colon \mu F.\, \{unfold_{\mu F}\, y/x\}M)$, and similarly for νF (just change the μ's to ν's). Unfortunately, when we consider the reduction system which provides the operational semantics for $\lambda^{\mu\nu}$, we will find that these inverses can not be computed in constant time; instead, they take time proportional to the size of their argument. The impact of this is that it would *not* be merely syntactic sugar to add a reduction rule such as

$$(\lambda\, fold_{\mu F}(x\colon F(\mu F)).\, M)(fold_{\mu F}\, N) \longrightarrow \{N/x\}M,$$

since it conflates an unknown number of reduction steps into one. This is in contrast to the other pattern matching abbreviations; for example, it makes no essential difference in running time to use the following reduction rules for the conditional statement introduced above:

$$\texttt{case}\ \mathit{true}\ \texttt{of}\ \mathit{true}.\, M,\ \mathit{false}.\, N \longrightarrow M$$
$$\texttt{case}\ \mathit{false}\ \texttt{of}\ \mathit{true}.\, M,\ \mathit{false}.\, N \longrightarrow N.$$

One solution to this is to simply add $unfold_{\mu F}$ and $fold_{\nu F}$ as constants, with an extra (one step) reduction rule $unfold(fold\, M) \longrightarrow M$ for each pair of constants; this will be discussed further below.

3.2 Equational proof system for $\lambda^{\mu\nu}$

Now that we have introduced the syntax of $\lambda^{\mu\nu}$, it is time to give a formal semantics for the language, in the form of a set of equational axioms and proof rules very similar to those given for PCF in the previous chapter. The categorical interpretation will continue to be our guide in describing the equations that hold between terms. Thus, the equation $\Gamma \triangleright M = N : \sigma$ will be interpreted as saying that the arrows corresponding to the terms $\Gamma \triangleright M\colon \sigma$ and $\Gamma \triangleright N\colon \sigma$, which both have source γ and target σ, are equal. We start by listing the usual structural rules necessary to have an equivalence relation

that respects term formation, including adding and renaming variables:

$$(ref) \qquad \Gamma \triangleright M = M : \sigma$$

$$(sym) \qquad \frac{\Gamma \triangleright M = N : \sigma}{\Gamma \triangleright N = M : \sigma}$$

$$(trans) \qquad \frac{\Gamma \triangleright M = N : \sigma, \quad \Gamma \triangleright N = P : \sigma}{\Gamma \triangleright M = P : \sigma}$$

$$(abs) \qquad \frac{\Gamma, x{:}\,\sigma \triangleright M = N : \tau}{\Gamma \triangleright (\lambda x{:}\,\sigma.\, M) = (\lambda x{:}\,\sigma.\, N) : \sigma{\to}\tau}$$

$$(app) \qquad \frac{\Gamma \triangleright M = N : \sigma{\to}\tau, \quad \Gamma \triangleright P = Q : \sigma}{\Gamma \triangleright MP = NQ : \tau}$$

$$(add\ var) \qquad \frac{\Gamma \triangleright M = N : \sigma}{\Gamma, x{:}\,\tau \triangleright M = N : \sigma}$$

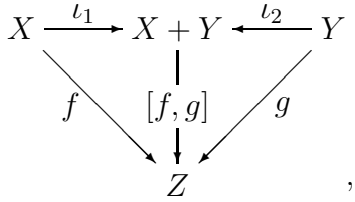
$$(\alpha) \qquad \Gamma \triangleright (\lambda x{:}\,\sigma.\, M) = (\lambda y{:}\,\sigma.\, \{y/x\}M) : \sigma{\to}\tau, \text{ if } y \notin \mathrm{FV}(M).$$

The rules (abs) and (app), often referred to as (ξ) and (μ), together with the $(\to\beta)$ axiom below are sufficient to establish that $=$ is a congruence, since we may prove the following substitution lemma:

Lemma 3.2.1 (Substitution) *If* $\Gamma, x{:}\,\sigma \triangleright M = N : \tau$ *and* $\Gamma \triangleright P = Q : \sigma$ *are provable, then so is* $\Gamma \triangleright \{P/x\}M = \{Q/x\}N : \tau$.

Proof. Using (abs) on the first equation yields $\Gamma \triangleright (\lambda x{:}\,\sigma.\, M) = (\lambda x{:}\,\sigma.\, N) : \sigma{\to}\tau$; by (app) we may combine this with the second equation to get $\Gamma \triangleright (\lambda x{:}\,\sigma.\, M)P = (\lambda x{:}\,\sigma.\, N)Q : \tau$. From $(\to\beta)$ we find that $\Gamma \triangleright (\lambda x{:}\,\sigma.\, M)P = \{P/x\}M : \tau$ and $\Gamma \triangleright (\lambda x{:}\,\sigma.\, N)Q = \{Q/x\}N : \tau$, hence an application of (sym) and two instances of $(trans)$ establish that $\Gamma \triangleright \{P/x\}M = \{Q/x\}N : \tau$. (We will not ordinarily go through proofs in such detail; the purpose here was to demonstrate that the rules are sufficient to carry out complete proofs if desired.) ∎

The rest of the axioms and inference rules come in two forms; in terms of the categorical interpretations of the type constructors, the (β) axioms state that the arrows provided for the type make a particular diagram commute, while the (η) axioms and rules establish that those arrows do so uniquely. For example, the categorical definition of the sum states that for any two objects X and Y there is an object $X + Y$, injection arrows $\iota_1 \colon X \to X + Y$ and $\iota_2 \colon Y \to X + Y$, and a sum arrow $[f, g] \colon X + Y \to Z$ for every pair of arrows $f \colon X \to Z$ and $g \colon Y \to Z$. The (β) axioms will assert that these arrows make the following diagram commute:



,

while the (η) axiom says that $[f, g]$ is the unique arrow with this property. Thus, we have the following axioms for $+$:

($+\beta_1$) $$\Gamma \rhd [M, N] \circ \iota_1 = M : \sigma \to \upsilon$$

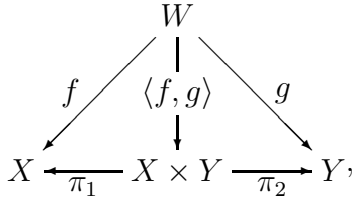
($+\beta_2$) $$\Gamma \rhd [M, N] \circ \iota_2 = N : \tau \to \upsilon$$

($+\eta$) $$\Gamma \rhd [M \circ \iota_1, M \circ \iota_2] = M : \sigma + \tau \to \upsilon.$$

Since 0 is a nullary sum, we only have an (η) rule, which says that $\Box^\upsilon$ is the unique arrow from 0 to υ:

(0η) $$\Gamma \rhd \Box^\upsilon = M : 0 \to \upsilon.$$

Similarly, the (β) axioms for $\times$ assert the commutativity of



,

and the (η) axiom says that $\langle f, g\rangle$ is unique; the (η) axiom for 1 says that $\diamond$ is the unique term of type 1:

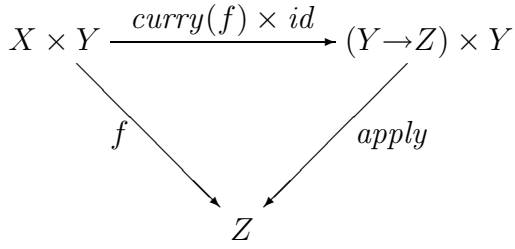
$$(\times\beta_1) \qquad \Gamma \triangleright \pi_1\langle M, N\rangle = M : \sigma$$

$$(\times\beta_2) \qquad \Gamma \triangleright \pi_2\langle M, N\rangle = N : \tau$$

$$(\times\eta) \qquad \Gamma \triangleright \langle \pi_1 M, \pi_2 M\rangle = M : \sigma \times \tau$$

$$(1\eta) \qquad \Gamma \triangleright \diamond = M : 1.$$

The situation for $\rightarrow$ is somewhat more complex, since the appropriate diagram results not simply from a limit or colimit, but instead from the adjunction between $- \times Y$ and $Y{\rightarrow}-$; nevertheless, it is a standard result that the commutativity of



is expressed by

$$(\rightarrow\beta) \qquad \Gamma \triangleright (\lambda x{:}\sigma.\, M)N = \{N/x\}M : \tau,$$

while the uniqueness of $curry(f)$ is given by

$$(\rightarrow\eta) \qquad \Gamma \triangleright (\lambda x{:}\sigma.\, Mx) = M : \sigma{\rightarrow}\tau, \text{ for } x \notin \mathrm{FV}(M).$$

Recall that we have already described the commutative diagrams corresponding to the inductive and projective types. For μF, the commutativity of

$$\begin{array}{ccc} F(\mu F) & \xrightarrow{F(h_f)} & F(X) \\ {\scriptstyle\varphi}\downarrow & & \downarrow{\scriptstyle f} \\ \mu F & \xrightarrow[h_f]{} & X \end{array}$$

is given by

$$(\mu\beta) \qquad \Gamma \rhd (it^{\tau}_{\mu F}\, M) \circ fold_{\mu F} = M \circ F(it^{\tau}_{\mu F}\, M) : F(\mu F) \rightarrow \tau,$$

while the uniqueness of h_f is established by

$$(\mu\eta) \qquad \frac{\Gamma \rhd N \circ fold_{\mu F} = M \circ F(N) : F(\mu F) \rightarrow \tau}{\Gamma \rhd N = it^{\tau}_{\mu F}\, M : \mu F \rightarrow \tau.}$$

Dually, the diagram

$$\begin{array}{ccc} Y & \xrightarrow{k_g} & \nu F \\ {\scriptstyle g}\downarrow & & \downarrow{\scriptstyle \psi} \\ F(Y) & \xrightarrow[F(k_g)]{} & F(\nu F) \end{array}$$

for the projective type νF leads to

$$(\nu\beta) \qquad \Gamma \rhd unfold_{\nu F} \circ (new^{\tau}_{\nu F}\, M) = F(new^{\tau}_{\nu F}\, M) \circ M : \tau \rightarrow F(\nu F)$$

and

$$(\nu\eta) \qquad \frac{\Gamma \rhd unfold_{\nu F} \circ N = F(N) \circ M : \tau \rightarrow F(\nu F)}{\Gamma \rhd N = new^{\tau}_{\nu F}\, M : \tau \rightarrow \nu F.}$$

As promised in the previous section, we may now prove some lemmas about the behavior of our abbreviated terms.

Lemma 3.2.2 *Composition is associative, and id is an identity, i.e.,* $M \circ (N \circ P) = (M \circ N) \circ P$ *and* $M \circ id = id \circ M = M$.

Proof. Expanding $M \circ (N \circ P)$ yields (assuming P has type $\sigma \to \tau$)

$$(\lambda x{:}\,\sigma.\, M((\lambda y{:}\,\sigma.\, N(Py))x)),$$

which by $(\to\beta)$ is equal to $(\lambda x{:}\,\sigma.\, M(N(Px)))$; this is also equal to

$$(\lambda x{:}\,\sigma.\, (\lambda y{:}\,\tau.\, M(Ny))(Px)),$$

which is the expanded form of $(M \circ N) \circ P$. As for the identity, both of the terms $M \circ id$ and $id \circ M$ are equal by $(\to\beta)$ to $(\lambda x{:}\,\sigma.\, Mx)$; using $(\to\eta)$, this is equal to M, since in the definition of $\circ$ we required that x be a new variable. ∎

Lemma 3.2.3 *Application of a "functor" F to a term preserves composition and identities, i.e.,* $F(M \circ N) = F(M) \circ F(N)$ *and* $F(id) = id$ *(thus justifying use of the term functor for substitution in a type).*

Proof. By induction on the structure of F; we will only show a few of the more interesting cases. For the induction to go through, we actually need to prove more — namely, that this result can be extended to functors with more than one argument, *e.g.*, $G(M \circ N, P \circ Q) = G(M, P) \circ G(N, Q)$; this extension is quite easy and giving the full details would not add enough to the presentation to be worth the extra notation. We assume that $\Gamma \rhd M{:}\,\tau \to \upsilon$ and $\Gamma \rhd N{:}\,\sigma \to \tau$ are well-formed for some Γ.

- if $F(t) = G(t) + H(t)$, then

$$\begin{aligned} F(M) \circ F(N) &= [F(M) \circ F(N) \circ \iota_1, F(M) \circ F(N) \circ \iota_2] \\ &= [F(M) \circ \iota_1 \circ G(N), F(M) \circ \iota_2 \circ H(N)] \\ &= [\iota_1 \circ G(M) \circ G(N), \iota_2 \circ H(M) \circ H(N)] \end{aligned}$$

$$\begin{aligned} &= [\iota_1 \circ G(M \circ N), \iota_2 \circ H(M \circ N)] \\ &= F(M \circ N); \end{aligned}$$

also, $F(id^\sigma) = [\iota_1, \iota_2] = id^{F(\sigma)}$.

- if $F(t) = G(t) \rightarrow H(t)$, then

$$\begin{aligned} F(M \circ N) &= \lambda f\colon F(\sigma).\, H(M \circ N) \circ f \circ G((M \circ N)^{op}) \\ &= \lambda f\colon F(\sigma).\, H(M) \circ H(N) \circ f \circ G(N^{op}) \circ G(M^{op}) \\ &= \lambda f\colon F(\sigma).\, H(M) \circ (F(N)f) \circ G(M^{op}) \\ &= \lambda f\colon F(\sigma).\, F(M)(F(N)f) \\ &= F(M) \circ F(N); \end{aligned}$$

for the identity, we have

$$\begin{aligned} F(id^\sigma) &= \lambda f\colon F(\sigma).\, H(id^\sigma) \circ f \circ G((id^\sigma)^{op}) \\ &= \lambda f\colon F(\sigma).\, id^{H(\sigma)} \circ f \circ id^{G(\sigma)} \\ &= id^{F(\sigma)}\,. \end{aligned}$$

- if $F(t) = \mu s.\, G(s,t)$, then

$$\begin{aligned} &F(M) \circ F(N) \circ fold_{F(\sigma)} \\ &= F(M) \circ fold_{F(\tau)} \circ G(F(\tau), N) \circ G(F(N), \sigma) \\ &= fold_{F(\upsilon)} \circ G(F(\upsilon), M) \circ G(F(M), \tau) \circ G(F(\tau), N) \circ G(F(N), \sigma) \\ &= fold_{F(\upsilon)} \circ G(F(\upsilon), M) \circ G(F(\upsilon), N) \circ G(F(M), \sigma) \circ G(F(N), \sigma) \\ &= fold_{F(\upsilon)} \circ G(F(\upsilon), M \circ N) \circ G(F(M) \circ F(N), \sigma), \end{aligned}$$

where the interchange $G(F(M), \tau) \circ G(F(\tau), N) = G(F(\upsilon), N) \circ G(F(M), \sigma)$ in the middle is possible because both sides are equal to $G(F(M), N)$, noticing that, *e.g.*, $G(F(M), \tau) = G(F(M), id^{F(\tau)})$ and using the multiple argument form of the induction hypothesis. From the above we may then use $(\mu\eta)$ to deduce that $F(M) \circ F(N) = it^{F(\upsilon)}_{F(\sigma)}(fold_{F(\upsilon)} \circ G(F(\upsilon), M \circ N)) = F(M \circ N)$.

For the identity, we must show that $F(id^\sigma) = it^{F(\sigma)}_{F(\sigma)} fold_{F(\sigma)}$ is the identity; but $id^{F(\sigma)} \circ fold_{F(\sigma)} = fold_{F(\sigma)} \circ G(id^{F(\sigma)}, \sigma)$ by the induction hypothesis, so $id^{F(\sigma)} = it^{F(\sigma)}_{F(\sigma)} fold_{F(\sigma)}$ by $(\mu\eta)$.

■

We may now use this lemma about functors to prove that the defined terms for $unfold_{\mu F}$ and $fold_{\nu F}$ are really inverses.

Lemma 3.2.4 *The term $unfold_{\mu F} \equiv it^{F(\mu F)}_{\mu F} F(fold_{\mu F})$ is an inverse for $fold_{\mu F}$, i.e., $fold_{\mu F} \circ unfold_{\mu F} = id^{\mu F}$ and $unfold_{\mu F} \circ fold_{\mu F} = id^{F(\mu F)}$. Similarly, $fold_{\nu F} \equiv new^{F(\nu F)}_{\nu F} F(unfold_{\nu F})$ is an inverse for $unfold_{\nu F}$.*

Proof. For variety, we will show the proof for the projective type νF; the inductive case is quite similar. First, note that $unfold_{\nu F} \circ fold_{\nu F} = F(fold_{\nu F}) \circ F(unfold_{\nu F}) = F(fold_{\nu F} \circ unfold_{\nu F})$ by $(\nu\beta)$ and the previous lemma; thus we only need to show the direction $fold_{\nu F} \circ unfold_{\nu F} = id^{\nu F}$. From what we have just shown, we know that $unfold_{\nu F} \circ fold_{\nu F} \circ unfold_{\nu F} = F(fold_{\nu F} \circ unfold_{\nu F}) \circ unfold_{\nu F}$, hence the $(\nu\eta)$ rule allows us to derive $fold_{\nu F} \circ unfold_{\nu F} = new^{\nu F}_{\nu F}\, unfold_{\nu F}$. This last term is the identity, by using $(\nu\eta)$ on the equation $unfold_{\nu F} \circ id^{\nu F} = F(id^{\nu F}) \circ unfold_{\nu F}$, so we are done. ■

3.3 The reduction system $\lambda^{\mu\nu}_r$

In this section we present an operational semantics for $\lambda^{\mu\nu}$, in the form of a set of reduction rules similar to those given for PCF in the previous chapter. We will show that this system is confluent and strongly normalizing, hence it will be meaningful to speak of the (unique) result of evaluating a term by reducing it to normal form. We then study the set of definable functions and find that this set of functions properly includes those that may be proven total in Peano arithmetic, hence we have a system which is more expressive than Gödel's System **T** of primitive recursive functionals.

The reduction rules we will take for $\lambda^{\mu\nu}_r$ are essentially the (β) axioms of the previous section, oriented in the direction of "computation." Since we do not include any (η) rules, the form of some of the axioms will be changed to better match our

notion of normal form — instead of dealing with functional terms and composition, we will apply such terms to dummy arguments to get rid of the $\circ$'s. Here are the reduction rules:

$$(+\beta_1)_r \qquad [M,N](\iota_1 P) \longrightarrow MP$$

$$(+\beta_2)_r \qquad [M,N](\iota_2 P) \longrightarrow NP$$

$$(\times\beta_1)_r \qquad \pi_1\langle M,N\rangle \longrightarrow M$$

$$(\times\beta_2)_r \qquad \pi_2\langle M,N\rangle \longrightarrow N$$

$$(\to\beta)_r \qquad (\lambda x{:}\,\sigma.\, M)N \longrightarrow \{N/x\}M$$

$$(\mu\beta)_r \qquad it^{\tau}_{\mu F}\, M(fold_{\mu F}\, P) \longrightarrow M(F(it^{\tau}_{\mu F}\, M)P)$$

$$(\nu\beta)_r \qquad unfold_{\nu F}(new^{\tau}_{\nu F}\, MP) \longrightarrow F(new^{\tau}_{\nu F}\, M)(MP).$$

As usual, we write $M \longrightarrow N$ if there is some subterm P of M (*i.e.*, $M \equiv \mathcal{C}[P]$ for some context $\mathcal{C}[\ ]$) and substitution θ such that $P \equiv \theta L$ and $N \equiv \mathcal{C}[\theta R]$, where $L \longrightarrow R$ is one of the above rules. The subterm P in M which is replaced is the *redex*, the term θR which replaces it is the *contractum*, and the result N is the *reduct*. A term which contains no redexes is a *normal form*. We write $\longrightarrow\!\!\!\!\!\rightarrow$ for the reflexive transitive closure of $\longrightarrow$; if we only want the transitive closure, which means that we must perform at least one reduction step, then we write $\longrightarrow^{+}$. Note that, in this section, we will use $\equiv$ to mean α-equivalent, rather than strictly identical — two different reductions from a given term may result in different names for "corresponding" bound variables, so for confluence we need to be able to rename them; however, using (α) as a reduction rule is a bad idea, since it would need special handling to preserve strong normalization (it does not have the same intuitive feeling of "performing a computation step" as the (β) rules, either).

We have left off the types and typing contexts in defining reduction, but it is an easy lemma to prove that reduction preserves type:

Lemma 3.3.1 (Subject Reduction) *If $\Gamma \triangleright M{:}\,\sigma$ is well-formed and $M \longrightarrow\!\!\!\!\!\rightarrow N$, then $\Gamma \triangleright N{:}\,\sigma$ is also well-formed.*

The first important theorems of this section are that $\lambda_r^{\mu\nu}$ is confluent and strongly normalizing. As a result, we will be able to speak of the unique normal form $\lambda_r^{\mu\nu}(M)$ of an arbitrary term M, independent of any specific reduction strategy — one strategy may be more efficient than another at finding the normal form of a given term, but every reduction path will eventually terminate with the same result. We will prove strong normalization first, as this result is used in the proof of confluence.

Theorem 3.3.2 (Strong Normalization) *There is no infinite sequence of reductions* $M_1 \longrightarrow M_2 \longrightarrow M_3 \longrightarrow \cdots$.

Proof. In section 3.5 we give a translation of $\lambda^{\mu\nu}$ into Girard's System **F**, with the property that if $M \longrightarrow N$ in $\lambda_r^{\mu\nu}$ then $\overline{M} \longrightarrow^+ \overline{N}$ in **F**; since **F** is strongly normalizing [17], this proves that $\lambda_r^{\mu\nu}$ is also, since otherwise we would be able to construct the infinite reduction sequence $\overline{M_1} \longrightarrow^+ \overline{M_2} \longrightarrow^+ \overline{M_3} \longrightarrow^+ \cdots$ in **F**. ∎

Theorem 3.3.3 (Confluence) *If* $M \longrightarrow\!\!\!\!\rightarrow N$ *and* $M \longrightarrow\!\!\!\!\rightarrow P$, *then there is a term* Q *such that* $N \longrightarrow\!\!\!\!\rightarrow Q$ *and* $P \longrightarrow\!\!\!\!\rightarrow Q$.

Proof. We only need to show that $\lambda_r^{\mu\nu}$ is weakly confluent, *i.e.*, if $M \longrightarrow N$ and $M \longrightarrow P$ then N and P have a common reduct, since by the previous theorem $\lambda_r^{\mu\nu}$ is strongly normalizing — using Newman's theorem [39] we may conclude that it is confluent. It is easy to see that it is weakly confluent, since there are no critical pairs, hence we are done. ∎

To continue the comparison of $\lambda_r^{\mu\nu}$ with PCF, we need to establish which types are observable. The criterion we use for whether a type is observable is if provable equality of closed terms of that type may be decided by reducing the terms to normal form and checking for syntactic identity. We restrict ourselves to closed terms because free variables can block the application of reduction rules — for example, the equation $x{:}\,\sigma \times \tau \rhd \langle \pi_1 x, \pi_2 x \rangle = x : \sigma \times \tau$ is true, but the two sides are distinct normal forms. If σ and τ are observable types, then we will see below that any closed term which could be substituted for x will be reducible to a pair $\langle N_1, N_2 \rangle$ of normal forms, whence the left-hand side will reduce by two additional $(\times\beta)$ steps to a term identical to the right-hand side; however, substituting every possible closed term for x and reducing

is not a very effective way to try to observe equality.[2] Functional types will not be observable, since a λ-abstraction introduces a variable which may cause the above problems in the body of the abstraction. Therefore, let us consider the possible normal forms which do not contain any free variables or λ-abstractions. In addition, although we have other ways than λ-abstraction to create terms of functional type, we will not want to try to observe a normal form containing such a subterm unless it is at the head of an application, thus guaranteeing that the term is not equal to another normal form with a λ-abstraction in that position.

Lemma 3.3.4 *The closed normal forms that do not contain any subterms of function type except at the head of an application are those well-formed terms that may be generated by the following grammar:*

$$R ::= \Diamond \mid \iota_1 R \mid \iota_2 R \mid \langle R, R' \rangle \mid \mathit{fold}_{\mu F} R;$$

we refer to these terms as results.

Proof. We could not have a result of the form $\pi_1 R$, for example, because for it to be a normal form we would need R to be a result of product type which is not a pair. Similarly, we could not have a result of the form $\pi_2 R$, $\mathit{it}_{\mu F}\, RR'$, $\mathit{unfold}_{\nu F}\, R$, or $[R, R']R''$. The only other way to get a result of type σ that is not of the form above is by $\Box^\sigma R$, where R has type 0. There are no closed terms of type 0, so this is impossible. Finally, note that there are no closed normal forms of type νF without functional subterms, since normal forms of that type would have to be of the form $\mathit{new}_{\nu F}\, RR'$, where R has a functional type. ∎

Therefore, the observable types are 1, $\sigma + \tau$, $\sigma \times \tau$, and μF, where σ and τ are observable and $F(\upsilon)$ is observable whenever υ is. The definition of a result as a closed normal form of observable type then matches the characterization given above.

Some examples of observable types are:

- $\mathit{bool} \equiv 1 + 1$, the type of booleans introduced above; the results are $\mathit{true} \equiv \iota_1 \Diamond$ and $\mathit{false} \equiv \iota_2 \Diamond$.

[2] And not necessarily even valid — this is the (ω) rule, which Plotkin [41] showed is not always sufficient for extensionality.

- $nat \equiv \mu t.\, 1 + t$, the set of natural numbers; results of this type take the form $zero \equiv fold(\iota_1 \diamond)$ or $succ(n) \equiv fold(\iota_2 n)$, where n is another such result.
- $bool \times nat$, pairs $\langle b, n \rangle$ of booleans and naturals.
- $natlist \equiv \mu t.\, 1 + nat \times t$, the type of lists of natural numbers; the results of this type are of the form $nil \equiv fold(\iota_1 \diamond)$ and $cons(n, \ell) \equiv fold(\iota_2 \langle n, \ell \rangle)$.

Thus we regain many of the observable types of PCF as a consequence of our more general definition.

We may now state the final theorem of this section, which asserts that $\lambda_r^{\mu\nu}$ is adequate for the computation of results of programs. This is analogous to the result property for PCF, although the method of proof will be somewhat different. Because of the form of the (η) rules for inductive and projective types, we do not have a reasonable definition for either η-reduction or η-expansion. For example, not only would use of the rule $M \longrightarrow it^{\tau}_{\mu F} N$ be conditional on having proven the equation $M \circ fold_{\mu F} = N \circ F(M)$, but it also introduces the term N in the contractum which does not occur in the redex — since N could contain free variables not present in M, we would need to reintroduce typing contexts to insure subject reduction. The same observation holds for the opposite direction, $it^{\tau}_{\mu F} N \longrightarrow M$; therefore, we will not be able to proceed through the intermediate step of an η-postponement lemma.

Theorem 3.3.5 *If $\emptyset \triangleright P = R : \sigma$ is provable for R a result, then $P \longrightarrow\!\!\!\!\!\rightarrow R$ in $\lambda_r^{\mu\nu}$.*

Proof. It is sufficient to show that if $\emptyset \triangleright P \stackrel{\eta}{=}_1 Q : \sigma$ is provable, and $Q \longrightarrow\!\!\!\!\!\rightarrow R$ for R a result, then $P \longrightarrow\!\!\!\!\!\rightarrow R$, where $\stackrel{\eta}{=}_1$ means that exactly one (η) rule is used (along with whatever structural rules are needed). The theorem then follows by the normal form property of $\longrightarrow\!\!\!\!\!\rightarrow$ and induction on the number of (η) rules.

The problem then is to construct a reduction sequence $P \longrightarrow\!\!\!\!\!\rightarrow R$ from the given sequence $Q \longrightarrow\!\!\!\!\!\rightarrow R$. If the (η) rule is $(+\eta)$, $(\times\eta)$, or $(\rightarrow\eta)$, then this chiefly consists of mimicking the reduction from Q on P, either adding or deleting steps corresponding to the destruction of an (η) redex by a (β) reduction. One difficulty arises in these cases from the non-linearity of $(+\eta)$ and $(\times\eta)$ — if Q contains the subterm $[M \circ \iota_1, M \circ \iota_2]$ where P only has M, for example, then we must choose to follow the reduction on

only the first, say, of the two components of the choice in reducing P; since R is a result and reduction is confluent, we may make this choice arbitrarily.

We are thus left with the case of the (η) step for one of the recursive types. We will consider $(\mu\eta)$ — the situation for $(\nu\eta)$ is entirely similar. If P contains a subterm M and Q contains $it^{\tau}_{\mu F}\, N$ in the same position, then in constructing the reduction from P we will need to use the hypothesis from the (η) rule, *i.e.*, $M \circ \mathit{fold}_{\mu F} = N \circ F(M)$. Now, for every $(\mu\beta)$ step in the original reduction of the form $it^{\tau}_{\mu F}\, N'(\mathit{fold}_{\mu F}\, K) \longrightarrow N'(F(it^{\tau}_{\mu F}\, N')K)$, we must replace it with the *equational* step $M'(\mathit{fold}_{\mu F}\, K) = N'(F(M')K)$, where M' and N' are corresponding residuals of M and N. We must then go back and apply the current theorem to convert this new equational proof of $P = R$ into a reduction $P \longrightarrow\!\!\!\!\!\rightarrow R$; we avoid circularity by noting that the height of the new proof tree for $P = R$ is shorter than before, measured in the number of nested applications of the $(\mu\eta)$ and $(\nu\eta)$ rules. In the situation where P contains $it^{\tau}_{\mu F}\, N$ and Q contains M we go through the same process, with the additional requirement that the reduction from Q must be rearranged so that if M is a λ-abstraction it will only be β-reduced when applied to arguments of the form $\mathit{fold}_{\mu F}\, K$. ∎

Corollary 3.3.6 *If $\emptyset \triangleright P = Q : \sigma$ is provable at observable type σ, then $\lambda^{\mu\nu}_r(P) \equiv \lambda^{\mu\nu}_r(Q)$.*

Proof. Trivial, since reduction is confluent and strongly normalizing. ∎

If we define *observational congruence* again as $\Gamma \triangleright M \simeq N : \sigma$ if $\lambda^{\mu\nu}_r(\mathcal{P}[M]) \equiv \lambda^{\mu\nu}_r(\mathcal{P}[N])$ for every well-formed program context $\mathcal{P}[\,]$, then we find that the full equational proof system, including the extensional rules, is sound for reasoning about programs:

Corollary 3.3.7 *The equational proof system for $\lambda^{\mu\nu}$ is sound for $\simeq$.*

Proof. Follows directly from the previous corollary. ∎

Therefore, we conclude that $\lambda^{\mu\nu}_r$ provides a suitable operational semantics for the language $\lambda^{\mu\nu}$.

3.4 Comparison with System T

The question naturally arises of what functions are expressible in $\lambda^{\mu\nu}$. We speak here of functions on the natural numbers, as given by the (observable) type *nat* described in the previous section. Since $\lambda_r^{\mu\nu}$ is strongly normalizing, all the functions must be total. We can enumerate the terms of $\lambda^{\mu\nu}$, therefore we must not be able to represent all of the total functions (since that set is not recursively enumerable). In this section we will see that all of the functions that are provably total in Peano arithmetic are definable, as well as some that are not. In the next section we show that $\lambda_r^{\mu\nu}$ is strongly normalizing by simulating it in System **F**; since the functions expressible in **F** are exactly those which are provably total in second-order arithmetic, we thus have a range in which the answer must fall:

Theorem 3.4.1 *The class of functions definable in $\lambda^{\mu\nu}$ properly includes those which are provably total in Peano arithmetic, and is included in the class of functions provably total in second-order arithmetic.*

To prove the first part of this theorem, we will compare $\lambda^{\mu\nu}$ to Gödel's System **T** of primitive recursive functionals of finite type ([18]; see also [17], for example). It is well-known that the natural number functions representable in **T** are precisely the partial recursive functions that are provably total in Peano arithmetic. Therefore, we start by showing that all those functions are also representable in $\lambda^{\mu\nu}$.

System **T** consists of the simply-typed lambda calculus $\lambda^{\rightarrow}$ plus the following constant terms and equations:

$$
\begin{aligned}
&0\colon \mathit{nat}\\
&\mathit{succ}\colon \mathit{nat}\rightarrow\mathit{nat}\\
&\mathbf{R}^{\tau}\colon \tau\rightarrow(\tau\rightarrow\mathit{nat}\rightarrow\tau)\rightarrow\mathit{nat}\rightarrow\tau
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{R}^{\tau}\, MN0 = M\\
&\mathbf{R}^{\tau}\, MN(\mathit{succ}\, P) = N(\mathbf{R}^{\tau}\, MNP)P.
\end{aligned}
$$

Here the type identifier *nat* is a new type constant, whose interpretation is the set of natural numbers; the terms 0 and *succ* represent the natural number zero and the

successor function. The term $\mathbf{R}$ gives a *recursor*; it is a generalization of definition by primitive recursion — if a function $f\colon \overline{nat}\to nat\to nat$ is defined using primitive recursion as

$$f(\bar{x})(0) = g(\bar{x})$$
$$f(\bar{x})(succ\, n) = h(\bar{x})(f(\bar{x})(n))(n),$$

where $g\colon \overline{nat}\to nat$ and $h\colon \overline{nat}\to nat\to nat\to nat$ are given, then it may be written using $\mathbf{R}$ as $(\lambda\bar{x}\colon\bar{\sigma}.\ \mathbf{R}^{nat}(g\bar{x})(h\bar{x}))$. The type τ in $\mathbf{R}^\tau$ may be any type, not just nat, hence we are able to define primitive recursive *functionals*. The well-known non-primitive recursive function, Ackermann's function[3], specified by

$$Ack(0)(y) = succ\, y$$
$$Ack(succ\, x)(0) = Ack(x)(1)$$
$$Ack(succ\, x)(succ\, y) = Ack(x)(Ack(succ\, x)(y)),$$

is easily defined with $\mathbf{R}$ as

$$\mathbf{R}^{nat\to nat}(succ)(\lambda f\colon nat\to nat.\ \lambda x\colon nat.\ \mathbf{R}^{nat}(f1)(\lambda z\colon nat.\ \lambda y\colon nat.\ fz)).$$

The variable f in the second argument is bound to the curried function $Ack(x)$ for the appropriate current value of x each time the outer recursion is unfolded; this is not possible in the ordinary scheme of primitive recursive function definition.

We have already seen how to define a type nat with constants 0 and $succ$ in $\lambda^{\mu\nu}$; thus we only need to define a recursor $\mathbf{R}^\tau$ and show that it satisfies the above equations. The syntax of $\lambda^{\mu\nu}$ only gives us an *iterator*, *i.e.*, a term $\mathbf{Z}^\tau\colon\tau\to(\tau\to\tau)\to nat\to\tau$ satisfying

$$\mathbf{Z}^\tau\, MN\, 0 = M$$
$$\mathbf{Z}^\tau\, MN(succ\, P) = N(\mathbf{Z}^\tau\, MNP).$$

Here we use the compound $\mathbf{Z}^\tau\, MN$ as an abbreviation for the term $it^\tau_{nat}[\lambda\diamond.\, M, N]$. Note that the argument P is not available to the function N in the recursive step, unlike the case for the recursor. The second argument to N is not used very often;

[3]Actually due to Péter; Ackermann's original function was somewhat different; see [47] for example.

for example, we may write the Ackermann function using $\mathbf{Z}$ as

$$\mathbf{Z}^{nat\to nat}(succ)(\lambda f\colon nat\to nat.\ \mathbf{Z}^{nat}(f1)f).$$

One significant function which *does* use the second argument is predecessor: $pred = \mathbf{R}^{nat}\,0(\lambda x\colon nat.\,\lambda y\colon nat.\,y)$.

In a language with products, a standard way to imitate a recursor with an iterator is to have the iterative call return both the current result and the corresponding value of the unavailable second argument. That is,

$$\mathbf{Z}^{\tau\times nat}\,M'N'0 = \langle M, 0\rangle$$
$$\mathbf{Z}^{\tau\times nat}\,M'N'(succ\,P) = (\texttt{let}\ \langle x\colon\tau, p\colon nat\rangle = (\mathbf{Z}^{\tau\times nat}\,M'N'P)\ \texttt{in}\ \langle Nxp, succ\,p\rangle),$$

where we must take $M' \equiv \langle M, 0\rangle$ and $N' \equiv (\lambda\langle x\colon\tau, p\colon nat\rangle.\,\langle Nxp, succ\,p\rangle)$. Thus the recursor-defined function $\mathbf{R}^{\tau}\,MN$ is also given by $\pi_1 \circ (\mathbf{Z}^{\tau\times nat}\,M'N')$.

Using this replacement for the recursor has an embarrassing property when applied to the definition of the predecessor — the equation $pred(succ\,P) = P$ only holds if P is equal to a numeral $succ^n\,0$. Even worse, in terms of the reduction system, *pred* must first evaluate its argument to normal form (which is wasteful if, for example, we are only going to test whether the predecessor is zero), and then, to compute $pred(succ^{n+1}\,0)$, it will apply the successor function to zero n times to reconstruct $succ^n\,0$, instead of just peeling off the outermost *succ*. Thus predecessor has been turned from a constant time lazy operation into a linear time strict operation.[4]

Our solution to this problem was hinted at earlier — we introduce the constant $unfold_{\mu F}\colon \mu F\to F(\mu F)$, and add the axioms

$$(\mu\beta') \qquad \Gamma \rhd unfold_{\mu F}\circ fold_{\mu F} = id^{F(\mu F)} : F(\mu F)\to F(\mu F)$$

[4]The lazy versus strict distinction will be important in the next chapter, where we allow non-termination.

and

$$(\mu\eta') \qquad \Gamma \triangleright \mathit{fold}_{\mu F} \circ \mathit{unfold}_{\mu F} = id^{\mu F} : \mu F \to \mu F.$$

Using the $(\mu\eta)$ rule, we proved in lemma 3.2.4 that this new constant is equal to the term $it^{F(\mu F)}_{\mu F}\, F(\mathit{fold}_{\mu F})$, which we have also been calling $\mathit{unfold}_{\mu F}$. To $\lambda^{\mu\nu}_r$ we add the corresponding β-reduction

$$(\mu\beta')_r \qquad \mathit{unfold}_{\mu F}(\mathit{fold}_{\mu F}\, P) \longrightarrow P;$$

it is not difficult to prove that all the theorems of the previous section continue to hold for this expanded system. Of course, because these new constants are definable in the unexpanded language, we are not changing the set of definable functions, only the set of expressible algorithms.

Dually, we also add a constant $\mathit{fold}_{\nu F}\colon F(\nu F)\to\nu F$ and rules

$$(\nu\beta') \qquad \Gamma \triangleright \mathit{unfold}_{\nu F} \circ \mathit{fold}_{\nu F} = id^{F(\nu F)} : F(\nu F) \to F(\nu F)$$

$$(\nu\eta') \qquad \Gamma \triangleright \mathit{fold}_{\nu F} \circ \mathit{unfold}_{\nu F} = id^{\nu F} : \nu F \to \nu F$$

$$(\nu\beta')_r \qquad \mathit{unfold}_{\nu F}(\mathit{fold}_{\nu F}\, P) \longrightarrow P;$$

this avoids a similar problem for projective types.

For example, let us define a type $\mathit{bstr} \equiv \nu t.\, \mathit{bool} \times t$ of boolean streams, with destructors $\mathit{head} \equiv \pi_1 \circ \mathit{unfold}_{\mathit{bstr}}\colon \mathit{bstr}\to\mathit{bool}$ and $\mathit{tail} \equiv \pi_2 \circ \mathit{unfold}_{\mathit{bstr}}\colon \mathit{bstr}\to\mathit{bstr}$, as well as a constructor $\mathit{all} \equiv \mathit{new}^{\mathit{bool}}_{\mathit{bstr}}\, \mathit{dup}\colon \mathit{bool}\to\mathit{bstr}$ which creates a constant stream, where $\mathit{dup} \equiv (\lambda b\colon \mathit{bool}.\, \langle b, b\rangle)\colon \mathit{bool}\to\mathit{bool}\times\mathit{bool}$. Then we find that $\mathit{head}(\mathit{all}\, \mathit{false}) \longrightarrow\!\!\!\!\!\rightarrow \mathit{false}$ and $\mathit{tail}(\mathit{all}\, \mathit{false}) \longrightarrow\!\!\!\!\!\rightarrow \mathit{all}\, \mathit{false}$, as expected. If we want to define a constructor to cons a boolean onto the head of a stream, then one way is to use $\mathit{cons} \equiv \mathit{new}^{\mathit{bool}\times\mathit{bstr}}_{\mathit{bstr}}(id^{\mathit{bool}} \times \mathit{unfold}_{\mathit{bstr}})\colon \mathit{bool} \times \mathit{bstr}\to\mathit{bstr}$. Unfortunately, this introduces an unnecessary unfolding when evaluating $\mathit{tail}(\mathit{cons}\langle \mathit{true}, \mathit{all}\, \mathit{false}\rangle)$, for example; the normal form of this term is $\mathit{cons}\langle \mathit{false}, \mathit{all}\, \mathit{false}\rangle$ instead of simply $\mathit{all}\, \mathit{false}$. This is not entirely unexpected, since bstr is not an observable type, hence there may be

many distinct normal forms for a given stream. What is worse is that this extra unfolding will propagate through an entire sequence of *cons*'s, *i.e.*, to compute the tail of a stream $b_1 :: b_2 :: \ldots :: b_n :: s$, where we write $b :: s$ as shorthand for $cons\langle b, s\rangle$, the reduction will pick off b_1 through b_n, unfold s once, and then cons b_2 through b_n back onto the head! The solution is to use the constant $fold_{bstr}\colon bool \times bstr \to bstr$ instead of *cons*; the extra unfolding is avoided, and the tail can be computed in constant time.

With $unfold_{\mu F}$ we may make an *ad hoc* definition of predecessor simply as $pred = [\lambda\Diamond.\, 0, id] \circ unfold_{nat}$. A more interesting discovery is that we may use this predecessor function to define a recursor $\mathbf{R}_{it}$ with the same reduction behavior as $\mathbf{R}$. The idea is to use the iterator to create a function that passes the needed second argument *down* from the outside, rather than creating it up from the inside. Here is the term that defines $\mathbf{R}^{\tau}_{it}\, MN$:

$$(\lambda n\colon nat.\ \mathbf{Z}^{nat\to\tau}(\lambda x\colon nat.\, M)(\lambda f\colon nat\to\tau.\, \lambda m\colon nat.\, N(f(pred\, m))(pred\, m))nn).$$

Theorem 3.4.2 *A function defined with the recursor $\mathbf{R}_{it}$ will have the same running time in $\lambda^{\mu\nu}_{r} + (\mu\beta')_r$ (within a constant factor) as the equivalent function defined with $\mathbf{R}$ will have in $\mathbf{T}$.*

Proof. All we need to show is that the two reduction rules for $\mathbf{R}$ in $\mathbf{T}$ can be performed in constant time in $\lambda^{\mu\nu}_{r} + (\mu\beta')_r$. We omit a number of steps showing the details of reducing $\mathbf{Z}$ and *pred*; it is easy to verify that the total number of steps remains independent of the size of the input. Also, we abbreviate the term $\mathbf{Z}^{nat\to\tau}(\lambda x\colon nat.\, M)(\lambda f\colon nat\to\tau.\, \lambda m\colon nat.\, N(f(pred\, m))(pred\, m))$ as $\mathbf{Q}^{\tau}_{M,N}$.

$$\begin{aligned}
\mathbf{R}^{\tau}_{it}\, MN\, 0 &\longrightarrow \mathbf{Q}^{\tau}_{M,N}\ 0\, 0\\
&\longrightarrow\!\!\!\!\to (\lambda x\colon nat.\, M)\, 0\\
&\longrightarrow M
\end{aligned}$$

$$\begin{aligned}
\mathbf{R}^{\tau}_{it}\, MN(succ\, P) &\longrightarrow \mathbf{Q}^{\tau}_{M,N}(succ\, P)(succ\, P)\\
&\longrightarrow\!\!\!\!\to (\lambda f\colon nat\to\tau.\, \lambda m\colon nat.\, N(f(pred\, m))(pred\, m))(\mathbf{Q}^{\tau}_{M,N}\, P)\\
&\qquad\quad (succ\, P)
\end{aligned}$$

$$\begin{aligned}
&\longrightarrow \quad (\lambda m\colon nat.\, N(\mathbf{Q}^{\tau}_{M,N}\, P(pred\, m))(pred\, m))(succ\, P)\\
&\longrightarrow \quad N(\mathbf{Q}^{\tau}_{M,N}\, P(pred(succ\, P)))(pred(succ\, P))\\
&\longrightarrow\!\!\!\!\rightarrow \quad N(\mathbf{Q}^{\tau}_{M,N}\, PP)P.
\end{aligned}$$

This last term is one ($\rightarrow\beta$) step from $N(\mathbf{R}^{\tau}_{it}\, MNP)P$, which corresponds to the term we would get after one reduction step from $\mathbf{R}^{\tau}\, MN(succ\, P)$ in $\mathbf{T}$. ∎

We have already mentioned the relation between System $\mathbf{T}$ and the functions that are provably total in Peano arithmetic. Another characterization of the functions expressible in $\mathbf{T}$ is that they are the functions definable by transfinite recursion up to some ordinal $\alpha < \epsilon_0$, where ϵ_0 is the least ordinal ϵ such that $\epsilon = \omega^{\epsilon}$ (see [30] or [48]; a good text covering this subject is [47]). We will show that $\lambda^{\mu\nu}$ is more expressive than $\mathbf{T}$, thus completing the proof of Theorem 3.4.1, by constructing the Hardy function H_{ϵ_0}, which requires recursion up to ϵ_0 itself [8]. The method we use to represent ordinal numbers and construct the hierarchy of Hardy functions is based on an example given by Coquand and Paulin [11].

To define H_α for $\alpha \leq \epsilon_0$ we will need to choose a *fundamental sequence* for each limit ordinal $\leq \epsilon_0$; that is, for each limit ordinal λ we need an increasing, natural number indexed sequence $\langle \lambda[0], \lambda[1], \ldots \rangle$ of ordinals less than λ whose limit is λ. A convenient choice makes use of the Cantor normal form, which writes each ordinal $\alpha < \epsilon_0$ uniquely as $\omega^{\alpha_1} + \cdots + \omega^{\alpha_k} + m$, for some natural numbers k and m and ordinals (themselves in normal form) $0 < \alpha_k \leq \ldots \leq \alpha_1 < \alpha$. If α is a limit ordinal, then we take the ordinal $\omega^{\alpha_1} + \cdots + \omega^{\alpha_k}[n]$ as the nth element of the fundamental sequence for α, where $\omega^{\beta+1}[n] = \omega^{\beta} \cdot n$ and $\omega^{\lambda}[n] = \omega^{\lambda[n]}$ for λ a limit. We extend this definition to ϵ_0 by taking $\epsilon_0[0] = 1$ and $\epsilon_0[n+1] = \omega^{\epsilon_0[n]}$. Now we may define the functions H_α as follows:

$$\begin{aligned}
H_0(n) &= n\\
H_{\alpha+1}(n) &= H_\alpha(n+1)\\
H_\lambda(n) &= H_{\lambda[n]}(n).
\end{aligned}$$

The type of notations for countable ordinals may be specified as the inductive type $ord \equiv \mu t.\, 1 + t + (nat \to t)$; the constructors are thus

$$\begin{aligned} ordzero &\equiv fold_{ord} \circ \iota_1^3 \colon 1 \to ord \\ ordsucc &\equiv fold_{ord} \circ \iota_2^3 \colon ord \to ord \\ lim &\equiv fold_{ord} \circ \iota_3^3 \colon (nat \to ord) \to ord. \end{aligned}$$

The interpretation of *lim* is that it creates an ordinal given a function which specifies the fundamental sequence for the ordinal; for example, we may define $\omega \equiv lim\, inord$, where $inord \equiv it_{nat}^{ord}[ordzero, ordsucc]$ is the natural injection from *nat* to *ord*, since $\langle 0, 1, 2, \ldots \rangle$ is the fundamental sequence for ω. Addition, multiplication, and exponentiation of ordinals may be defined as follows:

$$\begin{aligned} ordplus &\equiv (\lambda \alpha \colon ord.\ it_{ord}^{ord}[(\lambda \diamond.\, \alpha), ordsucc, lim]) \\ ordtimes &\equiv (\lambda \alpha \colon ord.\ it_{ord}^{ord}[ordzero, (\lambda \beta \colon ord.\ ordplus\, \beta \alpha), lim]) \\ ordexp &\equiv (\lambda \alpha \colon ord.\ it_{ord}^{ord}[ordone, (\lambda \beta \colon ord.\ ordtimes\, \beta \alpha), lim]), \end{aligned}$$

where $ordone \equiv ordsucc \circ ordzero$. A function that creates an exponential stack of n ω's when applied to a natural number n is $omegaexp \equiv it_{nat}^{ord}[ordone, ordexp\, \omega]$; we may thus define a notation for ϵ_0 by stating $\epsilon_0 \equiv lim\, omegaexp$.

The following term represents the Hardy function $H \colon ord \to nat \to nat$ in $\lambda^{\mu\nu}$:

$$it_{ord}^{nat \to nat}[(\lambda \diamond.\ id^{nat}), (\lambda f \colon nat \to nat.\ f \circ succ), (\lambda g \colon nat \to nat \to nat.\ \lambda n \colon nat.\ gnn)].$$

We will demonstrate the use of these definitions by evaluating $H_{\epsilon_0}(0)$:

$$\begin{aligned} H\epsilon_0\, 0 &\equiv H(lim\, omegaexp)\, 0 \\ &\longrightarrow\!\!\!\!\!\rightarrow (H \circ omegaexp)\, 0\, 0 \\ &\longrightarrow\!\!\!\!\!\rightarrow H(ordone \diamond)\, 0 \\ &\equiv H(ordsucc(ordzero \diamond))\, 0 \\ &\longrightarrow\!\!\!\!\!\rightarrow ((H(ordzero \diamond)) \circ succ)\, 0 \\ &\longrightarrow\!\!\!\!\!\rightarrow (id^{nat} \circ succ)\, 0 \\ &\longrightarrow\!\!\!\!\!\rightarrow succ\, 0. \end{aligned}$$

Being able to construct H_{ϵ_0} is sufficient for showing that $\lambda^{\mu\nu}$ can express more than the primitive recursive functionals of $\mathbf{T}$, but there is no reason to stop at ϵ_0. Indeed, since we may define the Veblen hierarchy of functions $\varphi_\alpha\colon ord\to ord$ for all $\alpha\colon ord$, we have a system of notation for all the ordinals less than Γ_0, the first "strongly critical" ordinal (see [15] for a very readable discussion of the significance of Γ_0). The particular Veblen hierarchy to which we refer is that starting from $\varphi_0(\beta) = \omega^\beta$; then the function φ_α for $\alpha > 0$ enumerates the common fixed points of all the functions φ_γ for $\gamma < \alpha$. For example, φ_1 enumerates the fixed points of $\varphi_0 = \lambda\beta.\,\omega^\beta$; these are known as the epsilon numbers, and indeed the first fixed point $\varphi_1(0)$ is the ordinal ϵ_0 discussed above. We will not give the term which computes φ here, but a detailed description of how to define it in terms of fundamental sequences is given in [10].

The ordinal Γ_0 is still not the largest ordinal we can express in $\lambda^{\mu\nu}$. As Miller shows in [34], we may extend the definition of φ_α to uncountable ordinals α that satisfy certain conditions. For example, φ_Ω is the function which enumerates the strongly critical ordinals (so $\varphi_\Omega(0) = \Gamma_0$), where Ω is the least uncountable ordinal. We can express Ω, and many other uncountable ordinals, in $\lambda^{\mu\nu}$ by introducing the type $ord_1 \equiv \mu t.\,1 + t + (nat\to t) + (ord\to t)$. As for ord, the first three components of the body of the recursive type represent zero, successor ordinals, and (nat-indexed) limit ordinals. But ord_1 also has a fourth component which allows ord-indexed limits. If we define $\lim_1^1 \equiv fold_{ord_1} \circ \iota_4^4\colon (ord\to ord_1)\to ord_1$, and make the obvious definition for $inord_1\colon ord\to ord_1$, then we may set $\Omega \equiv \lim_1^1 inord_1$. We may then define the usual arithmetic operations on ord_1, allowing the construction of such ordinals as Ω^2, Ω^Ω, and even $\epsilon_{\Omega+1} = \Omega^{\Omega^{\cdots}}$, with which we may go back and construct the countable (but *very* large) ordinal $\varphi_{\epsilon_{\Omega+1}+1}(0)$, known as "Howard's ordinal."[5]

We may make one more step in the production of ever-larger ordinals. By generalizing the construction of the types ord and ord_1, we may construct the class ord_n of *abstract tree ordinals* (see [56]), all of which will have cardinality $\aleph_n$, by the inductive type $\mu t.\,1 + t + (nat\to t) + (ord\to t) + (ord_1\to t) + \cdots + (ord_{n-1}\to t)$. Elements of this type may be defined as limits of order type up to Ω_n, the nth regular ordinal

[5]No relation.

beyond $\Omega_0 \equiv \omega$. Further discussion of these ordinals is far beyond the scope of this dissertation; for more details see for example [34, 56].

3.5 Comparison with System F

In this section we will give a translation of $\lambda^{\mu\nu}$ into Girard's System **F** that will allow us to prove the strong normalization of $\lambda_r^{\mu\nu}$ from that of **F**. The essential part of this translation is the well-known representation of finite sums and products and initial and terminal fixed points in **F** (see for example [17] or [24]), although the details of the translation for inductive and projective types are original. We have already noted that the functions expressible in **F** are precisely those that are provably total in second order arithmetic, so the fact that all the functions computable by $\lambda_r^{\mu\nu}$ are also computable in **F** gives us an upper limit on the expressibility of $\lambda^{\mu\nu}$. We suspect that the inclusion is proper, although we do not know an example of a function computable in **F** and not in $\lambda_r^{\mu\nu}$. Interestingly, there are *algorithms* computable in $\lambda_r^{\mu\nu} + (\mu\beta')_r$ which are not computable in **F** — the simplest example is the constant time predecessor mentioned in the previous section. This is a symptom of a more general lack in System **F**, namely, that types such as products, sums, and least fixed points are not extensional; as a result, many desirable equations between terms are not provable.

System **F**, which was discovered independently by Girard [16] and Reynolds [46], extends the simply typed lambda calculus $\lambda^{\rightarrow}$ with type variables and the polymorphic type $\forall t.\, \sigma$. Formally, we add the following term formation rules:

$$(\forall\ \textit{Intro}) \qquad \frac{\Gamma \rhd M\colon \sigma}{\Gamma \rhd (\Lambda t.\, M)\colon \forall t.\, \sigma} \text{ for } t \text{ not free in } \Gamma$$

$$(\forall\ \textit{Elim}) \qquad \frac{\Gamma \rhd M\colon \forall t.\, \sigma}{\Gamma \rhd M\tau\colon \{\tau/t\}\sigma.}$$

The metavariables σ and τ now refer to arbitrary type expressions formed with $\rightarrow$ and $\forall$; that is, they may contain free type variables. The type abstraction operator

Λ binds type variables in its body in the same manner as λ binds regular variables, thus we need equational rules for Λ analogous to those for λ:

$$(\forall\ abs) \qquad \frac{\Gamma \triangleright M = N : \sigma}{\Gamma \triangleright (\Lambda t.\, M) = (\Lambda t.\, N) : \forall t.\, \sigma}$$

$$(\forall\ app) \qquad \frac{\Gamma \triangleright M = N : \forall t.\, \sigma}{\Gamma \triangleright M\tau = N\tau : \{\tau/t\}\sigma}$$

$$(\forall\alpha) \qquad \Gamma \triangleright (\Lambda t.\, M) = (\Lambda s.\, \{s/t\}M) : \forall t.\, \sigma, \text{ if } s \text{ not free in } M$$

$$(\forall\beta) \qquad \Gamma \triangleright (\Lambda t.\, M)\tau = \{\tau/t\}M : \{\tau/t\}\sigma$$

$$(\forall\eta) \qquad \Gamma \triangleright (\Lambda t.\, Mt) = M : \forall t.\, \sigma, \text{ for } t \text{ not free in } M.$$

As usual, the reduction relation obtained by directing the $(\rightarrow\beta)$ and $(\forall\beta)$ axioms from left to right will be denoted $\longrightarrow$; if a term M reduces to another term N in one or more steps, then we write $M \longrightarrow^+ N$.

We will now start to give a translation from $\lambda^{\mu\nu}$ into $\mathbf{F}$, such that if $\Gamma \triangleright M{:}\,\sigma$ is a well-formed term in $\lambda^{\mu\nu}$, then $\overline{\Gamma} \triangleright \overline{M}{:}\,\overline{\sigma}$ is a well-formed term of $\mathbf{F}$. Type and term variables will stay the same under translation, as will lambda abstraction and application. The translation of the binary sum type $\sigma + \tau$ will be $\forall t.\, (\overline{\sigma}\rightarrow t)\rightarrow(\overline{\tau}\rightarrow t)\rightarrow t$, where t is not free in $\overline{\sigma}$ or $\overline{\tau}$; the empty type 0 is represented by $\forall t.\, t$. Dually, the binary product $\sigma \times \tau$ translates to $\forall t.\, (\overline{\sigma}\rightarrow\overline{\tau}\rightarrow t)\rightarrow t$, while the singleton type 1 becomes $\forall t.\, t\rightarrow t$. The translation of terms for these types is given in Table 3.1, where of course it is understood that all of the variables introduced on the right hand side are new.

Before we give the translation for the inductive and projective types, we will need a few abbreviations. It will be convenient to continue our practice of treating a type expression σ as a functor with respect to substitution for a type variable t, thus we will write $\overline{F}(\overline{\tau})$ to mean $\{\overline{\tau}/t\}\overline{\sigma}$; it is an easy exercise to show that $\overline{F(\tau)} \equiv \overline{F}(\overline{\tau})$. Similarly, if M is a term of type $\tau\rightarrow\upsilon$, then we write $\overline{F}(\overline{M})$ for the term $\overline{F(M)}$ of type $\overline{F(\tau)}\rightarrow\overline{F(\upsilon)}$. The "functor" $\overline{F}$ will no longer preserve composition or identities, since most types under the translation into $\mathbf{F}$ are no longer extensional (see below),

$M\colon\sigma$	$\overline{M}\colon\overline{\sigma}$
$x\colon\sigma$	$x\colon\overline{\sigma}$
$(\lambda x\colon\sigma.\, M)\colon\sigma\to\tau$	$(\lambda x\colon\overline{\sigma}.\, \overline{M})\colon\overline{\sigma}\to\overline{\tau}$
$MN\colon\tau$	$\overline{M}\,\overline{N}\colon\overline{\tau}$
$\iota_1\colon\sigma\to\sigma+\tau$	$(\lambda x\colon\overline{\sigma}.\, \Lambda t.\, \lambda f\colon\overline{\sigma}\to t.\, \lambda g\colon\overline{\tau}\to t.\, fx)\colon\overline{\sigma}\to\overline{\sigma+\tau}$
$\iota_2\colon\tau\to\sigma+\tau$	$(\lambda x\colon\overline{\tau}.\, \Lambda t.\, \lambda f\colon\overline{\sigma}\to t.\, \lambda g\colon\overline{\tau}\to t.\, gx)\colon\overline{\tau}\to\overline{\sigma+\tau}$
$[M,N]\colon\sigma+\tau\to\upsilon$	$(\lambda x\colon\overline{\sigma+\tau}.\, x\overline{\upsilon}\,\overline{M}\,\overline{N})\colon\overline{\sigma+\tau}\to\overline{\upsilon}$
$\Box^{\upsilon}\colon 0\to\upsilon$	$(\lambda x\colon\overline{0}.\, x\overline{\upsilon})\colon\overline{0}\to\overline{\upsilon}$
$\pi_1\colon\sigma\times\tau\to\sigma$	$(\lambda x\colon\overline{\sigma\times\tau}.\, x\overline{\sigma}(\lambda y\colon\overline{\sigma}.\, \lambda z\colon\overline{\tau}.\, y))\colon\overline{\sigma\times\tau}\to\overline{\sigma}$
$\pi_2\colon\sigma\times\tau\to\tau$	$(\lambda x\colon\overline{\sigma\times\tau}.\, x\overline{\tau}(\lambda y\colon\overline{\sigma}.\, \lambda z\colon\overline{\tau}.\, z))\colon\overline{\sigma\times\tau}\to\overline{\tau}$
$\langle M,N\rangle\colon\sigma\times\tau$	$(\Lambda t.\, \lambda f\colon\overline{\sigma}\to\overline{\tau}\to t.\, f\overline{M}\,\overline{N})\colon\overline{\sigma\times\tau}$
$\diamond\colon 1$	$(\Lambda t.\, \lambda x\colon t.\, x)\colon\overline{1}$

Table 3.1: Translation of function, sum, and product terms

but it will suffice for the purposes of this section because we are only interested in showing that the β reductions of $\lambda^{\mu\nu}$ are strongly normalizing.

We will also need the existentially quantified type $\exists t.\, \sigma$. This may be expressed in terms of $\forall$ and $\to$ as $\forall s.\, (\forall t.\, \sigma\to s)\to s$, where s is a fresh type variable. A term of the existential type $\exists t.\, \sigma$ is like a pair $\langle\tau, M\rangle$ of a type τ and a term M of type $\{\tau/t\}\sigma$; we will exploit this analogy by introducing (as in [17]) the syntactic sugar

$$\begin{aligned}\langle\tau, N\rangle &\equiv (\Lambda s.\, \lambda x\colon\forall t.\, \sigma\to s.\, x\tau N)\colon\exists t.\, \sigma\\ (\lambda\langle t, x\colon\sigma\rangle.\, M) &\equiv (\lambda y\colon\exists t.\, \sigma.\, y\upsilon(\Lambda t.\, \lambda x\colon\sigma.\, M))\colon(\exists t.\, \sigma)\to\upsilon,\end{aligned}$$

where in the latter definition the term M has type υ. It will be convenient to have this pattern matching syntax for terms of (translated) product type as well; thus, the **F** term $(\lambda\langle x\colon\sigma, y\colon\tau\rangle.\, M)$ will be an abbreviation for

$$(\lambda p\colon\forall t.\, (\sigma\to\tau\to t)\to t.\, p\upsilon(\lambda x\colon\sigma.\, \lambda y\colon\tau.\, M).$$

$M\colon\sigma$	$\overline{M}\colon\overline{\sigma}$
$fold_{\mu F}$	$(\lambda x\colon\overline{F}(\overline{\mu F}).\,\Lambda t.\,\lambda f\colon\overline{F}(t)\to t.\,f(\overline{F}(\overline{it}\,t\,f)\,x))\colon\overline{F}(\overline{\mu F})\to\overline{\mu F}$
$it^{\tau}_{\mu F}$	$\overline{it}\,\overline{\tau}\equiv(\Lambda s.\,\lambda f\colon\overline{F}(s)\to s.\,\lambda x\colon\overline{\mu F}.\,x\,s\,f)\overline{\tau}\colon(\overline{F}(\overline{\tau})\to\overline{\tau})\to\overline{\mu F}\to\overline{\tau}$
$unfold_{\nu F}$	$(\lambda\langle t,\langle f\colon t\to\overline{\sigma},x\colon t\rangle\rangle.\,\overline{F}(\overline{new}\,t\,f)(f\,x))\colon\overline{\nu F}\to\overline{F}(\overline{\nu F})$
$new^{\tau}_{\nu F}$	$\overline{new}\,\overline{\tau}\equiv(\Lambda s.\,\lambda f\colon s\to\overline{F}(s).\,\lambda x\colon s.\,\langle s,\langle f,x\rangle\rangle)\overline{\tau}\colon(\overline{\tau}\to\overline{F}(\overline{\tau}))\to\overline{\tau}\to\overline{\nu F}$

Table 3.2: Translation of inductive and projective terms

If we also write $\langle N,P\rangle$ for $(\Lambda t.\,\lambda f\colon\sigma\to\tau\to t.\,f\,N\,P)$, then it is easy to see that both of these pattern matching terms behave as expected under reduction, *i.e.*,

$$\begin{aligned}(\lambda\langle t,x\colon\sigma\rangle.\,M)\langle\tau,N\rangle &\longrightarrow^{+} \{N/x\}\{\tau/t\}M\\ (\lambda\langle x\colon\sigma,y\colon\tau\rangle.\,M)\langle N,P\rangle &\longrightarrow^{+} \{P/y\}\{N/x\}M.\end{aligned}$$

The general translation for an inductive type $\mu t.\,\sigma$ was essentially given by Böhm and Berarducci [4], although they were mainly concerned with representing iteratively defined functions over the term algebra of an algebraic signature, touching only briefly on iteration at higher types. The corresponding translation for a projective type $\nu t.\,\sigma$ was given independently by Hasegawa [24] and Wraith [58]. Here are the translations for the types:

$$\begin{aligned}\overline{\mu t.\,\sigma} &\equiv \forall t.\,(\overline{\sigma}\to t)\to t\\ \overline{\nu t.\,\sigma} &\equiv \exists t.\,\overline{(t\to\sigma)\times t}\\ &\equiv \exists t.\,\forall u.\,((t\to\overline{\sigma})\to t\to u)\to u\end{aligned}$$

The translations of the terms for these types are given in Table 3.2.

It is now a simple matter to verify that this translation preserves reduction in $\lambda^{\mu\nu}_{r}$.

Lemma 3.5.1 *If $M\longrightarrow N$ in $\lambda^{\mu\nu}_{r}$, then $\overline{M}\longrightarrow^{+}\overline{N}$.*

Proof. We will show two of the cases; the rest of the proof is entirely similar. We

omit most types for brevity.

$$\begin{array}{rcl}
\overline{[M,N](\iota_1 P)} & \equiv & (\lambda x.\, x\overline{v}\,\overline{M}\,\overline{N})((\lambda y.\,\Lambda t.\,\lambda f.\,\lambda g.\, f\, y)\,\overline{P}) \\
& \longrightarrow^+ & (\Lambda t.\,\lambda f.\,\lambda g.\, f\,\overline{P})\,\overline{v}\,\overline{M}\,\overline{N} \\
& \longrightarrow^+ & \overline{M}\,\overline{P} \\
& \equiv & \overline{MP} \\
\overline{\mathit{unfold}(\mathit{new}^\tau\, M\, N)} & \equiv & (\lambda\langle t,\langle f,x\rangle\rangle.\,\overline{F}(\overline{\mathit{new}}\,t\,f)(f\,x)) \\
& & \quad ((\Lambda s.\,\lambda g.\,\lambda y.\,\langle s,\langle g,y\rangle\rangle)\,\overline{\tau}\,\overline{M}\,\overline{N}) \\
& \longrightarrow^+ & (\lambda\langle t,\langle f,x\rangle\rangle.\,\overline{F}(\overline{\mathit{new}}\,t\,f)(f\,x))\langle\overline{\tau},\langle\overline{M},\overline{N}\rangle\rangle \\
& \longrightarrow^+ & \overline{F}(\overline{\mathit{new}}\,\overline{\tau}\,\overline{M})(\overline{M}\,\overline{N}) \\
& \equiv & \overline{F(\mathit{new}^\tau\, M)(MN)}
\end{array}$$

■

This lemma is precisely what we needed to complete the proof of strong normalization for $\lambda_r^{\mu\nu}$, since if there were an infinite reduction sequence from a term M in $\lambda_r^{\mu\nu}$ then we would be able to construct an infinite reduction from $\overline{M}$ in **F**. System **F** is strongly normalizing (see [17], for example), so we are done.

Chapter 4

Retractive Types and Non-termination

The inductive and projective solutions are only defined if the functor F is covariant. In general, however, we want to be able to solve RDEs where the type variable may occur negatively; for example, to model the untyped λ-calculus we need to solve equations such as $X \cong X \to X$. The standard approach, due to Smyth and Plotkin [51], is to consider the category of *embedding-projection pairs*, also known as *retracts*. Given a category $\mathcal{C}$ such that each homset $\mathcal{C}(A, B)$ is enriched with a partial order structure $\leq$, we say that a pair $(f\colon A \to B, g\colon B \to A)$ is a retract from A to B (and f is an embedding, g is a projection) if $f; g = A$ (*i.e.*, f followed by g is the identity arrow on A) and $g; f \leq B$. The category of retracts, $\mathcal{C}^R$, then has the same objects as $\mathcal{C}$, with retracts for the arrows. The advantage of considering this category is that mixed variant functors on $\mathcal{C}$ may be replaced by covariant functors on $\mathcal{C}^R$, which allows us to construct their least fixed point by taking a colimit of a suitable chain of retracts. For lack of a better name, we refer to types constructed in this manner as *retractive*; the notation we use is $\rho t.\, \sigma$, or ρF.

In the category $\mathbf{CPPO}^R$, the terminal object 1 from **CPPO** serves as the initial object, since $(\bot, \diamond)$ is a retract from 1 to any object A, where $\bot\colon 1 \to A$ is the least arrow in the homset $\mathbf{CPPO}(1, A)$, being the constant function which picks out the bottom element of A. We may thus construct a chain of retracts just as for the

inductive case in the previous chapter, using $(\bot, \diamond)\colon 1 \to F1$ in place of $\Box\colon 0 \to F0$. However, we are interested in $\mathbf{CPO}^R$, which has neither initial nor terminal objects. To find a chain of retracts in $\mathbf{CPO}^R$, we observe that if $F1$ is pointed, then there is a retract from 1 to $F1$, given as before by $(\bot, \diamond)$, since $F1$ has a bottom element; if F preserves retracts, then we may continue the chain as usual — indeed, it is the same chain as would have been formed in the subcategory $\mathbf{CPPO}^R$. The key to being able to say which functors F will allow this construction is to introduce the lifting functor, which adds a bottom element to a cpo. Formally, we define $\bot\colon \mathbf{CPO} \to \mathbf{CPO}$ to be UL, where $L\colon \mathbf{CPO} \to \mathbf{CPPO}_\bot$ is the left adjoint to the forgetful functor U from the category $\mathbf{CPPO}_\bot$ of cppo's and *strict* (*i.e.*, bottom-preserving) functions into $\mathbf{CPO}$. The advantage of this categorical definition is that we can read off most of the desired equations for terms of lifted type from the equations in the model, just as we may read off the usual (β) and (η) rules from the categorical properties of cartesian closedness.

We will say that a functor $F\colon \mathbf{CPO} \to \mathbf{CPO}$ is (unconditionally) *pointed* if FX has a bottom element for any object X; it is *conditionally pointed* if FX has a bottom element whenever X does (a trivial example is the identity functor). The allowable functors then are the conditionally pointed ones (since 1 is pointed, $F1$ will be) which preserve retracts. We may now use the lifting functor to form a syntactic class of functors which allow retractive fixed points.

When we have a type expression σ that is both covariant and conditionally pointed in a free type variable t, we may form the three recursive types $\mu t.\,\sigma$, $\nu t.\,\sigma$, and $\rho t.\,\sigma$. An interesting fact in $\mathbf{CPO}$ is that the last two will always be isomorphic. If σ is unconditionally pointed, then all three coincide. This result is related to recent work of Barr, Freyd, and Fokkinga and Meijer [2, 14, 13], although it was arrived at independently.

4.1 Syntax of the language $\lambda^{\bot\rho}$

In this section we will describe the syntax for the types and terms of $\lambda^{\bot\rho}$ as an extension to the language $\lambda^{\mu\nu}$ of the previous chapter. We start by adding two type constructors: the *lifted* type $\sigma_\bot$, which corresponds to the operation which adds a

bottom element to a cpo, and the *retractive* type $\rho t.\,\sigma$, which corresponds to the recursive type found by the usual Smyth-Plotkin construction in $\mathbf{CPO}^R$.

Not all type expressions σ may appear as the body of a retractive type, just as the bodies of inductive and projective types were restricted in the previous chapter to type expressions strictly positive in the type variable being bound. To state the restriction for retractive types (and also to relax the restriction for inductive and projective types) we need to formally introduce the concept of a *pointed* type. Intuitively, a pointed type is one which contains a bottom element, *i.e.*, a least element with respect to the information-content ordering on the type. The bottom element represents a complete lack of information about the value, generally due to non-termination of an evaluation function. Following our cpo interpretation of the type constructors, we may see that the types 1 and $\sigma_\perp$ are always pointed; it will also turn out that the retractive type $\rho t.\,\sigma$ is always pointed. If the types σ and τ are both pointed, then the product $\sigma \times \tau$ will be pointed, since the pair consisting of the bottom elements of σ and τ will be the least element under the pointwise ordering of $\sigma \times \tau$. Similarly, if τ is pointed, then the function type $\sigma{\to}\tau$ will be pointed for any σ, since the constant bottom function is less defined than any other element of $\sigma{\to}\tau$. The empty type 0 and the disjoint sum $\sigma + \tau$ will never be pointed; we do not even consider a type such as $\sigma_\perp + 0$ to be pointed, despite the fact that its cpo representation has a least element, since an element of a sum type necessarily conveys at least the information that it comes from, say, the first summand.

Since a recursive type $\mu t.\,\sigma$ (or, mutatis mutandis, $\nu t.\,\sigma$ or $\rho t.\,\sigma$) is isomorphic to the unfolded type $\{\mu t.\,\sigma/t\}\sigma$, it is reasonable to consider a recursive type to be pointed whenever the body σ is. We have two choices when defining the pointedness of type expressions with free variables. If σ is pointed no matter what types are substituted for the free variables, then it is said to be *unconditionally pointed*. For example, the type expression $s{\to}t_\perp$ is unconditionally pointed, since $t_\perp$ is always a pointed type expression. If the pointedness of σ depends on that of a free variable t, as for example in $\tau{\to}t$, then it is *conditionally pointed with respect to* t. Thus, our rule for inductive types will be that $\mu t.\,\sigma$ is pointed if σ is unconditionally pointed. We may go further with projective and retractive types, saying that $\nu t.\,\sigma$ and $\rho t.\,\sigma$

will be pointed if σ is conditionally pointed with respect to t. The reason for this difference comes from the respective constructions of the recursive types in **CPO**: an inductive type μF essentially results from an infinite number of applications of the functor F to the initial object 0, while the projective and retractive types start from 1. If F is only conditionally pointed, *i.e.*, $F(\tau)$ is only pointed if τ is, then none of the finite approximations 0, $F(0)$, $F^2(0)$, ..., to μF will be pointed; by continuity we thus expect that μF itself will not be pointed.[1] By contrast, all of the approximations 1, $F(1)$, $F^2(1)$, ..., to νF and ρF are pointed, so the limit types will be as well.

The introduction of retractive types necessitates a refinement of the definition of a type variable occurring covariantly in a type expression. Consider the expression $\rho s.\, s{\to}t$; at first glance it seems to be covariant in t, even strictly positive. However, one unfolding of this type expression produces $(\rho s.\, s{\to}t){\to}t$, in which t occurs both positively and negatively. The rule must be that $\rho s.\, \sigma$ is covariant in t only if σ is covariant in both s and t, or t does not occur at all in σ. This ensures that covariance is preserved under unfolding.

We now state in words the full conditions on the bodies of recursive types; a system of inference rules is given below. These conditions will be justified in the final section of this chapter, which compares the three ways of defining recursive types. An inductive type $\mu t.\, \sigma$ or a projective type $\nu t.\, \sigma$ is well-formed if t occurs covariantly in σ and if each occurrence of t on the left of an arrow (hence on the left of an even number of arrows) is contained in some (unconditionally) pointed subterm of σ. For a retractive type $\rho t.\, \sigma$ to be well-formed, t may be of mixed variance in σ, but σ must be conditionally pointed with respect to t. Thus, for example, $\mu t.\, (t{\to}\mathit{bool}){\to}\mathit{bool}$ is not well-formed, since $\mathit{bool} \equiv 1 + 1$ is not pointed, but $\mu t.\, (t_\perp{\to}\mathit{bool}){\to}\mathit{bool}$ is. The intuitive reason for this is that the latter type is isomorphic to $((\mu s.\, ((s{\to}\mathit{bool}){\to}\mathit{bool})_\perp){\to}\mathit{bool}){\to}\mathit{bool}$; the body of this modified recursive type is pointed, so by the reasoning in section 4.5 the necessary inductive limit in **CPO** may instead be computed in $\mathbf{CPO}^R$, where it is easy to show that the limit

[1] This is another reason why we do not recognize types such as $\sigma_\perp + 0$ as pointed. If $F(t) = \sigma_\perp + t$, then $F(0)$ would be pointed, although $F^2(0)$, $F^3(0)$, ... would not. The intuition we prefer is that 0 is the canonical unpointed type, in the sense that if $F(0)$ is pointed then $F(\tau)$ is pointed for all τ, *i.e.*, F is unconditionally pointed. This is supported by the first lemma below.

actually gives a fixed point. Note that, since the only pointed types in $\lambda^{\mu\nu}$ are isomorphic to 1 (this may be proved by induction on the type formation rules, or by observing that the types of $\lambda^{\mu\nu}$ may be modelled by sets and, up to isomorphism, 1 is the only pointed set), we did not lose any power in the previous chapter by restricting the bodies of inductive and projective types to be strictly positive in the bound variable. Also note that a retractive type will always be pointed, because the condition on the body for it to be well-formed is the same as the condition for it to be pointed.

These rules for type formation may be summarized by the inference system in Table 4.1. There are two kinds of judgement; the first, of the form $\Gamma \triangleright \texttt{type}\ \sigma$, asserts that σ is a well-formed type expression given the context Γ; the second, of the form $\Gamma \triangleright \texttt{ptd}\ \sigma$, asserts that σ is also pointed with respect to the context Γ. A context is a set of assumptions of the form $*\ t$, where $*$ may be either `type` or `ptd`; thus the judgement $\texttt{type}\ s, \texttt{ptd}\ t \triangleright \texttt{ptd}\ s{\rightarrow}t$ asserts (correctly) that the well-formed type expression $s{\rightarrow}t$ is conditionally pointed with respect to t. If a judgement $\emptyset \triangleright \texttt{type}\ \sigma$ is provable, then σ must be a well-formed type, *i.e.*, a closed type expression; similarly, if $\emptyset \triangleright \texttt{ptd}\ \sigma$ is provable then σ is a pointed type. The side condition $(*)$ on the μ and ν rules is that σ must be covariant with respect to t, and each occurrence of t on the left of an arrow must be contained in a subterm τ of σ such that $\Gamma, \texttt{type}\ t \triangleright \texttt{ptd}\ \tau$ is provable.

We may now confirm our intuition that 0 is in a sense canonical among unpointed types:

Lemma 4.1.1 *If* $\emptyset \triangleright \texttt{ptd}\ F(0)$ *is provable, then so is* $\texttt{type}\ t \triangleright \texttt{ptd}\ F(t)$*; therefore,* F *is unconditionally pointed iff* $F(0)$ *is pointed.*

Proof. From a proof of $\emptyset \triangleright \texttt{ptd}\ F(0)$ we may easily construct the desired proof of $\texttt{type}\ t \triangleright \texttt{ptd}\ F(t)$ by replacing each occurrence of the $(:0)$ axiom with the instance $\texttt{type}\ t \triangleright \texttt{type}\ t$ of the $(:\mathit{var})$ axiom, then renaming bound type variables away from t and adding $\texttt{type}\ t$ to contexts as necessary. ∎

Now that we have extended the types of $\lambda^{\mu\nu}$, we will describe the term formation rules corresponding to the additional types. For a retractive type ρF, we have terms

$(:var)$	$* \ t \triangleright \texttt{type} \ t$	$\texttt{ptd} \ t \triangleright \texttt{ptd} \ t$	$(.var)$
$(:add \ var)$	$\dfrac{\Gamma \triangleright \texttt{type} \ \sigma}{\Gamma, * \ t \triangleright \texttt{type} \ \sigma}$	$\dfrac{\Gamma \triangleright \texttt{ptd} \ \sigma}{\Gamma, * \ t \triangleright \texttt{ptd} \ \sigma}$	$(.add \ var)$
$(:0)$	$\emptyset \triangleright \texttt{type} \ 0$		
$(:1)$	$\emptyset \triangleright \texttt{type} \ 1$	$\emptyset \triangleright \texttt{ptd} \ 1$	$(.1)$
$(:+)$	$\dfrac{\Gamma \triangleright \texttt{type} \ \sigma, \quad \Gamma \triangleright \texttt{type} \ \tau}{\Gamma \triangleright \texttt{type} \ \sigma + \tau}$		
$(:\times)$	$\dfrac{\Gamma \triangleright \texttt{type} \ \sigma, \quad \Gamma \triangleright \texttt{type} \ \tau}{\Gamma \triangleright \texttt{type} \ \sigma \times \tau}$	$\dfrac{\Gamma \triangleright \texttt{ptd} \ \sigma, \quad \Gamma \triangleright \texttt{ptd} \ \tau}{\Gamma \triangleright \texttt{ptd} \ \sigma \times \tau}$	$(.\times)$
$(:\rightarrow)$	$\dfrac{\Gamma \triangleright \texttt{type} \ \sigma, \quad \Gamma \triangleright \texttt{type} \ \tau}{\Gamma \triangleright \texttt{type} \ \sigma \rightarrow \tau}$	$\dfrac{\Gamma \triangleright \texttt{type} \ \sigma, \quad \Gamma \triangleright \texttt{ptd} \ \tau}{\Gamma \triangleright \texttt{ptd} \ \sigma \rightarrow \tau}$	$(.\rightarrow)$
$(:\perp)$	$\dfrac{\Gamma \triangleright \texttt{type} \ \sigma}{\Gamma \triangleright \texttt{type} \ \sigma_\perp}$	$\dfrac{\Gamma \triangleright \texttt{type} \ \sigma}{\Gamma \triangleright \texttt{ptd} \ \sigma_\perp}$	$(.\perp)$
$(:\mu)$	$\dfrac{\Gamma, \texttt{type} \ t \triangleright \texttt{type} \ \sigma}{\Gamma \triangleright \texttt{type} \ \mu t.\, \sigma}(*)$	$\dfrac{\Gamma, \texttt{type} \ t \triangleright \texttt{ptd} \ \sigma}{\Gamma \triangleright \texttt{ptd} \ \mu t.\, \sigma}(*)$	$(.\mu)$
$(:\nu)$	$\dfrac{\Gamma, * \ t \triangleright \texttt{type} \ \sigma}{\Gamma \triangleright \texttt{type} \ \nu t.\, \sigma}(*)$	$\dfrac{\Gamma, \texttt{ptd} \ t \triangleright \texttt{ptd} \ \sigma}{\Gamma \triangleright \texttt{ptd} \ \nu t.\, \sigma}(*)$	$(.\nu)$
$(:\rho)$	$\dfrac{\Gamma, \texttt{ptd} \ t \triangleright \texttt{ptd} \ \sigma}{\Gamma \triangleright \texttt{type} \ \rho t.\, \sigma}$	$\dfrac{\Gamma, \texttt{ptd} \ t \triangleright \texttt{ptd} \ \sigma}{\Gamma \triangleright \texttt{ptd} \ \rho t.\, \sigma}$	$(.\rho)$

Table 4.1: Type formation and pointedness rules

which give the two directions of the isomorphism $\rho F \simeq F(\rho F)$:

(ρ *Intro*) $$\emptyset \triangleright \mathit{fold}_{\rho F} \colon F(\rho F) \to \rho F$$

(ρ *Elim*) $$\emptyset \triangleright \mathit{unfold}_{\rho F} \colon \rho F \to F(\rho F).$$

We could also introduce terms analogous to $\mathit{it}_{\mu F}$ and $\mathit{new}_{\nu F}$, representing the fact that a retractive type is both an initial F-algebra and a terminal F-coalgebra; however, the additional apparatus needed to define these terms is prohibitive — *e.g.*, since ρF is an initial F-algebra only in $\mathbf{CPPO}^R$, the subcategory of pointed cpo's and retractions, we would first have to prove that $M \colon F(\tau) \to \tau$ was a retraction before constructing the term $\mathit{it}^{\tau}_{\rho F}\, M \colon \rho F \to \tau$. We avoid this complexity at the expense of obtaining a less powerful proof system which does not directly reflect the categorical interpretation. The fact that a retractive type is a fixed point of F is sufficient to define a term $\mathit{fix}^{\sigma} \colon (\sigma \to \sigma) \to \sigma$ which acts as a fixed point operator on terms, *i.e.*, $\mathit{fix}^{\sigma} M \twoheadrightarrow M(\mathit{fix}^{\sigma} M)$ for any term $M \colon \sigma \to \sigma$. Using fix^{σ} we may define $\mathit{it}^{\tau}_{\rho F}\, M$ and $\mathit{new}^{\tau}_{\rho F}\, M$ as desired; the cost of being able to do this is that we must add a proof rule for fixed point induction to (partially) substitute for the missing extensional rules involving $\mathit{it}_{\rho F}$ and $\mathit{new}_{\rho F}$.

Here is the definition of the least fixed point operator using retractive types. The definition we use is based on Turing's combinator Θ (see [1], for example):

$$(\lambda x f.\, f(x x f))(\lambda x f.\, f(x x f)).$$

This untyped term may be typed if x has the retractive type $\upsilon \equiv \rho t.\, (t \to (\sigma \to \sigma) \to \sigma)$, where f will have type $\sigma \to \sigma$ for some pointed type σ, and t does not occur free in σ. The self-application of x thus becomes well-formed when it is written $\mathit{unfold}_{\upsilon}\, x x$. Therefore, we make the following definition for $\mathit{fix}^{\sigma} \colon (\sigma \to \sigma) \to \sigma$:

$$(\lambda x \colon \upsilon.\, \lambda f \colon \sigma \to \sigma.\, f(\mathit{unfold}_{\upsilon}\, x x f))(\mathit{fold}_{\upsilon}(\lambda x \colon \upsilon.\, \lambda f \colon \sigma \to \sigma.\, f(\mathit{unfold}_{\upsilon}\, x x f))).$$

To motivate the terms for the lifted type $\sigma_{\perp}$, we first consider the subcategory $\mathbf{CPPO}_{\perp}$ of pointed cpo's and strict functions. If X is a pointed cpo, then it must

have a bottom element, $\perp^X$. The strict functions from X to Y are those which preserve bottom, *i.e.*, $f: X \to Y$ is strict if $f(\perp^X) = \perp^Y$. If we let $L: \mathbf{CPO} \to \mathbf{CPPO}_\perp$ be the functor such that $L(X)$ is the pointed cpo X with a bottom element added and $L(f: X \to Y)$ is the function f extended by $f(\perp^{L(X)}) = \perp^{L(Y)}$, then there is an isomorphism φ from the homset $\mathbf{CPPO}_\perp(L(X), Y)$ to $\mathbf{CPO}(X, U(Y))$, where $U: \mathbf{CPPO}_\perp \to \mathbf{CPO}$ is the obvious forgetful functor. That is, given any strict function $f: L(X) \to Y$, there is a unique function $\varphi(f): X \to U(Y)$, obtained by dropping the bottom element from the domain; also, given any function $g: X \to U(Y)$ there is a unique strict function $\varphi^{-1}(g): L(X) \to Y$, obtained by adding a bottom element to the domain and requiring that $\varphi^{-1}(g)$ map it to the bottom of Y (which must be pointed because it is an object of $\mathbf{CPPO}_\perp$). This isomorphism is natural in X and Y, hence we have an adjunction $L \dashv U$.

Lifting arises from this adjunction by taking the interpretation of $\sigma_\perp$ to be the application of the endofunctor UL to the object corresponding to σ. One way of describing the adjunction $L \dashv U$ is by giving a natural transformation $\eta: \mathbf{CPO} \dot{\to} UL$, the *unit* of the adjunction, such that each arrow $\eta_X: X \to UL(X)$ is universal from X to U, *i.e.*, for any other arrow $f: X \to U(Y)$ there is a unique arrow $g: L(X) \to Y$ such that $f = U(g) \circ \eta_X$ (see [32], for example). This arrow g is the strict function $\varphi^{-1}(f)$ described above.

In general, an adjunction $L \dashv U$ for functors $L: \mathcal{C} \to \mathcal{D}$ and $U: \mathcal{D} \to \mathcal{C}$ creates a monad in $\mathcal{C}$. The monad endofunctor T is given by UL, the unit $\eta: \mathcal{C} \dot{\to} T$ is just the unit of the adjunction, and the multiplication $\mu: TT \dot{\to} T$ is $U\varepsilon L$, where $\varepsilon: LU \dot{\to} \mathcal{D}$ is the counit of the adjunction; details may be found in [31, 32], for example. For our purposes, it will be more useful to consider an alternate description of a monad, the *Kleisli triple*, which is discussed in [37]. Instead of the multiplication μ, a Kleisli triple provides an arrow $f^*: TX \to TY$ for every arrow $f: X \to TY$ of $\mathcal{C}$. The multiplication μ_X may then be obtained from $(id^{TX})^*$; conversely, the arrow f^* may be defined in terms of μ as $\mu_Y \circ Tf$. In terms of the adjunction description above, the arrow f^* is simply the arrow $U(\varphi^{-1}(f))$ found by the universality of η_X.

A monad $(T, \eta, ^*)$ may be represented in our language by the following pair of

term formation rules:

$$(T\ \textit{Intro}) \qquad \frac{\Gamma \triangleright M\colon \sigma}{\Gamma \triangleright \lfloor M \rfloor\colon T\sigma}$$

$$(T\ \textit{Elim}) \qquad \frac{\Gamma, x\colon \sigma \triangleright M\colon T\tau}{\Gamma \triangleright (\lambda \lfloor x\colon \sigma \rfloor .\, M)\colon T\sigma \to T\tau}$$

The introduction rule represents composition with η, and the elimination rule is the operation of constructing f^* from f.[2] There is as usual some freedom about whether to express functions externally via a free variable or internally via a binding operator; we have chosen the form here because it fits in well with our pattern-matching syntax (although in this case the term $(\lambda \lfloor x\colon \sigma \rfloor .\, M)$ is part of the basic syntax instead of syntactic sugar). Indeed, we will add $\lfloor \mathcal{P} \rfloor$ to our list of patterns, so that, for example, `let` $\lfloor x\colon \sigma \rfloor = M$ `in` N is an abbreviation for $(\lambda \lfloor x\colon \sigma \rfloor .\, N)M$. This `let` syntax is essentially the same as that used by Moggi for his computational lambda calculus [37].

In the special case that the category $\mathcal{D}$ is a subcategory of $\mathcal{C}$ and the functor $U\colon \mathcal{D} \to \mathcal{C}$ is the inclusion, then we may regard the objects and arrows of $\mathcal{D}$ as being objects and arrows of $\mathcal{C}$ with some special property (namely, that they belong to the subcategory). If we have judgements of the form $\triangleright$ "σ is in $\mathcal{D}$" and $\Gamma \triangleright$ "$M\colon \sigma \to \tau$ is in $\mathcal{D}$", then we may use this extra information to obtain a more expressive representation of the adjunction $L \dashv U$. We will have the same formation rule for the unit η:

$$(L\ \textit{Intro}) \qquad \frac{\Gamma \triangleright M\colon \sigma}{\Gamma \triangleright \lfloor M \rfloor\colon L\sigma.}$$

For the elimination rule, however, we may start with any arrow $f\colon X \to UY$, for Y an

[2] Actually, because the rest of the variables in the context Γ are not affected by the elimination rule, this correspondence only holds if we have a *strong* monad. A monad is strong if there is a natural transformation t, the *tensorial strength*, such that the components $t_{X,Y}\colon X \times TY \to T(X \times Y)$ satisfy certain equations. Thus, if the interpretation of $\Gamma, x\colon \sigma \triangleright M\colon T\tau$ is the arrow $f\colon \gamma \times \sigma \to T\tau$, then the correct interpretation of $\Gamma \triangleright (\lambda \lfloor x\colon \sigma \rfloor .\, M)\colon T\sigma \to T\tau$ is the arrow $\Lambda(f^* \circ t_{\gamma,\sigma})\colon \gamma \to (T\sigma \to T\tau)$, where Λ is the abstraction map. See [37] for more details.

object of $\mathcal{D}$, and construct the arrow $\varphi^{-1}(f)\colon LX \to Y$, which is actually an arrow in $\mathcal{D}$:

$$(L\ \textit{Elim}) \qquad \frac{\Gamma, x\colon\sigma \rhd M\colon\tau, \quad \rhd \text{“}\tau \text{ is in } \mathcal{D}\text{”}}{\Gamma \rhd (\lambda\lfloor x\colon\sigma\rfloor.\, M)\colon L\sigma\to\tau.}$$

We should thus include in the rules for these special judgements the facts that “$L\sigma$ is in $\mathcal{D}$” for all types σ, and “$(\lambda\lfloor x\colon\sigma\rfloor.\, M)\colon L\sigma\to\tau$ is in $\mathcal{D}$”.

We are finally ready to introduce the term formation rules for lifting. The subcategory $\mathcal{D}$ is $\mathbf{CPPO}_\perp$, so the judgement saying that a type σ is in $\mathcal{D}$ becomes the side-condition that σ is pointed. The judgement that a term $M\colon\sigma\to\tau$ is in $\mathcal{D}$ becomes the statement that M is strict. To be able to reason about strict functions in the proof system, we will need a term constant at each pointed type giving the bottom element of that type:

$$(\textit{bottom}) \qquad \frac{\emptyset \rhd \mathtt{ptd}\ \ \sigma}{\emptyset \rhd \perp^\sigma\colon\sigma.}$$

The natural transformation η is then represented in $\lambda^{\perp\rho}$ by the term formation rule

$$(\perp\ \textit{Intro}) \qquad \frac{\Gamma \rhd M\colon\sigma}{\Gamma \rhd \lfloor M\rfloor\colon\sigma_\perp.}$$

The term demonstrating the universality of η is given by the rule

$$(\perp\ \textit{Elim}) \qquad \frac{\Gamma, x\colon\sigma \rhd M\colon\tau, \quad \emptyset \rhd \mathtt{ptd}\ \ \tau}{\Gamma \rhd (\lambda\lfloor x\colon\sigma\rfloor.\, M)\colon\sigma_\perp\to\tau;}$$

when we present the equational proof system in the next section, the statement that this abstraction is strict will be given by the axiom

$$\Gamma \rhd (\lambda\lfloor x\colon\sigma\rfloor.\, M)\perp^{\sigma_\perp} = \perp^\tau : \tau.$$

In comparing our calculus to Moggi’s Computational Lambda Calculus, we note

that our treatment of lifting differs from Moggi's in two important respects. First, because we are dealing with the specific operation of lifting instead of an arbitrary monad of computations, we will get more terms and more provable equations. Specifically, by introducing strict functions and defining lifting as an adjunction, we will find some equations that hold for all pointed types, whereas if we only defined lifting as a monad they would only hold for lifted types. The second difference is that in Moggi's system, all functions return values of the computation type; we require lifting to be indicated more explicitly, so that functions are expressible which do not represent computations of the lifted type (indeed, the entire sublanguage $\lambda^{\mu\nu}$ is concerned with defining functions which do not return lifted types, because they always terminate).

Now we may extend the definition of $F(M)$ to cover lifted and retractive types:

- if $F(t) = G(t)_\perp$, then $F(M) = (\lambda\lfloor x{:}\, G(\sigma)\rfloor.\, \lfloor G(M)x\rfloor)$;
- if $F(t) = \rho s.\, G(s,t)$, then
  $F(M) = \mathit{fix}^{F(\sigma)\to F(\tau)}(\lambda f{:}\, F(\sigma)\to F(\tau).\, \mathit{fold}_{\rho s.\, G(s,\tau)} \circ G(f,M) \circ \mathit{unfold}_{\rho s.\, G(s,\sigma)})$.

Note that, because t must occur covariantly in $F(t)$, the type expression $G(s,t)$ in the retractive case must be covariant in both s and t. The term $G(f,M)$ was discussed in the proof that "functor" application preserves composition; it is equal to both $G(f,\tau) \circ G(F(\sigma),M)$ and $G(F(\tau),M) \circ G(f,\sigma)$, by the usual definition of the application of a bifunctor.

4.2 Equational proof system for $\lambda^{\perp\rho}$

We will extend the proof system given in the previous chapter for $\lambda^{\mu\nu}$ to provide a formal semantics for the types and terms just introduced. In addition to proving equations between terms, we will also need the auxiliary concept of an approximation ordering among terms. The categorical interpretation of this ordering is that we have a partial order structure on the homsets, reflecting the notion of increasing information content among the functions. A category whose homsets are enriched in this way is known as an $\mathcal{O}$-category ([57]; see also [51]). For example, among functions in a given homset whose codomain is pointed, the constant bottom function will be less

defined than all other functions; it is the least element of the homset. We will write $\Gamma \triangleright M \leq N : \sigma$ for the statement that the term $\Gamma \triangleright M{:}\,\sigma$ is less than or equal to $\Gamma \triangleright N{:}\,\sigma$, *i.e.*, that the arrow from γ to σ corresponding to M is less defined than that corresponding to N.

Just as for equality, we need some structural rules for approximation. Note that we need an extra "congruence" rule for strict abstraction, since it is a new binding operator. We do not need to go back and introduce a corresponding equality rule, since we may derive it from ($\leq$ *str abs*) by using ($\leq$ *eq*) and (*sym*) to split the hypothesis into two approximations, applying the abstraction rule, then recombining the approximations into the desired equality with ($\leq$ *antisym*).

($\leq$ *eq*)
$$\frac{\Gamma \triangleright M = N : \sigma}{\Gamma \triangleright M \leq N : \sigma}$$

($\leq$ *antisym*)
$$\frac{\Gamma \triangleright M \leq N : \sigma, \quad \Gamma \triangleright N \leq M : \sigma}{\Gamma \triangleright M = N : \sigma}$$

($\leq$ *trans*)
$$\frac{\Gamma \triangleright M \leq N : \sigma, \quad \Gamma \triangleright N \leq P : \sigma}{\Gamma \triangleright M \leq P : \sigma}$$

($\leq$ *abs*)
$$\frac{\Gamma, x{:}\,\sigma \triangleright M \leq N : \tau}{\Gamma \triangleright (\lambda x{:}\,\sigma.\, M) \leq (\lambda x{:}\,\sigma.\, N) : \sigma \to \tau}$$

($\leq$ *str abs*)
$$\frac{\Gamma, x{:}\,\sigma \triangleright M \leq N : \tau}{\Gamma \triangleright (\lambda \lfloor x{:}\,\sigma \rfloor.\, M) \leq (\lambda \lfloor x{:}\,\sigma \rfloor.\, N) : \sigma_\perp \to \tau} \text{ if } \tau \text{ is pointed}$$

($\leq$ *app*)
$$\frac{\Gamma \triangleright M \leq N : \sigma \to \tau, \quad \Gamma \triangleright P \leq Q : \sigma}{\Gamma \triangleright MP \leq NQ : \tau}$$

($\leq$ *add var*)
$$\frac{\Gamma \triangleright M \leq N : \sigma}{\Gamma, x{:}\,\tau \triangleright M \leq N : \sigma}$$

We may use these rules and the previous equational axioms and rules to establish the following analogue to the substitution lemma:

Lemma 4.2.1 *If the approximations* $\Gamma, x{:}\sigma \rhd M \leq N : \tau$ *and* $\Gamma \rhd P \leq Q : \sigma$ *are provable, then so is* $\Gamma \rhd \{P/x\}M \leq \{Q/x\}N : \tau$.

Proof. The proof is identical to the equational version in the previous chapter, with the addition of two applications of the $(\leq\ eq)$ rule to establish that $\Gamma \rhd \{P/x\}M \leq (\lambda x{:}\sigma.\, M)P : \tau$ and $\Gamma \rhd (\lambda x{:}\sigma.\, N)Q \leq \{Q/x\}N : \tau$ are both true. ∎

The two remaining rules for approximation establish some of the properties of recursively defined functions. Of course, we cannot expect to obtain a proof system which is complete for reasoning about equality between terms (or even programs) of $\lambda^{\perp\rho}$, since all partial recursive functions are representable in the language. Equality of partial recursive functions is not recursively enumerable, so we can not have a complete proof system.

The first of the remaining approximation rules asserts that $\perp^\sigma$ is the least element of pointed type σ:

$$(\perp \leq) \qquad \frac{\emptyset \rhd \mathtt{ptd}\ \sigma}{\Gamma \rhd \perp^\sigma \leq M : \sigma.}$$

The other rule is a form of Scott's fixed point induction ([50]; see also [33]). We only consider the special case where the inclusive predicate is an approximation. This is the most complex inference rule of the proof system; the second hypothesis should be read as saying that if the approximation $\Gamma \rhd Pc \leq Qc : \tau$ is provable for some fresh term constant c of type σ, then so is $\Gamma \rhd P(Mc) \leq Q(Mc) : \tau$. Since c must be a previously uninterpreted constant, this is proof-theoretically equivalent to the (non-finitary) hypothesis that $\Gamma \rhd PN \leq QN : \tau$ implies $\Gamma \rhd P(MN) \leq Q(MN) : \tau$, for every term $\Gamma \rhd N{:}\sigma$.

$$(\mathit{fpind}) \qquad \frac{\Gamma \rhd P\perp^\sigma \leq Q\perp^\sigma : \tau, \quad \left(\dfrac{\Gamma \rhd Pc \leq Qc : \tau}{\Gamma \rhd P(Mc) \leq Q(Mc) : \tau}\right)}{\Gamma \rhd P(\mathit{fix}^\sigma M) \leq Q(\mathit{fix}^\sigma M) : \tau} \ c \text{ fresh}$$

This rule is somewhat unsatisfactory for two reasons. First is its complexity; however, it is a common rule used for proving facts about fixed points, and the examples below

will make frequent use of it in ways that no simpler rule we tried could handle. Second, it is the only major rule without a direct categorical motivation; indeed, since it asserts a fact about fix^σ, which is only a defined term in $\lambda^{\perp\rho}$, it seems quite *ad hoc*. It would be more satisfying to derive this rule from a rule expressing the fact that a retractive type is an initial algebra, but as we have already noted, there are technical problems with this approach which we have not yet solved. The price we will pay for using (*fpind*) is that we have not been able to prove some equations about retractive types that would have followed easily from initiality; for example, we do not know how to use fixed point induction to show that $F(id) = id$, when F contains a retractive type (see below). The recent work of Crole and Pitts on their FIX-Logic [12] may offer a solution to this difficulty, although we have not yet been able to work out the precise relation between their system and ours.

To finish the proof system for $\lambda^{\perp\rho}$, we need to present the (β) and (η) rules for the new types. For a retractive type ρF, we simply take these to be the two equations establishing that $\mathit{fold}_{\rho F}$ and $\mathit{unfold}_{\rho F}$ are inverses:

$$(\rho\beta) \qquad \emptyset \rhd \mathit{unfold}_{\rho F} \circ \mathit{fold}_{\rho F} = id^{F(\rho F)} : F(\rho F) \to F(\rho F)$$

$$(\rho\eta) \qquad \emptyset \rhd \mathit{fold}_{\rho F} \circ \mathit{unfold}_{\rho F} = id^{\rho F} : \rho F \to \rho F.$$

In terms of our general discussion of monads in the previous section, the rules for an arbitrary monad $(T, \eta, {}^*)$ would be

$$(T\beta) \qquad \Gamma \rhd (\lambda\lfloor x{:}\,\sigma\rfloor.\, M)\lfloor N\rfloor = \{N/x\}M : T\tau$$

$$(T\eta) \qquad \Gamma \rhd (\lambda\lfloor x{:}\,\sigma\rfloor.\, \lfloor x\rfloor) = id^{T\sigma} : T\sigma \to T\sigma$$

$$(T\eta') \quad \Gamma \rhd (\lambda\lfloor y{:}\,\tau\rfloor.\, N) \circ (\lambda\lfloor x{:}\,\sigma\rfloor.\, M) = (\lambda\lfloor x{:}\,\sigma\rfloor.\, (\lambda\lfloor y{:}\,\tau\rfloor.\, N)M) : T\sigma \to T\upsilon;$$

these are a direct translation of the equations $f^* \circ \eta_X = f$, $\eta_X^* = id^{TX}$, and $g^* \circ f^* = (g^* \circ f)^*$ for a Kleisli triple, as found in [37].

In the special case that we have an adjunction $L \dashv U$ where $U{:}\,\mathcal{D} \to \mathcal{C}$ is an inclusion functor, we may improve on the monad equations by using information about the

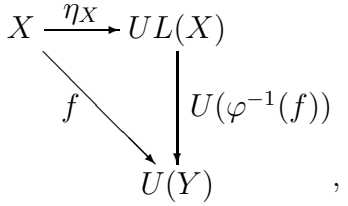
objects and arrows of $\mathcal{D}$:

$$(L\beta) \qquad \Gamma \triangleright (\lambda\lfloor x{:}\,\sigma\rfloor.\, M)\lfloor N\rfloor = \{N/x\}M : \tau$$

$$(L\beta') \qquad \Gamma \triangleright \text{“}(\lambda\lfloor x{:}\,\sigma\rfloor.\, M){:}\, L\sigma{\to}\tau \text{ is in } \mathcal{D}\text{”}$$

$$(L\eta) \qquad \frac{\Gamma \triangleright \text{“}M{:}\, L\sigma{\to}\tau \text{ is in } \mathcal{D}\text{”}}{\Gamma \triangleright (\lambda\lfloor x{:}\,\sigma\rfloor.\, M\lfloor x\rfloor) = M : L\sigma{\to}\tau} \text{ for } x \notin \mathrm{FV}(M).$$

The $(L\beta)$ axiom asserts the commutativity of the diagram



,

whereas the $(L\eta)$ rule establishes that $\varphi^{-1}(f)$ is the unique arrow which makes the diagram commute; this is just the statement discussed above that η_X is universal from X to U. The hypothesis on the $(L\eta)$ rule is necessary because the arrow represented by M needs to be an arrow in $\mathcal{D}$. Similarly, we also need the $(L\beta')$ rule to establish that the term representing $\varphi^{-1}(f)$ is in $\mathcal{D}$. With the additional knowledge that the subcategory $\mathcal{D}$ must be closed under formation of identities and composition, *i.e.*, id^σ is in $\mathcal{D}$ if σ is in $\mathcal{D}$ and $M \circ N$ is in $\mathcal{D}$ if M and N are in $\mathcal{D}$, it is easy to show that these rules for L imply those given for T above. For example, since $id^{T\sigma}$ is in $\mathcal{D}$, we find using $(L\eta)$ that $(\lambda\lfloor x{:}\,\sigma\rfloor.\, \lfloor x\rfloor) = (\lambda\lfloor x{:}\,\sigma\rfloor.\, id^{T\sigma}\lfloor x\rfloor) = id^{T\sigma}$, which is $(T\eta)$.

The particular instance of these rules for lifting leads to the following:

$$(\bot\beta) \qquad \Gamma \triangleright (\lambda\lfloor x{:}\,\sigma\rfloor.\, M)\lfloor N\rfloor = \{N/x\}M : \tau$$

$$(\bot\beta') \qquad \Gamma \triangleright (\lambda\lfloor x{:}\,\sigma\rfloor.\, M)\bot^{\sigma_\bot} = \bot^\tau : \tau$$

$$(\bot\eta) \qquad \frac{\Gamma \triangleright M\bot^{\sigma_\bot} = \bot^\tau : \tau}{\Gamma \triangleright (\lambda\lfloor x{:}\,\sigma\rfloor.\, M\lfloor x\rfloor) = M : \sigma_\bot{\to}\tau} \text{ for } x \notin \mathrm{FV}(M).$$

However, after some experience using these rules we discover that they are not quite

strong enough to represent our intuition about lifting. In particular, they do not seem to reflect the desired property that the only elements of the lifted type $\sigma_\perp$ are the elements of σ plus bottom. Therefore, we will use the following more powerful inference rule in place of $(\perp\eta)$:

$$(\perp ext) \qquad \frac{\Gamma \rhd M\perp^{\sigma_\perp} = N\perp^{\sigma_\perp} : \tau, \quad \Gamma, x{:}\sigma \rhd M\lfloor x\rfloor = N\lfloor x\rfloor : \tau}{\Gamma \rhd M = N : \sigma_\perp{\to}\tau.}$$

It is easy to see that taking $N \equiv (\lambda\lfloor x{:}\sigma\rfloor.\, M\lfloor x\rfloor)$ in this rule lets us derive $(\perp\eta)$.

Two equations which intuitively hold for the lifting monad and yet which do not seem provable without using $(\perp ext)$ were suggested to us by Eugenio Moggi:

$$\begin{array}{rcll} (\lambda\lfloor x{:}\sigma\rfloor.\,(\lambda\lfloor y{:}\sigma\rfloor.\,M)N)N & = & (\lambda\lfloor x{:}\sigma\rfloor.\,\{x/y\}M)N, & x \text{ not in } N \\ (\lambda\lfloor x{:}\sigma\rfloor.\,(\lambda\lfloor y{:}\tau\rfloor.\,M)N)P & = & (\lambda\lfloor y{:}\tau\rfloor.\,(\lambda\lfloor x{:}\sigma\rfloor.\,M)P)N, & x \text{ not in } N \\ & & & y \text{ not in } P \end{array}$$

These equations are not true in arbitrary monads. The first one says that, for the purposes of strict evaluation, repeating the evaluation step makes no difference (since if the argument is going to diverge then it will do so the first time); the second says that order of strict evaluation does not matter, as long as both N and P must be evaluated eventually. These are very easy to prove using the "reasoning by cases" made available by the $(\perp ext)$ rule; for example, the terms $(\lambda z{:}\sigma_\perp.\,(\lambda\lfloor x{:}\sigma\rfloor.\,(\lambda\lfloor y{:}\sigma\rfloor.\,M)z)z)$ and $(\lambda z{:}\sigma_\perp.\,(\lambda\lfloor x{:}\sigma\rfloor.\,\{x/y\}M)z)$ both are equal to $\perp^\tau$ when applied to $\perp^{\sigma_\perp}$, and similarly both are equal to $\{w/x\}\{w/y\}M$ when applied to $\lfloor w\rfloor$, hence they are equal by $(\perp ext)$; apply each to N to get the first equation. We would of course prefer it if the categorically motivated rules were enough to make such reasoning possible, but we have not managed to do this.

We may now prove several useful lemmas about the terms of $\lambda^{\perp\rho}$. As mentioned above, we are not able to prove that construction of the term $F(M)$ preserves identities or composition when $F(t)$ contains a retractive type which depends on t. The reason is that fixed point induction does not seem strong enough to prove that, for example, the identity function id^σ is the *least* fixed point of the functional

$(\lambda f{:}\,\sigma{\to}\sigma.\ \mathit{fold}_{\rho G} \circ G(f) \circ \mathit{unfold}_{\rho G})$, where G is a covariant functor which is assumed to preserve identities by the inductive hypothesis. However, for practical purposes this is not a problem. Since the only way $F(t)$ can contain a retractive type $\rho s.\, G(s,t)$ which depends on t is if G is covariant in both s and t, we will find in Section 4.5 that, at least with respect to creating isomorphic types in the cpo model, the retractive type may be replaced by the projective type $\nu s.\, G(s,t)$, for which we have the appropriate induction rule.

First of all, we will need to know how functor application interacts with approximation:

Lemma 4.2.2 *Functor application is* locally monotonic*, i.e., if the approximation* $\Gamma \triangleright M \leq N : \sigma{\to}\tau$ *is provable, then so is* $\Gamma \triangleright F(M) \leq F(N) : F(\sigma){\to}F(\tau)$.

Proof. We merely need to apply the substitution lemma for approximation to the equation $\Gamma, f{:}\,\sigma{\to}\tau \triangleright F(f) = F(f) : F(\sigma){\to}F(\tau)$. ∎

Now we may prove a partial extension of the fact that substituting a term in a type behaves like functor application:

Lemma 4.2.3 *If* $F(t)$ *does not contain any subterms of the form* $\rho s.\, \upsilon$*, where* t *occurs free in* υ*, then application of* F *to a term preserves composition and identities, i.e.,* $F(M \circ N) = F(M) \circ F(N)$ *and* $F(id) = id$.

Proof. Most of this lemma was proved in Section 3.2 as Lemma 3.2.3; we only need to consider the case where $F(t)$ is a lifted type expression. If $F(t) = G(t)_\perp$, then

$$\begin{aligned}
F(M) \circ F(N) &= F(M) \circ (\lambda \lfloor x{:}\,G(\sigma) \rfloor.\, \lfloor G(N)x \rfloor) \\
&= (\lambda \lfloor x{:}\,G(\sigma) \rfloor.\, F(M)((\lambda \lfloor x{:}\,G(\sigma) \rfloor.\, \lfloor G(N)x \rfloor)\lfloor x \rfloor)) \\
&= (\lambda \lfloor x{:}\,G(\sigma) \rfloor.\, (\lambda \lfloor y{:}\,G(\tau) \rfloor.\, \lfloor G(M)y \rfloor)\lfloor G(N)x \rfloor) \\
&= (\lambda x{:}\,G(\sigma).\, \lfloor G(M)(G(N)x) \rfloor) \\
&= F(M \circ N);
\end{aligned}$$

similarly, $F(id^\sigma) = (\lambda \lfloor x{:}\,G(\sigma) \rfloor.\, \lfloor x \rfloor) = id^{F(\sigma)}$. ∎

The impact on previous results of the restriction on the form of F in this lemma is that the proof of Lemma 3.2.4, which states that the defined terms for $\mathit{unfold}_{\mu F}$ and

$fold_{\nu F}$ are inverses for the primitives $fold_{\mu F}$ and $unfold_{\nu F}$, is only valid for functors of the restricted form. This provides another argument for including the inverses as primitives.

As an example of the kind of proof that is almost trivial given fixed point induction, we observe that we could have done away with the bottom constants $\bot^\sigma$ for each pointed type σ by defining them to be the least fixed points of the identity:

Lemma 4.2.4 *If σ is pointed, then $fix^\sigma\ id^\sigma = \bot^\sigma$.*

Proof. Since $id^\sigma\ \bot^\sigma \leq \bot^\sigma$ and, whenever $id^\sigma\ c \leq \bot^\sigma$, then also $id^\sigma(id^\sigma\ c) \leq \bot^\sigma$, it follows by (*fpind*) that $fix^\sigma\ id^\sigma = id^\sigma(fix^\sigma\ id^\sigma) \leq \bot^\sigma$. Using $(\bot \leq)$ we may change this approximation into an equality. ∎

We may easily obtain more information about the structure of the bottom elements for pointed types which do not contain any retractive types:

Lemma 4.2.5 *If all the necessary type expressions are pointed, then we may prove*

- $\bot^1 = \Diamond$;
- $\bot^{\sigma\times\tau} = \langle \bot^\sigma, \bot^\tau \rangle$;
- $\bot^{\sigma\to\tau} = (\lambda x{:}\sigma.\, \bot^\tau)$;
- $\bot^{\mu F} = fold_{\mu F}\, \bot^{F(\mu F)}$;
- $\bot^{\nu F} = new^1_{\nu F}\, \bot^{1\to F(1)} \bot^1$.

Proof. All the cases follow the general pattern of using the substitution lemma for approximation to build up a term which is equivalent by one of the extensional rules to the desired bottom.

- $\bot^1 = \Diamond$ is just an instance of the (1η) axiom.
- $\bot^\sigma \leq \pi_1 \bot^{\sigma\times\tau}$ and $\bot^\tau \leq \pi_2 \bot^{\sigma\times\tau}$, so by $(\leq \bot)$, substitution, and $(\times\eta)$ we find $\bot^{\sigma\times\tau} \leq \langle \bot^\sigma, \bot^\tau \rangle \leq \langle \pi_1 \bot^{\sigma\times\tau}, \pi_2 \bot^{\sigma\times\tau} \rangle = \bot^{\sigma\times\tau}$.
- $\bot^\tau \leq \bot^{\sigma\to\tau} x$, hence $\bot^{\sigma\to\tau} \leq (\lambda x{:}\sigma.\, \bot^\tau) \leq (\lambda x{:}\sigma.\, \bot^{\sigma\to\tau} x) = \bot^{\sigma\to\tau}$.

- $\bot^{F(\mu F)} \leq \mathit{unfold}_{\mu F}\, \bot^{\mu F}$, so similarly we find (using either $(\mu\eta')$ or, since F satisfies the restrictions mentioned above, Lemma 3.2.4) $\bot^{\mu F} \leq \mathit{fold}_{\mu F}\, \bot^{F(\mu F)} \leq \mathit{fold}_{\mu F}(\mathit{unfold}_{\mu F}\, \bot^{\mu F}) = \bot^{\mu F}$. Note that if we were to use this as a definition for $\bot^{\mu F}$ it would be well-founded, because we may show that, since F is unconditionally pointed, the definition of $\bot^{F(\mu F)}$ will not need the term $\bot^{\mu F}$. The same reasoning would not hold if we tried to define $\bot^{\nu F}$ as $\mathit{fold}_{\nu F}\, \bot^{F(\nu F)}$, or similarly for $\bot^{\rho F}$, because F need only be conditionally pointed; for example, $\nu t.\, t$ is a pointed type which would lead to the non-well-founded definition $\bot^{\nu t.\, t} = \mathit{fold}_{\nu t.\, t}\, \bot^{\nu t.\, t}$.

- We first note that $F(\bot^{1\to\nu F})$ is strict, because

$$\begin{aligned}
\bot^{F(\nu F)} &\leq F(\bot^{1\to\nu F}) \circ \bot^{F(1)} \\
&\leq (F(\bot^{1\to\nu F}) \circ F(\bot^{\nu F\to 1}))\bot^{F(\nu F)} \\
&\leq F(id^{\nu F})\bot^{F(\nu F)} \\
&= \bot^{F(\nu F)}.
\end{aligned}$$

  We may also show that $\mathit{unfold}_{\nu F}$ is strict, in the same way we proved that $\mathit{fold}_{\mu F}$ is strict in the previous item. Therefore, the equation $\mathit{unfold}_{\nu F} \circ \bot^{1\to\nu F} = F(\bot^{1\to\nu F}) \circ \bot^{1\to F(1)}$ is true because both sides are equal to the bottom function $\bot^{1\to F(\nu F)}$. This is the form we need to apply $(\nu\eta)$, resulting in $\bot^{1\to\nu F} = \mathit{new}^1_{\nu F}\, \bot^{1\to F(1)}$; applying both sides to $\bot^1$ and using the above characterization of a bottom of functional type yields the desired result.

■

In the interpretation of $\lambda^{\bot\rho}$ in the category **CPO**, we observe that application of the lifting functor to the initial object, *i.e.*, the empty cpo, yields a terminal object, *i.e.*, a cpo with exactly one element. It is an interesting consequence of our rules for lifting that this is true in all models of $\lambda^{\bot\rho}$:

Lemma 4.2.6 *The types* $0_\bot$ *and* 1 *are isomorphic.*

Proof. We will show that the terms $\bot^{0_\bot\to 1}$ and $\bot^{1\to 0_\bot}$ are inverses. Since both terms are strict, either way of composing them will result in a bottom function, hence we

only need to show that $\bot^{1\to 1} = id^1$ and $\bot^{0_\bot\to 0_\bot} = id^{0_\bot}$. The first equation is trivial, since by the (1η) rule, all terms of type $1\to 1$ are equal to $(\lambda\Diamond.\,\Diamond)$. For the second equation, since both sides are strict functions, if we can prove $x\colon 0 \triangleright \bot^{0_\bot\to 0_\bot}\lfloor x\rfloor = id^{0_\bot}\lfloor x\rfloor : 0_\bot$, then by using the (derived) strict abstraction congruence rule and $(\bot\eta)$ we are done. But this last equation is true, since for any term $x\colon 0 \triangleright M\colon\sigma$ we find by $(\to\beta)$ and (0η) that $M = (\lambda x\colon 0.\,M)x = \Box^\sigma x$ is true, *i.e.*, all such terms are equal. ∎

This isomorphism is an instance of a general condition for showing that lifting is a *monad with zero*, a concept introduced by Wadler [54]. The monad natural transformations $\eta\colon \mathcal{C} \dot{\to} T$ and $\mu\colon TT \dot{\to} T$ receive their names "unit" and "multiplication" in part because of an analogy with the corresponding concepts in a monoid; the diagrams which must commute for a monad may then be seen as statements that multiplication is associative and has the unit as an identity:

$$\begin{array}{ccc} TTTX & \xrightarrow{T\mu_X} & TTX \\ {\scriptstyle \mu_{TX}}\downarrow & & \downarrow{\scriptstyle \mu_X} \\ TTX & \xrightarrow[\mu_X]{} & TX \end{array} \qquad\qquad \begin{array}{ccccc} TX & \xrightarrow{\eta_{TX}} & TTX & \xleftarrow{T\eta_X} & TX \\ & {\scriptstyle id}\searrow & \downarrow{\scriptstyle \mu_X} & \swarrow{\scriptstyle id} & \\ & & TX & & \end{array} .$$

Wadler observed that many common monads also have zeroes with respect to multiplication. In a category with a terminal object, we may present a zero for a monad by giving a natural transformation $\zeta\colon 1 \dot{\to} T$ such that the following diagram commutes:

$$\begin{array}{ccccc} 1 & \xrightarrow{\zeta_{TX}} & TTX & \xleftarrow{T\zeta_X} & T1 \\ & {\scriptstyle \zeta_X}\searrow & \downarrow{\scriptstyle \mu_X} & \swarrow{\scriptstyle \zeta_X \circ \Diamond_{T1}} & \\ & & TX & & \end{array} ,$$

where $\Diamond_{T1}$ is the unique arrow from $T1$ to 1. For the lifting monad, it is easy to see that the bottom arrow from 1 to $X_\bot$ will act as a zero, since the multiplication μ_X collapses both the bottom element $\bot^{(X_\bot)_\bot}$ and the lifted bottom element $\lfloor\bot^{X_\bot}\rfloor$ down to the bottom of $X_\bot$.

This author and Michael Johnson [55] independently noticed the following special

case of this definition: if a category with a monad (T, η, μ) and a terminal object 1 also has an initial object 0 and if there is an isomorphism $\bot\colon 1 \to T0$, then the natural transformation whose X component is the arrow $T\Box_X \circ \bot\colon 1 \to TX$ will act as a zero for the monad. This follows by a simple diagram chase, using the uniqueness of the arrows $\Diamond_X\colon X \to 1$ and $\Box_X\colon 0 \to X$, plus the fact that $\Diamond_{T0}$ is an inverse for the given arrow $\bot$. The preceding lemma thus allows us to confirm the above observation about the zero of the lifting monad, since the term corresponding to $T\Box_X \circ \bot$ is $(\lambda\lfloor x\colon X\rfloor.\, \lfloor \Box^X x\rfloor) \circ \bot^{1\to 0_\bot}$, which is equal to $\bot^{1\to X_\bot}$ by the $(\bot\beta')$ rule.

4.3 The reduction system $\lambda_r^{\bot\rho}$

We may now consider an operational semantics for $\lambda^{\bot\rho}$, given as the obvious extension to that presented for $\lambda^{\mu\nu}$. The full reduction system $\lambda_r^{\bot\rho}$ will still be confluent, although it will of course no longer be normalizing. A notion of evaluating a term must therefore be relative to some strategy for choosing a reduction path. We will show that a lazy strategy is computationally adequate with respect to finding normal forms of programs.

As usual, we obtain the reduction rules from the equational proof system by directing the (β) axioms in the direction of decreasing complexity of terms. Here is the list of rules:

$$(+\beta_1)_r \qquad [M, N](\iota_1 P) \longrightarrow MP$$

$$(+\beta_2)_r \qquad [M, N](\iota_2 P) \longrightarrow NP$$

$$(\times\beta_1)_r \qquad \pi_1\langle M, N\rangle \longrightarrow M$$

$$(\times\beta_2)_r \qquad \pi_2\langle M, N\rangle \longrightarrow N$$

$$(\to\beta)_r \qquad (\lambda x\colon \sigma.\, M)N \longrightarrow \{N/x\}M$$

$$(\mu\beta)_r \qquad it_{\mu F}\, M(fold_{\mu F}\, P) \longrightarrow M(F(it_{\mu F}\, M)P)$$

$$(\nu\beta)_r \qquad unfold_{\nu F}(new_{\nu F}\, MP) \longrightarrow F(new_{\nu F}\, M)(MP)$$

$$(\mu\beta')_r \qquad unfold_{\mu F}(fold_{\mu F}\, M) \longrightarrow M$$

$$(\nu\beta')_r \qquad unfold_{\nu F}(fold_{\nu F}\, M) \longrightarrow M$$

$$(\rho\beta)_r \qquad unfold_{\rho F}(fold_{\rho F}\, M) \longrightarrow M$$

$$(\bot\beta)_r \qquad (\lambda\lfloor x{:}\,\sigma\rfloor.\, M)\lfloor N\rfloor \longrightarrow \{N/x\}M$$

$$(\bot\beta')_r \qquad (\lambda\lfloor x{:}\,\sigma\rfloor.\, M)\bot^{\sigma_\bot} \longrightarrow \bot^{\tau}.$$

As with $\lambda_r^{\mu\nu}$, we may leave the types and typing contexts implicit because we have a subject reduction lemma; intuitively, types are only need at compile time.

Lemma 4.3.1 (Subject Reduction) *If $\Gamma \triangleright M{:}\,\sigma$ is well-formed and $M \longrightarrow\!\!\!\!\rightarrow N$, then $\Gamma \triangleright N{:}\,\sigma$ is also well-formed.*

For the proof of confluence we will simply observe that $\lambda_r^{\bot\rho}$ is a regular combinatory reduction system (after erasing the types, which the previous lemma has shown are unnecessary for reduction). A combinatory reduction system (CRS), as introduced by Klop [28], is a generalization of term rewriting systems to include variable binding operators and substitution. Our system $\lambda_r^{\bot\rho}$ is a CRS with two binding forms, each of which binds one variable: $(\lambda x.\,\cdot)$ and $(\lambda\lfloor x\rfloor.\,\cdot)$. In fact it is a regular CRS, because it is left-linear (no meta-variable appears more than once on the left-hand side of a rule) and non-ambiguous (there are no critical pairs).

Theorem 4.3.2 (Confluence) *If $M \longrightarrow\!\!\!\!\rightarrow N$ and $M \longrightarrow\!\!\!\!\rightarrow P$, then there is a term Q such that $N \longrightarrow\!\!\!\!\rightarrow Q$ and $P \longrightarrow\!\!\!\!\rightarrow Q$.*

Proof. Immediate from [28], Theorem II.3.11. ∎

Klop also shows that if a regular CRS has the additional property that it is *left-normal*, then it satisfies a standardization theorem. Recall from Chapter 2 that the definition Klop gives for left-normal is that all of the constants (including binding operators) are to the left of the meta-variables in all of the reduction rules. The effect of this restriction is that if terms are evaluated from left to right, then all the necessary redexes will be created in an orderly fashion.

Our system $\lambda_r^{\bot\rho}$ as it stands is not left-normal; for example, in the $(+\beta_1)_r$ rule,

the constant ι_1 appears to the right of the meta-variables M and N. To see concretely how this would affect a left-to-right reduction strategy, consider the term $[id, \mathit{fix}\ id](id(\iota_1\Diamond))$. If we employed a strategy of strict left-to-right evaluation, then the reduction would get hung in a loop trying to evaluate the right arm of the choice, never reaching the $(\rightarrow\beta)_r$ redex at the right, which must be evaluated to complete the $(+\beta_1)_r$ redex and reach the normal form $\Diamond$.

The non-left-normal rules are $(+\beta_1)_r$, $(+\beta_2)_r$, $(\mu\beta)_r$, $(\bot\beta)_r$, and $(\bot\beta')_r$. There are two easy ways to fix these and obtain a left-normal reduction system. Either we may rearrange the syntax of terms, so that these rules become left-normal by Klop's definition, or we may modify the meaning of "left" with respect to the ordering of subterms.

The first fix involves replacing the term constructors $[\cdot,\cdot]\cdot$, $\mathit{it}_{\mu F}\,\cdot\cdot$, and $(\lambda\lfloor\cdot\rfloor.\,\cdot)\cdot$ with the following, two of which we have used (slightly modified) as abbreviations:

$$(+\ \mathit{Elim}') \qquad \frac{\Gamma \triangleright M\colon \sigma + \tau, \quad \Gamma \triangleright P\colon \sigma\rightarrow\upsilon, \quad \Gamma \triangleright Q\colon \tau\rightarrow\upsilon}{\Gamma \triangleright \mathtt{case}\ M\ \mathtt{of}\ P, Q\colon \upsilon}$$

$$(\mu\ \mathit{Elim}') \qquad \frac{\Gamma \triangleright M\colon \mu F, \quad \Gamma \triangleright N\colon F(\tau)\rightarrow\tau}{\Gamma \triangleright \mathit{induct}^{\tau}_{\mu F} MN\colon \tau}$$

$$(\bot\ \mathit{Elim}') \qquad \frac{\Gamma \triangleright M\colon \sigma_\bot, \quad \Gamma, x\colon\sigma \triangleright N\colon \tau}{\Gamma \triangleright \mathtt{let}\ \lfloor x\colon\sigma\rfloor = M\ \mathtt{in}\ N\colon\tau}\ \tau \text{ pointed.}$$

The previously non-left-normal reduction rules now become

$$(+\beta_1)'_r \qquad \mathtt{case}\ \iota_1 N\ \mathtt{of}\ P, Q \longrightarrow PN$$

$$(+\beta_2)'_r \qquad \mathtt{case}\ \iota_2 N\ \mathtt{of}\ P, Q \longrightarrow QN$$

$$(\mu\beta)'_r \qquad \mathit{induct}_{\mu F}(\mathit{fold}_{\mu F}\, P)N \longrightarrow N(F(\lambda x\colon\mu F.\, \mathit{induct}_{\mu F} xN)P)$$

$$(\bot\beta)'_r \qquad \mathtt{let}\ \lfloor x\colon\sigma\rfloor = \lfloor P\rfloor\ \mathtt{in}\ N \longrightarrow \{P/x\}N$$

$$(\bot\beta')'_r \qquad \mathtt{let}\ \lfloor x\colon\sigma\rfloor = \bot\ \mathtt{in}\ N \longrightarrow \bot,$$

which are all left-normal.

The other solution is to modify the left-to-right order of evaluation by altering the definition of when one redex, R, is to the left of another, S, written $R \prec S$. The effect of this will be to direct reduction into the appropriate part of a potential redex (for example, the test P of a choice expression $[M, N]P$) so that the reduction does not get needlessly sidetracked. The definition of $R \prec S$ that we start with is that a subterm R of a term M (which we write $R \subseteq M$; if $R \not\equiv M$ then as usual we write $R \subset M$) is to the left of another subterm $S \subseteq M$ if one of the following conditions holds:

- $S \subset R$;
- $R \subseteq M_1$ and $S \subseteq M_2$ for some subterm $[M_1, M_2]$ of M;
- $R \subseteq M_1$ and $S \subseteq M_2$ for some subterm $\langle M_1, M_2 \rangle$ of M; or
- $R \subseteq M_1$ and $S \subseteq M_2$ for some subterm $M_1 M_2$ of M.

Our modification will be to the last clause, based on whether or not the term M_1 is a strict abstraction, choice, or iterated function:

- if $R \subseteq M_1$ and $S \subseteq M_2$ for some $M_1 M_2 \subseteq M$, then $R \prec S$ if $M_1 \not\equiv (\lambda \lfloor x{:}\,\sigma \rfloor .\, P)$, $[P, Q]$, or $it_{\mu F}\, P$, otherwise $S \prec R$.

Now, following Klop, we may define standard reductions and the process of standardization. If we have a reduction sequence $\mathcal{R}{:}\, M_0 \longrightarrow M_1 \longrightarrow \ldots$, then let us form a labelled sequence by the following process: if the redex contracted in the term M_i is R, then we label every redex $S \prec R$; descendents of labelled redexes keep their labels until they are themselves reduced. For example, here is the labelled form of a reduction from a term discussed above:

$$\begin{aligned}
[id, \mathit{fix}\ id](id(\iota_1 \diamond))^* &\longrightarrow\!\!\!\!\rightarrow\ [id, id(\mathit{fix}\ id)](id(\iota_1 \diamond))^* \\
&\longrightarrow\ ([id, id(\mathit{fix}\ id)](\iota_1 \diamond))^* \\
&\longrightarrow\ ([id, \mathit{fix}\ id](\iota_1 \diamond))^* \\
&\longrightarrow\ id\, \diamond \\
&\longrightarrow\ \diamond .
\end{aligned}$$

A *standard reduction* is one in which no labelled redexes are contracted. The above reduction is not standard because two labelled redexes contract, producing the second and fourth lines respectively. An equivalent standard reduction is

$$\begin{array}{rcl} [id, \mathit{fix}\ id](id(\iota_1\Diamond)) & \longrightarrow & [id, \mathit{fix}\ id](\iota_1\Diamond) \\ & \longrightarrow & id\,\Diamond \\ & \longrightarrow & \Diamond. \end{array}$$

An *anti-standard pair* of reduction steps is a reduction $\mathcal{R}_1 \colon M_0 \longrightarrow M_1 \longrightarrow M_2$ which is not standard. The process of *meta-reduction* is the replacement in a reduction $\mathcal{R}$ of an anti-standard pair $\mathcal{R}_1$ by an equivalent standard reduction $\mathcal{R}_2$ to obtain a new reduction $\mathcal{R}'$; we write $\mathcal{R} \Longrightarrow \mathcal{R}'$. For example, in the reduction sequence

$$\mathcal{R} \colon (\lambda x \colon \sigma.\ id\langle x, x\rangle)(id\,\Diamond) \longrightarrow (\lambda x \colon \sigma.\ id\langle x, x\rangle)\Diamond \longrightarrow id\langle\Diamond, \Diamond\rangle \longrightarrow \langle\Diamond, \Diamond\rangle,$$

the first two reduction steps form an anti-standard pair; the pair may be replaced by a standard reduction to obtain the reduction sequence

$$\begin{array}{rcl} \mathcal{R}' \colon (\lambda x \colon \sigma.\ id\langle x, x\rangle)(id\,\Diamond) & \longrightarrow & id\langle id\,\Diamond, id\,\Diamond\rangle \longrightarrow id\langle\Diamond, id\,\Diamond\rangle \\ & \longrightarrow & id\langle\Diamond, \Diamond\rangle \longrightarrow \langle\Diamond, \Diamond\rangle, \end{array}$$

which is intuitively closer to a standard reduction than $\mathcal{R}$. One can imagine how repeating this process of meta-reduction will eventually produce a standard reduction sequence.

This intuition is formalized by the following theorem:

Theorem 4.3.3 (Standardization) *For every $\lambda_r^{\perp\rho}$ reduction sequence $\mathcal{R} \colon M \longrightarrow\!\!\!\!\!\rightarrow N$ there is a standard reduction sequence $\mathcal{R}_{st} \colon M \longrightarrow\!\!\!\!\!\rightarrow N$, obtained as a normal form of meta-reduction.*

Proof. See [28], Section II.6.2.8. In fact, Klop does not give the proof for the fully general case of a left-normal regular combinatory reduction system, he only indicates that it is possible. However, a minor modification to $\lambda_r^{\perp\rho}$ will put it in

the form of a left-normal term rewriting system plus lambda abstraction, for which Klop does provide a full proof. The required change is that the binding operation of strict abstraction ($\lambda\lfloor x{:}\,\sigma\rfloor.\, M$) be replaced by the ordinary lambda abstraction ($\lambda z{:}\,\sigma_\perp.\, test^\tau_\sigma z(\lambda x{:}\,\sigma.\, M)$), where $test^\tau_\sigma{:}\,\sigma_\perp{\to}(\sigma{\to}\tau){\to}\tau$ is a new function constant. The action of $test^\tau_\sigma$ is given by the following reduction rules:

$$\begin{aligned} test^\tau_\sigma\lfloor M\rfloor N &\longrightarrow NM \\ test^\tau_\sigma\perp N &\longrightarrow \perp; \end{aligned}$$

that is, it strictly evaluates the first argument and then applies the second argument to the result. It is easy to see that this modified system will have essentially the same reduction behavior as $\lambda_r^{\perp\rho}$; in particular, the standard reductions in the two systems will be isomorphic. ∎

As a corollary we find that a leftmost strategy for $\lambda_r^{\perp\rho}$ is normalizing, either using the straight definition of "left" and the modified syntax or using the adjusted definition of "left" and the original syntax:

Corollary 4.3.4 (Normalization) *If there is a $\lambda_r^{\perp\rho}$ reduction $M \longrightarrow\!\!\!\!\!\rightarrow N$ where N is a normal form, then the reduction sequence starting at M in which the leftmost redex is contracted at each step will eventually reach N.*

Proof. By the Standardization theorem, there is a standard reduction $\mathcal{R}_{st}$ from M to N. Suppose that there is some step in $\mathcal{R}_{st}$ in which the contracted redex is not leftmost; then since the reduction is standard, no descendents of the redexes to the left can ever be contracted, nor can they be erased, hence N must contain uncontracted redexes, contradicting the fact that it is a normal form. Therefore $\mathcal{R}_{st}$ is the desired normal reduction sequence. ∎

Note that this result is weaker than the computational adequacy results of the previous two chapters. If there is some reduction from a term to a normal form then we are guaranteed to find it, but there is no guarantee that the reduction system is strong enough to produce a normal form whenever the initial term is provably equal to one. Indeed, consider the term $fix^\sigma\ id^\sigma$; it is provably equal to $\perp^\sigma$, but there is no $\lambda_r^{\perp\rho}$ reduction between these terms. The reason for this is that we added several

ad hoc proof rules, such as fixed point induction and reasoning by cases for lifted types, which were necessary to get a strong enough proof system but which were not categorically motivated. It is a subject of future research to find categorically-based proof rules which permit similarly powerful arguments about programs and which are better-behaved than the *ad hoc* rules given here.

4.4 Example: call-by-name and call-by-value

One historical motivation for adding the capability to solve mixed-variant RDEs is the desire to find models for the untyped lambda calculus, Λ. That is, we wish to have a type u of untyped lambda terms, along with a translation $\mathcal{U}$ into a typed calculus such that $\mathcal{U}(M)$ is a term of type u that has the same behavior (in some sense) as the untyped term M.

A first attempt to model Λ in $\lambda^{\perp\rho}$ might take $u \equiv \rho t.\, t{\to}t$; this would allow us to translate the application MN as $\mathit{unfold}_u\,\mathcal{U}(M)\mathcal{U}(N)$. Unfortunately, since $1 \cong 1{\to}1$ and 1 is initial in $\mathbf{CPPO}^R$, the interpretation of u is just the trivial cpo. The usual semantic approach is to build the fixed point on top of a domain which has some non-trivial structure to begin with (see for example [20, 51]). We may use a much simpler syntactic reformulation to avoid the trivial solution by explicitly introducing extra structure with lifting. The result we will discuss in this section is that two particularly obvious choices for u using lifting correspond to call-by-name and call-by-value versions of Λ (see [42] for the original discussion of call-by-name and call-by-value; a presentation of domain models for these calculi which are essentially the same as the types n and v below is in [5]).

The type $n \equiv \rho t.\, (t{\to}t)_\perp$ adds to u the structure representing the possibility that a term might be undefined after an unfolding. Unfolding a term of type n will correspond to getting the head of an application into the form of an abstraction; evaluating the head may not terminate, and the bottom element takes care of this possibility. If we do get the head into the form of an abstraction, then that corresponds to obtaining the lift of a function of type $n{\to}n$ after the unfolding. Such a function will necessarily produce a term of type n when applied to any argument of type n —

it does not need to perform any extra evaluation on the argument; for this reason, the type n corresponds to a call-by-name calculus.

By contrast, the type $v \equiv \rho t.\, t{\to}t_\perp$ will correspond to a call-by-value calculus, because once the head has unfolded to a function, it is only a *partial* function (by the standard interpretation of $v{\to}v_\perp$ as the type of partial functions from v to v); the result of application may not terminate. Since we also have the possibility that simply trying to get the head of the application into abstraction form may not terminate, the type we will actually use to model terms of the call-by-value calculus is $v_\perp$.

To put this discussion in more concrete terms, consider the following translations from Λ to $\lambda^{\perp\rho}$:

$$\begin{aligned}
\mathcal{N}(x) &\equiv x{:}\, n \\
\mathcal{N}(\lambda x.\, M) &\equiv \mathit{fold}_n \lfloor \lambda x{:}\, n.\, \mathcal{N}(M) \rfloor {:}\, n \\
\mathcal{N}(MN) &\equiv \mathit{App}^N\, \mathcal{N}(M) \mathcal{N}(N) {:}\, n
\end{aligned}$$

$$\begin{aligned}
\mathcal{V}(x) &\equiv \lfloor x \rfloor {:}\, v_\perp \\
\mathcal{V}(\lambda x.\, M) &\equiv \lfloor \mathit{fold}_v (\lambda x{:}\, v.\, \mathcal{V}(M)) \rfloor {:}\, v_\perp \\
\mathcal{V}(MN) &\equiv \mathit{App}^V\, \mathcal{V}(M) \mathcal{V}(N) {:}\, v_\perp,
\end{aligned}$$

where $\mathit{App}^N \equiv (\lambda\, \mathit{fold}_n \lfloor f{:}\, n{\to}n \rfloor .\, f)$ and $\mathit{App}^V \equiv (\lambda \lfloor \mathit{fold}_v (f{:}\, v{\to}v_\perp) \rfloor .\, \lambda \lfloor z{:}\, v \rfloor .\, f z)$ (recall that, in our pattern matching syntax, $(\lambda\, \mathit{fold}_\tau\, \mathcal{P}.\, M)$ is shorthand for the term $(\lambda x{:}\, \tau.\, (\lambda \mathcal{P}.\, M)(\mathit{unfold}_\tau\, x)))$. In evaluating an application MN in either call-by-name or call-by-value, M must first reduce to an abstraction. The strict abstractions on f in the App combinators will ensure that this evaluation takes place first. In the case of the call-by-value translation, the additional strict abstraction on z forces the evaluation of the argument next; in call-by-name, the function f is simply applied to the argument right away.

To formalize the connection with the call-by-name and call-by-value calculi, recall

from [42] that the reductions $\xrightarrow{cbn}\!\!\!\!\rightarrow$ and $\xrightarrow{cbv}\!\!\!\!\rightarrow$ are built up from the axioms

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

$$(\beta_V) \qquad (\lambda x.\, M)V \longrightarrow \{V/x\}M, \quad V \text{ a value},$$

respectively, where a *value* is either a variable or an abstraction (we will omit the optional set of constants and δ-rules for simplicity). Then we may prove the following theorem:

Theorem 4.4.1 *$M \xrightarrow{cbn}\!\!\!\!\rightarrow N$ iff $\mathcal{N}(M) \longrightarrow\!\!\!\!\rightarrow \mathcal{N}(N)$, and $M \xrightarrow{cbv}\!\!\!\!\rightarrow N$ iff $\mathcal{V}(M) \longrightarrow\!\!\!\!\rightarrow \mathcal{V}(N)$.*

Proof. The forward direction of each part proceeds by induction on the length of the reduction; we only need to show that $\mathcal{N}((\lambda x.\, M)N) \longrightarrow\!\!\!\!\rightarrow \mathcal{N}(\{N/x\}M)$ and $\mathcal{V}((\lambda x.\, M)V) \longrightarrow\!\!\!\!\rightarrow \mathcal{V}(\{V/x\}M)$, for V a value. We will only show the details of the second, since the first is very similar (and easier):

$$\begin{aligned} \mathcal{V}((\lambda x.\, M)V) \quad &\equiv \quad App^V \lfloor fold_v(\lambda x\colon v.\, \mathcal{V}(M))\rfloor \mathcal{V}(V) \\ &\longrightarrow\!\!\!\!\rightarrow \quad (\lambda \lfloor z\colon v\rfloor.\, (\lambda x\colon v.\, \mathcal{V}(M))z)\mathcal{V}(V); \end{aligned}$$

since V is a value, $\mathcal{V}(V)$ must be of the form $\lfloor N\rfloor$ for some term N, hence the strict application may reduce, leading to $\{N/x\}\mathcal{V}(M)$. Now, examination of the definition of $\mathcal{V}(M)$ reveals that this substitution is identical to the term $\mathcal{V}(\{V/x\}M)$, as desired.

In the other direction we will make use of the standardization theorem of the previous section. Again, we will only give the details for the more complex call-by-value case. We will proceed by induction on the structure of M and the length of the reduction sequence. If $M \equiv x$, then $\mathcal{V}(M) \equiv \lfloor x\rfloor$, which is a normal form, so $M \equiv N$ and we are done. If $M \equiv (\lambda x.\, P)$, then $\mathcal{V}(M) \equiv \lfloor fold_v(\lambda x\colon v.\, \mathcal{V}(P))\rfloor$, so the only possible reduction is to $\lfloor fold_v(\lambda x\colon v.\, Q)\rfloor$. For this to be $\mathcal{V}(N)$ we must have that $Q \equiv \mathcal{V}(P')$ for $N \equiv (\lambda x.\, P')$; by the induction hypothesis then we know that $P \xrightarrow{cbv}\!\!\!\!\rightarrow P'$ and hence $M \xrightarrow{cbv}\!\!\!\!\rightarrow N$.

The remaining case is that $M \equiv PQ$. If $\mathcal{V}(M) \equiv App^V\, \mathcal{V}(P)\mathcal{V}(Q)$ reduces to $\mathcal{V}(N)$, then either $\mathcal{V}(N) \equiv App^V\, \mathcal{V}(P')\mathcal{V}(Q')$, whence $M \xrightarrow{cbv}\!\!\!\!\rightarrow P'Q' \equiv N$ by the

induction hypothesis, or the App^V must participate in the reduction. For this to be true, the standardization of the reduction must look like the following:

$$\begin{aligned} App^V\,\mathcal{V}(P)\mathcal{V}(Q) &\longrightarrow\!\!\!\!\!\rightarrow\; App^V \lfloor fold_v(\lambda x{:}\, v.\, \mathcal{V}(P'))\rfloor \mathcal{V}(Q) \\ &\longrightarrow\!\!\!\!\!\rightarrow\; (\lambda \lfloor z{:}\, v\rfloor .\, (\lambda x{:}\, v.\, \mathcal{V}(P'))z)\mathcal{V}(Q) \\ &\longrightarrow\!\!\!\!\!\rightarrow\; (\lambda \lfloor z{:}\, v\rfloor .\, (\lambda x{:}\, v.\, \mathcal{V}(P'))z)\lfloor Q_1\rfloor \\ &\longrightarrow\!\!\!\!\!\rightarrow\; \{Q_1/x\}\mathcal{V}(P') \equiv \mathcal{V}(\{Q'/x\}P') \\ &\longrightarrow\!\!\!\!\!\rightarrow\; \mathcal{V}(N), \end{aligned}$$

where $\mathcal{V}(Q') \equiv \lfloor Q_1\rfloor$. This uses the fact that in any standard reduction from $\mathcal{V}(Q)$, the first term in the reduction sequence which is of the form $\lfloor Q_1\rfloor$ must in fact be $\mathcal{V}(Q')$ for some value Q'. By the induction hypothesis we thus have the corresponding reduction sequence

$$\begin{aligned} M \equiv PQ &\xrightarrow{cbv}\!\!\!\!\!\rightarrow\; (\lambda x.\, P')Q \\ &\xrightarrow{cbv}\!\!\!\!\!\rightarrow\; (\lambda x.\, P')Q' \\ &\xrightarrow{cbv}\; \{Q'/x\}P' \\ &\xrightarrow{cbv}\!\!\!\!\!\rightarrow\; N. \end{aligned}$$

■

4.5 Comparison of recursive types

In this section we will prove the results mentioned in the introduction to this chapter about the relations between the recursive types in the cpo model. Although we only treat the case of **CPO**, we believe that the relations hold in many similar categories. A more abstract account is given by Barr [2], who shows that the initial and terminal fixed points of a functor F will coincide under fairly general conditions on F; he refers to such functors as *algebraically compact*. For the specific case of **CPO**, Barr's theorem proves that all of our pointed functors are algebraically compact, therefore

$\mu F \cong \nu F$. This is one part of our result below, which we obtained independently. Fokkinga and Meijer [13] also establish a similar result, although since they only work with pointed types they find that the initial and terminal fixed points always correspond (for suitably continuous functors).

We have already described how to construct μF and νF in the category **CPO**, and how we may use the Smyth-Plotkin technique to construct ρF in the category $\mathbf{CPO}^R$ of retracts. Now let us use the names of terms in $\lambda^{\perp\rho}$ to represent the actual functions on cpo's which model them. For example, if we have constructed the cpo μF, then we have an isomorphism $\mathit{fold}_{\mu F}\colon F(\mu F)\to\mu F$. Also, although we did not introduce the terms into our language, let us use $\mathit{it}_{\rho F}\colon (F(\tau)\to\tau)\to\rho F\to\tau$ for the map which describes the fact that ρF is an initial F-algebra in $\mathbf{CPO}^R$, and similarly for $\mathit{new}_{\rho F}\colon (\tau\to F(\tau))\to\tau\to\rho F$.

Theorem 4.5.1 *If νF is pointed, then $\nu F \cong \rho F$; if μF is pointed, then $\mu F \cong \nu F \cong \rho F$.*

Proof. If νF is pointed, then the isomorphism $\mathit{unfold}_{\nu F}$ and its inverse $\mathit{fold}_{\nu F}$ provide a retract in $\mathbf{CPO}^R$ both from νF to $F(\nu F)$ and also from $F(\nu F)$ back to νF. Therefore we have retracts $\mathit{it}_{\rho F}\,\mathit{fold}_{\nu F}$ and $\mathit{new}_{\rho F}\,\mathit{unfold}_{\nu F}$ going both ways between νF and ρF. If there are two cpo's X and Y and each has a retract into the other, then they must be isomorphic, hence νF and ρF are isomorphic types.

Similarly we may show that if μF is pointed then it is isomorphic to ρF. Now, since μF pointed implies νF pointed, we must have in this case that all three recursive types are isomorphic. ∎

As an application of this theorem, we will provide an informal semantic justification of our definition in Section 4.1 that inductive and projective types will be well-formed if the body is covariant in the bound type variable and if every occurrence of the bound variable to the left of an arrow is contained in a pointed subexpression. The base case is that either the body is strictly positive, so there will be no cardinality problems to prevent us from constructing the limit or colimit essentially as we would in **Set**, or the body is pointed, in which case the above theorem shows that we could just as well have used the Smyth-Plotkin construction to form the retractive type.

To form an arbitrary inductive type μF (or a projective type νF) we may first decompose the body into a strictly positive context $\sigma[\ ]$ and a sequence of pointed type expressions $\tau_1, \ldots, \tau_n$ in which the type variable t may occur to the left of an even number of arrows. Thus $F(t) \equiv \sigma[\tau_1, \ldots, \tau_n]$. Now, by a standard equivalence for recursive types, we know that $\mu t.\, F(t) \cong \mu t.\, \mu s.\, \sigma[\{s/t\}\tau_1, \tau_2, \ldots, \tau_n]$, where s is a new type variable. Unfolding the inner inductive type once results in the type expression $\mu t.\, \sigma[\{\mu s.\, \sigma[\tau_1', \tau_2, \ldots, \tau_n]/s\}\tau_1', \tau_2, \ldots, \tau_n]$, where $\tau_1' \equiv \{s/t\}\tau_1$.

By considering the unfoldings of the inner expression $\{\mu s.\, \sigma[\tau_1', \tau_2, \ldots, \tau_n]/s\}\tau_1'$, we may see that it is equivalent to the type $\mu r.\, \{\sigma[r, \tau_2, \ldots, \tau_n]/s\}\tau_1'$; let us call this υ_1. Note that since τ_1 was pointed, this type will also be pointed, hence we could have constructed it with ρ instead of μ. Now, repeating this process for τ_2 through τ_n, we find that $\mu t.\, F(t) \cong \mu t.\, \sigma[\upsilon_1, \ldots, \upsilon_n]$, where $\upsilon_k \equiv \mu r.\, \{\sigma[\upsilon_1, \ldots, \upsilon_{k-1}, r, \tau_{k+1}, \ldots, \tau_n]/t\}\tau_k$. The body of this type is strictly positive in t, and each υ_k is pointed, so it is a well-formed inductive type.

As an example of this construction, consider the type $\mu t.\, ((t \to \mathit{bool}) \to \mathit{bool})_\perp + t$. It is equivalent to $\mu t.\, \upsilon + t$, where $\upsilon \equiv \mu r.\, (((r + t) \to \mathit{bool}) \to \mathit{bool})_\perp$, which can be seen to correspond to a "partial" unfolding of the body to bring the pointed subterm to the outside.

Chapter 5

Conclusions

We have considered three functional programming languages based on the typed lambda calculus plus structured data types and recursive definitions of either functions or types. For each we have presented an equational proof system which includes extensional rules for reasoning about equivalences of types, as well as sets of reduction rules based on the equations. We have seen the interaction of recursion with the extensional rules, and in each case we have determined that an appropriate reduction system for evaluating the results of programs should not include the extensional rules.

One of our languages, $\lambda^{\mu\nu}$, considered in Chapter 3, could define only total functions; we showed that the class of definable functions properly includes those that are provably total in Peano Arithmetic. The other two languages, PCF and $\lambda^{\perp\rho}$, considered in Chapters 2 and 4 respectively, are each powerful enough to express all partial recursive functions. We proved for both of these languages that, under certain conditions, a leftmost reduction strategy is complete for finding normal forms relative to the "ideal" non-deterministic reduction system.

For the language PCF, we obtained a confluent reduction system for any set of confluent, left-linear algebraic rules defining the set of observable types. This reduction system is adequate for finding results of programs, relative to provability in the full equational proof system, including the extensional rules. Since the interaction of the surjective pairing contraction rule with the fixed-point operator causes confluence to fail even without any algebraic rules, this result provides a theoretical basis for

the folkloric belief that the extensional rules are not "computational." In addition, we showed that the extensional rules remain sound for reasoning about observational congruence.

We did not manage to prove such an adequacy theorem for $\lambda^{\perp\rho}$; the difficulty came in having to deal with several *ad hoc* proof rules for retractive and lifted types. For the sublanguage $\lambda^{\mu\nu}$, where all of the proof rules had clear categorical motivations, however, we were able to prove an adequacy theorem. A future line of research is to investigate alternative proof systems for languages similar to $\lambda^{\perp\rho}$ for which we can establish adequacy.

We did prove a standardization theorem for $\lambda^{\perp\rho}$, which allowed us to investigate use of the lifted and retractive types to model call-by-name and call-by-value versions of the untyped lambda calculus. A question which we did not have time to pursue is whether we can also model the partial lambda calculus of Moggi [36] in this manner.

Other open problems which remain are to find a more exact characterization of the functions definable in $\lambda^{\mu\nu}$, to extend the class of algebraic rules which support a left-most reduction strategy for PCF, and to examine the effect of adding computational monads other than lifting to $\lambda^{\perp\rho}$ in the manner of Moggi and Wadler [37, 54].

Bibliography

[1] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.

[2] M. Barr. Algebraically compact functors. Manuscript, 1991.

[3] G. Berry and P.L. Curien. Sequential algorithms on concrete data structures. *Theoretical Computer Science*, 20:265–321, 1982.

[4] C. Böhm and A. Berarducci. Automatic synthesis of typed Λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[5] G. Boudol. Lambda-calculi for (strict) parallel functions. Technical Report 1387, INRIA, January 1991.

[6] V. Breazu-Tannen. Combining algebra and higher-order types. In *Third Annual IEEE Symposium on Logic in Computer Science*, pages 82–90, 1988.

[7] V. Breazu-Tannen and J.H. Gallier. Polymorphic rewriting conserves algebraic strong normalization and confluence. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocha, editors, *16th International Colloquium on Automata, Languages, and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 137–159. Springer-Verlag, 1989.

[8] W. Buchholz and S.S. Wainer. Provably computable functions and the fast growing hierarchy. In S.G. Simpson, editor, *Logic and Combinatorics*, volume 65 of *Contemporary Mathematics*, pages 179–198. American Mathematical Society, 1987.

[9] A. Church. *The Calculi of Lambda Conversion*. Princeton Univ. Press, 1941. Reprinted 1963 by University Microfilms Inc., Ann Arbor, MI.

[10] E.A. Cichon and S.S. Wainer. The slow-growing and the Grzegorczyk hierarchies. *Journal of Symbolic Logic*, 48(2):399–408, 1983.

[11] T. Coquand and C. Paulin. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *COLOG-88*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer-Verlag, 1990.

[12] R.L. Crole and A.M. Pitts. New foundations for fixpoint computations. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 489–497, 1990.

[13] M.M. Fokkinga and E. Meijer. Program calculation properties of continuous algebras. Technical Report CS-R9104, CWI, January 1991.

[14] P. Freyd. Recursive types reduced to inductive types. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 498–507, 1990.

[15] J.H. Gallier. What's so special about Kruskal's theorem and the ordinal Γ_0? A survey of some results in proof theory. *Annals of Pure and Applied Logic*, 53(3):199–260, 1991.

[16] J.-Y. 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, *Second Scandinavian Logic Symposium*, pages 63–92. North-Holland, 1971.

[17] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[18] K. Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12:280–287, 1958. An English translation by W. Hodges and B. Watson appeared in *Journal of Philosophical Logic*, 9:133–142, 1980.

[19] J. Greiner. Programming with inductive and co-inductive types. Technical Report CMU-CS-92-109, Carnegie Mellon University, January 1992.

[20] C.A. Gunter and D.S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North-Holland, 1990.

[21] T. Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, 1987.

[22] T. Hagino. A typed lambda calculus with categorical type constructors. In *Category Theory in Computer Science*, pages 140–157, 1987.

[23] J.Y. Halpern, J.H. Williams, E.L. Wimmers, and T.C. Winkler. Denotational semantics and rewrite rules for FP. In *12th ACM Symposium on Principles of Programming Languages*, pages 108–120, January 1985.

[24] R. Hasegawa. Parametric polymorphism and internal representations of recursive type definitions. Master's thesis, Research Institute for Mathematical Science, Kyoto University, 1989.

[25] P. Henderson. *Functional Programming*. Prentice-Hall, 1980.

[26] B.T. Howard and J.C. Mitchell. Operational and axiomatic semantics of PCF. In *ACM Conference on* Lisp *and Functional Programming*, pages 298–306, 1990.

[27] C.B. Jay. Long $\beta\eta$ normal forms and confluence. Technical Report ECS-LFCS-91-183, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, October 1991.

[28] J.W. Klop. *Combinatory Reduction Systems*. PhD thesis, University of Utrecht, 1980. Published as Mathematical Center Tract 129.

[29] J.W. Klop and R.C. de Vrijer. Unique normal forms for lambda calculus with surjective pairing. *Information and Computation*, 80(2):97–113, 1989.

[30] G. Kreisel. Interpretation of analysis by means of constructive functionals of finite types. In A. Heyting, editor, *Constructivity in Mathematics*, pages 101–128. North-Holland, 1959.

[31] J. Lambek and P.J. Scott. *Introduction to Higher-Order Categorical Logic*, volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 1986.

[32] S. Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, 1971.

[33] Z. Manna, S. Ness, and J.E. Vuillemin. Inductive methods for proving properties of programs. In *ACM Conference on Proving Assertions about Programs*, pages 27–50, 1972. Proceedings appeared as *SIGPLAN Notices*, 7(1), 1972.

[34] L.W. Miller. Normal functions and constructive ordinal numbers. *Journal of Symbolic Logic*, 41(2):439–459, 1976.

[35] J.C. Mitchell. Type systems for programming languages. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*. North-Holland, 1990.

[36] E. Moggi. *The Partial Lambda-Calculus*. PhD thesis, University of Edinburgh, 1988.

[37] E. Moggi. Computational lambda-calculus and monads. In *Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 14–23, 1989.

[38] D. Nesmith. The Church-Rosser property in higher-order rewrite systems. Manuscript, 1989.

[39] M.H.A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.

[40] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[41] G.D. Plotkin. The λ-calculus is ω-incomplete. *Journal of Symbolic Logic*, 39:313–317, 1974.

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

[43] G.D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[44] G.D. Plotkin. Denotational semantics with partial functions. Lecture notes, C.S.L.I. Summer School, Stanford, 1985.

[45] G. Pottinger. The Church-Rosser theorem for typed λ-calculus with surjective pairing. *Notre Dame Journal of Formal Logic*, 22(3):264–268, 1981.

[46] J.C. Reynolds. Towards a theory of type structure. In *Paris Colloquium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer-Verlag, 1974.

[47] H.E. Rose. *Subrecursion: Functions and Hierarchies*, volume 9 of *Oxford Logic Guides*. Oxford University Press, 1984.

[48] H. Schwichtenberg. Elimination of higher type levels in definitions of primitive recursive functions by means of transfinite recursion. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium, '73*, pages 279–303. North-Holland, 1975.

[49] H. Schwichtenberg. Definierbare funktionen im λ-Kalkül mit typen. *Archiv für mathematische Logik und Grundlagenforschung*, 17:113–114, 1976.

[50] D.S. Scott. A type-theoretic alternative to CUCH, ISWIM, OWHY. Manuscript, 1969.

[51] M. Smyth and G.D. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM Journal on Computing*, 11:761–783, 1982.

[52] R. Statman. The typed lambda calculus is not elementary recursive. *Theoretical Computer Science*, 9:73–81, 1979.

[53] D.A. Turner. Miranda: a non-strict functional language with polymorphic types. In *IFIP International Conference on Functional Programming and Computer Architecture, Nancy*, volume 201 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[54] P. Wadler. Comprehending monads. In *ACM Conference on* Lisp *and Functional Programming*, pages 61–78, 1990.

[55] P. Wadler. Private communication, March 1991.

[56] S.S. Wainer. Slow growing versus fast growing. *Journal of Symbolic Logic*, 54(2):608–614, 1989.

[57] M. Wand. Fixed-point constructions in order-enriched categories. *Theoretical Computer Science*, 8:13–30, 1979.

[58] G.C. Wraith. A note on categorical datatypes. In D.H. Pitt, D.E. Rydeheard, P. Dybjer, A.M. Pitts, and A. Poigné, editors, *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 118–127. Springer-Verlag, 1989.