January 1989

Proof-Theoretic Methods for Analysis of Functional Programs

John J. Hannan
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-89-07.

Proof-Theoretic Methods for Analysis of Functional Programs

Abstract
We investigate how, in a natural deduction setting, we can specify concisely a wide variety of tasks that manipulate programs as data objects. This study will provide us with a better understanding of various kinds of manipulations of programs and also an operational understanding of numerous features and properties of a rich functional programming language. We present a technique, inspired by structural operational semantics and natural semantics, for specifying properties of, or operations on, programs. Specifications of this sort are presented as sets of inference rules and are encoded as clauses in a higher-order, intuitionistic meta-logic. Program properties are then proved by constructing proofs in this meta-logic. We argue the following points regarding these specifications and their proofs: *(i)* the specifications are clear and concise and they provide intuitive descriptions of the properties being described; *(ii)* a wide variety of program analysis tools can be specified in a single unified framework, and thus we can investigate and understand the relationship between various tools; *(iii)* proof theory provides a well-established and formal setting in which to examine meta-theoretic properties of these specifications; and *(iv)* the meta-logic we use can be implemented naturally in an extended logic programming language and thus we can produce experimental implementations of the specifications. We expect that our efforts will provide new perspectives and insights for many program manipulation tasks.

Comments
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-89-07.

PROOF-THEORETIC METHODS FOR ANALYSIS OF FUNCTIONAL PROGRAMS (Dissertation Proposal)

John J. Hannan

MS-CIS-89-07
LINC LAB 142

Department of Computer and Information Science
School of Engineering and Applied Science
University of Pennsylvania
Philadelphia, PA 19104

January 1989

Acknowledgements: This research was supported in part by NSF grants CCR-87-05596, MCS-8219196-CER, IRI84-10413-AO2, DARPA grant NOOO14-85-K-0018, and U.S. Army grants DAA29-84-K-0061, DAA29-84-9-0027.

PROOF-THEORETIC METHODS FOR ANALYSIS OF FUNCTIONAL PROGRAMS*

Dissertation Proposal

John J. Hannan†
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104 USA

December 1988

Abstract

We investigate how, in a natural deduction setting, we can specify concisely a wide variety of tasks that manipulate programs as data objects. This study will provide us with a better understanding of various kinds of manipulations of programs and also an operational understanding of numerous features and properties of a rich functional programming language. We present a technique, inspired by structural operational semantics and natural semantics, for specifying properties of, or operations on, programs. Specifications of this sort are presented as sets of inference rules and are encoded as clauses in a higher-order, intuitionistic meta-logic. Program properties are then proved by constructing proofs in this meta-logic. We argue the following points regarding these specifications and their proofs: (*i*) the specifications are clear and concise and they provide intuitive descriptions of the properties being described; (*ii*) a wide variety of program analysis tools can be specified in a single unified framework, and thus we can investigate and understand the relationship between various tools; (*iii*) proof theory provides a well-established and formal setting in which to examine meta-theoretic properties of these specifications; and (*iv*) the meta-logic we use can be implemented naturally in an extended logic programming language and thus we can produce experimental implementations of the specifications. We expect that our efforts will provide new perspectives and insights for many program manipulation tasks.

Advisor: Dale Miller

Committee: Val Breazu-Tannen
Jean Gallier (Chair)
David MacQueen
Andre Scedrov

*This report contains preliminary material that is subject to revision. Comments are welcomed. Address correspondence to John Hannan at the above address or at "hannan@linc.upenn.edu."

†Supported by a fellowship from the Corporate Research and Architecture Group, Digital Equipment Corporation, Maynard, MA USA.

Contents

List of Figures

1

Introduction

1.1 Meta-Programming

Meta-programming, in its most general setting, is any programming task in which programs are treated as data objects. We typically distinguish between the meta-language, in which we write the meta-programs, and the object-language, in which the programs being manipulated are written. These two languages may in fact be the same language in some cases, but in general they are two different ones. Considering this definition of meta-programming we observe that many common programs or procedures can be classified as meta-programs: (*i*) editors treat programs as objects that are to be modified; most editors do not treat object programs as a special data type (distinct from arbitrary text), though some recent programming systems include editors in which they are [54]; (*ii*) compilers treat programs as the source and target objects of a translation process; (*iii*) interpreters treat programs as input data and produce as output the result of executing the program. A more narrow definition of meta-programming includes only those programming tasks of an "experimental" nature or those that produce some auxiliary information, not typically required of a programming system. Examples of these include rapid-prototype interpreters for experimental extensions of languages, partial evaluators and programs for abstract interpretation. In this proposal we assume the former, more encompassing, definition of meta-programming. This choice does not greatly affect the issues to arise in this study, but merely enlarges the set of examples that we may present.

One purpose of this work is to present a general method for specifying meta-programming

tasks over functional programs. We cannot hope be completely general as we must make certain choices during our investigations: the choice of the functional programming language that we consider and the range of meta-programming tasks to be considered. For the former parameter we will use Standard ML as our guide, considering first only small subsets of this language, with the expectation of extending our techniques to the full language. For the latter parameter we hope that the methods we present will accommodate a wide range of meta-programming tasks but we do not expect to define formally the limits of our methods.

1.2 Natural Deduction and Meta-Programming

In a natural deduction theorem prover, one thinks of constructing proofs of propositions. These propositions are typically defined in some formal logic, *e.g.*, first-order predicate calculus. For our application, the propositions will denote statements about object programs. These statements may either be statements concerning a program property (*"program P is well-typed"*) or concerning an operation on the program (*"program P evaluates to value V"*). Considering our use of propositions we then have two questions to answer: (*i*) Over what logic do we define our propositions? and (*ii*) In this logic how do we encode programs as terms? These questions principally depend on the object language that we consider, though the kind of program property or operation that we consider, also matters.

The methods we present are based on proof-theoretic techniques of natural deduction. This approach owes much to the work on structural operational semantics by Plotkin [52] and to the work on natural semantics by Kahn and others at INRIA [5, 31]. (See Chapter 7 for more discussion of these works.) These meta-languages represent programs as first-order tree structures and provide a reasoning style similar to that of natural deduction. One strength of natural semantics is that it can be compiled directly into PROLOG by using first-order terms to represent programs and by using unification and backchaining to implement the natural deduction-style reasoning. While natural semantics provides methods for specifying many meta-programming tasks, we feel that its use of strictly first-order terms and limited types of inference figures restricts its general applicability and adaptability to a wider class of tasks. We shall present proof systems that extend the techniques of natural semantics by introducing higher-order terms (simply typed λ-terms) directly into the meta-language. Along with such an extension, we extend the underlying reasoning mechanism with two kinds of introduction and discharge rules. We argue that this extension yields a higher-level description of many program manipulations and provides a more natural specification of these tasks. Many low-level routines for manipulating program code, such as

substitutions for free variables, changing bound variable names, maintaining a context, *etc.*, are essentially moved to the meta-language and need not be written into the specification.

1.3 Natural Deduction and Logic Programming

As alluded to above there is a close relationship between natural semantics and logic programming. More precisely, it is the relationship between natural deduction and logic programming that we wish to exploit. Logic programming languages provide many features that make them suitable implementation languages for natural deduction theorem provers [12]. A brief examination of these features makes this connection obvious. A foremost aspect of computation in logic programming is the search operation. Taking a procedural view of logic programming we can describe the execution of a logic program by describing a search process through a space defined by the program. Search is also an important component of natural deduction theorem proving. The task of constructing a proof can be described as the exploration of a search space. Unification is a second characteristic feature of logic programming. It provides a mechanism for matching two terms. More specifically, we can specify "generic" clauses in a program and then use these clauses in specific instances via unification. A similar mechanism plays an important role in constructing proofs in natural deduction. A natural deduction system can be specified by a set of inference rules. These rules are typically given by rule templates, *i.e.*, ones that contain free variables. Proofs will contain only closed instances of these rules and so some matching or unification process is required to produce the required instances of these rules. Finally, most logic programming languages are constructed from clauses of the general form

$$Body \supset Head$$

with the intuitive reading "if *Body* is true then *Head* is true." Thus the inference rules used to specify a natural deduction system should have a natural translation into clauses of this form. The head and body of a clause will denote the consequent and the antecedent of an inference rule. This straightforward translation into clause form together with the declarative style of logic programming suggests that using logic programming to specify a natural deduction style theorem prover will yield a perspicuous implementation.

As part of this proposal we introduce a simple functional programming language based on a subset of Standard ML. We initially consider only the applicative aspect of the language, including abstraction, application and a polymorphic let. We then describe an encoding of this language into simply typed λ-terms. We will argue that this encoding provides a convenient representation for manipulating functional programs as objects. To manipulate

these programs we will require mechanisms for analyzing and modifying λ-terms. We will make a limited use of higher-order unification to provide an appropriate mechanism for analyzing terms representing programs. We will also make a simple use of β-conversion for manipulating λ-terms in useful ways.

In this proposal we will argue that first-order Horn clauses do not provide a suitable meta-language for manipulating λ-terms. Other work in natural semantics has used Prolog, which is based on first-order Horn clauses, as their meta-language, but we argue that a stronger meta-language is required to consider richer object-level programming languages (*i.e.*, those containing features such as modules, abstract data types and exceptions) *and* to specify manipulations of these languages in a clear and concise manner. We therefore introduce a meta-language based on *higher-order hereditary Harrop* formulas [39]. This language replaces the first-order terms of Prolog with simply typed λ-terms. Compared with Prolog, the language also provides a more flexible use of quantifiers in formulas and provides an important mechanism for the introduction and discharge of assumptions during the computation process. We will show how an essential use of these features contributes to the effectiveness of our meta-language. All the programs presented in this proposal have been implemented and tested in version LP2.6 of λProlog which is a logic programming language based on higher-order hereditary Harrop formulas. For a discussion on the various aspects of this language see [45, 39, 37].

1.4 Research Goals

The main purpose of this research is to demonstrate how, with a suitable meta-logic, a natural deduction paradigm provides a suitable framework for manipulating and analyzing functional programs. Previous work has used inference rules to specify the dynamic semantics and other properties of programs [52, 6], but their emphasis has typically been more towards software engineering issues and less towards a study of the proof theory. We are concerned with defining and characterizing a formal meta-language via proof-theoretic methods and understanding the nature of proofs that can be constructed in this language. From a practical standpoint this work finds immediate application in the development of programming languages and programming language environments. Such tasks require development tools that are both expressive and extensible (as well as other qualities as described below) and the methods pursued in this work are well suited.

Another goal of this research is to demonstrate how, in a single meta-language, we can specify a wide variety of tasks that treat programs as objects. We present specifications

for tasks such as evaluation, type inferencing and compilation, each presented by a set of inference rules. We intend to extend this list to include such tasks as strictness analysis, abstract interpretation and other flow analysis problems. By describing these apparently disparate tasks in a unified framework we hope to gain insight into the similarities and differences among these tasks. From a practical standpoint, this uniform treatment of tasks suggests the possibility of integrating various tools. From a theoretical standpoint, a detailed analysis of a variety of tools can be performed using uniform techniques. Thus the same (meta-theoretic) analysis techniques used on the static semantics of a language could apply to a compiler for the language.

A third goal of this research is to provide detailed analysis of the proof-theoretic analysis tools presented herein. By exploiting our foundations in proof theory we can reason about meta-theoretic properties via proof transformations and manipulations For example, we show that certain program transformers have a correctness-preserving property by demonstrating an equivalence between certain classes of proof trees. Thus, using some well-established methods of proof theory we can express and prove important (meta-)properties of our meta-programs. We argue that this analysis is more perspicuous than corresponding analyses for other type inference specifications (using other methods) owing to the nature of our proof system.

A fourth goal of this research is to provide an "operational" understanding of the effects of introducing additional features to a programming language. More precisely, we will examine the relationship between an object language and its meta-language. This will be done by considering changes required of the meta-language and various meta-programs to accommodate new object-language features.

1.5 Evaluating a Meta-Language

In this proposal we present a particular class of meta-languages for specifying a wide variety of program manipulations. We wish to claim that this class has inherent qualities that make it a suitable meta-language. However, we must first decide what the important characteristics of a meta-language are. We have adapted the following list of criteria from [34].

- *Expressibility.* We want a meta-language in which we can manipulate a rich functional programming language, such as Standard ML. This criteria has two aspects: the representation of programs as terms and manipulations on these terms. Some meta-languages seem suited only to handling small and relatively simple subsets of languages; they do

not appear to be capable of "scaling up" to a full language with features such as modules and exceptions.

- *Simplicity of Description.* We want our meta-language to be relatively easy to learn. This criterion is, of course, largely dependent on the individual user, but we would like a system that is available to a large audience.

- *Clarity of Specifications.* The meta-language should afford specifications that are clear and concise. Ideally, we would like these specifications to suggest intuitive explanations for their underlying tasks.

- *Ease of Modification.* An important use of a meta-language is during the development of a new or extended programming language. Therefore we would like a meta-language in which we can easily specify extensions or modifications to existing specifications.

An inherent difficulty in assessing a meta-language is the open nature of the specifications we wish to write. If we were only interested in a meta-language for specifying the dynamic semantics, then the notion of expressibility, and the other criteria, is simple to interpret. But we are interested in other types of tasks and do not want to limit ourselves to just a fixed set of possibilities. (We may, however, have a few tasks, such as dynamic semantics, type inference and compilation, that are most important.)

1.6 Organization of Proposal

The remainder of the proposal is organized as follows. In Chapter 2 we present some of the required mathematics and logic used later in the proposal. Some technical details relating the untyped and simply typed λ-calculi are carefully presented. The knowledgeable reader can skip this chapter. In Chapter 3 we describe a general framework for our proof systems and the methods used to encode functional programs as terms. We outline how program properties can be denoted by propositions in a suitable logic. In Chapter 4 we describe a simple functional language PCF_0 and present several standard semantics using our proof methods. We give a static semantics (for type inferencing), a dynamic semantics and a compilation semantics (for compiling the language into an abstract machine language). In Chapter 5 we give some non-standard semantics for PCF_0 and also enrich the language, showing how our methods extend to richer languages. In Chapter 6 we present some meta-theoretic results arguing for the correctness of some of the proof systems presented in previous chapters. In one case we present a direct proof in which the correctness of other systems is not assumed; but in the others we use indirect methods, showing how our systems

are equivalent to existing systems known to be correct. In Chapter 7 we present work in related areas and attempt to place the current work in proper perspective and finally in Chapter 8 we summarize our results and suggest directions for future work.

2

Mathematical Preliminaries

Before diving into a presentation of meta-languages, programs as objects, *etc.*, we present some preliminary definitions and concepts.

2.1 Untyped λ-Calculus

The terms of the untyped λ-calculus are defined by

$$M ::= x \mid MN \mid \lambda x.M.$$

in which x is a variable. We assume a countably infinite sequence of variables. As a notational convenience, we shall denote the set of all terms generated by this grammar as λ^u. We shall always consider λ-terms modulo α-conversion, *i.e.*,

$$\lambda x.M = \lambda y.[y/x]M, \quad \text{if } y \text{ is free for } x \text{ in } M \qquad (\alpha)$$

allowing us to rename bound variables. Two additional axioms that we shall consider are

$$(\lambda x.M)N = [N/x]M, \qquad (\beta)$$

$$\lambda x.Mx = M, \quad \text{if } x \text{ is not free in M}, \qquad (\eta)$$

in which the substitution operation $[N/x]M$, replacing free occurrences of x in M with N, is defined as usual, with renaming of bound variables of M to avoid capture. We naturally extend substitution to parallel substitution denoted by $[\overline{N}/\overline{x}]M$ in which $\overline{N}$ and $\overline{x}$ are of the form $(N_0, N_1, \ldots, N_k)$ and $(x_0, x_1, \ldots, x_k)$, respectively, for some $k > 0$ and tacitly assume that this operation is idempotent (*i.e.*, $\forall i, j$ no x_i occurs in any N_j). If we consider λ-terms

modulo these two equations, we shall say that two equal terms are $\beta\eta$-convertible. We may also consider (β) and (η) as directed rewrite rules, replacing a subterm that matches the left-hand side of either equation with the corresponding right-hand side. We refer to these rewrite operations as β-reduction and η-reduction, respectively. We write $M \longrightarrow N$, possibly subscripted with α, β or η, to denote the one-step reduction of M to N. For the transitive closure of this operation we write $M \longrightarrow\!\!\!\!\!\rightarrow N$.

The expressions corresponding to the second and third cases of the grammar above are termed *applications* and *abstractions*, respectively. Application is defined to be left-associative and we assume that when an abstraction such as $\lambda x.M$ occurs in a larger expression, M is taken as extending as far as possible, *i.e.*, to the first unmatched closing bracket or the end of the expression, whichever is first. A *β-redex* is a term of the form $(\lambda x.M)N$ and an *η-redex* is a term of the form $\lambda x.(Px)$ where x is not free in P. A λ-term is in *β-normal form* if it contains no β-redexes, and is in *$\beta\eta$-normal form* if it is in β-normal form and contains no η-redexes. For a more complete discussion of the λ-calculus and its properties see [1, 26].

DEFINITION 2.1 (Equality between λ-terms) Given two untyped terms M, N we say that the two terms are equal, written $M = N$ if we can construct a proof of $M = N$ in the following proof system:

axioms: All instances of (α), (β) and (η) rules.
inference rules:

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

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

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

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

We write $\vdash_{\lambda_u} M = N$ if the equation $M = N$ is provable.

2.2 Simply Typed λ-Calculus

As mentioned in the introduction, we will argue for the use of simply typed λ-terms as a data structure for representing object programs.

We first describe the type expressions which are constructed from type variables and constants using the connective $\rightarrow$. We adopt the notational convention that b_i denotes a base type (constant), $r, s, t, \ldots$ denote type variables and $\rho, \sigma, \tau, \ldots$ denote type expressions. The set of type expressions is defined by the grammar

$$\tau ::= b_i \mid t \mid \sigma \rightarrow \tau$$

The terms of the simply typed λ-calculus are defined by

$$M ::= x{:}\tau \mid MN \mid \lambda x{:}\tau.M.$$

in which x is a typed variable. For each type τ we assume a countably infinite sequence of variables. As a notational convenience, we shall denote the set of all terms generated by this grammar as $\lambda^{\rightarrow}$. Terms in $\lambda^{\rightarrow}$ are given types according to the following rules:

$$x{:}\tau \rhd \tau$$

$$\frac{M \rhd \sigma \rightarrow \tau \quad N \rhd \sigma}{(MN) \rhd \tau}$$

$$\frac{M \rhd \tau}{\lambda x \rhd \sigma.M \rhd \sigma \rightarrow \tau}$$

As with the untyped terms, we have the following three axioms:

$$\lambda x{:}\tau.M = \lambda y{:}\tau.[y/x]M, \quad \text{if } y \text{ is free for } x \text{ in } M \qquad (\alpha)$$

$$(\lambda x{:}\tau.M)N = [N/x]M, \quad \text{if } N \text{ is of type } \tau \qquad (\beta)$$

$$\lambda x{:}\tau.Mx = M, \quad \text{if } x \text{ is not free in M}, \qquad (\eta)$$

We shall not explicitly distinguish between the untyped and typed versions of these axioms as it should be clear from the particular context. As with untyped terms, we have an analogous definition for equality between terms. We also have a notion of normal form, analogous to that for untyped terms. A *β-redex* is a term of the form $(\lambda x{:}\tau.M)N$ and an *η-redex* is a term of the form $\lambda x{:}\tau.(Px)$ where x is not free in P. A λ-term is in *β-normal form* if it contains no β-redexes, and is in *$\beta\eta$-normal form* if it is in β-normal form and contains no η-redexes. For the simply typed calculus, every typed term has a unique $\beta\eta$-normal form (See [26].) As a convenience when writing typed terms, we do not explicitly tag occurrences of bound variables with their associated type (except at their binding occurrence).

DEFINITION 2.2 (Equality between typed λ-terms) Given two typed terms $M, N \in \lambda^{\rightarrow}$ we say that the two terms are equal, written $M = N$ if we can construct a proof of $M = N$ in the following proof system:

axioms: All instances of the typed (α), (β) and (η) rules.
inference rules:

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

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

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

$$\frac{M = N}{\lambda x{:}\tau.M = \lambda x{:}\tau.N} \quad (\xi)$$

We write $\vdash_{\lambda^{\rightarrow}} M = N$ if the equation $M = N$ is provable.

2.3 Embedding the Untyped λ-Calculus in a Simply Typed λ-Calculus

We now outline how the untyped λ-calculus can be embedded in the simply typed λ-calculus. We shall not include the detailed proofs, but only present the required arguments. The method we describe is based on material originally introduced in [59] and later in [36] and [56]. To accomplish this embedding we need to describe a mapping from terms in λ^u to terms in $\lambda^{\rightarrow}$and extend it to a mapping from equations in λ^u to equations in $\lambda^{\rightarrow}$. In particular, if we are given a set of equations E in the untyped calculus, then we shall produce a corresponding set of equations E' in a typed calculus such that if two terms $M, N \in \lambda^u$ map to the terms $M', N' \in \lambda^{\rightarrow}$ then

$$E \vdash_{\lambda^u} M = N \quad \Leftrightarrow \quad E' \vdash_{\lambda^{\rightarrow}} M' = N'.$$

As a hint of what is to follow consider the untyped term $\omega = (\lambda x.xx)(\lambda x.xx)$. It is well known that this term has no normal form; in particular, we have the infinite reduction sequence

$$(\lambda x.xx)(\lambda x.xx) \longrightarrow_\beta (\lambda x.xx)(\lambda x.xx) \longrightarrow_\beta \cdots$$

Now we must map ω to a simply typed term ω' (note that ω itself cannot be obtained by erasing the types some term in $\lambda^{\rightarrow}$) and in some simply typed calculus that somehow

"mirrors" the (β) rule of λ^u, there can be no "normal form" for ω'. The sense of the phrase "normal form" here is different than $\beta\eta$-normal described above. (This must be the case since ω' must have a $\beta\eta$-normal form.)

To begin we extend $\lambda^{\rightarrow}$ in the following ways. First, we introduce a new base type u. Terms of this type shall correspond to terms of λ^u. Second we introduce two new constants, $\Phi{:}u \rightarrow (u \rightarrow u)$, $\Psi{:}(u \rightarrow u) \rightarrow u$. Third, we introduce two new equivalences among lambda terms:

$$\Phi(\Psi(M)) = M, \qquad (\varphi)$$

$$\Psi(\Phi(M)) = M, \qquad (\psi)$$

We shall use these additional rules to capture the (β) and (η) rules of λ^u. A term of the form $\Phi(\Psi(M))$ is called a φ-redex and a term of the form $\Psi(\Phi(M))$ is called a ψ-redex. We shall say that a term is in u-normal form if it has no $\beta-$, $\eta-$, $\varphi-$ or $\psi-$ redexes. We shall denote this calculus, including the two new reduction rules, by $\lambda^{\rightarrow\varphi\psi}$.

Equality between terms in the enriched calculus is very similar to equality between terms of $\lambda^{\rightarrow}$.

DEFINITION 2.3 (Equality between $\lambda^{\rightarrow\varphi\psi}$-terms) Given two typed terms $M, N \in \lambda^{\rightarrow\varphi\psi}$ we say that the two terms are equal, written $M = N$ if we can construct a proof of $M = N$ in the proof system of Definition 2.2 extended with the two axioms (φ) and (ψ). We write $\vdash_{\lambda\rightarrow\varphi\psi} M = N$ if the equation $M = N$ is provable.

We now define a mapping from λ^u to $\lambda^{\rightarrow\varphi\psi}$.

DEFINITION 2.4 ($(\cdot)^*$) For any $M \in \lambda^u$ let $(M)^*$ be

$$\begin{aligned} (x)^* &= x^*{:}u \quad \text{for x a variable.} \\ (MN)^* &= \Phi(M^*)N^* \\ (\lambda x.M)^* &= \Psi(\lambda x^*{:}u.M^*) \end{aligned}$$

We assume that $(\cdot)^*$ defines a bijective mapping of untyped variables to typed variables. This is possible since we assumed a countably infinite sequence of both kinds of variables.

We assume x^* to be a typed variable in $\lambda^{\rightarrow}$ corresponding to the variable x in λ^u. We extend this definition to equations such that $(M = N)^*$ is defined as $M^* = N^*$.

We now show that our definition of $(\cdot)^*$ captures the proper notion of equivalence among terms.

THEOREM 2.5 (Well-definedness of $(\cdot)^*$)
Let $M, N \in \lambda^u$. Then $\vdash_{\lambda u} M = N \Leftrightarrow \vdash_{\lambda\rightarrow\varphi\psi} M^* = N^*$.

This tells us that our translation from untyped terms to typed terms is capturing the right amount of information and identifying equal terms with equal terms.

For example, consider again the untyped term ω and ω^*:

$$\omega \equiv (\lambda x.xx)(\lambda x.xx) \qquad \omega^* \equiv \Phi(\Psi(\lambda x^*{:}u.\Phi(x^*)x^*))\Psi(\lambda x^*{:}u.\Phi(x^*)x^*)$$

As noted previously, there is an infinite reduction sequence (in λ^u) starting from ω:

$$(\lambda x.xx)(\lambda x.xx) \longrightarrow_\beta (\lambda x.xx)(\lambda x.xx) \longrightarrow_\beta \cdots$$

Corresponding to this is an infinite reduction sequence in $\lambda^{\rightarrow\varphi\psi}$(which includes reduction rules for (φ) and (ψ)):

$$\begin{aligned} \Phi(\Psi(\lambda x^*{:}u.\Phi(x^*)x^*))\Psi(\lambda x^*{:}u.\Phi(x^*)x^*) &\longrightarrow\!\!\!\!\rightarrow_\varphi (\lambda x^*{:}u.\Phi(x^*)x^*)\Psi(\lambda x^*{:}u.\Phi(x^*)x^*) \\ &\longrightarrow\!\!\!\!\rightarrow_\beta \Phi(\Psi(\lambda x^*{:}u.\Phi(x^*)x^*))\Psi(\lambda x^*{:}u.\Phi(x^*)x^*) \\ &\longrightarrow\!\!\!\!\rightarrow_\varphi \cdots \end{aligned}$$

After one (φ) reduction and one (β) reduction the resulting term is again ω^* and so there is an infinite reduction sequence in $\lambda^{\rightarrow\varphi\psi}$. Note that at each stage of this sequence there is only one possible redex. This coincides with the untyped case. So as expected, not all terms in this calculus have a normal form.

3

General Proof Methods for Program Analysis

3.1 Programs as First-Class Objects

Consider using Lisp to write meta-programs for programs written in a simple functional language. While Lisp provides a representation of programs via its notation for λ-terms, the only primitive mechanisms for manipulating such terms in Lisp are essentially those for manipulating lists, namely, CAR, CDR, and CONS. Programs and lists are different objects, however, and the complexity of the structure of programs is not captured by simple list manipulation functions. While any meta-program can be implemented using lists to represent programs and CAR, CDR, and CONS to decompose and construct programs, the resulting implementation of such meta-programs is often complex and difficult to understand. Also, in Lisp, the equality operator EQUAL is not sensitive to the usual meaning of λ-terms. For example, if two Lisp terms differ only in their bound variable names, they are not EQUAL. Thus, while Lisp contains a notation for λ-terms, it does not treat them as being their own data type.

One characteristic that distinguishes between programs (especially functional programs) as values and list structures is that equality between λ-terms is typically considered modulo λ-conversion. This notion of equality is a much more complex operation than simple syntactic equality. In particular, using this notion of equality, a λ-term is equal to any alphabetic variant of itself. With respect to this notion of equality, accessing the name of a bound variable in a λ-term is not a meaningful operation since equal terms might return different values. Adhering to this notion of equality disqualifies most conventional methods

of analyzing the structure of programs.

Higher-order unification is a mechanism that can be used to probe the structure of programs, respecting congruence classes modulo λ-conversion. If the only method for manipulating λ-terms is via higher-order unification then it is impossible to distinguish between two programs which are equal modulo λ-conversion. In particular, it is impossible to access the names of bound variables.

The use of λ-terms and of higher-order unification to implement program manipulation systems has been proposed by various people. Huet and Lang in [29] employed second-order matching (a decidable subcase of higher-order unification) to express certain restricted, "template" program transformations. Miller and Nadathur in [38] extended their approach by adding to their scheme the flexibility of Horn clause programming and richer forms of unification. In [21] we argued that if the Prolog component of the TYPOL system [2] were enriched with higher-order features, logic programming could play a stronger role as a specification language for various kinds of interpreters and compilers.

The abstract syntax for programs and types of the object language we consider is based on the simply typed λ-calculus. We shall represent programs as simply typed terms by introducing an appropriate set of constants from which we can construct terms denoting programs. In general, for each programming language construct we introduce a new constant which is used to build a term representing this construct. We also define new base types (or sorts) corresponding to the different categories of the object language. For example, a simple functional language might require two sorts, one for object-level terms and one for object-level types. We provide an example of such an abstract syntax in the next chapter. In the rest of this chapter, we present the proof and reasoning components of our meta-theory.

While we are only concerned in the current work with the simply typed λ-calculus, richer and more flexible λ-calculi have been proposed as a suitable representation system for programs. For example, Pfenning and Elliot in [48] have extended the simply typed λ-calculus to include simple product types. They also discuss in depth the role of *higher-order abstract syntax*, *i.e.*, the representation of programs as λ-terms, in the construction of flexible and general program manipulation systems. The LF specification language [24] uses a λ-calculus with a strong typing mechanism to specify various components of proof systems: much of this specification language could profitably be used in the context we are concerned with here. While extensions of higher-order unification to such rich notions of terms and types are important, we do not consider them here.

Similar advantages of the blend of higher-order unification and logic programming have been exploited in systems that manipulate formulas and proofs of logical systems. Felty and Miller in [12] discuss the use of λProlog to specify and implement theorem provers and

proofs systems. Here again, λ-terms and higher-order unification are used to represent and manipulate formulas and proofs. The Isabelle theorem prover of Paulson [47] also makes use of these features to implement flexible theorem provers.

3.2 An Abstract Proof System

Given a representation of programs as terms we now describe the general structure of a proof system for manipulating these terms. We consider a natural deduction calculus patterned after Gentzen proof systems [17]. The propositions of this system will typically be binary statements of the form $E : \tau$ or $E \rightarrow F$. Here, of course, we are thinking of E, F, τ as variables which might range over λ-terms. Although propositions can have more complex structure, we shall restrict them to be λ-terms with a constant symbol as their head.

The proof system of our meta-language comes equipped with four built-in inference figures. The first has the structure:

$$\frac{A_1}{A_0}$$

in which the λ-terms representing the propositions in A_0 and A_1 are $\beta\eta$-convertible. By virtue of this rule, we generally think of any two λ-terms as equal if they are $\beta\eta$-convertible. The second inference figure is:

$$\frac{A_1 \quad A_2}{A_1 \;\&\; A_2} \qquad (\&-I)$$

This rule is called *conjunction introduction.* When implementing this inference rule, we interpret it in the following backward fashion: to establish the proposition in $A_1 \;\&\; A_2$, establish the two separate propositions found in A_1 and A_2.

The remaining two rules deal with introduction and discharge. To specify the introduction and discharge of assumptions needed to prove hypothetical propositions we use the following inference figure.

$$\frac{\begin{array}{c}(A_1)\\ \vdots \\ A_2\end{array}}{A_1 \Rightarrow A_2} \qquad (\Rightarrow -I)$$

That is, to prove $A_1 \Rightarrow A_2$, first assume that there is a proof of A_1 and attempt to build a proof for A_2 from it. If such a proof is found, then the implication is justified and the proof of this implication is the result of discharging the assumption about A_1. This rule is called *implication introduction.* Proving a universally quantified proposition has a similar structure, suggesting the following inference figure.

$$\frac{A[x \mapsto c]}{(\forall x)A} \qquad (\forall -I)$$

Here, to prove a universal instance, a new parameter (c) must be introduced and the resulting generic instance of the quantified formula must be proved. Of course, after that instance is proved, the parameter must be discharged, in the sense that c cannot occur free in A or in any undischarged hypotheses. This rule is called *universal introduction.* The corresponding discharge rules are also included but are not used in any of the examples presented.

A specification of a meta-level program will be a collection of atomic propositions which will denote axioms and a collection of inference figures, none of which introduce the symbols $\&, \Rightarrow, \forall$. Of course, the premises to user supplied inference figures can contain instances of these symbols. When providing examples of inference figures later in this proposal, we shall drop references to the connective & in premises. Inference figures of the form

$$\frac{A_1 \ \& \ A_2}{A_0} \quad \text{will simply be written as} \quad \frac{A_1 \quad A_2}{A_0} \ .$$

A proof in this language will be understood in the standard sense of proofs in natural deduction. For more information on natural deduction and its terminology (both of which are used in this proposal) see [17, 53].

3.3 An Implementation of the Meta-Language

Following the observation described in [31] that natural semantics has an intimate connection to logic programming, we show how the preceding four inference figures are related to logic programming. First-order Horn clauses, however, are not strong enough to directly implement these inference rules. First, the notion of equality between terms would be that of simple tree equality, not that of $\beta\eta$-conversion. Horn clauses also do not provide a mechanism for directly implementing the introduction and discharge of parameters and assumptions. It is not difficult to modify our proof system so that the explicit references to introducing and discharging assumptions could be eliminated in favor of treating basic propositions as essentially sequents. That is, a proposition *Prop* would be replaced by a proposition $\Gamma \rightarrow Prop$, in which Γ is used to store assumptions. This is, for example, used in natural semantics to handle contexts. For the examples in this proposal we actually used this approach to implement them in λProlog. We were required to do this since version LP2.6 of λProlog does not fully support implication in goal clauses. A more serious challenge to Horn clauses is that they cannot naturally implement the universally quantified proposition.

There is, however, a generalization of Horn clauses which adds both implications and universal quantifiers to the body of clauses and permits quantification over higher-order

variables. This extension, called *higher-order hereditary Harrop formulas* [39] has (partially) been implemented in the λProlog system. λProlog does, in fact, provide a natural implementation language for these inference rules. For example, the user can specify inference rules by directly writing program clauses containing conjunction, implication, and universal quantifiers, since these are understood on a primitive level of λProlog. For example, clauses of the form

$$A_0 \ :- \ A_1 \ \& \ (\forall x)(A_2 \Rightarrow A_3).$$

can be used to represent complex inference figures. Free (higher-order) variables here are assumed to be universally quantified over the scope of the full clause. Instead of using this kind of syntax to present example inference rules in later chapters, we shall continue to use the more graphically oriented inference figures. All the examples presented in this proposal have been implemented and tested in a version of λProlog.

4

Analysis of a Simple Functional Language $\mathbf{PCF}_0$

4.1 Presenting an Object Language

As an example of our proof-theoretic methods, we present a simple functional language PCF_0, (essentially the same language as introduced in [51]). This language is based on a simply typed λ-calculus and can be viewed as a subset of Standard ML, containing just the functional part of that language. (It does not contain exceptions, pattern matching, data type declarations or modules.) This presentation demonstrates how an abstract syntax for a functional language can be constructed using simply typed lambda terms and also how the unique properties of our methods can be exploited in the manipulation of programs. We take care in making the distinction between terms and types at the object (PCF_0) level and terms and types at the meta-level. We refer to the latter as meta-terms and meta-types. We have two base meta-types, *tm* and *tp*, representing object-level terms and types, respectively.

To define our abstract syntax for PCF_0 we begin by giving a signature for some meta-terms that we use to construct terms and types at the object level. (See Figure 4.1.) Notice that the constants **lamb**, **let** and **fix** are higher-order, that is, they each require a functional argument of type $tm \rightarrow tm$. In the examples that follow M will be used as a higher-order variable of this meta-type. '$\rightarrow$' is the function space constructors for *tp*. We have overloaded the symbol '$\rightarrow$', using it at both the object and meta levels; its use, however, should always be clear from context. The object types we consider are only monotypes (in the sense of [41] as we do allow type variables). Expressions with polytypes (i.e., monotypes that may be

prefixed by universal quantification over type variables) do arise, however, in PCF_0. Later in this chapter we present a separate discussion of polytypes.

meta-term	meta-type
true, false	tm
$0, 1, 2, \ldots$	tm
if	$tm \rightarrow tm \rightarrow tm \rightarrow tm$
'@'	$tm \rightarrow (tm \rightarrow tm)$
lamb	$(tm \rightarrow tm) \rightarrow tm$
let	$(tm \rightarrow tm) \rightarrow tm \rightarrow tm$
fix	$(tm \rightarrow tm) \rightarrow tm$

meta-term	meta-type
int	tp
$bool$	tp
'$\rightarrow$'	$tp \rightarrow tp \rightarrow tp$

FIGURE 4.1
Signature for PCF_0 Terms and Types

4.2 An Abstract Syntax for PCF_0

Figure 4.2 contains the higher-order abstract syntax for expressions in PCF_0. The first few lines treat constants, variables and the conditional in a traditional manner. Application is made explicit with the infix operator '@'. For lambda abstraction we introduce the constructor **lamb** which takes a meta-term M of the form $\lambda x.e$, in which x and e are of meta-type tm, and produces a PCF_0 term. Similar to **lamb**, the **let** construct uses a meta-term M of the form $\lambda x.e$ to represent the binding of an identifier. To accommodate recursion we introduce the **fix** construct which again uses an explicit abstraction to capture the binding. Thus, we employ the general principal that bindings at the object level have an associated abstraction at the meta-level.

This abstract syntax is essentially the embedding of the untyped lambda calculus into a simply typed calculus as described in Chapter 2. Using the notation of [36] our meta-term **lamb** corresponds to the function Ψ, for coercing functions into terms. The meta-term '@' corresponds to the function Φ for coercing terms into functions. Thus our representation of PCF_0 code is essentially the same as first encoding them as untyped lambda terms and then embedding them into the typed calculus using Φ and Ψ.

Throughout most of this proposal we will avoid discussing primitive operations of PCF_0, such as $+$, $-$, *etc.* They are, of course, important to have in the full language but including them here is neither difficult nor illuminating. In subsequent examples we shall systematically drop the apply "@" operator in order to make examples more readable.

syntax	description
c	constants: integers, true, false, etc.
x	variables
(if e_1 e_2 e_3)	if e_1 then e_2 else e_3
(e_1 @ e_2)	application
(lamb M)	$M = \lambda x.e$ (lambda abstraction)
(let M e_2)	$M = \lambda x.e_1$ (let $x = e_2$ in e_1)
(fix M)	$M = \lambda f.e$ (least fixed point operator)

FIGURE 4.2
Abstract Syntax for PCF_0

4.3 Environments versus Abstractions

Before presenting the type inference proof procedure we make another distinction between our method and typical approaches to natural or operational semantics. This distinction concerns the treatment of identifiers. The typical approach to analyzing programs uses an environment (or context) to denote a finite mapping from identifiers to some domain (*e.g.*, types or terms). When analyzing an abstraction, the bound variable is stripped from the abstraction and the identifier which names that bound variable is added to the context. The meaning of such an identifier within the body of the abstraction is then determined by "looking up" the value associated with the identifier in the current environment. We refer to this technique as the environment approach.

Given our commitment to representing program abstractions using abstractions with λ-terms and to equating such terms when they are $\beta\eta$-convertible, it is impossible to access the bound variable name of a λ-term at the meta-level, since such an operation would return different answers on equal terms. A combination of the $\forall$ and $\Rightarrow$ propositions, however, can provide a very simple solution to this problem. When an abstraction is encountered, typically within **lamb**, **let** and **fix** constructions, a $\forall$ judgement is used to introduce a new parameter. That parameter is then substituted into the abstraction using β-conversion. The value or type to be associated with this new parameter is then introduced as an assumed proposition. In this way, the newly introduced identifier is used to stand for the name of the bound variable.

This relation between the environment approach and our technique is similar to an observation by Plotkin about evaluations in the SECD machine [49]. There two different evaluation functions were defined: the awkward *Eval* function defined in terms of closures

and the simpler *eval* defined using substitution (β-conversion, here). While these two functions were shown to be equivalent, introducing the simpler definition for evaluation allowed properties of the SECD machine to be described much more naturally than with the first, more cumbersome, definition. Similarly, we believe that the use of abstractions and substitution in our meta-language will often produce this kind of advantage over programs using the environment approach. (For an example of this, see Section 6.3.) However we note that the environment approach is more amenable to optimizations and efficient implementation [31].

4.4 Examples of Static Semantics

Static semantics refers to a class of program analyses that provide information about programs based on their static structure (*i.e.*, not considering their behavior during some form of evaluation). One common example of a static semantics is type inferencing. An example of this kind of analysis is given below. Other kinds of static analysis include type checking, certain kinds of flow analysis and possibly complexity analyzers.

4.4.1 Type Inference

The proof system for type inference in our formulation of PCF_0 is given in Figure 4.3. A proof of the proposition $E \xrightarrow{ty} \tau$, in which E is a closed expression given in the above abstract syntax, states that E has type τ. To be precise we should prove certain properties about this typing system, *e.g.*, soundness, completeness and principal typing [9, 27]. We discuss these issues and others below and in a later chapter, but for now we concentrate on providing an informal description of the system. The first three clauses (actually axioms) are for typing the constants of the language; here N denotes any integer. The next clause gives the usual typing for the conditional statement.

Clause 5 is the typing rule for lambda abstraction and it is a bit different from the usual typing rule using environments. In the environment approach, typing the (first-order) term ($\lambda\ x.E$) would first require adding the type assignment $x : \tau_1$ to the environment, then computing the type of E in this new environment to be τ_2, and then finally inferring the type of the original term to be $\tau_1 \rightarrow \tau_2$. Our rule uses β-reduction and operationally works as follows. Given the term (**lamb** M) we first pick a new constant c and assume it has type τ_1 (i.e., we introduce the assumption $c \xrightarrow{ty} \tau_1$). Under this assumption we then type (the $\beta\eta$-normal form of) the term ($M\ c$). If M is of the form $\lambda x.e$ then the β-reduction

$$N \xrightarrow{ty} int \qquad \mathsf{true} \xrightarrow{ty} bool \qquad \mathsf{false} \xrightarrow{ty} bool \tag{1,2,3}$$

$$\frac{e_1 \xrightarrow{ty} bool \qquad e_2 \xrightarrow{ty} \tau \qquad e_3 \xrightarrow{ty} \tau}{(\mathsf{if}\ e_1\ e_2\ e_3) \xrightarrow{ty} \tau} \tag{4}$$

$$\frac{(\forall c)\ (c \xrightarrow{ty} \tau_1 \ \Rightarrow\ (M\ c) \xrightarrow{ty} \tau_2)}{(\mathsf{lamb}\ M) \xrightarrow{ty} (\tau_1 \rightarrow \tau_2)} \qquad \frac{e_1 \xrightarrow{ty} (\tau_1 \rightarrow \tau_2) \qquad e_2 \xrightarrow{ty} \tau_1}{(e_1 @ e_2) \xrightarrow{ty} \tau_2} \tag{5,6}$$

$$\frac{e_2 \xrightarrow{ty} \tau_2 \qquad (M\ e_2) \xrightarrow{ty} \tau_1}{(\mathsf{let}\ M\ e_2) \xrightarrow{ty} \tau_1} \qquad \frac{(\forall c)\ (c \xrightarrow{ty} \tau \ \Rightarrow\ (M\ c) \xrightarrow{ty} \tau)}{(\mathsf{fix}\ M) \xrightarrow{ty} \tau} \tag{7,8}$$

FIGURE 4.3
Type Inference for PCF_0

is, in this case, equivalent to the substitution $e[x \mapsto c]$. If we infer the type τ_2 for this term then we infer the type of the original term to be $\tau_1 \rightarrow \tau_2$. Informally, this infers the correct type because every occurrence of x bound by this abstraction has been replaced by a term c whose type will be inferred to be τ_1. Although this is in many ways similar to the environment approach, it avoids the need to access the names of bound variables.

Clause 6 is the usual typing rule for application. Clause 8 for fixed points uses the same technique as **lamb**, though in this case we know that M must be of type $\tau \rightarrow \tau$ for some τ. Clause 7 requires some explanation. The more standard implementation of type inference for **let** first infers the principle type for e_2, then generalizes that type with a universal quantifier over type variables, yielding a polytype. Later in the typing of the abstraction M, various universal instances of this polytype could be made for instances of the abstracted variable of M. Our meta-language, however, contains no method for generalizing a free variable into a bound variable, and so this kind of implementation is not possible here. Instead, we avoid inferring a polytype for e_2 explicitly. Clause 7 requires that e_2 have some type, but that type is then ignored. β-reduction is used to substitute e_2 into the abstraction M, and then the type of the result is inferred. If e_2 is placed into several different places in M, each of those instances will again have a type inferred for them; this time the types might be different. Therefore, e_2 could be polymorphic in that its occurrences in M might be at several different types.

We do not need a rule for typing identifiers because any identifier occurring in a term is replaced via β-reduction with either (i) a term explicitly typed via an assumption (**lamb**,

fix) or (*ii*) a term whose type has already been inferred (**let**). (Recall that we are typing only closed expressions.) Note that the three clauses that make significant use of higher-order features correspond precisely to the three clauses in the environment approach that extend the environment. This is not surprising as these are the only three clauses that introduce identifiers and bindings.

4.4.2 Coping With Open Expressions.

In the discussion above, and in later discussions, we tacitly assume that we are dealing with only closed expressions. We would like our methods to be more general, though, and we now present a technique for manipulating terms with free variables. Essentially, there is no proof of a typing judgment for an open expression e using the proof system given above. This is because that system does not provide an axiom for variables (as all bound variables are eventually replaced by universal constants). This observation suggests two possible approaches to coping with free variables: (1) add additional axioms and/or inference rules to the proof system, or (2) replace the free variables with new universal constants. The former approach then treats bound variables and free variables entirely differently, and this distinction seems counterintuitive. The latter approach, however, suggests a uniform treatment of variables and this is the one we pursue.

Clearly, given an untyped expression with free variables we require additional information (assumptions) about these variables to type the expression. This information can be presented as a set of assumptions such as $\{(x_0, \tau_0), \ldots, (x_k, \tau_k)\}$, associating each free variable x_i to a type τ_i. Thus given an expression e with the above assumptions on its free variables we shall consider the proposition

$$(\forall c_0 c_0 \xrightarrow{ty} \tau_0 \Rightarrow \cdots \forall c_k c_k \xrightarrow{ty} \tau_k \Rightarrow [c_0/x_0, \ldots, c_k/x_k]e \xrightarrow{ty} \tau)$$

and attempt to prove it in our type system. Informally, we argue that variables can no longer appear at the leaves of trees and only universal constants can appear there. A more formal argument is given in a later chapter.

Many approaches to type inference supply the additional information via a finite (partial) function Γ from variables to types such that $\Gamma(x_i) = \tau_i$. For convenience we will refer to the proposition above as the proposition for typing e with respect to Γ.

4.4.3 The Subsumes Relation for Polytypes

As a second example of using our meta-language to manipulate ML-like types, we present a proof system for the subsumes relation on polytypes [41]. For this purpose, we now introduce a higher-order constant for constructing ML types, namely the type quantifier forall which is of meta-type $(tp \rightarrow tp) \rightarrow tp$. Any term of type tp which does not contain an instance of this constant is a monotype. A term of type tp in which all of occurrences of forall are in its prefix (that is, no occurrence of forall is in the scope of $*$ or $\rightarrow$) is called a *polytype* (a monotype is a polytype). It is possible to construct terms (of meta-type tp) that are neither monotypes nor polytypes, but these will not interest us here. In the following discussion, the greek letter τ will represent a monotype and σ a polytype. Before defining the subsumes relation we define an auxiliary definition.

DEFINITION 4.1 (Instance of a Polytype) τ is an instance of polytype (forall $\lambda t_1(\ldots$(forall λt_n $(\tau'))\ldots)$) if there exists a substitution S of the variables $t_1, \ldots, t_n$ into monotypes such that $\mathcal{S}(\tau') = \tau$.

The subsumes relation on polytypes is then given by the following.

DEFINITION 4.2 (Subsumes) Let σ_1 and σ_2 be two polytypes. σ_1 subsumes σ_2, written $\sigma_1 \sqsubseteq \sigma_2$, if every instance of σ_2 is also an instance of σ_1.

For example, the polytype (forall $\lambda t.t$) subsumes all other polytypes. An informal operational description of this definition is the following. Given σ_1 and σ_2, erase the quantifiers of each yielding two monotypes, τ_1 and τ_2. Then $\sigma_1 \sqsubseteq \sigma_2$ iff there exists a substitution $\mathcal{S}$ such that $\mathcal{S}(\tau_1) = \tau_2$. Since the erasure of bound variables is another operation not available in our meta-language, we need to approach the implementation of subsumes differently.

In our meta-language we can construct a simple proof system for the subsumes relation; it is given in Figure 4.4. The first clause states the obvious: any polytype subsumes itself. The second clause produces a 'canonical' instance of σ_2. This step is essentially like the process of erasing a type quantifier. The meta-level universal quantifier used in this clause ensures that, after removing the quantifiers on σ_2, revealing a monotype, any future substitution does not affect this monotype (its free variables are, in a sense, protected). The third clause is used to build an instance of the first type by stripping off a quantifier (replacing a bound (type) variable with a free one).

Notice that these three proof rules have a simple declarative reading. Assume that types are interpreted as sets of objects of that type, that forall is interpreted as intersection, and $\sqsubseteq$ as subset. The second clause states that a type is a subset of the intersection of a

$$\sigma \sqsubseteq \sigma \qquad \frac{(\forall c)\ \sigma_1 \sqsubseteq (M\ c)}{\sigma_1 \sqsubseteq (\mathsf{forall}\ M)} \qquad \frac{(M\ x) \sqsubseteq \sigma_2}{(\mathsf{forall}\ M) \sqsubseteq \sigma_2}$$

FIGURE 4.4
Subsumes Relation for Polytypes

family of types if it is a subset of all members of the family. The third clauses similarly states that if some member of a family is a subset by a given type, then the intersection of that family is a subset of that type.

4.4.4 Properties of Type Inference Semantics for PCF_0.

We would like to consider some meta-theoretic properties of the proof system for type inference. We begin with a normal-form theorem. We want to show that for any proof derivable in this system, there exists a smallest or 'canonical' proof. For the present example this task is simplified somewhat by the lack of elimination rules in our specification of type inference. Roughly, an elimination rule is an inference rule in which some logical connective or constant appears in the antecedent but not in the consequent. Recall that the only built-in inference figures we have made use of are $(\&-I, \Rightarrow -I, \forall -I)$ and $(\beta\eta)$-rule. Therefore, in describing normal-form proofs, we need only be concerned with the use of the $\beta\eta$-rule.

Before presenting a normal-form theorem we first need some information about the structure of certain terms occurring in proofs. We begin by tacitly assuming that PCF_0 is extended with the set of all constants c that may be introduced via universal introduction. This is justified by the fact that we assume that $\forall$ ranges over elements of PCF_0. With this in mind, the following lemma characterizes the meta-terms appearing in proofs.

LEMMA 4.3 Let $e \in PCF_0$ be one of the terms (lamb M), (let M e_2) or (fix M) for some meta-term M of type $(tm \to tm)$, such that e is in $\beta\eta$-normal form. Then:

(i) M must be of the form $\lambda x.e_1$ or $(e_1\ @)$ or $(if\ e_1\ e_2)$, for some $e_1, e_2 \in PCF_0$ in $\beta\eta$-normal form; and

(ii) for any $e' \in PCF_0$ in $\beta\eta$-normal form, $(M\ e')$ is either in normal form or one-step β-reduces to a normal-form term.

The proof is by induction on the structure of terms and is not included here.

THEOREM 4.4 (Normal Form) If $\vdash e \xrightarrow{ty} \tau$, then there exists a proof Ξ of this proposition such that the following hold

(i) the proposition $e \xrightarrow{ty} \tau$, considered as a λ-term of type o, is in $\beta\eta$-normal form

(ii) all instances of the $(\beta\eta)$ inference figure occur only in the following contexts:

$$\frac{(\forall c)\ (c \xrightarrow{ty} \tau_1 \ \Rightarrow \ \dfrac{M' \xrightarrow{ty} \tau_2}{(M\ c) \xrightarrow{ty} \tau_2)\ (\beta\eta)}}{(\mathsf{lamb}\ M) \xrightarrow{ty} (\tau_1 \to \tau_2)}$$

$$\frac{e_2 \xrightarrow{ty} \tau_2 \qquad \dfrac{M'' \xrightarrow{ty} \tau_1}{(M\ e_2) \xrightarrow{ty} \tau_1}\ (\beta\eta)}{(\mathsf{let}\ M\ e_2) \xrightarrow{ty} \tau_1} \qquad \frac{(\forall c)\ (c \xrightarrow{ty} \tau \ \Rightarrow \ \dfrac{M' \xrightarrow{ty} \tau}{(M\ c) \xrightarrow{ty} \tau)\ (\beta\eta)}}{(\mathsf{fix}\ M) \xrightarrow{ty} \tau}$$

(for some terms M' and M'') such that $(M\ c) \longrightarrow_\beta M'$ and $(M\ e_2) \longrightarrow_\beta M''$ (one-step β-reduction) and M' and M'' are normal-form terms.

REMARK. Proofs in such form shall be called normal-form proofs (with respect to the given proof system). Note that $\beta\eta$-normal forms of propositions must always exists since the propositions are represented by simply typed λ-terms and the simply typed λ-calculus is strongly normalizing.

PROOF. The proof is by induction on the height h of proof tree Ξ with (at most) open assumptions of the form $c_i \xrightarrow{ty} \tau$ for some constants c_i excluding integers, true and false. We shall refer to such proof trees as *open proof trees.* (Note that a proof tree is also an open proof tree.)

base: $h = 0$. Then Ξ must consist of just an axiom or an open assumption and be of the form

$$N \xrightarrow{ty} int \qquad \text{or} \qquad \mathsf{true} \xrightarrow{ty} bool \qquad \text{or} \qquad \mathsf{false} \xrightarrow{ty} bool \qquad \text{or} \qquad c_i \xrightarrow{ty} \tau_i$$

(for some constant c_i and type τ_i). Then the conditions hold trivially.

step: $h = n$ for some $n > 1$. Assume the theorem hold for all $1 \leq m < n$. Then we consider the last inference rule of Ξ. We note that the last inference rule cannot be $\& - I$, $\Rightarrow - I$ or $\forall - I$ as the root of Ξ must be of the form $e \xrightarrow{ty} \tau$. We consider the remaining possible cases:

(i) The last inference rule is of the form

$$\frac{A_1}{A_2}\ (\beta\eta)$$

in which A_1 and A_2 are $\beta\eta$-convertible. By inductive hypothesis there exist a normal-form open proof tree Ξ' whose root is A_1' such that A_1' is a normal-form term and is $\beta\eta$-convertible to A_1. But then A_1' is also convertible to A_2 and so we take Ξ' as the normal-form open proof tree.

(ii) The last inference rule was

$$\frac{e_1 \xrightarrow{ty} bool \qquad e_2 \xrightarrow{ty} \tau \qquad e_3 \xrightarrow{ty} \tau}{(\mathsf{if}\ e_1\ e_2\ e_3) \xrightarrow{ty} \tau}$$

By inductive hypothesis there exist normal-form open proof trees Ξ_1', Ξ_2' and Ξ_3' for the propositions $e_1' \xrightarrow{ty} bool$, $e_2' \xrightarrow{ty} \tau$ and $e_3' \xrightarrow{ty} \tau$, respectively. corresponding to the proofs Ξ_1, Ξ_2 and Ξ_3 for the propositions $e_1 \xrightarrow{ty} bool$, $e_1 \xrightarrow{ty} \tau$ and $e_2 \xrightarrow{ty} \tau$, respectively. e_1', e_2', e_3' are the normal-form terms for e_1, e_2, e_3, respectively. But then we can construct an open proof tree Ξ' of the form

$$\frac{\Xi_1' \qquad \Xi_2' \qquad \Xi_3'}{(\mathsf{if}\ e_1'\ e_2'\ e_3') \xrightarrow{ty} \tau}$$

that satisfies the conditions of the theorem.

The step for other constants (not involving meta-level bound variables) is similar and not presented here.

(iii) The last inference rule was

$$\frac{(\forall c)\ (c \xrightarrow{ty} \tau_1 \ \Rightarrow\ (M\ c) \xrightarrow{ty} \tau_2)}{(\mathsf{lamb}\ M) \xrightarrow{ty} (\tau_1 \rightarrow \tau_2)}.$$

By inductive hypothesis there exists a normal-form open proof tree Ξ' for the proposition $(e' \xrightarrow{ty} \tau_2)$ in which e' is the normal form of $(M\ c)$ (whose open assumptions may include $(c \xrightarrow{ty} \tau_1)$). Now let M' be the normal form of M. Then by Lemma 4.3, $(M'\ c)$ (at most) one-step β-reduces to e'. Hence we can construct an open proof of the normal form term $(\mathsf{lamb}\ M') \xrightarrow{ty} (\tau_1 \rightarrow \tau_2)$ that satisfies condition (ii) of the theorem (and whose open assumptions do not include $(c \xrightarrow{ty} \tau_1)$).

The cases for **let** and **fix** are similar and not presented here. □

We also have that normal-form proofs are unique:

COROLLARY 4.5 If Ξ and Ξ' are both normal-form proofs of $e \xrightarrow{ty} \tau$, then Ξ and Ξ' differ only in a consistent renaming of bound variables.

Note that if we were to consider proofs as simply typed terms of, say, type *pf*, we would have that Ξ and Ξ' are just α-convertible as terms.

A second property of our type inference system is the existence of principal types.

THEOREM 4.6 (Principal Type) Let $e \in PCF_0$. If for some type τ', $\vdash e \xrightarrow{ty} \tau'$, then there exists some type τ such that $\vdash e \xrightarrow{ty} \tau$, and for any other type τ'', $\vdash e \xrightarrow{ty} \tau'' \Rightarrow \tau \sqsubseteq \tau''$, *i.e.*, there exists some substitution $\mathcal{S}$ from type variables to monotypes such that $\mathcal{S}(\tau) = \tau''$.

We don't give the proof of this theorem here, but in Chapter 6 we will give an indirect proof.

4.5 An Example of Dynamic Semantics

Dynamic semantics refers to a class of program analyses that provide information about programs based on a dynamic behavior, *i.e.*, some set of evaluation rules is assumed and the behavior of programs under these rules in considered. In this section we present a standard evaluation semantics that provides a declarative specification for a PCF_0 interpreter. Other, non-standard semantics are also possible and two such are given in the next chapter.

4.5.1 Standard Evaluation Semantics for PCF_0

We would like to specify the evaluation of expressions in PCF_0, based on a simple interpreter for the language. (We say standard here to distinguish from a non-standard semantics.) Following [31] we refer to a formal specification of an evaluator for a language as the language's *dynamic semantics*. We characterize the dynamic semantics of an object language via judgements of the form $\vdash E \longrightarrow \alpha$ in which E is an expression of the object language and α is the result of evaluating E. Informally, the terms appearing to the left of $\longrightarrow$ denote PCF_0 expressions and the terms appearing to the right are the "values" or meanings of the expressions. Thus, we may have a specification in which the set of values is not a subset of the language (as is the case in [31]). We will not discuss further the technical merits of having the set of values contained in the language. By providing rules corresponding to the operational behavior of the language (with the general guideline of having one rule for each programming language construct) we can specify the declarative aspects of interpreters (or evaluators)

for the language, isolated from control issues. As mentioned previously this provides a convenient tool for analyzing and experimenting with new programming languages.

We now present a dynamic semantics for PCF_0, using the same higher-order abstract syntax as given in the previous chapter. Propositions in our system are of the form $E \xrightarrow{se} \alpha$ in which E and α are expressions in PCF_0 and α is the result of "evaluating" E. Proofs of these propositions are constructed from the proof system given in Figure 4.5. If a proposition P is provable in this system we write $\vdash P$. The first three rules treat the constants of the language (N here is any integer) as just evaluating to themselves. The next two rules treat the conditional in the natural way. Rule (5) states that an abstraction evaluates to itself. In the rule for application (6), meta-level β-reduction correctly captures the notion of function application (with a call-by-value semantics). Similar comments apply to our rule for let (7). In the rule for recursion (8) we introduce a fixed point operator with its intuitive operational semantics (*i.e.*, unfolding). This again makes explicit use of meta-level β-reduction as the meta-term M is applied to the term (fix M). The result of β-converting this expression substitutes the recursive call, namely (fix M), within the body of the recursive program, given by M. Static scoping is ensured with this specification because β-reduction, as a means of propagating binding information, guarantees that the identifiers occurring free within a lambda abstraction are replaced (with their associated value) prior to manipulating the abstraction. To present an example later, we shall add some basic list manipulation operations to the syntax of PCF_0 and its evaluation semantics: these operations are not central to our analysis of PCF_0.

The values implicitly defined by this specification (*i.e.*, the set of terms that can appear to the right of $\xrightarrow{se}$) are just the set of constants, lambda abstractions and primitive constructors. In general, the set of values may not always be a subset of the language (as is the case in [30]). Now given some closed expression e we can think of evaluating e by finding some value α such that $\vdash e \xrightarrow{se} \alpha$. We assume some non-deterministic search procedure is used to find such an α and construct such a proof.

4.5.2 Properties of the Dynamic Semantics for PCF_0.

As we did with the static semantics, we would like to consider some meta-theoretic properties of the proof system for the dynamic semantics of PCF_0. We begin with a normal-form theorem. We want to show that for any proof derivable in this system, there exists a smallest or 'canonical' proof. As with the static semantics, this task is simplified somewhat by the lack of elimination rules in our proof system. Considering the inference rules of Figure 4.5 we observe that each rule introduces (in its consequent) a top-level constant of the object

$$N \xrightarrow{se} N \qquad \mathsf{true} \xrightarrow{se} \mathsf{true} \qquad \mathsf{false} \xrightarrow{se} \mathsf{false} \qquad (1,2,3)$$

$$\frac{e_1 \xrightarrow{se} \mathsf{true} \qquad e_2 \xrightarrow{se} \alpha}{(\mathsf{if}\ e_1\ e_2\ e_3) \xrightarrow{se} \alpha} \qquad \frac{e_1 \xrightarrow{se} \mathsf{false} \qquad e_3 \xrightarrow{se} \alpha}{(\mathsf{if}\ e_1\ e_2\ e_3) \xrightarrow{se} \alpha} \qquad (4a, 4b)$$

$$(\mathsf{lamb}\ M) \xrightarrow{se} (\mathsf{lamb}\ M) \qquad (5)$$

$$\frac{e_1 \xrightarrow{se} (\mathsf{lamb}\ M) \qquad e_2 \xrightarrow{se} \alpha_2 \qquad (M\ \alpha_2) \xrightarrow{se} \alpha}{(e_1 @ e_2) \xrightarrow{se} \alpha} \qquad (6)$$

$$\frac{e_2 \xrightarrow{se} \alpha_2 \qquad (M\ \alpha_2) \xrightarrow{se} \alpha}{(\mathsf{let}\ M\ e_2) \xrightarrow{se} \alpha} \qquad \frac{(M\ (\mathsf{fix}\ M)) \xrightarrow{se} \alpha}{(\mathsf{fix}\ M) \xrightarrow{se} \alpha} \qquad (7,8)$$

$$\frac{e_1 \xrightarrow{se} \alpha_1 \quad \cdots \quad e_n \xrightarrow{se} \alpha_n \qquad f_{op}(\alpha_1, \ldots, \alpha_n) = \alpha}{(op\ e_1 \ldots e_n) \xrightarrow{se} \alpha} \qquad (9)$$

FIGURE 4.5
Standard Evaluation Semantics for PCF_0

language (*e.g.*, lamb, let, *etc.*).

THEOREM 4.7 (Normal Form) If there exists a proof of a formula $e \xrightarrow{se} \alpha$, for normal-form terms e and α, then there exists a unique proof Ξ of this formula such that all instances of the $(\beta\eta)$ inference figure occur in one of the following contexts:

$$\frac{e_1 \xrightarrow{se} (\mathsf{lamb}\ M) \qquad e_2 \xrightarrow{se} \alpha_2 \qquad \dfrac{M' \xrightarrow{se} \alpha}{(M\ \alpha_2) \xrightarrow{se} \alpha}\ (\beta\eta)}{(e_1 @ e_2) \xrightarrow{se} \alpha}$$

$$\frac{e_2 \xrightarrow{se} \alpha_2 \qquad \dfrac{M' \xrightarrow{se} \alpha_1}{(M\ \alpha_2) \xrightarrow{se} \alpha_1}\ (\beta\eta)}{(\mathsf{let}\ M\ e_2) \xrightarrow{se} \alpha} \qquad \frac{\dfrac{M'' \xrightarrow{se} \alpha}{(M\ (\mathsf{fix}\ M)) \xrightarrow{se} \alpha}\ (\beta\eta)}{(\mathsf{fix}\ M) \xrightarrow{se} \alpha}$$

such that $(M\ \alpha_2) \longrightarrow_\beta M'$ and $(M\ (\mathsf{fix}\ M)) \longrightarrow_\beta M''$ (one-step β-reduction) and M' and M'' are normal-form terms.

The proof of this theorem is similar to the one for the static semantics and is not presented here. If we assume that the specifications for all primitive functions are deterministic,

then we have the following theorem.

THEOREM 4.8 Let $e \in PCF_0$. If for some expression α, there exists a proof Ξ of the formula $e \xrightarrow{se} \alpha$, then for any other proof Ξ' of the formula $e \xrightarrow{se} \alpha'$, we have $\Xi = \Xi'$ and $\alpha = \alpha'$.

The equality of proofs used here is defined by considering normal-form proofs modulo α-conversion and then comparing the structure of the proofs as trees. This is well-defined since all proofs have a normal form. The proof is straightforward and not given here. In a complete system, *i.e.*, one that includes rules for primitive functions, we must ensure that the specification for each of these functions is also deterministic (*e.g.*, a given boolean expression evaluates to either **true** or **false**, but not possibly both.

To assist us later we formalize the notion of "values" in PCF_0 and the undecidability of program termination.

DEFINITION 4.9 The set $\mathcal{V}$ of *Values* of PCF_0 is the smallest set containing all constants, all terms with a primitive constructor as their head and values for subterms and all expressions of the form (**lamb** M) for some M.

LEMMA 4.10 If $\vdash e \xrightarrow{se} \alpha$ then $\alpha \in \mathcal{V}$.

The proof is straightforward by induction on the structure of e.

COROLLARY 4.11 If $e \in \mathcal{V}$ then $\vdash e \xrightarrow{se} \alpha$ implies $e = \alpha$ (up to $\beta\eta$-convertibility).

I.e., values only evaluate to themselves.

And finally, we include an "obvious" result on computability of proofs:

THEOREM 4.12 It is not decidable whether given any $e \in PCF_0$ there exists some $\alpha \in \mathcal{V}$ such that $\vdash e \xrightarrow{se} \alpha$.

This undecidability stems from the presence of the fixed point operation. If we consider only those expressions e not containing **fix**, then the question is decidable.

4.6 An Example of Compilation Semantics

Compilation semantics refers to a class of program analyses that describe translations from a source language (PCF_0) to some low-level machine language. For the final example in

this chapter we take the translation from PCF_0 to CAM given in [11] and present it using our techniques.

4.6.1 The Categorical Abstract Machine

The Categorical Abstract Machine (CAM) has its roots in both categories and de Bruijn's notation for λ-terms. It is a very simple stack-based machine in which, according to its inventors, "categorical terms can be considered as code for acting on a graph of values." For further information on the CAM see [7]. The architecture consists of only a few instructions and are quite similar to traditional (architecture) machine instructions. We have specified a proof system providing a dynamic semantics for the CAM. As the CAM is a low-level stack-based machine, higher-order syntax provides little advantage in specifying its semantics. Values in the CAM must be explicitly maintained on a stack, thus forming a kind of environment; hence we could not dispense with environments. We were, however, able to avoid the use of infinite structures for handling the **rec** command. In the first-order system of [30], the **rec** command, which allows recursion, is handled by constructing a cyclic (hence, infinite) environment. We construct a higher-order object for the environment and then represent this recursive environment by a fixed point. This specification, we believe, provides a clear picture of the underlying stack manipulation of the CAM. Our specification for the CAM dynamic semantics is given in Figure 4.6.

4.6.2 Translation from PCF_0 to CAM

The translation from PCF_0 to CAM is based on the translation presented in [11] and [30]. The inference figures for this translation are given in Figure 4.7. We were able to replace the use of environments with de Bruijn indices (the D's occurring in the proof rules). Such a simple addressing scheme is due partly to our restriction that bindings refer only to individual identifiers. When dealing with identifiers, our presentation is somewhat simpler than that of [11]. We give only an overview here of the functioning of this proof system. We have not presented the constructors for the abstract syntax for the CAM since they all have straightforward first-order types.

Given a PCF_0 term e we define the *depth* of a subterm e_1 of e to be the number of variable bindings in e of which e_1 is in the scope. The proposition $D : e \xrightarrow{\text{pcf-cam}} C$ then has the declarative reading: "the PCF_0 term e, occurring at a depth D in some term, translates to the CAM code C." The depth of a subterm is needed in order to generate the correct CAM code for accessing the value of PCF_0 identifiers. Identifiers are translated into access paths

$$\frac{(\mathit{init\text{-}stack},\ \mathrm{coms}) \xrightarrow{cam} \alpha \cdot s}{\mathit{program}(\mathrm{coms}) \xrightarrow{cam} \alpha} \tag{1}$$

$$(s,\ \emptyset) \xrightarrow{cam} s \qquad \frac{(s,\ \mathrm{com}) \xrightarrow{cam} s_1 \qquad (s_1,\ \mathrm{coms}) \xrightarrow{cam} s_2}{(s,\ \mathrm{com;coms}) \xrightarrow{cam} s_2} \tag{2,3}$$

$$(\alpha \cdot s,\ \mathit{quote}(v)) \xrightarrow{cam} v \cdot s \tag{4}$$

$$((\alpha, \beta) \cdot s,\ \mathit{car}) \xrightarrow{cam} \alpha \cdot s \qquad ((\alpha, \beta) \cdot s,\ \mathit{cdr}) \xrightarrow{cam} \beta \cdot s \qquad (\alpha \cdot \beta \cdot s,\ \mathit{cons}) \xrightarrow{cam} (\alpha, \beta) \cdot s \tag{5,67}$$

$$(\alpha \cdot s,\ \mathit{push}) \xrightarrow{cam} \alpha \cdot \alpha \cdot s \qquad (\alpha \cdot \beta \cdot s,\ \mathit{swap}) \xrightarrow{cam} \beta \cdot \alpha \cdot s \tag{8,9}$$

$$\frac{\mathrm{op}, \alpha \xrightarrow{\mathrm{eval}} \beta}{(\alpha \cdot s,\ \mathit{op}(\mathrm{op})) \xrightarrow{cam} \beta \cdot s} \tag{10}$$

$$\frac{(s,\ C_1) \xrightarrow{cam} s_1}{(\mathit{true} \cdot s,\ \mathit{branch}(C_1, C_2)) \xrightarrow{cam} s_1} \qquad \frac{(s,\ C_2) \xrightarrow{cam} s_1}{(\mathit{false} \cdot s,\ \mathit{branch}(C_1, C_2)) \xrightarrow{cam} s_1} \tag{11,12}$$

$$(\rho \cdot s,\ \mathit{cur}(C)) \xrightarrow{cam} [\![C, \rho]\!] \cdot s \tag{13}$$

$$\frac{((\rho, \alpha) \cdot s,\ C) \xrightarrow{cam} s_1}{(([\![C, \rho]\!], \alpha) \cdot s,\ \mathit{app}) \xrightarrow{cam} s_1} \qquad \frac{((\rho, \rho_1) \cdot s,\ C) \xrightarrow{cam} (P\ \rho_1) \cdot s}{(\rho \cdot s,\ \mathit{rec}(C)) \xrightarrow{cam} (\mathit{fix}\ P) \cdot s} \tag{14,15}$$

FIGURE 4.6
Dynamic Semantics for CAM

into an environment on top of the CAM's stack. The precise nature of this environment is not important; we only note that it is, in general, a tree structure with values at its leaves. An access path is a sequence of **fst**s and **snd**s for descending through this environment to retrieve a desired value. Due to the uniform manner in which identifiers are introduced into (our simplified) PCF_0 the access path for an identifier has the form "$\mathsf{fst}^{d-1};\mathsf{snd}$" in which d is the usual de Bruijn index for the identifier [3]. We can compute this index during translation by noting that $d = D - D_1 + 1$ where D is the depth of the occurrence of the identifier and D_1 is the depth of the binding occurrence for the identifier. For example, in the term $\lambda x \lambda y.x$ the occurrence of the identifier x is at depth 2 and the binding occurrence of x is at depth 1 (the top level). The de Bruijn index for the occurrence of x is then computed to be 2 ($= 2 - 1 + 1$). (Compare this with the same lambda term given in a syntax using de Bruijn indices: $\lambda\lambda.2$.)

To implement this translation in our meta-language we use a technique similar to our handling of lambda abstraction in the PCF_0 type inference system. Consider rule 6 in Figure 4.7. To translate the term (**lamb** M) we introduce a new parameter c and apply the meta-term M to it. This substitutes c for the abstracted variable of the term. Since the term (**lamb** M) represents the introduction of a new binding we must increment by one the depth D for translating subterms in M. The assumption $D+1 \xrightarrow{\text{dp}} c$ asserts that c is an identifier which was abstracted at depth $D+1$. This will be precisely the information required to produce the access path for this identifier (given by rule 4). When the subterm c is reached during the translation process the depth (D_1) of its binding occurrence is obtained from the assumptions of the form $D_1 \xrightarrow{\text{dp}} c$. Noting the above relation between de Bruijn indices and our depths we form rule 4 to generate the correct access path. This is essentially the rule for the "categorical combinator" $n!$ given in [7], though they work directly with de Bruijn indices and so their translation of identifiers into such indices is simpler.

The translations for **let** and **fix** (rules 8 and 9, respectively) use the same approach for manipulating the identifiers. The translations for the remaining constructs are almost identical to their counterparts in [11] and we do not discuss them here.

$$D,\ N \xrightarrow{\text{pcf-cam}} (\text{quote } N) \tag{1}$$

$$D,\ \mathsf{true} \xrightarrow{\text{pcf-cam}} (\text{quote true}) \qquad D,\ \mathsf{false} \xrightarrow{\text{pcf-cam}} (\text{quote false}) \tag{2, 3}$$

$$\frac{D_1 \xrightarrow{\text{dp}} x}{D,\ x \xrightarrow{\text{pcf-cam}} \text{fst}^{D-D_1+1};\ \text{snd}} \tag{4}$$

$$\frac{D,\ e_1 \xrightarrow{\text{pcf-cam}} C_1 \qquad D,\ e_2 \xrightarrow{\text{pcf-cam}} C_2 \qquad D,\ e_3 \xrightarrow{\text{pcf-cam}} C_3}{D,\ (\mathsf{if}\ e_1\ e_2\ e_3) \xrightarrow{\text{pcf-cam}} (\text{push};\ C_1;\ \text{branch}(C_2, C_3))} \tag{5}$$

$$\frac{(\forall c)\ (D+1 \xrightarrow{\text{dp}} c\ \Rightarrow\ D+1,\ (M\ c) \xrightarrow{\text{pcf-cam}} C)}{D,\ (\mathsf{lamb}\ M) \xrightarrow{\text{pcf-cam}} \text{cur}(C)} \tag{6}$$

$$\frac{D,\ e_1 \xrightarrow{\text{pcf-cam}} C_1 \qquad D,\ e_2 \xrightarrow{\text{pcf-cam}} C_2}{D,\ (e_1 @ e_2) \xrightarrow{\text{pcf-cam}} (\text{push};\ C_1;\ \text{swap};\ C_2;\ \text{cons};\ \text{app})} \tag{7}$$

$$\frac{D,\ e_2 \xrightarrow{\text{pcf-cam}} C_2 \qquad (\forall c)\ (D+1 \xrightarrow{\text{dp}} c\ \Rightarrow\ D+1,\ (M\ c) \xrightarrow{\text{pcf-cam}} C_1)}{D,\ (\mathsf{let}\ M\ e_2) \xrightarrow{\text{pcf-cam}} (\text{push};\ C_2;\ \text{cons};\ C_1)} \tag{8}$$

$$\frac{(\forall c)\ (D+1 \xrightarrow{\text{dp}} c\ \Rightarrow\ D+1,\ (M\ c) \xrightarrow{\text{pcf-cam}} C)}{D,\ (\mathsf{fix}\ M) \xrightarrow{\text{pcf-cam}} (\text{push};\ \text{rec}(C))} \tag{9}$$

FIGURE 4.7
Translation from PCF_0 to CAM

5

Extended Examples Using $\mathbf{PCF_0}$

In the previous chapter we considered the static and dynamic semantics of the language PCF_0 and we presented various proof systems that specified these semantics. In this chapter we consider additional examples involving the analysis of PCF_0. First we consider the problem of abstract interpretation, more specifically strictness analysis. We demonstrate how this task can be elegantly expressed using our methods. The current work does not extend previous results for strictness analysis but only serves to show how our methods are flexible enough to accommodate non-standard semantics. Second, we consider an extended dynamic semantics that has applications in addressing the tasks of partial evaluation and pre-compile-time optimizations. Finally, we extend PCF_0 to a language that contains data type definitions. We show how our methods extend naturally to handle a richer object language.

5.1 Strictness Analysis for PCF_0

Abstract interpretation is a general technique for establishing facts (or proving properties) about a program that may then be used during compile-time optimizations or during a source-to-source program transformation [19]. This technique typically involves interpreting programs in a non-standard domain such that the elements of this domain correspond, for example, to specific judgments about program properties. An example of this is given below. A particular use of abstract interpretation is the strictness analysis of functional

programs, originally demonstrated in [42]. A function f is strict in an argument if f is undefined whenever the argument is undefined (*e.g.*, $f\perp = \perp$), whether or not the function's behavior was dependent upon the argument. Strictness analysis is an essential technique for incorporating call-by-value safely into a functional language with call-by-name or call-by-need semantics, without affecting the semantics of the language (see [43] for a discussion of these issues). The safe use of call-by-value can be a significant gain in execution efficiency since it provides information regarding safe parallel evaluation strategies for functional arguments and also reduces the need for constructing closures required by both call-by-name and call-by-need. We shall show how strictness information can be obtained by specifying a non-standard semantics of PCF_0.

5.1.1 Expressing Strictness Information

In this section we present a non-standard semantics of PCF_0 that performs strictness analysis based on a structural analysis of expressions. We informally develop this semantics by transforming the dynamic semantics of PCF_0 given above into a non-standard dynamic semantics. In the discussion below we have taken material originally presented in [4] and recast it using the proof methods described above.

For strictness analysis, we wish to know whether a function f always needs its argument, *i.e.*, whether $f\perp = \perp$, where $\perp$ represents the "undefined" or divergent value of the language. More generally, we may have a function g of n arguments and then we consider strictness with respect to a particular argument. We say that g is strict in its i^{th} argument if

$$\forall a_1, \ldots, a_{i_1}, a_{i+1}, \ldots, a_n \; (g(a_1, \ldots, a_{i_1}, \perp, a_{i+1}, \ldots, a_n) = \perp)$$

in which the a_i's range over the appropriate types or domains. Generalizing this problem even further we shall consider expressions in a language (PCF_0) and ask whether or not an expression's meaning, as given by a suitable semantics, is $\perp$. Of course it is impossible to provide an exact answer to this question, for all possible functions g (this we know from computability theory). Thus we must settle for some approximation. For this purpose we introduce a two-point abstract domain

$$\mathbf{2} = \{\mathbf{0}, \mathbf{1}\} \qquad \text{ordered by } \; \mathbf{1} \sqsubseteq \mathbf{0}$$

that we will use to give abstract meaning to expressions. (Intuitively, **0** has more information than **1**, as explained below.) Then for any expression $e \in PCF_0$ we would like a function $\mathbf{ABS} : PCF_0 \to \mathbf{2}$ (providing a non-standard semantics) with the intended meaning:

$$\mathbf{ABS}\; e \;=\; \mathbf{0} \;\text{ then } e \text{ fails to terminate}$$

There may, in fact, be expressions that fail to terminate but get mapped by **ABS** into **1**. This, informally, provides us with a level of safety: if (**ABS** e) = **0**, then we have some precise information about e; however, if (**ABS** e) = **1**, then we do not know whether or not e terminates. Such is the nature of this approximation. Some examples of expressions that definitely fail to terminate (in a typical standard semantics) are $\perp$ and *if true then* $\perp$ *else* e'.[1] Notice that the expression *if* $p(x)$ *then* 3 *else* $\perp$ diverges iff $p(x) =$ *false* or $p(x)$ diverges and terminates otherwise; if we know that $p(x) =$ *false* for all x (or at least for all possible values that x could ever receive), we would map this expression, via **ABS**, to **0**.

As an alternative to the function **ABS**, we can consider first translating an expression $e \in PCF_0$ into $e^\sharp \in PCF_0{}^\sharp$. $PCF_0{}^\sharp$ is similar to PCF_0, but it is defined over the constants **0** and **1** and has some additional operators to manipulate these constants. We then need only provide a semantics for $PCF_0{}^\sharp$ and show that under the semantics for $PCF_0{}^\sharp$, if $e^\sharp$ has value **0**, then e is undefined. Intuitively, we would like both of these approaches to produce the same results, but, furthermore, we would like a notion of safety of our computations, *i.e.*, if such a computation gives an expression e the value **0**, then it must be that e, when evaluated by some "standard" interpreter does not terminate. (If e gets the value **1**, then e may terminate or may not terminate.)

If we are able to define the function $(\cdot)^\sharp$ and an interpretation for $PCF_0{}^\sharp$ then we can perform strictness analysis in the following way. Given an expression $f \in PCF_0$ denoting a function of, say, one argument, we then compute $f^\sharp(\mathbf{0})$. If $f^\sharp(\mathbf{0}) = \mathbf{0}$ then we will show that $f\perp = \perp$ *i.e.*, that the function f is strict.

5.1.2 The "Undefined" Value.

In the discussion above we informally introduced the constant $\perp$, denoting the undefined value, into the language PCF_0. This allowed us to simply define the strictness of a function f via $f\perp = \perp$. If we were working in a denotational-semantics setting, then the introduction of $\perp$ would be quite natural.

More specifically, a denotational semantics typically includes some domain $\mathcal{D}$ of denotable values, and a denotation function $[\![\cdot]\!] : \mathcal{E} \to Env \to \mathcal{D}$, in which $\mathcal{E}$ denotes the expressions in the language and $Env : Var \to \mathcal{D}$ denotes environments, mapping variables to denotable values. Using this type of semantics a function f would be strict if

[1]These two examples assume that the constant $\perp$ has been added to the language PCF_0. This is done only for convenience in providing examples.

$(\forall\rho)[\![f]\!]\rho \circ \bot = \bot$ (in which $\circ$ denotes application in the domain $\mathcal{D}$ and $\bot \in \mathcal{D}$ denotes the undefined value).

In our presentation of dynamic semantics via proof systems, we give meaning to expressions via proofs of formulas, in which the formulas express that an expression reduces to a certain canonical value in the language. Thus we could introduce a new constant $\bot$ into our semantics only by introducing it into the language. Furthermore, we would need to introduce axioms and inference rules that essentially define this new constant. To understand why this approach is not applicable we consider a simple example for which this method fails.

Consider the expression fix $\lambda f(f)$. In a standard denotational semantics, in which the meaning of recursion is given by a least fixed-point construction, this expression would be given the denotation $\bot$ (as is easily shown since $(\lambda f.f\ \bot) = \bot$). Using the dynamic semantics for PCF_0 given above, we evaluate, or give meaning to, this term by constructing (searching for) a proof of the formula fix $\lambda f(f) \longrightarrow V$ in which V is initially an unbound variable. We expect that when a proof has been constructed, V is bound to a value and this (canonical) value is the meaning of the original expression. We easily see that we can construct the following infinite tree:

$$\dfrac{\dfrac{\dfrac{\vdots}{\text{fix } \lambda f.f \xrightarrow{se} V}}{(\lambda f.f\ (\text{fix } \lambda f.f)) \xrightarrow{se} V}}{\text{fix } \lambda f.f \xrightarrow{se} V}$$

Furthermore, we note that by Theorem 4.8, this is the unique tree with fix $\lambda f.f \xrightarrow{se} V$ as its root. Thus it must be that no proof tree exists for this formula (since proof trees must be finite). We might then consider adding additional inference rules to avoid the construction of infinite proof trees. These additional rules would have as consequents formulas of the form $\delta_i \xrightarrow{se} \bot$ in which the δ_i's form a set of expressions, all of which have no canonical value. Of course we know immediately that such a set of δ_i is not effectively computable as the set of nonterminating expressions is not recursively enumerable. And so we cannot hope to augment our inference rules without still leaving some "gaps" that would allow the construction of infinite trees. We would then have two specifications for undefined values: the non-existence of proof trees and the constant $\bot$. Clearly it is undesirable to require both notions. Thus infinite trees, or the lack of proof trees, must play an important role in our understanding of undefined values, while the use of additional constants seems incongruous to our approach. We will, however, relate our notion of undefined values to the constant $\bot$ used in denotational semantics.

As this example suggests, to express the notion of undefined values in terms of proofs

in the dynamic semantics requires a statement about the (lack of) existence of proof trees. So corresponding to the notion "$[\![e]\!] = \perp$" we would have $(\forall\alpha) \not\vdash e \xrightarrow{se} \alpha$. Now to express strictness properties of functional expressions we must first specify a non-strict dynamic semantics. The dynamic semantics for PCF_0 given above defines a strict semantics via the inference rule

$$\frac{e_1 \xrightarrow{se} (\mathsf{lamb}\ M) \qquad e_2 \xrightarrow{se} \alpha_2 \qquad (M\ \alpha_2) \xrightarrow{se} \alpha}{(e_1 @ e_2) \xrightarrow{se} \alpha}$$

Note that the function's argument, e_2, is evaluated *before* function application. For example, for no value V is there is a proof of the formula $((\mathsf{lamb}\ \lambda x.3)@(\mathsf{fix}\ \lambda f.f)) \longrightarrow V$ using this definition of dynamic semantics. Alternatively, we can define a non-strict dynamic semantics by changing the above inference rule to

$$\frac{e_1 \xrightarrow{se} (\mathsf{lamb}\ M) \qquad (M\ e_2) \xrightarrow{se} \alpha}{(e_1 @ e_2) \xrightarrow{se} \alpha}$$

In this new proof system there is a proof of the above formula. The lack of existence of proof trees suggest that to express strictness we should make a statement similar to the following:

> Let $f \in PCF_0$ be some expression representing a function. Then we say that f is strict if the following holds: for all expressions e and types α, $\not\vdash (e \xrightarrow{se} \alpha)$ implies for all types β, $\not\vdash ((f@e) \xrightarrow{se} \beta)$.
>
> Taking the contrapositive we have the following definition of strictness.

DEFINITION 5.1 (Strictness) Let $f, e \in PCF_0$ such that $(f@e) \in PCF_0$. If there exists some β such that $\vdash ((f@e) \xrightarrow{se} \beta)$ implies that there exists some α such that $\vdash (e \xrightarrow{se} \alpha)$ then we say that f is strict (in its argument).

5.1.3 Defining the Abstraction Function

Recall that we would like a function $(\cdot)^\sharp$ to translate expressions $e \in PCF_0$ into new expressions on which we can simply perform strictness analysis. Let us first informally develop an intuitive definition for the function $(\cdot)^\sharp$ and then give a compete definition in terms of a proof system. What is meant by intuitive is, at best, vague, but certainly we want a definition for $(\cdot)^\sharp$ that is computable and one that provides at least some minimum strictness information. Clearly, there are many possible definition for $(\cdot)^\sharp$. In fact, there exists a partial order of functions for $(\cdot)^\sharp$, in which the order is the typical pointwise ordering: $f \leq g$ implies that g is more defined than f. The least defined function, $(\cdot)^\sharp_\perp$ is the one

that maps every expression to **1**. This is the safest approximation possible as it provides absolutely no strictness information. Theoretically, a maximal element in the order that satisfies the safety requirement is the one that provides exact information. As mentioned above, this function is not computable.

In defining $(\cdot)^\sharp$ we note that constants always terminate. Therefore we have

$$c^\sharp = \mathbf{1} \quad \text{for all constants } c$$

Next we can see that primitive operators are strict in both arguments, *e.g.*, that

$$(e_1 = e_2) \text{ terminates} \Leftrightarrow (e_1 \text{ terminates}) \textbf{ AND } (e_2 \text{ terminates})$$

thus we have:

$$(e_1 = e_2)^\sharp = e_1{}^\sharp \wedge e_2{}^\sharp$$

where $\wedge$ is the boolean **AND** operator in $PCF_0{}^\sharp$. Similarly, $\vee$ is the boolean **OR** operator in $PCF_0{}^\sharp$. It should be clear that all the relational operators and arithmetic operators are strict in both arguments and so for every primitive binary operator OP, we define

$$(e_1 \text{ OP } e_2)^\sharp = e_1{}^\sharp \wedge e_2{}^\sharp$$

We now consider the interpretation of the *if* construct of PCF_0. In particular, a naive call-by-value conditional function is inappropriate as nonterminating values can turn perfectly reasonable expressions into nonterminating ones. This problem results from the fact that for any given execution of a conditional, at most one branch need be evaluated (none are evaluated if the condition part fails to terminate). Therefore, we do not want to evaluate both the 'then' clause and the 'else' clause ahead of time. We can ask, then, under what conditions may the conditional statement $(\mathit{if}\ e_1\ e_2\ e_3)$ terminate? Clearly e_1 must terminate and at least one of e_2 and e_3 must terminate. Hence, an appropriate definition appears to be

$$(\mathit{if}\ x\ y\ z)^\sharp = x^\sharp \wedge (y^\sharp \vee z^\sharp)$$

As demonstrated in a previous section, we can describe translations via proof systems in our meta-logic. The complete definition of $(\cdot)^\sharp$ is given in Figure 5.1. We use the formula $e \xrightarrow{\sharp} b$ to denote the application of $(\cdot)^\sharp$: $e^\sharp = b$.

For example, consider the expression g:

$$\mathsf{lamb}\ \lambda p(\mathsf{lamb}\ \lambda q(\mathsf{lamb}\ \lambda r(\mathit{if}\ (p = 0)\ (q + r)\ (q + p))))$$

By applying the translation given by Figure 5.1 we have:

$$\begin{aligned} g^\sharp &= \mathsf{lamb}\ \lambda p_b(\mathsf{lamb}\ \lambda q_b(\mathsf{lamb}\ \lambda r_b(p_b \wedge \mathbf{1}) \wedge ((q_b \wedge r_b) \vee (q_b \wedge p_b)))) \\ &= \mathsf{lamb}\ \lambda p_b(\mathsf{lamb}\ \lambda q_b(\mathsf{lamb}\ \lambda r_b(p_b \wedge q_b \wedge (p_b \vee r_b)) \end{aligned}$$

$$N \xrightarrow{\sharp} 1 \qquad \mathsf{true} \xrightarrow{\sharp} 1 \qquad \mathsf{false} \xrightarrow{\sharp} 1 \tag{1,2,3}$$

$$\frac{e_1 \xrightarrow{\sharp} b_1 \qquad e_2 \xrightarrow{\sharp} b_2 \qquad e_3 \xrightarrow{\sharp} b_3}{(\mathsf{if}\ e_1\ e_2\ e_3) \longrightarrow b_1 \wedge (b_2 \vee b_3)} \tag{4}$$

$$\frac{\forall c(c \xrightarrow{\sharp} c \Rightarrow (M\ c) \xrightarrow{\sharp} (M'\ c))}{(\mathsf{lamb}\ M) \longrightarrow (\mathsf{lamb}\ M')} \qquad \frac{e_1 \xrightarrow{\sharp} b_1 \qquad e_2 \xrightarrow{\sharp} b_2}{(e_1 @ e_2) \xrightarrow{\sharp} (b_1 @ b_2)} \tag{5,6}$$

$$\frac{\forall c(c \xrightarrow{\sharp} c \Rightarrow (M\ c) \xrightarrow{\sharp} (M'\ c)) \qquad e_2 \xrightarrow{\sharp} b_2}{(\mathsf{let}\ M\ e_2) \xrightarrow{\sharp} (\mathsf{let}\ M'\ b_2)} \qquad \frac{\forall c(c \xrightarrow{\sharp} c \Rightarrow (M\ c) \xrightarrow{\sharp} (M'\ c))}{(\mathsf{fix}\ M) \xrightarrow{\sharp} (\mathsf{fix}\ M')} \tag{7,8}$$

FIGURE 5.1
Translating PCF_0 to $PCF_0{}^\sharp$

Now to interpret these translated expressions we need a dynamic semantics for $PCF_0{}^\sharp$. It is similar to the dynamic semantics for PCF_0 with the addition of rules for the boolean operators $\wedge$ and $\vee$. This dynamic semantics is given in Figure 5.2.

We can now check whether g is strict in its i^{th} argument (for $i \in \{1,2,3\}$) simply by interpreting $g^\sharp$ (using the dynamic semantics of $PCF_0{}^\sharp$) with all of its arguments set to **1** except its i^{th} which is set to **0**. Informally, the choice of **1** for all the other arguments is justified since **1** contains the least amount of information (hence it doesn't eliminate any possibilities). Formally, the choice is justified by the fact that the dynamic semantics for $PCF_0{}^\sharp$ when viewed as a function (for all $e^\sharp \in PCF_0{}^\sharp$, there is at most one value α such that there exists a proof of the formula $e^\sharp \xrightarrow{\sharp} \alpha$) is monotonic, using a pointwise ordering on the cartesian product over the domain **2**. The element (of the appropriate cpo) $(\mathbf{1},\ldots,\mathbf{0}_i,\ldots,\mathbf{1})$ is the least element such that the i^{th} projection is **0**. Hence if $g^\sharp(\mathbf{1},\ldots,\mathbf{0}_i,\ldots,\mathbf{1}) \xrightarrow{\sharp} \mathbf{0}$, then for any other $(b_1,\ldots,\mathbf{0}_i,\ldots,b_n)$ $(b_j \in \{\mathbf{1},\mathbf{0}\})$, $g^\sharp(b_1,\ldots,\mathbf{0}_i,\ldots,b_n) \xrightarrow{\sharp} \mathbf{0}$.

For example, to determine whether g is strict in each of its arguments we compute (*i.e.*, derive proof trees for the following formulas):

$(g^\sharp\ \mathbf{0}\ \mathbf{1}\ \mathbf{1})$	$\xrightarrow{S}$	$\mathbf{0}$	g is strict in its first argument (p)
$(g^\sharp\ \mathbf{1}\ \mathbf{0}\ \mathbf{1})$	$\xrightarrow{S}$	$\mathbf{0}$	g is strict in its second argument (q)
$(g^\sharp\ \mathbf{1}\ \mathbf{1}\ \mathbf{0})$	$\xrightarrow{S}$	$\mathbf{1}$	g is not strict in its third argument (r)

Note that by composing the definitions of $\sharp$ and the dynamic semantics of $PCF_0{}^\sharp$, we

$$\mathbf{1} \xrightarrow{S} \mathbf{1} \qquad \mathbf{0} \xrightarrow{S} \mathbf{0} \tag{1,2}$$

$$\frac{b \xrightarrow{S} c}{\mathbf{1} \wedge b \xrightarrow{S} c} \qquad \frac{b \xrightarrow{S} c}{b \wedge \mathbf{1} \xrightarrow{S} c} \qquad \mathbf{0} \wedge b \xrightarrow{S} \mathbf{0} \qquad b \wedge \mathbf{0} \xrightarrow{S} \mathbf{0} \tag{3,4,5,6}$$

$$\frac{b \xrightarrow{S} c}{\mathbf{0} \vee b \xrightarrow{S} c} \qquad \frac{b \xrightarrow{S} c}{b \vee \mathbf{0} \xrightarrow{S} c} \qquad \mathbf{1} \vee b \xrightarrow{S} \mathbf{1} \qquad b \vee \mathbf{1} \xrightarrow{S} \mathbf{1} \tag{7,8,9,10}$$

$$(\mathsf{lamb}\ \lambda x.b) \xrightarrow{S} (\mathsf{lamb}\ \lambda x.b) \tag{11}$$

$$\frac{b_1 \xrightarrow{S} (\mathsf{lamb}\ \lambda x.b) \qquad b_2 \xrightarrow{S} c_2 \qquad (\mathsf{lamb}\ \lambda x.b)c_2 \xrightarrow{S} c}{(b_1 @ b_2) \xrightarrow{S} c} \tag{12}$$

$$\frac{b_2 \xrightarrow{S} c_2 \qquad (\lambda x.b_1)c_2 \xrightarrow{S} c}{(\mathsf{let}\ \lambda x.b_1\ b_2) \xrightarrow{S} c} \qquad \frac{(\lambda f.b)(\mathsf{fix}\ \lambda f.b) \xrightarrow{S} c}{(\mathsf{fix}\ \lambda f.b) \xrightarrow{S} c} \tag{13,14}$$

FIGURE 5.2
Dynamic Semantics of $PCF_0^\sharp$

have, essentially, a definition for **ABS**. If we consider the dynamic semantics of $PCF_0^\sharp$ as a function $\mathcal{S}$ ($\mathcal{S}(e) = b \Leftrightarrow$ there exists a proof Ξ of $e \xrightarrow{\mathcal{S}} b$) then we can define **ABS** precisely as

$$\mathbf{ABS} = \mathcal{S} \circ \sharp$$

For example,

$$\begin{aligned} (\mathbf{ABS}\ ((\mathsf{lamb}\ \lambda x.x)@(3)) &= \mathcal{S}\ ((\mathsf{lamb}\ \lambda x.x)@(3))^\sharp \\ &= \mathcal{S}\ ((\mathsf{lamb}\ \lambda x_b.x_b)@(1)) \\ &= 1 \end{aligned}$$

However, applying **ABS** gives us an abstract (non-standard) interpretation of an expression e but to perform strictness analysis what we really require is information about a family of expressions, *i.e.*, $g(a_1, \ldots, \perp_i, \ldots, a_n)$ for all $(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n)$. We did this above by taking the expression $g^\sharp$ and arguing that by using **1** for a_j, $j \neq i$ we could "evaluate" just the single expression $g^\sharp(\mathbf{1}_1, \ldots, \mathbf{0}_i, \ldots, \mathbf{1}_n)$ in place of the family of expressions above. Also note that using **ABS** for perform strictness analysis requires the introduction of a new constant $\perp$ into PCF_0.

5.1.4 Handling Recursive Function Definitions.

The above example is a simple non-recursive function. Of course, the most interesting functions are typically the recursive ones such as the expression f defined as

$$\text{fix } \lambda f.(\text{lamb } \lambda x(\text{lamb } \lambda y(\text{lamb } \lambda z(\mathit{if}\,(y = 0)\ (f\ 0\ 1\ x)\ x))))$$

To understand strictness analysis of recursive functions we first describe our approach using proof systems and then compare this with a method presented in [4].

Now using our proof system, we do not define recursion by computing a least fix point of some ascending sequence (in the style of a denotational semantics for recursion) but rather use the operational style of unfolding (as we did in the dynamic semantics of PCF_0). Consider the function f given above. Using our translation, we have $f^\sharp$ equal to

$$\text{fix } \lambda f_b.(\text{lamb } \lambda x_b(\text{lamb } \lambda y_b(\text{lamb } \lambda z_b(y_b \wedge ((f_b\ \mathbf{1}\ \mathbf{1}\ x) \vee x)))))$$

To determine if f is strict in its second argument (y) we try to construct a proof of the formula $(f^\sharp\ \mathbf{101}) \xrightarrow{\mathcal{S}} \mathbf{0}$. A proof of this formula is given by the following (abbreviated) proof:

$$\frac{(\mathbf{0} \wedge ((f^\sharp\ \mathbf{1}\ \mathbf{1}\ \mathbf{1}) \vee \mathbf{1})) \xrightarrow{\mathcal{S}} \mathbf{0}}{(f^\sharp\ \mathbf{1}\ \mathbf{0}\ \mathbf{1}) \xrightarrow{\mathcal{S}} \mathbf{0}}$$

This example is a positive test for strictness. For a negative test, we search for a proof of some appropriate expression evaluating to $\mathbf{1}$. For example, we can find a proof of the proposition $(f^\sharp\ \mathbf{1}\ \mathbf{1}\ \mathbf{0}) \xrightarrow{\sharp} \mathbf{1}$, from which we infer that f is not strict in its third argument (z). This fact is also obvious by inspection of the original function f. We also can construct a proof of $(f^\sharp\ \mathbf{0}\ \mathbf{1}\ \mathbf{1}) \xrightarrow{\sharp} \mathbf{1}$, and hence f is not strict in its first argument (x).

We want to show that our method of strictness analysis has the required safety property. First we have the following lemma about the translation from PCF_0 to $PCF_0{}^\sharp$ and the semantics of $PCF_0{}^\sharp$.

LEMMA 5.2 (i) For all $e \in PCF_0$, no subterm of $e^\sharp$ is $\mathbf{0}$;
(ii) For all $e^\sharp \in PCF_0{}^\sharp$ if $\vdash e^\sharp \xrightarrow{\mathcal{S}} \mathbf{0}$ then some subterm of $e^\sharp$ must be $\mathbf{0}$;
(iii) For all $e^\sharp \in PCF_0{}^\sharp$, if $\vdash e^\sharp \xrightarrow{\mathcal{S}} \mathbf{0}$ then $\not\vdash e^\sharp \xrightarrow{\mathcal{S}} \mathbf{1}$ and if $\vdash e^\sharp \xrightarrow{\mathcal{S}} \mathbf{1}$ then $\not\vdash e^\sharp \xrightarrow{\mathcal{S}} \mathbf{0}$

The proofs for these lemmas follow from the definitions of $\xrightarrow{\sharp}$ and $\xrightarrow{\mathcal{S}}$. The safety theorem for strictness is then stated as:

THEOREM 5.3 (Safety of Strictness) For all $f \in PCF_0$ we say that f is *strict* if $\vdash (f^\sharp @ \mathbf{0}) \xrightarrow{\mathcal{S}} \mathbf{0}$ implies that for all $e \in PCF_0$, if $\vdash ((f @ e) \xrightarrow{se} \beta)$ for some β then $\vdash (e \xrightarrow{se} \alpha)$ for some α (in which $\xrightarrow{se}$ is the non-strict variant described above).

Rather than give the proof of this theorem we outline the general reasoning involved. The implication of $(f^\sharp @ \mathbf{0})$ evaluating to $\mathbf{0}$ is that the result $\mathbf{0}$ must have been introduced by the argument $(\mathbf{0})$ of the function. (We use the lemma to show this). Hence the argument to $f^\sharp$ was used. As we have informally reasoned above, the definition of $f^\sharp$ enables us to "simulate" all undefined values by just $\mathbf{0}$. The fact that just one evaluation (in the $\sharp$ domain) categorizes a family of evaluations in the standard domain is what makes abstract interpretation so powerful.

5.1.5 Comparison with Other Methods

The work reported above were based on results found in a paper by Clack and Peyton Jones [4]. They use a purely functional approach, defining functions analogous to our $(\cdot)^\sharp$ and $\xrightarrow{\sharp}$. We shall overload the symbol $\sharp$, but its use should be clear from context. In the discussion to follow we use the notation found in their paper, *e.g.*, for f we would write

$$f\ x\ y\ z\ =\ \mathit{if}\,(y = 0)\ \mathit{then}\ (f\ 0\ 1\ x)\ \mathit{else}\ x$$

A reasonable guess at a definition for $f^\sharp$ would the least upper bound of an ascending sequence of approximations to $f^\sharp$. As an initial approximation, they use $f^{\sharp 0}\ x\ y\ z\ =\ \mathbf{0}$, the function with the 'least' amount of information. They argue that a fixed point can be obtained by taking the limit of the approximations. The sequence of approximations must be finite, so a fixed point can be found. Informally, the justification for applying this method is the following. We are working in the abstract domain $\mathbf{2}$ of two elements. Therefore, there is a finite number of functions of four arguments (2^4) and these functions are monotonic.

We can then compute successive approximations as:

$$f^{\sharp n+1}\ x\ y\ z\ =\ y\ \wedge\ ((f^{\sharp n}\ \mathbf{1}\ \mathbf{1}\ x)\ \vee\ x)$$

and define $f^\sharp = \bigcup f^{\sharp n}$. (The translation of *if*, constants and primitive operations is identical to ours.) This yields the following sequence of approximations:

$$
\begin{array}{lcll}
f^{\sharp 0}\ x\ y\ z & = & \mathbf{0} & \text{(strict in } x, y, z\text{)} \\
f^{\sharp 1}\ x\ y\ z & = & y \wedge ((f^{\sharp 0}\ \mathbf{1}\ \mathbf{1}\ x) \vee x) & \\
 & = & y \wedge (x \vee x) & \\
 & = & y \wedge x & \text{(strict in } x, y\text{)} \\
f^{\sharp 2}\ x\ y\ z & = & y \wedge ((f^{\sharp 1}\ \mathbf{1}\ \mathbf{1}\ x) \vee x) & \\
 & = & y \wedge (\mathbf{1} \vee x) & \\
 & = & y & \text{(strict in } y\text{)} \\
f^{\sharp 3}\ x\ y\ z & = & y \wedge ((f^{\sharp 2}\ \mathbf{1}\ \mathbf{1}\ x) \vee x) & \\
 & = & y \wedge (\mathbf{1} \vee x) & \\
 & = & y & \text{(strict in } y\text{)}
\end{array}
$$

Hence, we have reached a fixed point and we can conclude that f is strict only in y. This is indeed the correct answer.

Now observe that this is precisely the same answer which we computed (and, indeed, is an exact answer, rather than an approximation, since it predicts the precise strictness information for all three arguments. With other, more complex examples, the results produced by both systems may be less accurate. By this we mean that the methods will return safe approximations, characterizing a function as being non-strict in its i^{th} argument (which is always safe), when in fact it is strict (as determined by some "more sophisticated" method). This property just suggests that both of these methods perform only simple or naive strictness tests and more intelligent ones may exist. We do, however, make the following conjecture regarding the relationship between our method and the one in [4].

CONJECTURE: Our strictness system and the one from [4] presented above calculate precisely the same strictness information, *i.e.*,

$$\vdash (f^{\sharp}\ \mathbf{0}) \xrightarrow{\sharp} \mathbf{0} \quad \Leftrightarrow \quad f^{\sharp}\ \mathbf{0} = \mathbf{0}$$

Again, we have overload the definition of $f^{\sharp}$, but each use should be clear from context.

5.2 Mixed Evaluation Semantics

Program transformations form a general class of program manipulations in which a new program is constructed from a given program. In this section we consider a particular kind of program transformation that performs a simplification task. The notion of simplification here intuitively refers to the reduction or elimination of subparts of a program and can be viewed as a kind of compilation step. Specifications for such transformations are called *mixed evaluation semantics.* This kind of simplification has also been called *partial evaluation* and

mixed computation because one attempts to evaluate as much of a program as possible given only part of the program's input, mixing known and unknown quantities. A key feature in the development of this section is that we informally derive a specification for a mixed evaluation semantics from a specification for a standard evaluation semantics.

5.2.1 Motivating Mixed Evaluation

Mixed evaluation is a systematic method of constructing an efficient program based on a given program and a part of its input [?]. In general terms it can be described as follows. Let f be some functional program of two arguments x and y and consider the application $f(c, y)$ for some constant (known) value c and variable (unknown) value y. We wish to construct a new functional program f_c such that $f_c(y) = f(c, y)$ for all values of y, such that for any value of y, computing $f_c(y)$ should be easier (*e.g.*, faster) than computing $f(c, y)$. Such improvement is possible by "compiling" the information of $x = c$ in f into the definition of f_c. The problem with such "compiling" is that one may encounter known (c) and unknown (y) information. Thus a formal approach to mixed or mixed evaluation must deal with the proper treatment of the interaction of known and unknown values (hence the adjective "mixed").

The importance of partial evaluation was elucidated by Futamura [?] when he derived the construction of compiled programs, compilers, and compiler generators via partial evaluation. Thus partial evaluation was viewed as a process for understanding and constructing a wide range of translation tools. More recent work has generalized the notion of partial evaluation, incorporating theorem provers as part of their specification [13]. *But where do partial evaluators come from?* Few research efforts have addressed the formal construction of partial evaluators using principled techniques. But such work is essential to the development and advancement of partial evaluation methods. We address this issue by demonstrating how, for a simple functional programming language, a specification for mixed evaluation can be derived from a specification for standard evaluation. We hope that such work will suggest general construction methods that are language independent.

5.2.2 Deriving a Mixed Evaluation Semantics

To derive a proof system for mixed evaluation from the proof system for standard evaluation we need a method of characterizing evaluation semantics. We shall find the following syntactic version of logical relations useful [?].

DEFINITION 5.5 (Logical Relations) Let $\{D_\sigma\}_\sigma$ be a type indexed family of sets of λ-terms, and let $\{R_\sigma\}_\sigma$ be a type indexed family of binary relations such that for all simple types σ, R_σ is a binary relation on D_σ. R is a *logical relation* if for all simple types τ_1, τ_2, $R_{\tau_1 \to \tau_2}(f,g)$ *iff* for all $a, b \in D_{\tau_1}$, $R_{\tau_1}(a,b)$ implies $R_{\tau_2}(f(a), g(b))$.

Let Σ be the signature for PCF_0. In this section, we shall consider types to be only those simple types built from the primitive type tm and D_σ will be the set of closed λ-terms of type σ whose constants are taken from Σ. One logical relation R' is *smaller* than another logical relation R'' if the binary relation R'_{tm} is contained in R''_{tm}.

Let SE_{tm} be a binary relation on D_{tm} such that $SE_{tm}(e_1, e_2)$ *iff* $\vdash e_1 \xrightarrow{se} e_2$ and let SE be the unique logical relation defined by induction on types that extends SE_{tm}. We shall only be interested in how SE behaves at the base type tm (*i.e.* the relation SE_{tm}) because little of interest happens at higher types. The importance of SE is that it characterizes the expressiveness of the standard evaluation semantics. We next consider a relation whose behavior at all types is more interesting.

Observe that for any term e of the form (lamb M), e is related (via SE) only to itself. This is because SE is essentially a relation over tm. The only higher-type relations that it includes are the ones induced by the definition of logical relation. Clearly for terms M, M' of type $(tm \to tm)$ if $SE(M, M')$ then we would like to have $SE((\mathsf{lamb}\ M), (\mathsf{lamb}\ M'))$. But this is not the case for SE. To achieve this result we shall construct an extension to SE that relates more terms (than SE) at higher types and these additional relations will "filter down" to the base type and produce some additional relations between terms of type tm. Once we have this relation we shall construct a proof system (over expressions of type tm) that has the same extension as this relation (at type tm).

To define the new logical relation MIX we start by observing that for constants $c \in \Sigma$ of type higher than tm, c is not related to itself in SE. But relating expressions of the form (lamb M) and (lamb M') suggests that lamb should be related to itself. With this insight we propose a definition for the extension of SE.

DEFINITION 5.6 (MIX) Let MIX be the smallest logical relation such that SE_{tm} is contained in MIX_{tm} and $MIX(c,c)$ for every $c \in \Sigma$.

The inclusion of $MIX(c,c)$ enriches the relation (as compared to SE) at higher types. Consider the constant if for building conditional statements. It has type $(tm \to tm \to tm \to tm)$. Now from the definition of logical relations, for $MIX(\mathsf{if}, \mathsf{if})$ to hold, we have that the following proposition must also hold:

$$\forall e_1, e_1' {:} tm\ (MIX(e_1, e_1') \supset \forall e_2, e_2' {:} tm\ (MIX(e_2, e_2') \supset \forall e_3, e_3' {:} tm\ (MIX(e_3, e_3') \supset$$

$$MIX((\mathit{if}\ e_1\ e_2\ e_3),\ (\mathit{if}\ e_1'\ e_2'\ e_3'))))).$$

Similarly consider the constant $\mathsf{lamb} \in \Sigma$ of type $(tm \to tm) \to tm$. From the definition of logical relations for $MIX(\mathsf{lamb}, \mathsf{lamb})$ to hold, we have that the following proposition must also hold:

$$\forall f, f' {:}(tm \to tm)\ (\forall c, c' {:} tm\ (MIX(c, c') \supset\ MIX(f(c), f'(c'))) \supset\ MIX((\mathsf{lamb}\ f), (\mathsf{lamb}\ f')))$$

Note the negative occurrences of the universal quantifier. These arise for constants of higher type. The remaining constants of the signature produce similar propositions.

Now we would like to define an extension to the proof system of Figure 4.5 that has the same extension as the MIX relation (at type tm). Fortunately the way to such a proof system is provided for us by the propositions above. The key observation is that these propositions can be encoded as inference rules in our proof system. For the proposition for if the translation yields

$$\frac{e_1 \xrightarrow{mix} e_1' \qquad e_2 \xrightarrow{mix} e_2' \qquad e_3 \xrightarrow{mix} e_3'}{(\mathsf{if}\ e_1\ e_2\ e_3) \xrightarrow{mix} (\mathsf{if}\ e_1'\ e_2'\ e_3')}.$$

(replacing the prefix MIX with the infix $\xrightarrow{mix}$) and for the proposition for lamb the translation yields (modulo renaming for clarity)

$$\frac{\forall c \forall d\ (c \xrightarrow{mix} d\ \Rightarrow\ ((M\ c) \xrightarrow{mix} (M'\ d)))}{(\mathsf{lamb}\ M) \xrightarrow{mix} (\mathsf{lamb}\ M')}.$$

The formulation of the remaining inference rules proceeds similarly and the complete definition of the mixed evaluation semantics is given in Figure 5.3.

Now we can precisely state the relation between provability in this proof system and MIX.

THEOREM 5.7 For all terms e, e', $\vdash e \xrightarrow{mix} e'$ iff $MIX(e, e')$.

The proof in the forward direction is straightforward by induction on the height of proof trees. The proof in the reverse direction follows from the construction of MIX and is also straightforward. Note that since we have constructed this proof system by augmenting the one for PCF_0's standard evaluation semantics, $\vdash (e \xrightarrow{se} \alpha)$ implies $\vdash (e \xrightarrow{mix} \alpha)$. But note that the converse is not true and for a given e there may now be many e' such that $\vdash e \xrightarrow{mix} e'$ with e' *not* a value.

$$N \xrightarrow{mix} N \qquad \mathsf{true} \xrightarrow{mix} \mathsf{true} \qquad \mathsf{false} \xrightarrow{mix} \mathsf{false} \tag{1,2,3}$$

$$\frac{e_1 \xrightarrow{mix} \mathsf{true} \qquad e_2 \xrightarrow{mix} e'}{(\mathsf{if}\ e_1\ e_2\ e_3) \xrightarrow{mix} e'} \qquad \frac{e_1 \xrightarrow{mix} \mathsf{false} \qquad e_3 \xrightarrow{mix} e'}{(\mathsf{if}\ e_1\ e_2\ e_3) \xrightarrow{mix} e'} \tag{4a,4b}$$

$$\frac{e_1 \xrightarrow{mix} e_1' \qquad e_2 \xrightarrow{mix} e_2' \qquad e_3 \xrightarrow{mix} e_3'}{(\mathsf{if}\ e_1\ e_2\ e_3) \xrightarrow{mix} (\mathsf{if}\ e_1'\ e_2'\ e_3')} \tag{4c}$$

$$(\mathsf{lamb}\ M) \xrightarrow{mix} (\mathsf{lamb}\ M) \qquad \frac{\forall c \forall d\ (c \xrightarrow{mix} d \Rightarrow ((M\ c) \xrightarrow{mix} (M'\ d)))}{(\mathsf{lamb}\ M) \xrightarrow{mix} (\mathsf{lamb}\ M')} \tag{5,5a}$$

$$\frac{e_1 \xrightarrow{mix} (\mathsf{lamb}\ M) \qquad e_2 \xrightarrow{mix} e_2' \qquad (M\ e_2') \xrightarrow{mix} e'}{(e_1 @ e_2) \xrightarrow{mix} e'} \qquad \frac{e_1 \xrightarrow{mix} e_1' \qquad e_2 \xrightarrow{mix} e_2'}{(e_1 @ e_2) \xrightarrow{mix} (e_1' @ e_2')} \tag{6,6a}$$

$$\frac{e_2 \xrightarrow{mix} e_2' \qquad (M\ e_2') \xrightarrow{mix} e'}{(\mathsf{let}\ M\ e_2) \xrightarrow{mix} e'} \qquad \frac{e_2 \xrightarrow{mix} e_2' \qquad \forall c \forall d\ (c \xrightarrow{mix} d \Rightarrow ((M\ c) \xrightarrow{mix} (M'\ d)))}{(\mathsf{let}\ M\ e_2) \xrightarrow{mix} (\mathsf{let}\ M'\ e_2')} \tag{7,7a}$$

$$\frac{(M\ (\mathsf{fix}\ M)) \xrightarrow{mix} e'}{(\mathsf{fix}\ M) \xrightarrow{mix} e'} \qquad \frac{\forall c \forall d\ (c \xrightarrow{mix} d \Rightarrow ((M\ c) \xrightarrow{mix} (M'\ d)))}{(\mathsf{fix}\ M) \xrightarrow{mix} (\mathsf{fix}\ M')} \tag{8,8a}$$

$$\frac{e_1 \xrightarrow{mix} \alpha_1 \quad \cdots \quad e_n \xrightarrow{mix} \alpha_n \qquad f_{op}(\alpha_1, \ldots, \alpha_n) = \alpha}{(op\ e_1 \ldots e_n) \xrightarrow{mix} \alpha} \qquad \frac{e_1 \xrightarrow{mix} e_1' \quad \cdots \quad e_n \xrightarrow{mix} e_n'}{(op\ e_1 \ldots e_n) \xrightarrow{mix} (op\ e_1' \ldots e_n')} \tag{9,9a}$$

FIGURE 5.3
Mixed Evaluation Semantics for PCF_0

5.2.3 A Simple Example

Now let us consider an example of this mixed evaluation semantics. To pick an interesting example, we shall add to PCF_0 the constants *nil*, *cons*, *car*, *cdr*, and *null* for manipulating lists. The inference rules for evaluation semantics for these additional constants are given in Figure 5.4. The corresponding inference rules denoting the fact that $MIX(cdr, cdr)$, *etc*, are obvious and are also assumed. Consider the append function given by the term

$$(\mathsf{fix}\ \lambda f.(\mathsf{lamb}\ \lambda x.(\mathsf{lamb}\ \lambda y.(\mathit{if}\ (\mathit{null}\ x)\ y\ (\mathit{cons}\ (\mathit{car}\ x)\ (f\ (\mathit{cdr}\ x)\ y)))))).$$

(All occurrences of application "@" have been dropped in this example to make it more readable.) For convenience, we abbreviate this term by A. Now suppose we try to show $\vdash (A\ @\ (\mathit{cons}\ 1\ \mathit{nil})) \xrightarrow{se} \alpha$ for some α. It is not hard to see that the only possible value for

$$nil \xrightarrow{se} nil \qquad \frac{e_1 \xrightarrow{se} \alpha_1 \quad e_2 \xrightarrow{se} \alpha_2}{(cons\ e_1\ e_2) \xrightarrow{se} (cons\ \alpha_1\ \alpha_2)}$$

$$\frac{e \xrightarrow{se} (cons\ \alpha_1\ \alpha_2)}{(car\ e) \xrightarrow{se} \alpha_1} \qquad \frac{e \xrightarrow{se} (cons\ \alpha_1\ \alpha_2)}{(cdr\ e) \xrightarrow{se} \alpha_2}$$

$$\frac{e \xrightarrow{se} nil}{(null\ e) \xrightarrow{se} true} \qquad \frac{e \xrightarrow{se} (cons\ \alpha_1\ \alpha_2)}{(null\ e) \xrightarrow{se} false}$$

FIGURE 5.4
Standard Evaluation Specification for Primitive Functions

α is

$$(\textsf{lamb}\ \lambda y.(if\ (null\ (cons\ 1\ nil))\ y\ (cons\ (car\ (cons\ 1\ nil))\ (A\ (cdr\ (cons\ 1\ nil))\ y)))).$$

No further evaluation is possible.

Now consider showing $\vdash (A\ @\ (cons\ 1\ nil)) \xrightarrow{mix} \alpha$ for some α. The additional rules of the $\vdash$ proof system provide for further simplification of this expression. In particular, the partial instantiation of a list structure often provides enough information for the evaluation of some functions [17] *e.g.*, the function **null** applied to the "cons" of *any* two (terminating) expressions is always false. Clearly we can have the same value for α as above. Further evaluation is possible, however, by applying rule (5a) followed by (4b) to the α above. We are then able to show $\vdash (A@(cons\ 1\ nil)) \xrightarrow{mix} (\textsf{lamb}\ \lambda y.(cons\ 1\ y))$.

5.2.4 Comparison with Other Work

This approach to simplifying expressions is similar in spirit to the work on partial evaluation with inference rules found in [24]. However, our approach is less general in that it is given for a specific language (PCF_0) rather than for a given meta-language (TYPOL). Also we derive our "mixed evaluation" rules from the existing rules for dynamic semantics. But the manipulation of proofs is a general idea that is quite attractive. The current work also shares much with the recent work of Futamura on Generalized Partial Computation [14, 13]. Unifying our notion of mixed evaluation semantics with Futamura's framework for partial computation may yield a unified framework for understanding the task of partial computation. A distinct aspect of our work is the use of a formal meta-language for expressing a partial evaluator. In most other work on mixed computation the connection between

specification and implementation of a partial evaluator is, at best, tenuous. We argue that in our work the corresponding connection is intimate and easily understood. In the next chapter we present a proof of correctness for our mixed evaluation semantics and we shall comment further on the importance of this connection.

5.3 Extending PCF_0 with Data Type Definitions

In [21] we provide a proof-theoretic semantics for the introduction and proper scoping of constructors and destructors during the evaluation of data type definitions in an extension of PCF_0. We present that work here, showing how we can extend PCF_0 with a data type definition facility, reminiscent of that found in Standard ML.

5.3.1 Considering Data Types

To see how a data type definition facility can be added to PCF_0, consider the following constant.

pairtype	$(tm \longrightarrow tm \longrightarrow tm \longrightarrow tm) \longrightarrow tm$

The argument to **pairtype** must be an abstraction over three variables of type *tm*. We shall assume that these abstracted variables are named **pair**, **fst** and **snd**, respectively, and that the body of this abstraction is a term in which these three variables are treated as if they were new constants. To define the dynamic semantics of **pairtype**, let the formulas D_1, D_2 and D_3, respectively, be the formulas of our meta-language that denote the inference rules in Figure 5.5, namely

$$D_1 = (\forall e_1, e_2, v_1, v_2)(e_1 \longrightarrow v_1 \Rightarrow e_2 \longrightarrow v_2 \Rightarrow (\mathsf{pair}\ e_1\ e_2) \longrightarrow (\mathsf{pair}\ v_1\ v_2))$$
$$D_2 = (\forall e, v_1, v_2)(e \longrightarrow (\mathsf{pair}\ v_1\ v_2) \Rightarrow (\mathsf{fst}\ e) \longrightarrow v_1)$$
$$D_3 = (\forall e, v_1, v_2)(e \longrightarrow (\mathsf{pair}\ v_1\ v_2) \Rightarrow (\mathsf{snd}\ e) \longrightarrow v_2).$$

Then the semantics of **pairtype** is given by rule (13) in Figure 5.6.

Using the introduction and discharge rules of natural deduction, the correct dynamic scoping behavior of the pair data type is captured. For example, there is a proof of the judgment

$$\mathsf{pairtype}\ \lambda\mathsf{pair}\lambda\mathsf{fst}\lambda\mathsf{snd}(\mathsf{fst}\ (\mathsf{pair}\ 1\ 2)) \longrightarrow 1$$

while there is no proof of the judgment

$$\mathsf{pairtype}\ \lambda\mathsf{pair}\lambda\mathsf{fst}\lambda\mathsf{snd}(\mathsf{pair}\ 1\ 2) \longrightarrow V$$

$$\frac{e_1 \xrightarrow{se} \alpha_1 \qquad e_2 \xrightarrow{se} \alpha_2}{(\mathsf{pair}\ e_1\ e_2) \xrightarrow{se} (\mathsf{pair}\ \alpha_1\ \alpha_2)} \tag{10}$$

$$\frac{e \xrightarrow{se} (\alpha_1, \alpha_2)}{(\mathsf{fst}\ e) \xrightarrow{se} \alpha_1} \qquad \frac{e \xrightarrow{se} (\alpha_1, \alpha_2)}{(\mathsf{snd}\ e) \xrightarrow{se} \alpha_2} \tag{11, 12}$$

FIGURE 5.5
Dynamic Semantics for pair, fst, snd

$$\frac{(\forall \mathsf{pair}, \mathsf{fst}, \mathsf{snd})(D_1 \Rightarrow D_2 \Rightarrow D_3 \Rightarrow (E\ \mathsf{pair}\ \mathsf{fst}\ \mathsf{snd}) \longrightarrow V)}{(\mathsf{pairtype}\ E) \longrightarrow V} \tag{13}$$

FIGURE 5.6
Dynamic Semantics for pairtype.

for any value of V. In the latter case, the only possible value for V contains the parameter substituted for **pair** but this is not possible given the proviso on universal introduction that the parameter must be discharged (not free in the resulting judgment).

5.3.2 Specifying Data Type Definitions in PCF_0

Given this motivation, we now define the language PCF_1 that contains a general mechanism for data type definition. To generalize on the **pairtype** constant described above, we must be able to parametrize the data type definition not only with expressions (over which constants have been abstracted) but also with the implementation of those abstracted constants. Thus, the data type definition must take as an argument the inference rules encoded as terms representing the dynamic semantics of the introduced constants.

In particular, we define the signature of PCF_1 to be that of PCF_0 (see Figure 4.1) plus the constant

datatype	$(\alpha \longrightarrow o) \longrightarrow (\alpha \longrightarrow tm) \longrightarrow tm,$

in which α is first-order in tm, *i.e.*, of the form tm, $(tm \rightarrow tm)$, $(tm \rightarrow tm \rightarrow tm)$, *etc.*

$$\frac{(\forall c)((R\ c)\ \Rightarrow\ (E\ c) \longrightarrow V)}{(\text{datatype } R\ E) \longrightarrow V} \qquad (14)$$

FIGURE 5.7
Dynamic Semantics for Data Type Definition

The semantics of this constant is supplied by proof rule (14) in Figure 5.7. To illustrate the behavior of this new constant, consider the following term of type *tm*.

```
(datatype λpair[D1]
          λpair[datatype λfst[D2]
                        λfst[datatype λsnd[D3]
                                      λsnd[ E ]]])
```

Here, the expression E is built up from constants of PCF_0 and the abstracted variables **pair**, **fst**, and **snd**. The evaluation of such a term makes use of three universal introductions and three implication introductions to introduce the three new parameters and their evaluation rules. Of course, the expression E can contain uses of the constant **datatype**. The **datatype** term above has the same dynamic semantics as the term (**pairtype** E) (using rule (13)).

We can now state the following theorem relating computations (proofs) in PCF_1 with computations (proofs) in PCF_0. Let $PCF_0(\Sigma, I)$ be the language PCF_0 extended with the new constants of signature Σ and new inference rules I (for I a set of inference rules).

THEOREM 5.8 Let E be a PCF_1 expression and let Ξ be a PCF_1 proof of $E \longrightarrow V$. Let Ξ' be a subproof of Ξ of the formula $E' \longrightarrow V'$ where E' contains no **datatype** constants. Let I be the set of all undischarged assumptions of Ξ' and let Σ be the set of constants in these assumptions that are not part of PCF_0. Then Ξ' is a $PCF_0(\Sigma, I)$ proof of $E' \longrightarrow V'$.

In essence this theorem states that computations in PCF_1 are collections of subcomputations in various extensions of PCF_0. Thus we can describe the evaluation of data type definitions as a three phase process: (*i*) new constants and clauses are introduced (*e.g.*, a language $\mathcal{P}$ is extended to $\mathcal{P}(\Sigma, I)$ in which Σ is the signature for the new constants and I is the set of new clauses); (*ii*) evaluation (of the body of the data type definition) continues over this extended language; and (*iii*) the constants and clauses are discharged. An evaluator can exploit this process in the following way. Assume that the body E of the data type definition contains no occurrence of **datatype**. Then the meta-logic (*i.e.*, meta-interpreter)

required to evaluate E is the one for PCF_0.

There is, however, a significant problem with the dynamic semantics of data type given in Figure 5.7: the rule for **datatype** is badly unconstrained. Clearly, the domain over which the first argument of **datatype** can vary must be constrained if its intended use is to introduce user defined data types into a functional programming setting. Thus, we shall assume that this first argument is picked using some "natural" restriction on the extension of an evaluator.

6

Correctness of Proof Systems

In Chapter 4 we presented a simple programming language PCF_0, an abstract syntax for it, and several proof systems for manipulating expressions in PCF_0. We now would like to justify or support these specifications with correctness proofs. Our reasons for doing this are two-fold. First, we would obviously like to demonstrate that the given specifications are indeed correct. Second, we would like to develop general principles for stating and deriving correctness results for our proof systems. This latter goal is indeed ambitious and is discussed later in this proposal (see Chapter 8).

6.1 Correctness of Type Inference Specification

To demonstrate that the type inference proof system is, in some way, correct we shall relate it to the well-known Damas-Milner type inference system [9]. The type inference system given above differs from the Damas-Milner system in several respects, most of which arise from our use of a higher-order meta-language. In particular our treatment of bound variables, with the introduction of universal constants and the application of β-reduction, must be shown to be correct. Additionally, the Damas-Milner system can infer polytypes for expressions and thus is more general than our system. Despite this fact we can still show a strong relation between the two systems. Below we shall make this relationship precise and discuss its relevance.

First, however, we begin by comparing our representation of terms with the one used by

Damas-Milner (DM) and for this we refer to the material in Chapter 2 in which we described an embedding of the untyped λ-calculus into a simply typed λ-calculus. Essentially, DM considers a set of pre-terms, that are untyped λ-terms, to which types can be inferred. They then present an inference system (read: proof system) in which types are given to well-formed terms. Only those pre-terms that can be typed are considered legal ML expressions. As described previously, with our abstract syntax these pre-terms are represented as simply typed λ-terms (of type *tm*) and we attempt to infer meta-types (terms of type *tp*) to these terms. For simplicity, we consider only the subset of the language with variables, abstraction, application and a polymorphic let. The presentation below is easily extended for the language containing additional constants (*e.g.*, *if*, *fix*, *etc.*)

6.1.1 The Damas-Milner Type Inference System

We briefly outline the Damas-Milner type inference system and then in a following section we provide the comparison between the two methods of type inference. As mentioned above, DM considers a set of pre-terms that are untyped terms generated by the following grammar

$$M \ ::= \ x \ \mid \ MN \ \mid \ \lambda x.M. \ \mid \ let\ x = e_1\ in\ e$$

With a suitable abstract syntax, the terms generated by this grammar can be viewed as a subset of terms of an untyped lambda calculus with constant *let*. (Variables, applications and abstractions directly translate into untyped λ-terms. The untyped term representing a **let** expression would be $(let\,\lambda x.e\,e_1)$. We will refer to this encoding as the DM abstract syntax.) The legal expressions of the language are then those pre-terms that can be given types according to a set of inference rules. This set is given in Figure 6.1 in which Γ is a finite mapping from variables to types and $\Gamma, x{:}\sigma$ is the usual extension of Γ with the proviso that x is not in the domain of Γ. This definition is slightly different than the one given in [9] in that the definition we give for (Inst) is simpler, but this results in no loss of expressiveness. Thus the term $\lambda x.x$ can be given the type $t \rightarrow t$ for type variable t while the term $\lambda x.xx$ cannot be given a type and thus is not a legal expression.

An important property of this type inference system is the existence of a unique principal type for all typable expressions. Given some Γ and e such that $\Gamma \ \rhd\ e : \sigma$ for some σ, then there exists a unique σ_p, called the *principal type-scheme* of e under assumptions Γ such that the following hold:

(*i*) $\Gamma \ \rhd\ e : \sigma_p$

(*ii*) for all σ, if $\Gamma \ \rhd\ e : \sigma$ then $(\sigma_p \sqsubseteq \sigma)$

$$\Gamma \rhd x : \sigma \quad (\Gamma(x) = \sigma) \qquad \text{(Taut)}$$

$$\frac{\Gamma \rhd e : \forall t.\sigma}{\Gamma \rhd e : [\tau/t]\sigma} \quad \textit{for some type } \tau \qquad \text{(Inst)}$$

$$\frac{\Gamma \rhd e : \sigma}{\Gamma \rhd e : (\forall t.\sigma)} \quad (t \textit{ not free in } \Gamma) \qquad \text{(Gen)}$$

$$\frac{\Gamma, x{:}\tau_1 \rhd e : \tau_2}{\Gamma \rhd (\lambda x.e) : (\tau_1 \to \tau_2)} \quad (x \notin \mathrm{FV}(\Gamma)) \qquad \text{(Abs)}$$

$$\frac{\Gamma \rhd e_1 : (\tau_1 \to \tau_2) \qquad \Gamma \rhd e_2 : \tau_1}{\Gamma \rhd (e_1\, e_2) : \tau_2} \qquad \text{(Comb)}$$

$$\frac{\Gamma \rhd e_2 : \sigma \qquad \Gamma, x{:}\sigma \rhd e_1 : \tau_1}{\Gamma \rhd (\mathit{let}\ x = e_2\ \mathit{in}\ e_1) : \tau_1} \quad (x \notin \mathrm{FV}(\Gamma)) \qquad \text{(Let)}$$

FIGURE 6.1
Damas-Milner Type Inference

6.1.2 Relating Abstract Syntaxes

As we want to relate our type inference system to the Damas-Milner one we must first define a relationship between terms in our abstract syntax and the DM abstract syntax. Recall the mappings $(\cdot)^*$ and $(\cdot)^\dagger$ defined in Chapter 2. We see that our abstract syntax is essentially the definition of $(\cdot)^*$, with Φ and Ψ renamed to **lamb** and @, respectively. (We can easily extend the definition of $(\cdot)^*$ to treat **let** or other constants.) Thus, applying the results from that chapter we have a correspondence between terms in our syntax and terms in the DM abstract syntax. So for any pre-term e we have the corresponding term e^* (in our syntax) and for any term e given in our syntax we have the corresponding pre-term $e^\dagger$.

We shall not make any distinction between the types constructed in the two systems. In our system we actually represent types as simply typed λ-terms of meta-type tp with the constructor '$\to$' denoting a meta-term of type $(tp \to tp \to tp)$. In the Damas-Milner formulation, types are built up from a grammar. Any distinction between the representation of object-level types, however, is inconsequential and we shall make no further distinction.

6.1.3 Relating Typing Judgments

Now given an untyped term e we wish to relate proof trees for proofs of the formulas $\Gamma \rhd e : \sigma$ and $(e)^* \xrightarrow{ty} \tau$. As before, τ denotes a monotype, σ a polytype, and Γ is some environment providing a finite map from identifiers to polytypes. We know that the two type inference systems are different because the DM system includes type generalization (Gen) and instantiation (Inst). But this difference can be viewed as a minor technical point as discused below.

To relate the two kinds of typing judgments, we first introduce the notion of generalizing a monotype to a polytype. The idea is to universally quantify the free variables occurring in the monotype. This operation, however, must be done with respect to an environment because the environment may have free type variables (the free type variables of an environment Γ are those free type variables occurring in type assignments) also occurring in the monotype. The definition then is as follows.

DEFINITION 6.1 (gen) Let Γ be some environment and τ some monotype. Then

$$gen(\Gamma, \tau) = \begin{cases} \forall t_1(\cdots(\forall t_n(\tau))) & \text{if } \mathrm{FV}(\tau) \setminus \mathrm{FV}(\Gamma) = \{t_1, \ldots, t_n\} \\ \tau & \text{if } \mathrm{FV}(\tau) \setminus \mathrm{FV}(\Gamma) = \emptyset \end{cases}$$

The order of the universal quantification is arbitrary as it does not affect the subsumes ordering of polytypes.

We can now give the theorem relating the two type inference systems. This theorem is similar to Theorem 2.1 in [5].

THEOREM 6.2 Let e be an arbitrary untyped term with at most free variables $x_0, \ldots, x_k$ and let Γ be a context such that $\Gamma(x_i) = \sigma_i$ for $0 \leq i \leq k$. Furthermore, let $\tau_0, \ldots, \tau_k$ be monotypes such that $gen(\Gamma, \tau_i) \sqsubseteq \sigma_i$ for $0 \leq i \leq k$ (*i.e.*, erasing the quantifiers of σ_i yields τ_i). Then the following hold:

(*i*) if there is a derivation of the proposition $e^* \xrightarrow{ty} \tau$ with at most open assumptions of the form $x_i{}^* \xrightarrow{ty} \tau_i$ (for $0 \leq i \leq k$) then $\vdash \Gamma \rhd e : \tau$;

(*ii*) if $\vdash \Gamma \rhd e : \sigma$ then there exists some type τ such that there is a derivation of the proposition $e^* \xrightarrow{ty} \tau$ with at most open assumptions of the form $x_i{}^* \xrightarrow{ty} \tau_i$ (for $0 \leq i \leq k$) and $(gen(\Gamma, \tau) \sqsubseteq \sigma)$

A general outline of the proof of the theorem is as follows. We begin by defining a variant of the Damas-Milner type inference system, called DM′, that provides a sort of cut-elimination. We then show the equivalence of these two systems. Then we prove the

soundness and completeness of our proof system by constructing a (two-way) transformation between proofs in our system and proofs in DM′.

6.1.4 Cut Elimination for Damas-Milner

Before proving Theorem 6.2 we need to prove a form of cut elimination for the Damas-Milner type inference system. Recall that the cut rule, in a Gentzen-style LJ and LK calculi, is of the form:

$$\frac{\Gamma \longrightarrow \Delta, A \qquad A, \Lambda \longrightarrow \Theta}{\Gamma, \Lambda \longrightarrow \Delta, \Theta} \qquad \text{(Cut)}$$

A is called the *cut formula* of this inference. Gentzen's Hauptsatz then says the following:

> HAUPTSATZ: Every LJ- or LK-derivation can be transformed into an LJ or LK-derivation with the same endsequent and in which the inference figure called 'cut' does not occur. [16]

Notice that this rule formalizes the use of an *auxiliary lemma* in a proof. This is a technique constantly used in practical mathematics. We can clarify this correspondence between cut formulas and auxiliary lemmas by considering the case in which Δ is empty. Then $\Gamma \longrightarrow A$ is the auxiliary lemma that can be taken as belonging to a catalogue of already-proven results. Now using A as an assumption, if we can show using other assumptions Λ, that Θ is provable, then we can conclude that $\Gamma, \Lambda \longrightarrow \Theta$ is provable. The conclusion does not refer to A. (See [15], pp. 109–110.)

Now recall the *let* rule in the DM type inference system:

$$\frac{\Gamma \rhd e_2 : \sigma \qquad \Gamma, x : \sigma \rhd e_1 : \tau_1}{\Gamma \rhd (let\ x = e_2\ in\ e_1) : \tau_1} \quad (x \notin \mathrm{FV}(\Gamma)) \qquad \text{(Let)}$$

Notice that a polymorphic type σ is assumed for x, but the inferred type for the entire expression is only a monotype. This intermediate use of polytypes when the resultant type is just a monotype appears to be analogous to the cut rule above.

We now give an important lemma that we will use in showing the relationship between DM and our type inference proof system.

LEMMA 6.3

(i) If $\vdash \Gamma \rhd e' : \sigma'$ and $\vdash \Gamma, x{:}\sigma' \rhd e : \sigma$ then $\vdash \Gamma \rhd [e'/x]e : \sigma$.
(ii) If $\vdash \Gamma \rhd e' : \sigma'$ and $\vdash \Gamma \rhd [e'/x]e : \sigma$ then $\vdash \Gamma, x{:}\sigma'' \rhd e : \sigma$ where $\sigma'' = gen(\Gamma, \sigma')$.

PROOF. We prove only (i); the proof for (ii) proceeds similarly, though one must justify

the use of *gen*. We assume a DM-proof Ξ of $\Gamma, x{:}\sigma' \rhd e : \sigma$ and some DM-proof of $\Gamma \rhd e' : \sigma'$. The proof is by induction on the height h of Ξ.

base: $h = 1$. Ξ must be of the form $\Gamma, x{:}\sigma' \rhd y : \sigma \quad (\Gamma(y) = \sigma)$ and so $e = y$ for some variable y. Now two cases to consider:

(i) $x = y$: then $[e'/y]y = e'$ and by assumption, $\vdash \Gamma \rhd e' : \sigma'$ $(\sigma = \sigma')$.

(ii) $x \neq y$: then $[e'/x]y = y$ and we have $\vdash \Gamma, x{:}\sigma' \rhd y : \sigma$, but also $\vdash \Gamma \rhd y : \sigma$ since $x \neq y$.

step: We consider the possible cases according to the last inference rule occurring in Ξ.

(i) The last inference rule is an instance of (Inst):

$$\frac{\Gamma, x{:}\sigma' \rhd e : \forall t.\sigma}{\Gamma, x{:}\sigma' \rhd e : [\tau/t]\sigma}$$

By induction hypothesis, $\vdash \Gamma \rhd [e'/x]e : \forall t.\sigma$ and hence we have $\vdash \Gamma \rhd [e'/x]e : [\tau/t]\sigma$ by application of (Inst).

(ii) The last inference rule is an instance of (Gen):

$$\frac{\Gamma, x{:}\sigma' \rhd e : \sigma}{\Gamma, x{:}\sigma' \rhd e : (\forall t.\sigma)} \quad (t \textit{ not free in } \Gamma)$$

By induction hypothesis, $\vdash \Gamma \rhd [e'/x]e : \sigma$ and hence we have $\vdash \Gamma \rhd [e'/x]e : (\forall t.\sigma)$ by application of (Gen).

(iii) The last inference rule is an instance of (Abs):

$$\frac{\Gamma, x{:}\sigma', y{:}\tau_1 \rhd e : \tau_2}{\Gamma, x{:}\sigma' \rhd (\lambda y.e) : (\tau_1 \to \tau_2)} \quad (y \notin \mathrm{FV}(\Gamma))$$

By induction hypothesis, $\vdash \Gamma, y{:}\tau_1 \rhd [e'/x]e : \tau_2$ and, applying (Abs) to this, we have $\vdash \Gamma \rhd \lambda y.[e'/x]e : (\tau_1 \to \tau_2)$. But $x \neq y$ (by assumption on contexts) and so this is equal to $\vdash \Gamma \rhd [e'/x]\lambda y.e : (\tau_1 \to \tau_2)$.

(iv) The last inference rule is an instance of (Comb);

$$\frac{\Gamma, x{:}\sigma' \rhd e_1 : (\tau_1 \to \tau_2) \qquad \Gamma, x{:}\sigma' \rhd e_2 : \tau_1}{\Gamma, x{:}\sigma' \rhd (e_1\, e_2) : \tau_2}$$

By induction hypothesis, $\vdash \Gamma \rhd [e'/x]e_1 : (\tau_1 \to \tau_2)$ and $\vdash \Gamma \rhd [e'/x]e_2 : \tau_1$ and, applying (Comb) to this, we have $\vdash \Gamma \rhd (([e'/x]e_1)\,([e'/x]e_2)) : \tau_2$. And since $(([e'/x]e_1)\,([e'/x]e_2))$ is equivalent to $[e'/x](e_1\, e_2)$, we also have $\vdash \Gamma \rhd [e'/x](e_1\, e_2) : \tau_2$.

(v) The last inference rule is an instance of (Let):

$$\frac{\Gamma, x{:}\sigma' \rhd e_2 : \sigma \qquad \Gamma, x{:}\sigma', y{:}\sigma \rhd e_1 : \tau_1}{\Gamma, x{:}\sigma' \rhd (\mathit{let}\ y = e_2\ \mathit{in}\ e_1) : \tau_1} \quad (y \notin \mathrm{FV}(\Gamma))$$

By induction hypothesis, $\vdash \Gamma \rhd [e'/x]e_2 : \sigma$ and $\vdash \Gamma, y{:}\sigma \rhd [e'/x]e_1 : \tau_1$ and, applying (Let) to these, we have $\vdash \Gamma \rhd (\mathit{let}\ y = [e'/x]e_2\ \mathit{in}\ [e'/x]e_1) : \tau_1$. And since $x \neq y$ (by assumption on contexts), we have $\vdash \Gamma \rhd [e'/x](\mathit{let}\ y = e_2\ \mathit{in}\ e_1) : \tau_1$. □

Next we describe a notion of a normalized DM-proof that we will find important in demonstrating the correspondence between proofs in our type inference system and DM-proofs.

LEMMA 6.4 (∀-normalization) If $\Gamma \rhd e : \sigma$ is provable then there exists a DM-proof Ξ such that no conclusion of an instance of (Gen) is a premise to an instance of (Inst). We call Ξ a ∀-normal proof.

This lemma follows directly from the notion of ∀-normalization as found in [51]. From this lemma we have the following corollary:

COROLLARY 6.5 If Ξ is a ∀-normal proof, then any instance of the (Gen) rule in Ξ occurs either

(a) as the last inference rule of Ξ;
(b) as the last inference rule for the left premise of an instance of (Let); or
(c) immediately above another instance of (Gen).

Now we can state and prove our version of cut-elimination for DM-proofs.

THEOREM 6.6 (Cut-Elimination for DM-Proofs) There exists a DM-proof Ξ of $\Gamma \rhd e : \sigma$ iff there exists a DM′-proof Ξ' of the same formula, where DM′ is the same as DM, except the (Let) rule is

$$\frac{\Gamma \rhd e_2 : \tau \qquad \Gamma \rhd [e_2/x]e_1 : \tau_1}{\Gamma \rhd (\mathit{let}\ x = e_2\ \mathit{in}\ e_1) : \tau_1} \quad (x \notin \mathrm{FV}(\Gamma)) \qquad (\mathrm{Let}').$$

PROOF. By Lemma 6.4 we need only show that given a DM-proof of $\Gamma \rhd e : \sigma$ we can construct a corresponding DM′-proof, and vice-versa. (We can easily extend the notion of ∀-normal DM-proofs to DM′-proofs.)

I. Assume we have a ∀-normal DM-proof Ξ of $\Gamma \rhd e : \sigma$; we prove, by induction on the height h of Ξ, the existence of a DM′-proof of the same proposition.

base: $h = 0$. Then Ξ is of the form

$$\Gamma \rhd x : \sigma \quad (\Gamma(x) = \sigma)$$

Then trivially, Ξ is also a DM′-proof.

step: $h = n$ for some $n > 0$. We assume theorem holds for all proof trees of size $m < n$. We divide the possible cases into two groups:

(a) The last inference rule of Ξ is one other than (Let). Then Ξ is of the form

$$\frac{\Xi_1}{\Gamma \rhd e : \sigma}$$

and by inductive hypothesis we can construct a DM′-proof Ξ_1' for Ξ_1. And since the last inference is not a (Let), we have the following DM′-proof:

$$\frac{\Xi_1'}{\Gamma \rhd e : \sigma}$$

as all other inference rules are the same in DM′.

(b) The last inference rule of Ξ is an instance of (Let). The let e be of the form $(let\ x = e_2\ in\ e_1)$. Now Ξ must be of the form

$$\frac{\Xi_2 \qquad \Xi_1}{\Gamma \rhd (let\ x = e_2\ in\ e_1) : \tau_1}$$

in which Ξ_2 is $\forall$-normal with root $\Gamma \rhd e_2 : \sigma$ and Ξ_1 is $\forall$-normal with root $\Gamma, x{:}\sigma \rhd e_1 : \tau_1$. Clearly, from Ξ_2 we can construct a DM-proof Ξ_3 of $\Gamma \rhd e_2 : \tau$, for some monotype τ, in one of two ways:

($b1$) if $e_2 = y$ for some variable y and $\Gamma(y) = \sigma_y$, then we just apply successive instance of (Inst) to σ_y, until we obtain a monotype;
($b2$) Ξ_2 must be of the form

$$\frac{\Gamma \rhd e_2 : \tau}{\Gamma \rhd e_2 : \sigma}\ (\text{Gen})^*$$

in which (Gen)* is some sequence of (Gen)'s. Then by induction hypothesis we can construct a DM′-proof Ξ_2' of $\Gamma \rhd e_2 : \tau$.

Now assume we have a DM-proof Ξ_1 of $\Gamma, x{:}\sigma \rhd e_1 : \tau_1$. Then by part ($i$) of Lemma 6.3, there exists a DM-proof of $\Gamma \rhd [e_2/x]e_1 : \tau_1$ and by induction hypothesis we can construct a DM′-proof Ξ_1' of $\Gamma \rhd [e_2/x]e_1 : \tau_1$. Hence we have the DM′-proof

$$\frac{\Xi_2' \qquad \Xi_1'}{\Gamma \rhd (let\ x = e_2\ in\ e_1) : \tau_1}$$

This completes Part I of the proof.

II. Assume we have a DM′-proof Ξ' of $\Gamma \rhd e : \sigma$; we prove the existence of a DM-proof.

base: $h = 0$. Then Ξ' is of the form

$$\Gamma \rhd x : \sigma \quad (\Gamma(x) = \sigma)$$

Then trivially, Ξ' is also a DM-proof.

step: $h = n$ for some $n > 0$. We assume the theorem holds for all proof trees of size $m < n$. We divide the possible cases into two groups:

(a) The last inference rule of Ξ is one other than (Let). Then, as in step (a) of Part I, we can trivially construct a DM-proof.

(b) The last inference rule of Ξ is an instance of (Let). Then let e be of the form $(let\ x = e_2\ in\ e_1)$. Now Ξ' must be of the form

$$\frac{\Xi_2' \qquad \Xi_1'}{\Gamma \rhd (let\ x = e_2\ in\ e_1) : \tau_1}$$

in which the root of Ξ_2' is $\Gamma \rhd e_2 : \tau$ and the root of Ξ_1' is $\Gamma \rhd [e_2/x]e_1 : \tau_1$. By induction hypothesis we can construct a DM-proof of $\Gamma \rhd e_2 : \tau$ but then we can also construct a DM-proof Ξ_2 of $\Gamma \rhd e_2 : \sigma$ in which $\sigma = gen(\Gamma, \tau)$. Also by induction hypothesis we can construct a DM-proof Ξ_1 of $\Gamma \rhd [e_2/x]e_1 : \tau_1$. Then by part (ii) of Lemma 6.3, there exists a DM-proof Ξ_3 of $\Gamma, x{:}\sigma \rhd e_1 : \tau_1$. Hence we have the DM-proof

$$\frac{\Xi_2 \qquad \Xi_3}{\Gamma \rhd (let\ x = e_2\ in\ e_1) : \tau_1}.$$

□

One important property of ∀-normal DM′-proofs is the following.

COROLLARY 6.7 In a ∀-normalized DM′-proof of $\Gamma \rhd e : \tau$, in which e is closed, there are no occurrences of (Inst) or (Gen).

This follows immediately from Corollary 6.5 and the definition of DM′-proofs. Finally we have the following lemma relating substitution to typing:

LEMMA 6.8 $e \xrightarrow{ty} \tau$ has a derivation with at most open assumptions $x_i \xrightarrow{ty} \tau_i$ for $0 \leq i \leq k$ *iff* $[c/x_j]e \xrightarrow{ty} \tau$ has a derivation with at most open assumptions $c \xrightarrow{ty} \tau_j$ and $x_i \xrightarrow{ty} \tau_i$ for $0 \leq i \leq k$, $i \neq j$, in which c is some constant not appearing in e.

The proof is by induction on the structure of e and not presented here.

Recall that in our proof system for $\xrightarrow{ty}$, we have no rules corresponding to (Gen) and (Inst) and our versions of (Abs) and (Let) use β-reduction (substitution). We now begin to see that $\forall$-normalized DM′-proofs are very similar in structure to proofs in our system. In fact, we claim that there is an isomorphism between $\forall$-normalized DM′-proofs and proofs in our system ($\xrightarrow{ty}$). For the reader convinced of this fact, the following soundness and completeness proofs can be skipped. The skeptical reader should continue on.

6.1.5 Proof of Correctness Theorem

We now show the soundness and completeness results for Theorem 6.2. We actually show this with respect to DM′-proofs for the Damas-Milner system, but as we have shown these to be equivalent to DM-proofs, no problem arises.

I. (Soundness)
We show that if there is a derivation of the proposition ${e^*} \xrightarrow{ty} \tau$ with at most open assumptions of the form ${x_i}^* \xrightarrow{ty} \tau_i$ (for $0 \leq i \leq k$) then there is a corresponding DM′-proof of $\Gamma \rhd e : \tau$ such that $\Gamma(x_i) = \sigma_i$ and $gen(\Gamma, \tau_i) \sqsubseteq \sigma_i$.

The proof is by induction on the size of proof tree Ξ. By the normal-form theorem for proofs involving $\xrightarrow{ty}$, we need only consider normal derivation trees Ξ.

base: Ξ is of height 1. Then e is some variable x_i.

Assume we have a derivation of ${x_i}^* \xrightarrow{ty} \tau$ for some τ. Then the derivation tree must simply be ${x_i}^* \xrightarrow{ty} \tau_i$.

Then, by definition $\Gamma(x_i) = \sigma_i$ and we have the following DM′-derivation tree:

$$\frac{\Gamma \rhd x_i : \sigma_i \quad (\Gamma(x_i) = \sigma_i)}{\Gamma \rhd x_i : \tau_i}\ (\text{Inst})^*$$

where $(\text{Inst})^*$ is an abbreviation for some sequence of instances of (Inst) that will produce the generic instance τ of σ. This is possible since $gen(\Gamma, \tau_i) \sqsubseteq \sigma_i$.

step: For the inductive case, *i.e.*, a proof tree Ξ of height > 1, we only need to consider the three cases for compound expressions.

(i) e is $(e_1\ e_2)$. Assume we have a derivation of ${(e_1}^* @ {e_2}^*) \xrightarrow{ty} \tau$. Hence, there exists some derivation tree

$$\frac{\Xi_1 \qquad \Xi_2}{{(e_1}^* @ {e_2}^*) \xrightarrow{ty} \tau}$$

in which Ξ_1 and Ξ_2 are subtrees whose roots are (respectively) ${e_1}^* \xrightarrow{ty} (\tau_1 \rightarrow \tau_2)$ and ${e_2}^* \xrightarrow{ty} \tau_1$ and each with at most open assumptions of the form $x_i \xrightarrow{ty} \tau_i$.

Now by inductive hypothesis we must have DM′-proofs of $\Gamma \rhd e_1 : (\tau_1 \rightarrow \tau_2)$ and $\Gamma \rhd e_2 : \tau_1$. Then we can construct the DM′-proof tree whose last inference rule is

$$\frac{\Gamma \rhd e_1 : (\tau_1 \rightarrow \tau_2) \qquad \Gamma \rhd e_2 : \tau_1}{\Gamma \rhd (e_1\, e_2) : \tau_2}.$$

(*ii*) e is $\lambda y.e_1$. Assume we have a derivation of $(\mathsf{lamb}\ \lambda y^*.{e_1}^*) \xrightarrow{ty} (\tau_1 \rightarrow \tau_2)$. Hence, there exists some derivation tree

$$\frac{\Xi_1}{(\mathsf{lamb}\ \lambda y^*.{e_1}^* \xrightarrow{ty} (\tau_1 \rightarrow \tau_2)))}$$

in which Ξ_1 is a subtree whose root is

$$\forall c\ (c \xrightarrow{ty} \tau_1) \ \Rightarrow\ (\lambda y^*.{e_1}^*\, c) \xrightarrow{ty} \tau_2$$

with at most open assumptions of the form $x_i \xrightarrow{ty} \tau_i$. But $(\lambda y^*.{e_1}^*\, c)$ β-reduces to $[c/y^*]{e_1}^*$. Now we must have some derivation of $[c/y^*]{e_1}^* \xrightarrow{ty} \tau_2$ with at most open assumptions of the form $x_i \xrightarrow{ty} \tau_i$ and $c \xrightarrow{ty} \tau_1$. By Lemma 6.8, we also have a derivation of ${e_1}^* \xrightarrow{ty} \tau_2$ with at most open assumptions of the form $x_i \xrightarrow{ty} \tau_i$ and $y \xrightarrow{ty} \tau_1$. By inductive hypothesis we can construct a proof tree of $\Gamma, y{:}\tau_1 \rhd e_1 : \tau_2$. Then we can construct the DM′-proof tree whose last inference rule is

$$\frac{\Gamma, y{:}\tau_1 \rhd e_1 : \tau_2}{\Gamma \rhd \lambda y.e_1 : (\tau_1 \rightarrow \tau_2)}.$$

(*iii*) e is $(let\ y = e_2\ in\ e_1)$. Assume we have a derivation of $(\mathsf{let}\ \lambda y^*.{e_1}^*\ {e_2}^*) \xrightarrow{ty} \tau_1$. Hence, there exists some derivation tree

$$\frac{\Xi_1 \qquad \Xi_2}{(\mathsf{let}\ \lambda y^*.{e_1}^*\ \ {e_2}^*) \xrightarrow{ty} \tau_1}$$

in which Ξ_1 and Ξ_2 are subtrees whose roots are ${e_2}^* \xrightarrow{ty} \tau_2$ and $(\lambda y^*.{e_1}^*\, {e_2}^*) \xrightarrow{ty} \tau_1$, respectively, and with at most open assumptions of the form $x_i \xrightarrow{ty} \tau_i$. But by one-step β-reduction, this latter formula is $([{e_2}^*/y^*]{e_1}^*) \xrightarrow{ty} \tau_1$.

By inductive hypothesis we must have DM′-proofs of $\Gamma \rhd e_2 : \tau_2$ and $\Gamma \rhd [e_2/y]e_1 : \tau_1$. And now from $\vdash \Gamma \rhd e_2 : \tau_2$, $\vdash \Gamma \rhd [e_2/y]e_1 : \tau_1$ we can construct a DM′-proof tree whose last inference rule is

$$\frac{\Gamma \rhd e_2 : \tau_2 \qquad \Gamma \rhd [e_2/y]e_1 : \tau_1}{\vdash \Gamma \rhd (let\ y = e_2\ in\ e_1) : \tau_1}.$$

II. (Completeness)
We show that for any DM′-proof of $\Gamma \rhd e : \sigma$, there is a derivation tree Ξ' of $e^* \xrightarrow{ty} \tau$ such that $gen(\Gamma, \tau) \sqsubseteq \sigma$, in which the free variables of e are at most $x_0, \ldots, x_k$, the open assumptions of Ξ' are of the form $x_i \xrightarrow{ty} \tau_i$, $\Gamma(x_i) = \sigma_i$ and $gen(\Gamma, \tau_i) \sqsubseteq \sigma_i$.

The proof is by induction on the size of proof tree Ξ for $\Gamma \rhd e : \sigma$. By Theorem 6.6 it is sufficient to consider $\forall$-normalized DM′-proofs. So let Ξ be a $\forall$-normalized DM′-proof of $\Gamma \rhd e : \sigma$.

base: Assume we have a proof Ξ of $\Gamma \rhd e : \sigma$ and the height of Ξ is 1. Then e must be some variable x and a proof tree for this expression must be of the form

$$\Gamma \rhd x_i : \sigma_i \qquad (\Gamma(x_i) = \sigma_i)$$

for some polytype σ_i. Then we have the derivation tree ${x_i}^* \xrightarrow{ty} \tau_i$ And by assumption, $gen(\Gamma, \tau_i) \sqsubseteq \sigma_i$.

step: We assume the theorem holds for all proof trees of size n and now consider proof trees of size $n+1$. Assume we have a proof Ξ of $\Gamma \rhd e : \sigma$ and the height of Ξ is $n+1$. We proceed by examining the last inference rule applied in these Ξ.

(i) The last inference rule applied is an instance of (Inst):

$$\frac{\Gamma \rhd e : \forall t.\sigma}{\Gamma \rhd e : [\tau/t]\sigma}$$

By inductive hypothesis we can construct a derivation tree for $e^* \xrightarrow{ty} \tau$ such that $gen(\Gamma, \tau) \sqsubseteq \forall t.\sigma$. But $\forall t.\sigma \sqsubseteq [\tau/t]\sigma$ and so (by transitivity of $\sqsubseteq$) $gen(\Gamma, \tau) \sqsubseteq [\tau/t]\sigma$.

(ii) The last inference rule applied is an instance of (Gen):

$$\frac{\Gamma \rhd e : \sigma}{\Gamma \rhd e : (\forall t.\sigma)} \qquad (t \textit{ not free in } \Gamma)$$

By inductive hypothesis we can construct a proof of $e^* \xrightarrow{ty} \tau$ such that $gen(\Gamma, \tau) \sqsubseteq \sigma$. But by definition of gen, we must have $gen(\Gamma, \tau) \sqsubseteq \forall t.\sigma$ (since t is not free in Γ).

(iii) The last inference rule applied is an instance of (Abs):

$$\frac{\Gamma, y{:}\tau_1 \rhd e : \tau_2}{\Gamma \rhd (\lambda y.e) : (\tau_1 \rightarrow \tau_2)}$$

in which $y \notin \mathrm{FV}(\Gamma)$. By inductive hypothesis we can construct a derivation tree for $e^* \xrightarrow{ty} \tau_2$ with at most open assumptions of the form $x_i \xrightarrow{ty} \tau_i$ and $y \xrightarrow{ty} \tau_1$. Then by Lemma 6.8 we can also construct a derivation tree for $[c/y^*]e^* \xrightarrow{ty} \tau_2$ with at

most open assumptions of the form $x_i \xrightarrow{ty} \tau_i$ and $c \xrightarrow{ty} \tau_1$. But then we can also construct a derivation tree for $\forall c\ (c \xrightarrow{ty} \tau_1 \ \Rightarrow\ [c/y^*]e^* \xrightarrow{ty} \tau_2)$ with at most open assumptions of the form $x_i \xrightarrow{ty} \tau_i$ and hence we can construct a derivation tree for $(\textsf{lamb}\ \lambda y^*.e^*) \xrightarrow{ty} (\tau_1 \rightarrow \tau_2)$. And by definition, $gen(\Gamma, (\tau_1 \rightarrow \tau_2)) \sqsubseteq (\tau_1 \rightarrow \tau_2)$.

(iv) The last inference rule applied is an instance of (Comb):

$$\frac{\Gamma \rhd e_1 : (\tau_1 \rightarrow \tau_2) \qquad \Gamma \rhd e_2 : \tau_1}{\Gamma \rhd (e_1\ e_2) : \tau_2}.$$

By the induction hypothesis we can construct derivation trees for ${e_1}^* \xrightarrow{ty} (\tau_1 \rightarrow \tau_2)$ and ${e_2}^* \xrightarrow{ty} \tau_1$ and hence we can construct a derivation tree for $({e_1}^* @ {e_2}^*) \xrightarrow{ty} \tau_2$. And by definition, $gen(\Gamma, \tau_2) \sqsubseteq \tau_2$.

(v) The last inference rule applied is an instance of (Let):

$$\frac{\Gamma \rhd e_2 : \tau \qquad \Gamma \rhd [e_2/x]e_1 : \tau_1}{\Gamma \rhd (let\ y = e_2\ in\ e_1) : \tau_1}$$

By the induction hypothesis we can construct derivation trees for ${e_2}^* \xrightarrow{ty} \tau$ and $([e_2/y]e_1)^* \xrightarrow{ty} \tau_1$ with at most open assumptions of the form $x_i \xrightarrow{ty} \tau_i$. Hence we can construct a derivation tree for $(\textsf{let}\ \lambda y^*.{e_1}^*\ \ {e_2}^*) \xrightarrow{ty} \tau_2$ with at most open assumptions of the form $x_i \xrightarrow{ty} \tau_i$. And by definition, $gen(\Gamma, \tau_1) \sqsubseteq \tau_1$.

This completes the proof of Theorem 6.2. □

6.1.6 Remarks on Correctness Result

We have thus shown that essentially, our type inference system and the Damas-Milner one produce the same typings for terms (up to generalization of type variables). Thus we have shown an implicit existence of principal types for our system (since Damas-Milner has principal types). We can then invoke the soundness and completeness results for Damas-Milner (with respect to a semantics of types) as discussed, for example, in [39], to conclude that our system is also sound and complete with respect to that semantics. We make two observations regarding the proof carried out in this section.

First, the correctness proof is of an indirect sort, in the sense that we prove correctness not by describing a semantics for types and then showing that certain terms are elements of the semantic value of types, but rather by showing an equivalence between systems. A more satisfying approach would be a direct one, but to date we have not developed the methods

for carrying out such proofs. We do hope, however, that such methods, when developed, will offer new insights and provide new techniques into proofs of this kind. We base such hopes on the relative strength, in terms of a higher-order logic, of our proof systems (*i.e.*, meta-language).

Second, we demonstrated a (two-way) translation between proofs in our system and proofs in another system. Specifically, the other system is one in which the extent of abstractions is less than in our system. This notion of extent can be explained in the following manner. In our proof systems, the concept of abstraction exists at two distinct levels: the object level (where it is denoted by **lamb** expressions) and the meta-level (where it is denoted by λ). We use the abstractions at the meta-level to assist in the manipulation of object-level abstractions. In the Damas-Milner system, however, the concept of abstraction exists only at the object level (where it is denoted by λ). At the meta-level, environments are introduced to manipulate abstractions. Thus we have informally provided a translation between two meta-languages. We say informally here because we have not formally described a language (a meta-meta-language?) for specifying this translation. This manipulation of proof systems appears in numerous places throughout this work and thus suggests that future work should attempt to define a language for specifying such translations. See Chapter 8 for more comments on this.

6.2 Correctness Issues for Dynamic and Compilation Semantics

We shall discuss the correctness of our dynamic and compilation semantics together, as we are able to present a general concept of translation between various dynamic semantics presented at different levels of abstraction. Compilation is just a particular kind of translation in which the source is "high-level" and the target is typically "low-level" machine code. In our case, the source and target both specify a dynamic semantics for PCF_0. We shall see how we can extend this notion of translation to be between other dynamic semantics specifications. We will then make observations about the differences among these specifications. We present four "meta-languages" (and object languages) for expressing the dynamic semantics of a basic functional programming language, in order of their level of abstraction.

6.2.1 Four Levels of Dynamic Semantics

The first dynamic semantics is the most abstract and represents programs as untyped λ-terms. The notion of β-reduction is defined as the only operation on λ-terms. (Equivalence of terms up to α-conversion is still assumed, but it is not provided as an explicit operation.) and the evaluation of a program is given by a chain of reduction steps. As an extreme, this language does not even have constants: Integers, booleans, a conditional expression and arithmetic can all be encoded via combinators (closed λ-terms). We refer to this specification as ds_0.

The second dynamic semantics is the one presented in Section 4.5 of this proposal which we shall refer to as ds_1. In this specification we have abstractions present at both the meta- and object levels. The object language is PCF_0 as presented in this proposal. This language has explicit abstractions *e.g.*, of the form $\lambda x.e$ (in concrete syntax). Our meta-language, recall, is a proof system in which the terms are simply typed λ-terms. We represent the object-level abstraction above by the term (**lamb** $\lambda x.e^*$). Note how we use the abstraction at the meta-level to treat the abstraction at the object level.

The third dynamic semantics is a simple variant of the one presented in [5] which we shall refer to as ds_2. For our purposes, we will take the object language again to be PCF_0 (this is a slight simplification but does not affect our arguments). The meta-language is first-order in that abstractions are not present. The terms of the language are first-order terms and the abstractions of the object-level are manipulated using contexts or environments in the meta-language. In this example a context is a partial function (with finite domain) from identifiers (in the object language) to values in the object language. (Note that this definition of context differs from the one given in [5] where a context is a list of pairs (x, α) with an implicit order.) The context maintains the object-level binding information explicitly as a partial function with finite domain. So, based on the level of abstraction, ds_2 is of a lower-level than ds_1.

Finally, the fourth dynamic semantics is the one given by the CAM, described in this proposal and also in [5]. We shall refer to this one as ds_3. In this case, the object language is CAM and the dynamic semantics is given by either our meta-language or the one for ds_2 (the difference is minimal). We choose the one for ds_2 as it will help point out certain differences. Now in ds_3 we no longer have abstractions explicitly present in either language (meta or object). Thus this dynamic semantics is of the "lowest" level considered.

6.2.2 Relating the Dynamic Semantics

We wish show how each of these dynamic semantics is, in a sense, equivalent, to the others. We can do this by showing transformations between proofs of ds_i and ds_{i+1}. Of course this is not quite true since in the object languages of ds_0 and ds_3 we can write and evaluate many programs that are not allowed in ds_1 and ds_2. The presentation of the four dynamic semantics provided insight into the varying degree and use of abstraction, but of practical concern to us shall be the relationship between ds_1 and ds_2, as they both have the same object language.

A complete specification for ds_2 is given in Figure 6.2. Note that in ds_2 expressions are denoted by first-order structures, and hence the symbol λ used in rules (7) and (10) is not a true binding operation. We make one subtle observation regarding this specification. In rules (8), (9) and (10) an environment is extended from ρ to $\rho \cdot P \mapsto \alpha$ but the implicit assumption here is that P is not in the domain of ρ. Hence there is actually a restriction on the abstract syntax: conflicts between bound variable names are not allowed. In ds_1, abstractions at the meta-level dispense with such restrictions as conflicts are managed by α-conversion.

The connection between ds_2 and ds_3 has been given in [11]. This proof of translation shows that if a program e translates to CAM code C, then the ds_2 semantics of e is equivalent to the ds_3 semantics of C. This is shown by demonstrating a correspondence between proof trees in the two systems. The proof itself is rather long and tedious and is not reproduced here. We believe that a proof of translation between ds_1 and ds_3, based on the compilation semantics given in Section 4.6, would be similar, but there may be some advantages owing to the higher level of our meta-language. However, note that there is a larger "gap" between the meta-languages of ds_1 and ds_3. Thus we might expect that some parts of the proof of translation would be more complex. (We are reasoning about two systems with less in common than in the case of [11].)

Instead of showing a direct correspondence between ds_1 and ds_3 we would like show the relationship between ds_1 and ds_2. From this we will be able to conclude that our compilation semantics is, in some sense, equivalent to that given in [11]. This will in turn prove correct our compilation semantics. We shall not comment further on ds_0 but only note that it is the most abstract of the four dynamic semantics presented here as its only method of evaluation is via β-reduction. We shall not present the formal proof of the equivalence between ds_1 and ds_2, but rather only present the salient points of the proof. To show an equivalence between ds_1 and ds_2 we must first have a bijection between programs in each system. This is trivial and is based on the material of Chapter 2. With an abuse of notation, we shall not

$$\rho \rhd N \Longrightarrow N \qquad \rho \rhd \text{true} \Longrightarrow \text{true} \qquad \rho \rhd \text{false} \Longrightarrow \text{false} \tag{1,2,3}$$

$$\rho \rhd x \Longrightarrow \rho(x) \tag{4}$$

$$\frac{\rho \rhd E_1 \Longrightarrow \text{true} \qquad \rho \rhd E_2 \Longrightarrow \alpha}{\rho \rhd \text{if } E_1 \text{ then } E_2 \text{ else } E_3 \Longrightarrow \alpha} \qquad \frac{\rho \rhd E_1 \Longrightarrow \text{false} \qquad \rho \rhd E_3 \Longrightarrow \alpha}{\rho \rhd \text{if } E_1 \text{ then } E_2 \text{ else } E_3 \Longrightarrow \alpha} \tag{5,6}$$

$$\rho \rhd \lambda P.E \Longrightarrow [\lambda P.E, \rho] \tag{7}$$

$$\frac{\rho \rhd E_1 \Longrightarrow [\lambda P.E, \rho_1] \qquad \rho \rhd E_2 \Longrightarrow \alpha \qquad \rho_1 \cdot P \mapsto \alpha \rhd E \Longrightarrow \beta}{\rho \rhd E_1 @ E_2 \Longrightarrow \beta} \tag{8}$$

$$\frac{\rho \rhd E_2 \Longrightarrow \alpha \qquad \rho \cdot P \mapsto \alpha \rhd E_1 \Longrightarrow \beta}{\rho \rhd \text{let } P = E_2 \text{ in } E_1 \Longrightarrow \beta} \tag{9}$$

$$\frac{\rho \cdot F \mapsto \alpha \rhd E_1 \Longrightarrow \alpha}{\rho \rhd \text{fix } \lambda F.E_1 \Longrightarrow \alpha} \tag{10}$$

FIGURE 6.2
ds_2 Specification

distinguish between expressions in the two languages. Naively then, we must show something like

$$\forall e \forall \alpha \ (e \xrightarrow{se} \alpha \ \Leftrightarrow \ \rho \rhd e \Longrightarrow \alpha)$$

(for closed e) in which the latter formula is the proposition from ds_2 expressing the evaluation of e to α. For simplicity we shall take ρ to be $\emptyset$ initially (*i.e.*, the context whose domain is empty). It should be clear that since e is closed, this assumption is safe.

However, this naive statement of correctness overlooks the fact that the "values" in ds_1 and ds_2 (*i.e.*, the expressions appearing on the right-hand side of an arrow) are not identical. In particular, ds_2 contains closures of the form $[\![\lambda P.E, \rho]\!]$ while ds_1 has no closure. (This difference is an artifact of the different levels of abstraction at which the two systems work.) As mentioned in Chapter 4, closures ensure the static scoping of ds_2. Thus before showing an equivalence between the two proof systems, we must define an appropriate equivalence between values. To do this we begin by relating the context ρ of ds_2 to the universally introduced constants of ds_1. This requires that we can refer to "corresponding" instances of propositions in any two proofs Ξ_1 and Ξ_2 (Ξ_1 in ds_1 and Ξ_2 in ds_2) of the

same proposition. We observe (without proof) that for arbitrary Ξ_1 and Ξ_2, both of which prove the same proposition, say $e \xrightarrow{se} \alpha$ ($\emptyset \rhd e \Longrightarrow \alpha$), have a similar structure and we are able to identify instances of propositions in each tree that correspond to the same abstract notion of computation. Then we show that given two such propositions, say $e' \xrightarrow{se} \alpha'$ and $\rho' \rhd e' \Longrightarrow \alpha'$, that $\rho(x) = \beta$ just in the case that at some proposition P occurring below $e' \xrightarrow{se} \alpha'$ in Ξ_1, P is a β-redex of the form $(\lambda x.e''\,\beta)$. This property essentially shows that our use of meta-level abstractions and β-conversion performs the same function as the contexts of ds_2.

We can then show an equivalence of values in the two semantics, with the only tricky part involving abstractions, *i.e.*, (**lamb** M) and $[\![\lambda P.E, \rho]\!]$. Combining this equivalence of values with the structural relationship between proof trees, we can then prove, by induction (on the height of the trees) the equivalence of the two semantic specifications.

An alternative to this proof is to first define an "intermediate" dynamic semantics. This specification would use the abstract syntax of ds_1, but it would use implication introduction corresponding to the extension of contexts of ds_2. Thus the relationship between ds_1 and ds_2 could be explained by examining this intermediate specification that shares some of the features of them both.

6.3 Correctness of Mixed Evaluation Semantics

Since we constructed the mixed evaluation semantics by extending the evaluation semantics[1] of PCF_0 we are able to express and prove the correctness of the mixed evaluation directly in terms of the evaluation semantics. Here we make further use of the logical relations as we shall specify the correctness conditions in terms of the relations SE and MIX.

Now for the correctness of our mixed evaluation semantics we need to show that the MIX relation preserves the standard evaluation semantics. This is given by the following theorem.

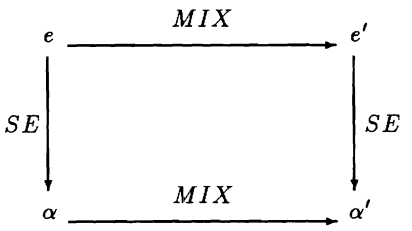
THEOREM 6.9 (Correctness of Mixed Evaluation Semantics) For all $e, e' \in PCF_0$, if $MIX(e, e')$ then the following hold:

(i) for all values α if $SE(e, \alpha)$ then there exists some value α' such that $MIX(\alpha, \alpha')$ and $SE(e', \alpha')$;

(ii) for all values α' if $SE(e', \alpha')$ then there exists some value α such that $MIX(\alpha', \alpha)$ and $SE(e, \alpha)$.

[1] Unless otherwise labeled the term "evaluation semantics" refers to the standard evaluation semantics.

Graphically, this relation among terms is depicted by the diagram:



The following lemma will help us in proving Theorem 6.9.

LEMMA 6.10

(i) for all constants $c \in PCF_0$, $SE(c,e)$ implies $c = e$ and $MIX(c,e)$ implies $c = e$.

(ii) for all terms (lamb M) $\in PCF_0$, $SE((\text{lamb } M),e)$ implies $e = (\text{lamb } M)$ and $MIX((\text{lamb } M),e)$ implies $e = (\text{lamb } M')$ for some M'.

The proof is trivial from the construction of SE and MIX.

PROOF. (of Theorem 6.9) We consider only the proof of (i), with the proof of (ii) proceeding similarly. For some $e, e', \alpha \in PCF_0$, we assume $MIX(e,e')$ and $SE(e,\alpha)$. We must show that there exists some α' such that $SE(e',\alpha')$ and $MIX(\alpha,\alpha')$. Recall that $SE(e',\alpha')$ iff $\vdash e \xrightarrow{se} \alpha$. The proof proceeds by induction on the height h of the proof tree Ξ of $e \xrightarrow{se} \alpha$. (The proof cannot proceed by induction on the structure of terms because certain steps in evaluating terms actually increase the size of terms.)

base: $h = 1$. Two cases apply:

(i) $e = c$ for some constant c. Then by Lemma 6.10, $\alpha = e' = \alpha' = c$.

(ii) $e = (\text{lamb } M)$ for some M. Then by Lemma 6.10, $\alpha = (\text{lamb } M)$ and $e' = \alpha' = (\text{lamb } M')$ for some M' and trivially, $SE((\text{lamb } M'),(\text{lamb } M'))$.

step: $h > 1$.

(i) $e = (\text{if } e_1\ e_2\ e_3)$. There are 6 possible sub-cases based on the structure of e' and α. We shall consider just one sub-case, with the others following similarly.

Assume $e' = (\text{if } e_1'\ e_2'\ e_3')$ such that $MIX(e_1,e_1')$, $MIX(e_2,e_2')$ and $MIX(e_3,e_3')$. Also assume the last inference rule of Ξ is of the form

$$\frac{e_1 \xrightarrow{se} \text{true} \qquad e_2 \xrightarrow{se} \alpha_2}{(\text{if } e_1'\ e_2'\ e_3') \xrightarrow{se} \alpha_2}.$$

By inductive hypothesis, there exists some α_1' such that $DS(e_1',\alpha_1')$ and $MIX(\alpha_1,\alpha_1')$, and by Lemma 6.10, $\alpha_1' = \text{true}$; also (by inductive hypothesis) there exists some

α_2' such that $SE(e_2', \alpha_2')$ and $MIX(\alpha_2, \alpha_2')$. But $SE(e_1', \textsf{true})$ iff $\vdash e_1' \xrightarrow{se} \textsf{true}$ and $SE(e_2', \alpha_2')$ iff $\vdash e_2' \xrightarrow{se} \alpha_2'$. Then we must also have $\vdash (\textsf{if } e_1' \ e_2' \ e_3') \xrightarrow{se} \alpha_2'$ and hence also $DS((\textsf{if } e_1' \ e_2' \ e_3'), \alpha_2')$.

The other 5 cases proceed similarly.

(*ii*) $e = (e_1 @ e_2)$. There are two cases to consider based on the structure of e'. We shall consider just one sub-case, with the other following similarly.

Assume $e' = (e_1' @ e_2')$ such that $MIX(e_1, e_1')$ and $MIX(e_2, e_2')$. Also assume the last inference rule of Ξ is of the form

$$\frac{e_1 \xrightarrow{se} (\textsf{lamb } M) \qquad e_2 \xrightarrow{se} \alpha_2 \qquad (M \alpha_2) \xrightarrow{se} \alpha}{(e_1 @ e_2) \xrightarrow{se} \alpha}.$$

By inductive hypothesis, there exists some α_1' such that $SE(e_1', \alpha_1')$ and $MIX((\textsf{lamb } M), \alpha_1')$, and by Lemma 6.10, $\alpha_1' = (\textsf{lamb } M')$ for some M'. Likewise, there exists some α_2' such that $SE(e_2', \alpha_2')$ and $MIX(\alpha_2, \alpha_2')$. But then we have $\vdash e_1' \xrightarrow{se} (\textsf{lamb } M')$ and $\vdash e_2' \xrightarrow{se} \alpha_2'$. Thus we can construct a proof Ξ' whose last inference rule is

$$\frac{e_1' \xrightarrow{se} (\textsf{lamb } M') \qquad e_2' \xrightarrow{se} \alpha_2' \qquad (M' \alpha_2') \xrightarrow{se} \alpha'}{(e_1' @ e_2') \xrightarrow{se} \alpha'}.$$

We now just need to show $MIX(\alpha, \alpha')$. From $MIX((\textsf{lamb } M), (\textsf{lamb } M'))$ and construction of MIX we have $MIX(M, M')$. But then for α_2, α_2' such that $MIX(\alpha_2, \alpha_2')$ we have $MIX((M \, \alpha_2), (M' \, \alpha_2'))$. And by inductive hypothesis, if $SE((M \, \alpha_2), \alpha)$ then there exists some β such that $SE((M' \, \alpha_2'), \beta)$ and $MIX(\alpha, \beta)$. But $SE((M' \, \alpha_2'), \alpha')$ and SE is functional; so we must have $MIX(\alpha, \alpha')$.

The cases for **let** and **fix** expressions follow similarly and are not presented here. □

The use of the MIX relation here instead of the provability in terms of the mix proof system simplifies the proof because MIX is defined at higher types while the proof system only treats terms of base type. This fact is important because we need to relate two terms M, M' (of type $(tm \rightarrow tm)$ in order to related the two terms (**lamb** M) and (**lamb** M').

We can now simply provide the following corollary that gives us the result we want (correctness of the mix proof system):

COROLLARY 6.11 For all $e, e' \in PCF_0$, if $\vdash e \xrightarrow{mix} e'$ then the following hold:

(*i*) for all values α if $\vdash e \xrightarrow{se} \alpha$then there exists some value α' such that $\vdash \alpha \xrightarrow{mix} \alpha'$ and $\vdash e' \xrightarrow{se} \alpha'$;

(*ii*) for all values α' if $\vdash e' \xrightarrow{se} \alpha'$ then there exists some value α such that $\vdash \alpha' \xrightarrow{mix} \alpha$ and $\vdash e \xrightarrow{se} \alpha$.

The proof is trivial from Theorem 6.9, Theorem 5.7 and the construction of SE.

REMARK. The discussion of correctness here is greatly simplified by two features of our standard evaluation and mixed evaluation semantics:

(*i*) The lack of explicit environments in our specification reduces the overall complexity of our arguments. The actual names of bound variables have no importance in discussing the equivalence of programs and in our specification we never refer to them explicitly.

(*ii*) As discussed previously, the values for our evaluation semantics are a subset of the language. Thus we can manipulate these values just as regular expressions in the language. For specifications that include, for example, closures as values, such uniform treatment is not possible.

The treatment of functional type expressions in both the definition of MIX and the specification of the mix proof system highlights the suitability of our meta-language.

6.4 General Remarks on Correctness Results

One argument for our particular choice of meta-language or proof system is the ability to specify and reason naturally about a variety of programming tasks. While our methods may not provide a panacea, we believe this argument to be correct and one aspect of our research efforts is the identification and elaboration of those tasks most amenable to our methods. In this chapter we have presented proofs and outlines of proofs for the correctness of a number of our specifications. A few salient features of our methods were highlighted by this work and are discussed below.

We noted that our specifications are generally given at a "higher level" than in many other approaches. For example, our standard evaluation semantics for PCF_0 did not require explicit manipulation of contexts. This feature can actually make certain kinds of reasoning, like the proof of translation from PCF_0 to CAM, more difficult. The reason for this can be explained in terms of levels of abstraction. The CAM is a low level machine that explicitly manipulates a stack or environment of values. Thus an evaluation semantics for PCF_0 that uses a context to maintain identifier information is, in some sense, "closer" in spirit to the CAM than a semantics such as our own. In reasoning about a translation from PCF_0 to CAM one must make a correspondence between identifiers in PCF_0 and environments in

CAM. With the lower-level evaluation semantics, this correspondence is just a translation between contexts and environments. However, with our higher-level semantics, the correspondence is not as obvious and one must manufacture some artificial structure to refer to identifiers in our evaluation semantics. From this we conclude that use of our higher-level methods do not necessarily serve to increase our understanding of low-level tasks.

By far the simplest proof we discuss is the one for our mixed evaluation semantics and, as mentioned above, this fact stems from a number of features of our specifications. Foremost is the fact that the correctness of our mixed semantics can be defined and proved in terms of our standard evaluation semantics. And as pointed out above, this is in part due to our use of values that are also expressions in PCF_0. Here we have a significant simplification over the approach given in [5] and this is a direct consequence of our higher-level treatment of identifiers. Thus in this application, our methods have simplified the meta-theoretic discussions. Making a generalization, we claim that meta-programming tasks that operate at a high level of abstraction are more suitable to our techniques than those at a low level. Thus, for example, we should not expect our methods to yield any startling results when applied to such tasks as peephole optimization or (low-level) abstract machine specification. This kind of characterization is important because it gives us direction for our future work.

7

Related Work

7.1 Structural Operational Semantics

The seminal work on a structured approach to operational semantics is by Plotkin [52]. This work introduced the general approach of describing semantics with inference rules.

The focus of this particular work was to present and natural and complete description of the static and dynamic semantics of a simple imperative programming language. While inference rules were introduced, they were presented as a method for specifying a "reduction machine" for the language, as opposed to our approach of describing a proof systems. For example, we view the statement "$E \longrightarrow V$" as a proposition denoting the *property* "expression E evaluates to value V." Plotkin views this statement as a *reduction* and describes a machine, not unlike Landin's SECD machine [33], for defining the reduction process for expressions. Another difference between the present proposal and Plotkin's work is the notion of a formal meta-language. Plotkin purposely remains at an informal level, in an effort to produce the most natural and perspicuous descriptions for the dynamic semantics. He was not specifically concerned with defining a system that could be directly implemented, nor was he concerned with a meta-language for defining a wide variety of meta-programs. These, of course, are principal concerns in this proposal.

7.2 Natural Semantics and TYPOL

Much of the work reported in this proposal was motivated by the research in natural semantics done by G. Kahn and his group at INRIA. They chose to study semantics of programming languages by developing proof systems similar to the ones we develop in this paper. Unlike Plotkin, however, they do define a formal language, Typol, in which they can specify inference rules. A crucial difference (from our approach) is that they view programs as first-order structures which can be manipulated by a first-order language (*e.g.*, PROLOG). While the use of a first-order language may lead directly to efficient implementations, the logical aspects of program systems are not always elucidated in a strictly first-order setting.

In natural semantics, like our own system, a semantic definition is given by a list of axioms and inference rules, given in the language Typol, that define a predicate of the form "$\rho \vdash E \triangleright V$." An informal reading of this predicate is "in context ρ, expression E has value, or has semantic meaning, equal to V." Then reasoning about such predicates is achieved by a restricted form of theorem proving in the logic of Typol. Computations such as type inferencing and evaluation can be viewed as a process of solving equations. For example, we might wish consider the predicate $\rho \vdash E : \tau$ for type inferencing (expression E has type τ in context ρ) in which E is instantiated to a closed term denoting a program expression and τ is a free variable. Computing a value for τ, *i.e.*, finding the type of E, is achieved by constructing a proof of this predicate. During this process we expect that the variable τ will become instantiated.

As suggested in [31] this formulation of reasoning about programs produces a variety of possibilities. For example, other kinds of equations could be of interest. We may as the question "Given some E and τ, does there exist some ρ such that $\rho \vdash E : \tau$?" This is a type inference problem of slightly different nature. Another point is that this presentation is relational (*e.g.*, the relation among some ρ, E and τ) rather than functional. Therefore, one should expect that non-determinism and overloading can be specified naturally.

7.3 Denotational Semantics

Other efforts to provide a flexible meta-language have considered denotational instead of operational or natural semantics. Typically, work has focused on generating evaluators or compilers (*i.e.*, a compiler-compiler) based on a denotational definition of an input programming language. We shall characterize this work using denotational semantics into three classes, based on the type of code that the resulting systems generate.

Direct Evaluation. The first type is really just an evaluator for denotational definitions. The semantic notation for a program is treated as a machine language and an evaluator for this language is implemented. Thus the denotational equations translate a program to its denotation and the evaluator applies simplification rules to the denotation until all possible simplifications are performed. Some of the earliest work in this area is due to Mosses and his SIS system [42]. SIS is a compiler generator which takes as input a specification of the denotational semantics of an object language and produces a compiler for this language.

Combinators. The second class of systems based on denotational definitions is based on combinators. A combinator is simply a λ-term that has no free variables. A combinator expression is any term constructed solely from combinators. Typically, a combinator is given a name and that name is used in place of the expression. Also, derived rewrite rules are supplied for manipulating combinators. For example, we might use the name I for the expression $\lambda f.f$ with the derived rewrite rule $IE \Rightarrow E$ in which E is any combinator expression. As this example suggests, using combinator expressions eliminates the explicit use of bound variables With combinators, one gives a denotational semantics that translates a language into combinator expressions. Then these expressions are evaluated according to a set of rewrite rules. An example of this is a technique for specifying the semantics of an applicative language, such as LISP, as described by Turner [58, 57]. Turner has shown how, by using combinators, expressions written in an applicative language, such as LISP, can be translated into a form that contains no bound variables.

Transformations. The third class takes a denotational definition and transforms it into a language with a known semantics (*i.e.*, its semantics has already been defined). Thus only the transformation rules need to be checked for correctness. Via these transformations, a language can be defined in terms of another. The approach using combinators can actually be viewed as an instance of this class, with a strong restriction on the type of target language.

There have been numerous other efforts with similar goals and their contributions are well documented in the literature [19, 30, 55]. While this abundance of work has produced some fruitful results certain limitations appear inherent with this approach. First the mathematical machinery required to specify a denotational semantics can become burdensome for practical language definitions. Furthermore, denotational semantics, in general, does not appear to be a convenient technique for specifying parallelism or nondeterminism. The techniques described in this proposal do not seem to suffer from these deficiencies. Furthermore, denotational semantics seems best suited only to describing the dynamic (standard-) semantics of programming language. The ability to specify naturally a wide variety of meta-programming tasks, which is certainly a priority of ours, does not appear to be within the limits of a denotational approach.

7.4 Attribute Grammars.

In [32] Knuth introduced attribute grammars as a tool for defining the semantics of context free languages. An attribute grammar (AG) is an ordinary context-free grammar augmented with *attributes* and *semantics functions*. Attribute grammars enable information to be passed down a parse tree as "inherited" attributes and information to be passed up the tree as "synthesized" attributes. This additional capability permits the specification of context sensitive grammars as well as the specification of a semantics for CFLs. More recently, extended attribute grammars (EAG) have been introduced as a more flexible tool for defining CFLs. With respect to the current work we can think of attributes as specifying program properties, such as the type of an expression. The relationship between attribute grammars and our work can be understood in terms of the relationship between the former and logic programming [10].

The basic idea of AGs is to associate, with each symbol of a CFG, a fixed number of *attributes*, each with a fixed domain. These attributes are used to convey information to and from other parts of the parse tree. Inherited attributes will convey information about a phrase's context and synthesized attributes will convey information about the phrase itself. Attributes are given names and, by convention, inherited attributes are prefixed by downward arrows ($\downarrow$) and synthesized attributes by upwards arrows ($\uparrow$). The production rules of an AG are like those of a CFG, but the symbols appearing in the rules (nonterminals and terminals) are given with their associated attributes. The attributes are used to specify context-sensitive constraints on a language with a context-free structure. Each AG rule is basically a context-free rule augmented by (*i*) *evaluation rules*, specifying the evaluation of certain attributes in of other attributes, and (*ii*) *constraints* which must be satisfied by the attributes in each application of this rule.

While the flexibility of attribute grammars has been sufficiently demonstrated (Algol 68 has been defined by an attribute grammar), their suitability as a general-purpose meta-language is severely restricted by their awkward syntax. Using attribute grammars to define even simple constructs often produces obscure descriptions with no intuitive explanation. Furthermore, attribute grammars suffer from some drawbacks including a lack of modularity and modifiability and an inability to express naturally a wide variety of meta-programming tasks.

7.5 PSP

Combining elements of both denotational definitions and attribute grammars, Paulson implemented a compiler generator for semantic grammars [46]. This implementation, named PSP (Paulson's Semantic Processor) by others, uses denotational definitions written in the form of attribute grammars. A semantic grammar includes function and type definitions and attribute grammar rules specifying syntax, static semantics and dynamic semantics. PSP consists of three programs: (*i*) the *Grammar Analyzer* which converts a semantic grammar for a language $\mathcal{L}$ into tables for parsing and compiling $\mathcal{L}$-programs; (*ii*) the *Universal Translator* which reads the tables for $\mathcal{L}$ and then translates $\mathcal{L}$-programs into SECD code [33]; and (*iii*) the *Stack Machine* which optimizes and interprets SECD instructions, executing user programs written in $\mathcal{L}$.

A semantic grammar is an attribute grammar that uses attributes for specifying the static and dynamic semantics of a language. It also includes the definitions of types, functions and structured constants that appear in the semantics. These definitions may be viewed as one or more functional programs. Consider a simple language $\mathcal{L}_0$ that, among other features, contains the syntactic category *iexp* for integer expressions. A semantic grammar for $\mathcal{L}_0$ might include the attribute declaration

attribute *exp*<**synth** *exp*→*integer*>

which indicates that the non-terminal *exp* has a synthesized attribute of type (*exp*→*integer*). The semantic grammar rules for expressions give the grammar symbol *exp* a synthesized attribute E, written inside angle brackets, for the dynamic semantics of expressions. For example we might have the rules

exp<E> = "(" *exp*<E> ")"
exp<*lookup ident*> = *var*<*ident*>
exp< $\lambda s.int$ > = *number*<*int*>
exp< $\lambda s.(E_1\ s) + (E_2\ s)$ > = *exp*< E_1 > "+" *exp*< E_2 >

In this simple example the synthesized attribute represents the denotation of the expression generated from the symbol *exp*. In a larger example, each expression would also have an inherited attribute containing the current declaration table and another synthesized attribute for returning the expression's type.

7.6 Higher-order Reasoning

From the perspective of reasoning in a higher-order setting this work shares much with several other projects. In [29, 23, 38] the authors argue that higher-order unification and logic programming can elegantly be used to manipulate programs in semantically meaningful ways. In [12] a logic programming language containing not only higher-order terms but also the ability to introduce and discharge assumptions and parameters is used to specify and implement various natural deduction-style theorem provers. Many techniques from that paper find immediate applications in this paper. The meta-language(s) presented in this proposal is essentially an application of the general notion of *higher-order abstract syntax* to a particular program manipulation system [48]. This meta-language can also be specified in the much richer proof system specification language of LF [24]. Although we outline briefly how this meta-language can be implemented in the λProlog logic programming language [12, 37, 45], it should also be possible to provide an immediate implementation in the theorem proving system Isabelle [47].

Our abstract syntax for $\mathbf{PCF}_0$ programs involves the use of higher-order terms (specifically, order 2). A discussion of the advantages to using such an abstract syntax in presented in [48]. As argued there, higher-order abstract syntax provides a uniform and language generic method for treating name-binding information in environments that manipulate syntactic objects. Once binding constructs are defined for a given language, the information is explicit in the higher-order abstract syntax. And as we have argued in the present paper, such a syntax, as part of a programming logic, affords concise specifications for typical programming tools.

7.7 Constructive Type Theory

In the current proposal we make essential use of λ-terms as part of a proof theory. Another avenue of research, constructive type theory, establishes an even more intimate relationship between λ-terms and proof theory. In this work judgments of a particular kind are derived from a simple set of inference rules. The judgments are typically of the form $p{:}P$ in which P is a proposition (*e.g.*, describing a program property) and p is a proof of P. The key observation made by Curry and later by Howard is the dual reading of judgments of this form. The judgment $p{:}P$ can also be interpreted as "program p has type P." And from this observation came the notion of "formulas as types" and the Curry-Howard isomorphism. Historically, in [8] the authors noted that the types of the combinators $\boldsymbol{K}_{\sigma\tau}$ and $\boldsymbol{S}_{\sigma\tau\rho}$ are

valid formulas of propositional logic if $\rightarrow$ is interpreted as implication. In 1960 Howard extended these results to show that given any type expression σ, the proposition denoted by σ (*i.e.*, interpreting $\rightarrow$ as implication) is valid iff there exists a closed λ-term of type σ [28].

Exploiting this correspondence between λ-terms and proof theory, Martin Löf developed a constructive type theory extending the results of Curry and Howard. He introduced quantified formulas to his system and was motivated, in part, to develop a formal basis for a programming language. In such a language, one would write programs by constructing proofs of a certain kind. Obtaining a proof p of a proposition P, one would be certain that the "program" p was well-typed and, in particular, of "type" P.

This connection between λ-terms and proofs is much stronger than that provided in our setting. Currently, we do not exploit the Curry-Howard isomorphism, though in fact our proofs could be constructed and represented as λ-terms, and perhaps manipulated or analyzed for some purpose. See Chapter 8 for more discussion of this.

8

Summary and Future Work

8.1 Summary

In this proposal we presented proof-theoretic methods, based on a natural deduction paradigm, for analyzing and manipulating functional programming languages. Using a higher-order, intuitionistic meta-logic we encoded axioms and inference rules as clauses in this logic. The expressive power of this logic provided us the ability to specify a wide variety of program manipulation tasks (*e.g.*, type inferencing, interpretation and compilation) as proof systems. We were then free to perform meta-theoretic analyses of these proof systems, using existing methods from proof theory.

While we expect that "direct" correctness proofs for our specifications should be simpler and perhaps contain new insights (owing to the strength of the meta-logic involved), the proofs presented in the proposal were based on two techniques. First, we argued a form of correctness for our dynamic semantics (an indirectly, compilation semantics) be giving a correspondence between proofs in our system and proofs in an existing system for which correctness is known. The key feature of this correspondence is the stratification of, for example, various dynamic semantics, into levels according to the level of abstraction present in the object language. Since compilation is actually the definition of a (one-way) correspondence between two dynamic semantics and we expect these relationships to be transitive, we can related two dynamic semantics in non-adjacent strata given only the correspondence between adjacent strata.

For the mixed evaluation semantics we exploited the relationship between this seman-

tics and the dynamic semantics to provide a notion of correctness. We take as correctness a general statement about the family of proofs (in a dynamic semantics) for two expressions related via the mixed evaluation semantics. This technique provides a natural way of expressing correctness and reasoning about it. We observed that this work provides a framework for presenting partial evaluation (also called mixed computation) in a natural and general setting with the ability to treat formally concepts of correctness and derivation (of partial evaluators). These results stem from the logical foundation of our proof systems treated as meta-languages.

8.2 Future Work

Our future work in this area has several directions and each is described below.

8.2.1 Richer Languages and Analyses

We would like to consider richer functional programming languages, *e.g.*, those including exceptions, modules, richer data types, *etc.*, and the specification of their various semantics (dynamic, static, *etc.*) using similar proof-theoretic techniques. As the object language becomes richer we wish to determine whether the meta-logic specified in this proposal is strong enough to express naturally these specifications. If they are not, then we would like to propose explanations for this inadequacy and suggest extensions to the logic.

We wish to apply the proof methods proposed herein to other types of program analyses. Recall that one of our research goals was to demonstrate how a wide variety of program analyses/manipulations could be cast into the unified framework of proof theory. To support this claim we suggest the following tasks for possible consideration:

- *Partial Evaluation.* We gave a preliminary discussion of the generalized notion of mixed evaluation semantics. We would like to consider the current state of research in partial evaluation and determine whether our methods are suitable for representing and addressing important issues. One immediate possibility is extending our mixed evaluation semantics to the language PCF_1 by following the methods discussed in [18]. We believe that the our meta-language is an ideal implementation language for this task.
- *Program Transformation.* We presented a restricted kind of program transformation in our mixed evaluation semantics. Other, more general kinds of transformations (*e.g.*, tail recursion elimination), could be expressed using a similar style of proof system.
- *Flow Analysis.* We gave an example of flow analysis by showing how strictness anal-

ysis could be specified in our setting. In general, flow analysis is aimed at providing information about the "flow" of information or values during some computation. More general types of flow analysis provide information that can be exploited by compilers.

- *Complexity Analysis.* To date little work has been in automating the analysis of program complexity. The most significant work in this area has been by Le Métayer [35].

This list is not meant to be exhaustive, but only to give an idea of the breadth of analysis tools we believe to be amenable to our methods.

8.2.2 Manipulating Proof Systems

In several parts of this paper we informally derived a new proof system from an existing one. Recall the section in which we discussed strictness analysis. We demonstrated how a non-standard (dynamic) semantics for PCF_0 could be constructed to provide strictness information. We constructed this proof system informally but noted its similarity to the original dynamic semantics of PCF_0. We also derived the mixed evaluation semantics from the dynamic semantics by making certain observations. Let us then reinterpret these non-standard proof systems as the result of some transformation on the original proof systems. If we could formalize this transformation in terms of a meta-language then we could formally derive these non-standard semantics (for strictness and other types of analysis) of programming languages. The advantage here is the obvious connection to the standard semantics which will assist in correctness proofs.

In Section 3.1 we argued that the representation of (object) programs as simply typed λ-terms is highly suitable for treating programs as objects. If we reflect this principle back onto our meta-language, we see that our proof systems, as a collection of axioms and inference rules, can actually be represented as simply typed λ-terms themselves. For example, consider the proof system for the dynamic semantics of PCF_0. Let the two-place function symbol $(\cdot \xrightarrow{se} \cdot)$ be a meta-term of meta-type $tm \to tm \to o$ where o is the type of propositions. Also, let the inference rule constructor (the horizontal bar) be a term for constructing proof-terms from propositions and other proof-terms. Using such a formulation, the axioms, inference rules and proofs are all just meta-terms and thus we can consider performing various types of analyses and manipulations on them.

The extent to which such forms of analyses and manipulations will prove useful is an open question, but we have already provided two illustrative example that suggests certain possibilities. This framework for generating new kinds of semantics from existing ones is an exciting possibility.

8.2.3 Relating Proof Systems

One point about our proof methods that we have yet to argue is the uniform specification of a wide variety of analysis and manipulation tools. Tasks from type inference to compilation to flow analysis can all be specified in a similar framework, using the same meta-language. The immediate advantage of having such a unified framework is the ability to relate different proof systems, comparing and contrasting them. Two basic kinds of relations are of interest. The first is relating two proof systems that provide the same kind of information but for different object languages. For example, we might compare the dynamic semantics for two different programming languages $\mathcal{P}$ and $\mathcal{Q}$, where $\mathcal{P}$ and $\mathcal{Q}$ may be totally unrelated (*e.g.*, a functional language and an imperative language) or one may be a subset of the other.

A second and more interesting kind of relation is between two proof systems that provide different semantics for the same language. (This is related to the idea discussed above.) Consider, for example, the language PCF_0 and its proof systems for type inference and compilation. Typical use of these tools might be first to perform type inference, to see that the program is well typed, and second to perform the compilation. But now suppose we wish to combine these two phases into a single phase. With our proof methods, this process amounts to merging the two proof systems into a single one. Now because of the uniform framework in which both of these systems have been developed, we might very well hope to develop formal methods for performing this merging process.

Combining proof systems is a specialization of the previous section in which we proposed general manipulation and analysis of inference rules. It deserves particular attention because, as the example above suggests, important derivations can be formalized with such techniques.

8.2.4 Control Issues

For all the proof systems presented in this work we have tacitly made a distinction between specification and implementation. We see this distinction as falling along the same lines as the declarative and procedural semantics of logic programming. This similarity is not surprising given the close correlation between our proof systems (or natural deduction) and logic programming that we described in the introduction.

The proof systems described in this work could have similar declarative and procedural perspectives, though we refer to them as specifications and implementations. We have been careful to describe our systems as *specifications* of certain program properties or semantics, rather than an implementation. The reason for this is simple: Implementations require a

notion of control which, in our setting, would include an ordering of the axioms and inference rules of a given proof system and also an ordering of the antecedents of inference rules. For example, in the proof system giving the dynamic semantics for PCF_0 we have the two rules

$$\frac{e_1 \xrightarrow{se} \textsf{true} \qquad e_2 \xrightarrow{se} \alpha}{(\textsf{if } e_1\ e_2\ e_3) \xrightarrow{se} \alpha} \qquad \frac{e_1 \xrightarrow{se} \textsf{false} \qquad e_3 \xrightarrow{se} \alpha}{(\textsf{if } e_1\ e_2\ e_3) \xrightarrow{se} \alpha}.$$

Assume we have an expression (if e_1 e_2 e_3) such that e_1 evaluates to **false**, e_2 diverges (as described in Section 5.1), and e_3 evaluates to 3. Now if the first of these two clauses is "tried" first during the search for a proof of (if e_1 e_2 e_3)$\xrightarrow{se}\alpha$ and if we arbitrarily choose to evaluate e_2 before e_1 (there is no specific order to the antecedents), then no proof will ever be found. From this simple example we see that our dynamic semantics, in its current incarnation, is *not* an interpreter for PCF_0. Rather, it is a specification of *valid* computations. More specifically, proof trees for dynamic semantics describe a relation between expressions and values. Constructing a proof of the expression (if e_1 e_2 e_3)$\xrightarrow{se}\alpha$ indicates the computations necessary to evaluate the conditional expression. When we say that we have *implemented* this proof system in the programming language λProlog what we have essentially done is added control information, or a procedural interpretation, to the clauses representing the axioms and inference rules. Unfortunately, this control is not flexible. (It provides a depth-first search, with a left-to-right ordering of atomic formulae in the antecedent.) While this control regime provides a sufficient structure for producing an interpreter for PCF_0, it prohibits more flexible strategies that might prove useful for manipulating other (object) programming languages.

One obvious example is a programming language with nondeterministic operations. To provide a dynamic semantics to such a language we would require a meta-language that could capture some notion of nondeterminism. From a specification perspective, we would expect a proof system (specifying the dynamic semantics) to have different properties than the one for PCF_0. First, we might expect there to be more than just one proof for a given proposition $e \longrightarrow \alpha$. This fact suggests that the proof system would not be "deterministic," in the sense that the dynamic semantics for PCF_0 is. Second, for a given expression e in this nondeterministic language we may have some distinct α_1 and α_2 such that $e \longrightarrow \alpha_1$ and $e \longrightarrow \alpha_2$. Thus as part of our future work we should investigate how to make control information a part of a specification.

Bibliography

[1] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Volume 103 of *Studies in Logic and the Foundations of Mathematics*, North-Holland, 1981.

[2] P. Borras *et. al. CENTAUR: the System.* Technical Report 777, INRIA, December 1987.

[3] N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser Theorem. *Indag. Math.*, 34(5):381–392, 1972.

[4] C. Clack and S. Peyton Jones. Strictness analysis – a practical approach. In *Functional Programming Languages and Computer Architecture*, pages 35–49, Springer-Verlag LNCS, Vol. 201, 1985.

[5] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: mini-ML. In *Proceedings of the ACM Lisp and Functional Programming Conference*, pages 13–27, 1986.

[6] D. Clément, J. Despeyroux, L. Hascoët, and G. Kahn. *Natural Semantics on the Computer.* Research Report 416, INRIA, June 1985.

[7] G. Cousineau, P-L. Curien, and M. Mauny. The categorical abstract machine. *The Science of Programming*, 8(2):173–202, 1987.

[8] H. B. Curry and R. Feys. *Combinatory Logic, Volume I.* North-Holland, 1958.

[9] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proceedings of the ACM Conference on Principles of Programming Languages*, pages 207–212, 1982.

[10] P. Deransart and J. Małuszyński. Relating logic programs and attribute grammars. *Journal of Logic Programming*, 2(2):119–155, 1985.

[11] J. Despeyroux. Proof of translation in natural semantics. In *Proceedings of the First ACM Conference on Logic in Computer Science*, pages 193–205, 1986.

[12] A. Felty and D. Miller. Specifying theorem provers in a higher-order logic programming language. In *Proceedings of the Ninth International Conference on Automated Deduction*, 1988.

[13] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Computer, Systems, Controls*, 2(5):45–50, 1971.

[14] Y. Futamura. Program evaluation and generalized partial computation. In *Proceedings of International Conference on Fifth Generation Computer Systems*, 1988.

[15] Y. Futamura and K. Nogi. Generalized partial computation. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Proceedings of the IFIP Workshop on Partial Evaluation and Mixed Computation*, North-Holland, 1987.

[16] J. Gallier. *Logic for Computer Science.* Harper & Row, 1986.

[17] G. Gentzen. Investigations into logical deduction. In M. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland Publishing Co., 1969.

[18] F. Giannotti, A. Matteucci, D. Pedreschi, and F. Turini. Symbolic evaluation with structural recursive symbolic constants. *Science of Computer Programming*, 9(2):161–177, 1987.

[19] M. J. C. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, 1979.

[20] J. Hannan. *Abstract Interpretation: Theory and Practice.* Area Exam Paper, University of Pennsylvania, October 1987.

[21] J. Hannan and D. Miller. Enriching a meta-language with higher-order features. In *Workshop on Meta-Programming in Logic Programming*, Bristol, June 1988.

[22] J. Hannan and D. Miller. A proof-theoretic description of the evaluation of data type definitions. 1988. (submitted).

[23] J. Hannan and D. Miller. Uses of higher-order unification for implementing program transformers. In K. Bowen and R. Kowalski, editors, *Fifth International Conference and Symposium on Logic Programming*, MIT Press, 1988.

[24] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204, 1987.

[25] L. Hascoët. Partial evaluation with inference rules. *New Generation Computing*, 6(2–3):187–209, 1988.

[26] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-calculus.* Cambridge University Press, 1986.

[27] R. Hindley. The completeness theorem for typing λ-terms. *Theoretical Computer Science*, 22(1–2):1–18, 1983.

[28] W. Howard. The formulas-as-types notion of construction. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490, Academic Press, 1980.

[29] G. Huet and B. Lang. Proving and applying program transformations expressed with second-order logic. *Acta Informatica*, 11:31–55, 1978.

[30] N. Jones, editor. *Semantics-Directed Compiler Generation.* Volume 94 of *Lecture Notes in Computer Science*, Springer-Verlag, 1980.

[31] G. Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39, Springer-Verlag LNCS, Vol. 247, 1987.

[32] D. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[33] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(5):308–320, 1964.

[34] M. Marcotty, H. Ledgard, and G. Bochmann. A sampler of formal definitions. *Computing Surveys*, 8(2):191–276, 1976.

[35] D. Le Métayer. ACE: an automatic complexity evaluator. *ACM TOPLAS*, 10(2):248–266, 1988.

[36] A. Meyer. What is a model of the lambda calculus? *Information and Control*, 52(1):87–122, 1981.

[37] D. Miller and G. Nadathur. Higher-order logic programming. In *Proceedings of the Third International Logic Programming Conference*, Springer-Verlag, 1986.

[38] D. Miller and G. Nadathur. A logic programming approach to manipulating formulas and programs. In *Proceedings of the IEEE Fourth Symposium on Logic Programming*, IEEE Press, 1987.

[39] D. Miller, G. Nadathur, and A. Scedrov. Hereditary Harrop formulas and uniform proof systems. In *Symposium on Logic in Computer Science*, pages 98–105, ACM Press, 1987.

[40] J. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76:211–249, 1988.

[41] J. Mitchell and B. Harper. The essence of ML. In *Proceedings of the ACM Conference on Principles of Programming Languages*, pages 28–46, 1988.

[42] P. Mosses. *SIS: a Compiler Generator System Using Denotational Semantics.* DAIMI MD-30, Aarhus University, Aarhus, Denmark, August 1979.

[43] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, University of Edinburgh, 1981.

[44] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proc. 4th International Symposium on Programming*, pages 269–281, Springer-Verlag LNCS, Vol. 83, 1980.

[45] G. Nadathur and D. Miller. An overview of λProlog. In K. Bowen and R. Kowalski, editors, *Fifth International Conference and Symposium on Logic Programming*, MIT Press, 1988.

[46] L. Paulson. *A Compiler Generator for Semantic Grammars.* PhD thesis, Stanford University, 1981.

[47] L. Paulson. The foundation of a generic theorem prover. (To appear in the *Journal of Automated Reasoning*).

[48] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation*, 1988.

[49] G. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(1):125–159, 1976.

[50] G. Plotkin. Lambda-definability in the full type hierarchy. In R. Hindley and J. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic*, pages 363–373, Academic Press, 1980.

[51] G. Plotkin. LCF as a programming language. *Theoretical Computer Science*, 5(3):223–257, 1977.

[52] G. Plotkin. *A Structural Approach to Operational Semantics.* DAIMI FN-19, Aarhus University, Aarhus, Denmark, September 1981.

[53] Dag Prawitz. *Natural Deduction.* Almqvist & Wiksell, Uppsala, 1965.

[54] T. Reps. *Generating Language-Based Environments.* MIT Press, 1985.

[55] D. Schmidt. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, 1986.

[56] D. Scott. Relating theories of the λ-calculus. In R. Hindley and J. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic*, pages 403–450, Academic Press, 1980.

[57] D. A. Turner. Another algorithm for bracket abstraction. *Journal of Symbolic Logic*, 44:267–270, 1979.

[58] D. A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9:31–49, 1979.

[59] C. P. Wadsworth. The relation between computational and denotational properties for Scott's D_∞ models of the lambda-calculus. *SIAM Journal of Computing*, 5(3):488–521, 1976.