Adequate Models for Recursive Program Schemes

by

Michael D. Ernst

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1989

Signature of Author ....
Department of Electrical Engineering and Computer Science
May 22, 1989

Certified by ....
Albert R. Meyer
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by ....
Leonard A. Gould
Chairman, Departmental Committee on Undergraduate Theses

Adequate Models for Recursive Program Schemes

by

Michael D. Ernst

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 1989, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Engineering

Abstract

This thesis is a pedagogical exposition of adequacy for recursive program schemes in the monotone frame. Adequacy relates the operational and denotational meanings of a term; it states that for any term of base type, the operational and denotational meanings are identical. Adequacy is typically proved in the continuous frame. This is a pedagogically questionable step; in order to prove adequacy (or some other property) of a pair of semantics, it would be desirable to show the property directly, without introducing superfluous notions. This difficulty is particularly acute for this thesis because, in general, not all monotone functions are continuous.

This thesis attempts to work out the concept of adequacy for a class of monotone first-order recursive program schemes, using Vuillemin and Manna's method of "safe" computation rules. The attempt is very nearly successful, but at a crucial point the fact that the scheme-definable functions are, in fact, continuous as well as monotone must be used.

Thesis Supervisor: Albert R. Meyer
Title: Professor of Computer Science and Engineering

Contents

List of Symbols

Many of the symbols used in this thesis are listed here; some which are used in a standard way (such as "iff" for "if and only if" and $\beta \to \gamma$ for the type of a function from type β to type γ) are not listed. The page is the page on which the symbol is defined or explained, which is usually its first occurrence.

F, 8
$\mathrm{ar}(\mathrm{F})$, 8
f, 8
$\Leftarrow$, 9
R, 9
F_m, 9
$\vec{x}_{\mathrm{ar}(\mathrm{F}_i)}$, 9
F, 9
$\tau_i[F_1, F_2, \ldots, F_n]$, 10
D_{F_i}, 10
D_τ, 10
$F_{1\ldots n}$, 10
τ, 11
$\mathcal{I}, \mathcal{I}_0$, 12
$\Longrightarrow_{\mathcal{I},R}$, 12
$D, D_{\mathcal{I}}$, 12
$\widehat{d}$, 12
$(\cdot)$, 12
$(t_i)^{\vec{s}_k}_{\vec{x}_k}$, 14
$\overset{*}{\Longrightarrow}_{\mathcal{I},R}$, 15
$\mapsto$, 16
$\mathcal{I}_\Omega$, 16
F_Ω, 16
$[\![t]\!]_{\mathrm{op}}$, 16
$\mathcal{M}_{\mathcal{I}}(R)$, 16
$\overset{\mathrm{sub}}{\Longrightarrow}_{\mathcal{I},R}$, 18
$S_{C,R}, S_{C,\mathcal{I},R}$, 20
$C_R, C_{\mathcal{I},R}$, 20
$[\![C_R]\!]$, 21
LI, PI, L, PO, FA, FS, 21
$\sqsubseteq, \sqsubset$, 22
$\bot$, 23
$D_\bot$, 23
$\bot_D$, 23
$\bigsqcup$, 23
$\mathcal{D}, (D_{\mathcal{I}})_\bot$, 24
Ω, 24
$\lfloor D \rfloor$, 24
$|D|$, 24
$H(g, \beta)$, 25
μ, 25
$\mathcal{D}_{F_i}$, 26
$\mathcal{D}_\tau$, 26
$[\![\tau_i]\!]$, 26
$[\![\tau]\!]$, 26
f_R, 27
$\alpha_C\{F^{1\ldots i}, F^{i+1\ldots j}\}$, 28
$S^i_{C,R}$, 28
$l_{1\ldots i}$, 29
$\sqsubseteq_t$, 29
O, 30
$[\![\alpha_C]\!]\{\mathrm{F}^{1\ldots i}, \mathrm{F}^{i+1\ldots j}\}$, 35
f^k, 35
$S_{C',R}, [\![\alpha_{C'}]\!]_{\mathrm{sem}}$, 37

Chapter 1

Introduction

When we write a program or a procedure in any programming language, we usually know what function we intend it to compute on its input. We would also like to know what function it actually computes on its input.[1] The *meaning* of a program is the function that its main procedure denotes.

There are several ways that the meaning of a program can be defined; chief among these are the operational and denotational meanings. The *operational* meaning of a program is the function which is computed if some particular interpreter is run on the program applied to inputs. The *denotational* meaning, on the other hand, is computed in a more mathematical fashion: it is the least fixed point of the defining equations of a program. Thus, the meaning of a program of the form[2]

$$\mathrm{F}(x) \Leftarrow \textit{if} \ \ x \leq 1 \ \textit{then} \ 1 \ \textit{else} \ x \times \mathrm{F}(x-1) \ \textit{fi}$$

can be determined to be the factorial function for nonnegative inputs either by running the program on various inputs (that is, evaluating the term $\mathrm{F}(y)$ for various values of y) or by determining that the least fixed point of F considered as a function over the nonnegative integers is the factorial function. We will not address the issue of which is the right point of view about the meaning of a program or term; either side can be plausibly argued. For "reasonable" (or "correct") interpreters the operational meaning is the same as the denotational meaning for "accurate" models of the functions and function spaces, so there is no need for such disputes. It is the goal of this thesis to determine for a particular, but quite general, class of recursive program schemes when the operational and denotational meanings coincide.

1.1 Why adequacy?

In particular, this thesis is a pedagogical exposition of adequacy for recursive program schemes in the monotone frame. *Adequacy* is a condition on models which relates the operational and denotational meanings of term; it guarantees that for any term of base

[1]The correspondence between the two functions, which matters a great deal to programmers, will not concern us in this thesis.

[2]The syntax of recursive program schemes will be explained in detail in section 2.1, page 8.

type, the operational and denotational meanings are identical. The stronger condition which guarantees equivalence of operational and denotational meanings at any type is known as *full abstraction*. Full abstraction will not be directly considered in this thesis for two reasons. First, full abstraction does not hold in the monotone frame, which is our arena of interest. While both the continuous and monotone semantic spaces are adequate, only the continuous one is fully abstract as well. We will briefly defer discussion of that point in favor of the second: full abstraction does not nontrivially apply in the program schemes that we will consider, because their syntax allows only first-order terms to be constructed. Thus, adequacy is a strong enough condition on these models.

Although they are simple enough to facilitate reasonably easy proofs, the program schemes that this thesis considers are anything but trivial. Their simplicity makes the results stronger since most programming languages are a superset of the one considered here.

In fact, many people are only interested in first-order functions and first-order terms anyway. Most programmers restrict their scope of attention to a subset of the programs which they could be writing. The higher-order procedures that they do use are usually second- or third-order at very most. It is too hard to think about functions of type higher than that anyway.

This argument also helps to explain why the monotone frame was chosen. Most expositions of adequacy consider the continuous frame, which is in many respects easier to deal with. For one thing, the least upper bound (lub) operator can be used to determine least fixed points. This use of continuity, however, is usually poorly-motivated. Why should this new notion be used in proofs when most people are perfectly content to remain in the realm of monotone models? The examples that are used to illustrate the need for adequacy, too, can usually be handled in the monotone frame: they demonstrate a need for adequacy in monotone models, not necessarily in the continuous ones.

This thesis represents an attempt to close that pedagogical gap by presenting a proof of adequacy for monotone models without resorting to continuity. It is the case that the monotone functions we will study also happen to be continuous, so use of continuity would not be wrong; it would just be overstepping the bounds of good taste in exposition, for in general not all monotone functions are continuous. The attempt, unfortunately, fails. With a single exception the proofs and explanations are able to strictly confine themselves to the self-imposed boundaries, but the proof of a crucial theorem fails for infinite values unless the fact of continuity is used. Although the theorem is true for the monotone case, it cannot be proved (by the method chosen in this thesis, at least) without the use of continuity.

1.2 Work of others

It is generally agreed that adequacy is an essential property of a reasonable link between the two semantics; there is less consensus on whether full abstraction and similar properties are so important. Because it is such a basic and important property, adequacy has been often shown for various interpreters (operational semantics) and denotational semantics [Vui73, Man74, Tai75, LS87]; the classic statement is found in [Plo77]. All of these expositions use the continuous model, which we reject for reasons outlined above. The proof methods of these references are, in general, valid even when their proofs do not apply to the monotone case. Because of this fact, the results shown in this thesis in many cases follow from the proofs, though not from the theorems, of other published work. Meyer [Mey88] presents the case for the use of monotone models and questions the gratuitous use of adequacy; it was such issues that inspired this thesis.

Plotkin [Plo77] proves adequacy very elegantly by Tait's method of computability, which requires a proof by structural induction on terms of a property defined by induction on types. This method is much more general than required here (it applies to a wider class of programs), and it demands continuity.

Loeckx and Sieber [LS87] present a fairly straightforward proof of adequacy based on the definitions of substitution and interpretation; however, the proof only works for interpreters which simultaneously evaluate all subexpressions of an expression. This isn't the case for many real interpreters, and we have no wish to so restrict ourselves in this thesis.

The method which seems to have the best promise for extension to the monotone case is that of "safe" computation rules [Man74, Vui73]; this thesis follows that method. A safe computation rule is one for which adequacy is guaranteed to hold; the formal definition will appear later. This proof works only in the presence of first-order terms; no functional abstraction is permitted. However, it is much simpler than the other proofs, and it can be disentangled from the issue of monotonicity vs. continuity.

Chapter 2

Syntax and Operational Semantics

2.1 Syntax

Recursive program schemes will be constructed from the symbols **(**, **)**, $\Leftarrow$, and **,**, a countable set of variables $V = \{x, y, \ldots\}$, and a countable set of function variables $W = \{\mathrm{F}, \mathrm{G}, \ldots\}$. Each function variable F has an associated arity ar(F).

A basis for a language of recursive program schemes is a pair B of sets of function and predicate symbols; these are specified, respectively, as funsym(B) and predsym(B). Each symbol in the basis has a non-negative arity; symbols of arity 0 are constants.

Recursive program schemes are constructed from *extended first-order terms*, henceforth called just *terms*, which are defined by the following BNF grammar. There are two kinds of terms: Boolean terms (b-terms) and domain terms (t-terms). b-terms denote elements of *Bool* and will be used in the construction of conditional (*if-then-else*) expressions; t-terms denote elements of the domain of the program. The domain is for now a set, often the integers. The domain will be more carefully defined on page 12.

$$\begin{array}{lll}
t & ::= x & \text{(variable)} \\
& \mid f(t_1, t_2, \ldots, t_{\mathrm{ar}(f)}) & \text{(function symbol application; } f \in \mathrm{funsym}(B)\text{)} \\
& \mid \mathrm{F}(t_1, t_2, \ldots, t_{\mathrm{ar}(\mathrm{F})}) & \text{(function variable application)} \\
& \mid \textit{if } b \textit{ then } t_1 \textit{ else } t_2 \textit{ fi} & \text{(conditional)} \\
b & ::= \textit{true} \mid \textit{false} & \text{(constants)} \\
& \mid \neg b & \text{(negation)} \\
& \mid b \wedge b & \text{(conjunction)} \\
& \mid t_1 = t_2 & \text{(equality test)} \\
& \mid p(t_1, t_2, \ldots, t_{\mathrm{ar}(p)}) & \text{(predicate symbol application; } p \in \mathrm{predsym}(B)\text{)}
\end{array}$$

Equality connects t-terms to b-terms, and the conditional expression connects b-terms to t-terms. Constants in B are not explicitly mentioned in the grammar because function and predicate constants are just applications of function and predicate symbols to zero arguments. The constants *true* and *false*, on the other hand, are explicitly mentioned because the rules of the operational semantics will depend on them. Of the Boolean connectives, only $\neg$ and $\wedge$ are included since all other connectives are definable from these two.

Definition 1. A *recursive program scheme* R over a basis B is defined to be a set of n *recursive equations* ($n \geq 1$), of the form

$$\begin{array}{rcl} \mathrm{F}_1(x_1, x_2, \ldots, x_{\mathrm{ar}(\mathrm{F}_1)}) & \Leftarrow & t_1 \\ \mathrm{F}_2(x_1, x_2, \ldots, x_{\mathrm{ar}(\mathrm{F}_2)}) & \Leftarrow & t_2 \\ & \vdots & \\ \mathrm{F}_n(x_1, x_2, \ldots, x_{\mathrm{ar}(\mathrm{F}_n)}) & \Leftarrow & t_n \end{array}$$

and a number $m \leq n$. F_m is designated as the *main function variable*; this function variable determines the starting point of the recursive program.

The function variables $\mathrm{F}_1, \mathrm{F}_2, \ldots, \mathrm{F}_n$ must be distinct and are called the function variables *declared* in R. When F is used without a subscript, any F_i can be used for the F; different F_i may be used for different instances or occurrences (we will formally define an *occurrence* in section 11 on page 17) of unsubscripted F which appear in an expression. The t-term t_i in the i^{th} recursive equation is called the *body* of the declaration of F_i; the only variables of base type that may occur in t_i are $\vec{x}_{\mathrm{ar}(F_i)}$ (*i.e.*, $x_1, x_2, \ldots, x_{\mathrm{ar}(F_i)}$). The only function variables allowed in the t_i terms are those declared in R.

Now that the syntax has been laid out, we can talk about the functions to which the recursive equations are intended to correspond. Since all terms are of base type (there is no abstraction mechanism provided in the syntax of terms or of recursive equations), all such functions are first-order.

The conditional operator is unique in the language because it is the only built-in operator which is not strict in all of its arguments: it is strict only in its first argument.[1]

The equality test is just a predicate symbol which is part of every interpretation, and the Boolean operators $\neg$ and $\wedge$ can be thought of as predicate symbols which are applied to Boolean instead of domain terms. Because these are straightforward generalizations, in what follows only function symbols and function variables will be explicitly considered. The comments directed toward function symbols will also apply to predicate symbols and to $=$, $\neg$, and $\wedge$, which will be known as *built-in symbols*. Note that the conditional expression is not a constant like $=$ and $\neg$; rather, it is a phrase constructor. Nevertheless, the comments on built-in constants will often apply; when they don't, *if-then-else* will be treated as a separate case.

F_i is not itself a semantic object or a function that takes terms (or domain elements) to terms (or domain elements); it is simply a piece of notation. $F_i : \mathrm{Term}^{\mathrm{ar}(F_i)} \rightarrow \mathrm{Term}$ is the function which, applied to terms $s_1, s_2, \ldots, s_{\mathrm{ar}(F_i)}$, returns the term $\mathrm{F}_i(s_1, s_2, \ldots, s_{\mathrm{ar}(F_i)})$. (Note the italics.) Such a function will be known as a *term constructor*. It is obvious how to write this function, and since $F_i(s_1, s_2, \ldots, s_{\mathrm{ar}(F_i)})$ is an expression which means (evaluates to) the term $\mathrm{F}_i(s_1, s_2, \ldots, s_{\mathrm{ar}(F_i)})$, this expression (which is a function application) can be used in place of the piece of syntax (the term). This gives us two ways to express a given

[1]Actually, the "conditional operator" is a term constructor, not an operator, but that is just a matter of taste; we could just as easily have defined constants which performed the *if-then-else* operation. The point is that the conditional is different from anything else in the language.

term; we can write the term itself, or we can write an expression which, when evaluated, produces the term. We have introduced a meta-syntactic level in the expression level. There are several other natural functions relating to a program scheme's function variables that we will want to specify. Immediately below we will define a notation for the other syntactic transformation from terms to terms (that is, the transformation from $\mathrm{F}_i(x_1, x_2, \ldots, x_{\mathrm{ar}(F_i)})$ to t_i). The function from domain elements to domain elements that the program calculates will be specified in several ways that will turn out to be identical.

An equivalent form for the equations of a recursive program scheme, which we shall prefer, is:

$$\begin{aligned}
F_1(x_1, x_2, \ldots, x_{\mathrm{ar}(F_1)}) &\Leftarrow \tau_1[F_1, F_2, \ldots, F_n](x_1, x_2, \ldots, x_{\mathrm{ar}(F_1)}) \\
F_2(x_1, x_2, \ldots, x_{\mathrm{ar}(F_2)}) &\Leftarrow \tau_2[F_1, F_2, \ldots, F_n](x_1, x_2, \ldots, x_{\mathrm{ar}(F_2)}) \\
&\vdots \\
F_n(x_1, x_2, \ldots, x_{\mathrm{ar}(F_n)}) &\Leftarrow \tau_n[F_1, F_2, \ldots, F_n](x_1, x_2, \ldots, x_{\mathrm{ar}(F_n)})
\end{aligned}$$

The type of $\tau_i[F_1, F_2, \ldots, F_n]$, like that of F_i, is $\mathrm{Term}^{\mathrm{ar}(F_i)} \to \mathrm{Term}$; this denotes the syntactic operation from $\vec{x}_{\mathrm{ar}(F_i)}$ to t_i suggested by the recursive equation for F_i. F_i takes $\mathrm{ar}(F_i)$ terms as arguments and returns another term which is the result of substituting its arguments into t_i, the body of the i^{th} recursive function definition, in place of the formal parameters of F_i (the x_i's). Providing other arguments to τ will result in different functions on terms.

Enclosing the function variable arguments to τ_i (the F's) in square brackets is a syntactic convention which abbreviates $\tau_i(\langle F_1, F_2, \ldots, F_n\rangle)$; τ_i takes an n-tuple of functions from terms to terms as its arguments and returns the same sort of function as its result. That each τ_i takes only a single argument (not a vector of functions) is required for the construction of the functional τ below. It is required that all functions be called on the proper number of arguments of the proper type; no currying is done, and the results of an improper application are unspecified.

This notation specifies precisely the same recursive program as the previous formulation, but it makes different information explicit. Each recursive program scheme R has a corresponding τ, and vice versa, so a recursive program scheme can be specified by either its R or its τ (along with specification of its main function variable). We will use each convention as convenient.

We will use the abbreviations $D_{F_i} = \mathrm{Term}^{\mathrm{ar}(F_i)} \to \mathrm{Term}$ for the type of F_i, $D_\tau = D_{F_1} \times D_{F_2} \times \ldots \times D_{F_n}$ for the type of the domain of each τ_i, and $F_{1\ldots n}$ for $F_1, F_2, \ldots, F_n$.

Let G_i be of type D_{F_i}. $\tau_i[G_1, G_2, \ldots, G_n](s_1, s_2, \ldots, s_{\mathrm{ar}(F_i)})$ represents expansion of t_i by replacing in t_i each x_j by s_j and each application of a function variable F_k to $\mathrm{ar}(F_i)$ arguments by the result of applying G_k to the corresponding terms. That is, each *term* $\mathrm{F}_j(\ldots)$ is replaced by the *result* of the application $G_j(\ldots)$; this result is also a term.

Each τ_i expands an application of one of the recursive program's function variables; when τ_i is applied to $\langle F_1, F_2, \ldots, F_n\rangle$, the expansion is the replacement of the application by t_i. These coordinate functionals are components of

$$\tau[F_1, F_2, \ldots, F_n] = \langle \tau_1[F_1, F_2, \ldots, F_n], \tau_2[F_1, F_2, \ldots, F_n], \ldots, \tau_n[F_1, F_2, \ldots, F_n]\rangle\,.$$

$\tau : D_\tau \rightarrow D_\tau$ is a functional which, given (a tuple of) n functions on terms, returns another tuple of functions on terms. $\tau_i^m[\ldots]$ denotes the i^{th} element of $\tau^m[\ldots]$. τ varies from program to program; the particular one intended will be indicated by context.

We can now see why the meta-syntactic level of expressions was introduced: operation on the expressions that make up that level is a powerful idea, and it allows us an easy way to change a term by changing the expression that refers to it. If we think of the expression as the primary method for representing a term, then making simple changes to the expression (such as substituting one term constructor for another) will change the term in question. An even more powerful idea is that of a function over term constructors such as τ_i: if we apply it to different term constructors, then it returns a different term constructor as well. By applying such a function to different a term constructors, we can specify substitution (or simultaneous substitution, or several sequential applications of substitution, among others) in the final term. An example of this is the application of a function over term constructors to F_i and to $\tau_i[F_{1\ldots n}]$; relative to the first one, the second has had some number of instances of $\mathrm{F}_i(s_1, s_2, \ldots, s_{\mathrm{ar}(F_i)})$ expanded to t_i with $\vec{x}_{\mathrm{ar}(F_i)}$ (the formal variables) replaced by $\vec{x}_{\mathrm{ar}(F_i)}$ (the actual arguments). This expansion is difficult to formally specify without (at least) a formal definition of an *occurrence* of an element in a term; we can specify an expansion of (part of) a term by doing a simple substitution on the expression that refers to the term or by applying a higher-level function to a different set of arguments. It makes no sense to say, "substitute for F_i" in a term unless it is another function variable that is being substituted; expansion cannot be specified by substitution on the term level. The notions of application, substitution, and expansion are successively more complicated and difficult to formalize, so the first will be used in this thesis. The price of using higher-order functions in expressions is a small one to pay for this convenience.

As an example, consider this recursive program scheme for exponentiation:

$$\begin{aligned}
\mathrm{F}(x,y) &\Leftarrow \textit{if } p_{ev}(y) \textit{ then } \mathrm{G}(x,y) \textit{ else } \mathrm{H}(x,y) \textit{ fi} \\
\mathrm{G}(x,y) &\Leftarrow \textit{if } p_z(y) \textit{ then } 1 \textit{ else } f_{sq}(\mathrm{F}(x,y/2)) \textit{ fi} \\
\mathrm{H}(x,y) &\Leftarrow x \times \mathrm{F}(x,y-1)
\end{aligned}$$

We have taken the liberty of writing arithmetic operators in infix form. p_{ev} test for evenness and p_z for the zero value; f_{sq} is the squaring function. In the following examples of expressions and the terms to which they evaluate, t_1 and t_2 are arbitrary terms.

$$\begin{aligned}
F(t_1,t_2) \quad &\text{is} \quad \mathrm{F}(t_1,t_2) \\
\tau_F[F,G,H](t_1,t_2) \quad &\text{is} \quad \textit{if } p_{ev}(t_2) \textit{ then } \mathrm{G}(t_1,t_2) \textit{ else } \mathrm{H}(t_1,t_2) \textit{ fi} \\
\tau_F[G,F,\tau_H[G,H,F]](t_1,t_2) \quad &\text{is} \quad \textit{if } p_{ev}(t_2) \textit{ then } \mathrm{F}(t_1,t_2) \textit{ else } t_1 \times \mathrm{G}(t_1,t_2-1) \textit{ fi}
\end{aligned}$$

The language in which the τ_i's are written is not specified. The terms on the right hand side of the examples are to be taken verbatim, not evaluated in any fashion (though they could be later on). No arbitrary restrictions are placed on the functional arguments to τ; any tuple may be passed in, so long as it is of the right type (*i.e.*, it has the correct length and each of its elements is of the right type).

2.2 Operational semantics

The operational semantics of a recursive program scheme R over a basis B is defined by fixing an interpretation $\mathcal{I}$ for B, defining a transition relation $\Longrightarrow_{\mathcal{I},R}$ that "does one step" of "evaluation" of terms with respect to $\mathcal{I}$ and R, and then defining the meaning $\mathcal{M}_{\mathcal{I}}(R)$ by repeating evaluation steps until a constant is reached.

Definition 2. An *interpretation* $\mathcal{I} = \langle D, \mathcal{I}_0 \rangle$ of a recursive program scheme R consists of:

1. A nonempty set of elements D called the *domain* of the interpretation and sometimes referred to as the domain of the program scheme. $D_{\mathcal{I}}$ will be used for the domain of a recursive program scheme; D will be used when an arbitrary domain is intended.
2. $\mathcal{I}_0$, which defines the meanings of the constants of R. To each function symbol f_i corresponds a total function mapping D^n into D. To each predicate symbol p_i corresponds a total predicate (*i.e.*, a total function mapping D^n into the true or the false Boolean value).

An interpretation is a partial function $\mathcal{I}$: Term $\rightarrow$ Env $\rightarrow D_{\mathcal{I}}$; that is, given a term as input, it returns a function from environments to domain elements which specifies the interpretation of the term in the environment. An environment provides assignments of variables x_i and F_i to elements of type $D_{\mathcal{I}}$ and $D_{\mathcal{I}}^{\mathrm{ar}(F_i)} \rightarrow D_{\mathcal{I}}$, respectively. The type of an environment is Var $\rightarrow (D_{\mathcal{I}}^{\mathrm{ar}(F_i)} \rightarrow D_{\mathcal{I}})$; given a symbol, it returns a function over domain elements. There are no variables of base type bound in the environment.

The interpretation of a term is defined inductively by

1. $\mathcal{I}(c)(\gamma) = \mathcal{I}_0(c)$ if c is a constant.
2. $\mathcal{I}(f(t_1, t_2, \ldots, t_{\mathrm{ar}(f)})) = \mathcal{I}_0(f)(\mathcal{I}(t_1)(\gamma), \mathcal{I}(t_2)(\gamma), \ldots, \mathcal{I}(t_{\mathrm{ar}(f)})(\gamma))$ if f is a function, predicate, or built-in constant and each t_i is any term.
3. $\mathcal{I}(\mathrm{F}(t_1, t_2, \ldots, t_{\mathrm{ar}(F)})) = \gamma(\mathrm{F})(\mathcal{I}(t_1)(\gamma), \mathcal{I}(t_2)(\gamma), \ldots, \mathcal{I}(t_{\mathrm{ar}(f)})(\gamma))$ if F is a function variable.

If none of the cases work or if the interpretation of a subterm cannot be determined, then the term has no interpretation. This will be the case when a term is nonsyntactic or an illegal variable or symbol is used, for instance. Note that there is no interpretation for ordinary (base) variables; we will always do substitution of base constants for ordinary variables before interpreting or evaluating the body of a function definition. This definition is closely mirrored by the operational semantics.

To simplify the technical development below, we require that there be a *unique* constant symbol for every element of the domain and that these be the only constant symbols. In other words, we assume that all constant symbols denote distinct values and that for every $d \in D_{\mathcal{I}}$ there is a constant symbol $\widehat{d} \in F$ such that $\mathcal{I}_0(\widehat{d}) = d$. Here $\hat{}$ denotes a metamathematical function that maps an element of the domain $D_{\mathcal{I}}$ to *the* appropriate constant symbol. We will also assume that given $\widehat{d}$ we can determine d and vice-versa.

The empty environment, represented by a centered dot $(\cdot)$, has no bindings whatever; it is used when the actual value of the argument is irrelevant. We write $\mathcal{I}(t)(\cdot)$ rather than

$\mathcal{I}(t)(\sigma)$ for the meaning of t with respect to $\mathcal{I}$ and any σ. The use of the empty environment does not affect the interpretation of constants such as $\widehat{d}$ and f; they are handled by $\mathcal{I}_0$, which is part of the interpretation.

Axioms and inference rules will be used to inductively define the transition relation $\Longrightarrow_{\mathcal{I},R}$. There are no soundness or completeness properties to be considered for the operational semantics, for it is by definition correct; it is what it is. One question we could ask is whether it is what we intended it to be (it is). For now, we are taking the value computed by the operational semantics as the definition the meaning of a term. Another question about the operational semantics, then, is whether its definition of the meaning of a term coincides with other definitions of meaning. Other definitions are possible; one natural one will be introduced in section 3.4. We will show in section 4.3 that the two notions coincide; they compute the same values for any program and input.

Given a recursive program scheme $R = \langle\{\mathrm{F}_i(\vec{x}_{\mathrm{ar}(F_i)}) \Leftarrow t_i\}, m\rangle$ and an interpretation $\mathcal{I}$, the transition relation $\Longrightarrow_{\mathcal{I},R}$ between terms is defined by the axioms and inference rules below. In each axiom, $\Longrightarrow$ will possibly be subscripted by R or $\mathcal{I}$ to indicate that it depends upon the recursive program scheme R or the interpretation $\mathcal{I}$. If no subscript is given, then the axiom describes the evaluation of constructs common to every recursive program scheme (for instance, the conditional and boolean operators). In the axioms and rules below, c stands for a b- or t-constant; in what follows them, c_j is usually a t-constant.

First, we introduce axioms for the "primitives" of R, namely the function, predicate, and built-in symbols of the basis B. Function and predicate symbols from the basis are given meaning by the interpretation $\mathcal{I}$, so we have the axioms

$$\begin{array}{lll} f(c_1,\ldots,c_k) \Longrightarrow_{\mathcal{I}} \widehat{d} & & (\text{when } \mathcal{I}_0(f)(\mathcal{I}_0(c_1),\ldots,\mathcal{I}_0(c_k)) = d) \ , \\ p(c_1,\ldots,c_k) \Longrightarrow_{\mathcal{I}} \widehat{d} & & (\text{when } \mathcal{I}_0(p)(\mathcal{I}_0(c_1),\ldots,\mathcal{I}_0(c_k)) = d) \ . \end{array}$$

Note that $\Longrightarrow$ does not relate constant symbols to anything, since constant symbols are "answers" or "values" and do not require further evaluation.

The axioms for the conditional and the equality test are:

$$\begin{array}{rl} \textit{if true then } s_1 \textit{ else } s_2 \textit{ fi} \Longrightarrow s_1 \ , & \\ \textit{if false then } s_1 \textit{ else } s_2 \textit{ fi} \Longrightarrow s_2 \ , & \\ c = c \Longrightarrow \textit{true} \ , & \\ c = c' \Longrightarrow \textit{false} & \quad (c \not\equiv c') \ . \end{array}$$

The axioms for $\wedge$ and $\neg$ are:

$$\begin{array}{r} \textit{true} \wedge \textit{true} \Longrightarrow \textit{true} \ , \\ \textit{true} \wedge \textit{false} \Longrightarrow \textit{false} \ , \\ \textit{false} \wedge \textit{true} \Longrightarrow \textit{false} \ , \\ \textit{false} \wedge \textit{false} \Longrightarrow \textit{false} \ , \\ \neg\textit{true} \Longrightarrow \textit{false} \ , \\ \neg\textit{false} \Longrightarrow \textit{true} \ . \end{array}$$

The above axioms require that certain subterms of the term being evaluated be constant symbols. In general this is not the case, so we need some inference rules that will allow subterms of a term to be simplified. To evaluate the application of a function symbol f to its arguments, we must first evaluate its arguments one at a time in left-to-right order.

$$\frac{s_i \Longrightarrow s_i'}{f(c_1, \ldots, c_{i-1}, s_i, s_{i+1}, \ldots, s_{\mathrm{ar}(f)}) \Longrightarrow f(c_1, \ldots, c_{i-1}, s_i', s_{i+1}, \ldots, s_{\mathrm{ar}(f)})} \; .$$

There is a similar inference rule for the application of a predicate symbol p:

$$\frac{s_i \Longrightarrow s_i'}{p(c_1, \ldots, c_{i-1}, s_i, s_{i+1}, \ldots, s_{\mathrm{ar}(p)}) \Longrightarrow p(c_1, \ldots, c_{i-1}, s_i', s_{i+1}, \ldots, s_{\mathrm{ar}(p)})} \; .$$

To evaluate a conditional term *if* b *then* s_1 *else* s_2 *fi*, the subterm b must be evaluated to either *true* or *false*:

$$\frac{b \Longrightarrow b'}{\textit{if } b \textit{ then } s_1 \textit{ else } s_2 \textit{ fi} \Longrightarrow \textit{if } b' \textit{ then } s_1 \textit{ else } s_2 \textit{ fi}} \; .$$

To evaluate an equality term $s_1 = s_2$, the subterms s_1 and s_2 must first be evaluated to constants:

$$\frac{s_1 \Longrightarrow s_1'}{s_1 = s_2 \Longrightarrow s_1' = s_2} \; ,$$

$$\frac{s_2 \Longrightarrow s_2'}{c = s_2 \Longrightarrow c = s_2'} \; .$$

As with function and predicate symbol application, the rules for computing $=$ enforce a left-to-right evaluation.

To evaluate a Boolean term $b_1 \wedge b_2$, the subterms b_1 and b_2 must be evaluated first:

$$\frac{b_1 \Longrightarrow b_1'}{b_1 \wedge b_2 \Longrightarrow b_1' \wedge b_2} \; ,$$

$$\frac{b_2 \Longrightarrow b_2'}{c \wedge b_2 \Longrightarrow c \wedge b_2'} \; .$$

Again, the rules for computing $\wedge$ enforce a left-to-right evaluation. The inference rule for $\neg b$ is:

$$\frac{b \Longrightarrow b'}{\neg b \Longrightarrow \neg b'} \; .$$

Finally, to evaluate the application of a function variable F_i to $k = \mathrm{ar}(\mathrm{F}_i)$ arguments, we substitute the arguments for the parameter variables in the body t_i of the declaration of the function variable.

$$\mathrm{F}_i(\vec{s}_k) \Longrightarrow_R (t_i)^{\vec{s}_k}_{\vec{x}_k} \qquad \text{(copy rule)}$$

This expansion of a function variable into its body with the substitution of arguments for parameter variables is called the *copy rule.* We can see that $\tau_i[F_{1\ldots n}](\vec{s}_k) = (t_i)^{\vec{s}_k}_{\vec{x}_k}$.

The transition relation $\Longrightarrow_{\mathcal{I},R}$ is *symbolic*. It relates terms to terms in a syntactic, symbol-pushing way without regard for the meanings of the symbols being manipulated. The choice of constants and meanings in the interpretation could lead to the impression that it is meaning, not syntax, that is being manipulated here; after all, the interpreter will replace 1+1 by 2 and *if true then* t_1 *else* t_2 *fi* by t_1, so it might seem that while the copy rule does symbol-pushing on function variables and their arguments, simplifications treat symbols semantically. All that's really happening, however, when a symbol application is "evaluated" is a table lookup of its arguments and replacement of the original expression by the result of the lookup. When we set up the interpretation, we had a specific semantics in mind, but the relations are computed without knowledge of what that intended meaning was.

Definition 3. A binary relation $\longrightarrow$ is *Church-Rosser* if for all M, M_1, and M_2,

$$M \longrightarrow^* M_1 \text{ and } M \longrightarrow^* M_1 \quad \text{implies} \quad \exists M_3 . M_1 \longrightarrow^* M_3 \text{ and } M_2 \longrightarrow^* M_3 \ .$$

($\longrightarrow^*$ is the transitive, reflexive closure of $\longrightarrow$.)

A binary relation $\longrightarrow$ is *determinate* if $M \longrightarrow M_1$ and $M \longrightarrow M_2$ implies that $M_1 = M_2$. Another way to state this is to say that there is a deterministic algorithm to compute $\longrightarrow$.

Obviously any determinate relation is Church-Rosser.

Fact 4. $\Longrightarrow$ is determinate. It follows that if a term evaluates to a constant, then the constant is unique; *i.e.*, if $t \Longrightarrow^*_{\mathcal{I},R} c_1$ and $t \Longrightarrow^*_{\mathcal{I},R} c_2$, then $c_1 \equiv c_2$. The relation $\Longrightarrow_{\mathcal{I},R}$ is in fact a partial function for any recursive program scheme R (it is partial because it is undefined for constants and variables).

Having defined a mechanism for symbolically evaluating terms, we verify that it works as expected for the primitives that are shared by predicate logic and recursive program schemes.

Lemma 5. If t is a closed, first-order term (*i.e.*, t contains no free variables, function variables, or conditional expressions), then $t \Longrightarrow^*_{\mathcal{I}} \widehat{d}$ iff $\mathcal{I}(t)(\cdot) = d$.

Proof. ($\Rightarrow$) By induction on computation length and the definition of $\Longrightarrow_{\mathcal{I}}$, verifying that each computation step is sound with respect to $\mathcal{I}$.

($\Leftarrow$) By structural induction on t. ■

Lemma 6. If b is a closed, first-order b-term, then $b \Longrightarrow^*_{\mathcal{I}} true$ iff $\mathcal{I}(b)(\cdot) = true$. Naturally enough, this also hold for *false*.

Proof. As for lemma 5. ■

Ω is the function which never returns, regardless of its argument; it is more formally defined in section 3.3. Its syntactic analog F_Ω is a function variable which never gets replaced in a computation, regardless of the computation rule in effect, or, alternatively, gets replaced by itself. The latter formulation, which is the one we will use, is identical to the former because $\mathrm{F}_i(s_1, s_2, \ldots, s_{\mathrm{ar}(F_i)})$ is replaced in any substitution step by $\mathrm{F}_i(s_1, s_2, \ldots, s_{\mathrm{ar}(F_i)})$;

that is, by itself. Thus, it does not matter whether the term is substituted or not, for the output is the same in either case. Our choice of F_Ω will be justified in section 3.6, page 31. F_Ω is the F_i such that $\tau_i[F_{1\ldots n}] = F_i$; we will assume that we can use it, whether or not it is one of the function variables explicitly defined in R.

$\mathcal{I}_\Omega$ is the interpretation in which all function variables are given the value Ω; that is, $\mathcal{I}_\Omega(t)(\sigma) = \mathcal{I}(t)(\sigma[\mathrm{F}_i \mapsto \Omega])$. $\sigma[\mathrm{F}_i \mapsto \Omega]$ is an environment which is the same as σ (has the same bindings of variables to values as σ) except that F_i is bound to Ω, whether or not F_i was bound to anything in σ. The free variable i in the substitution notation indicates that the substitution should be done for all appropriate values of the variable (in this case, for $1 \le i \le n$). Since the environment passed to $\mathcal{I}_\Omega$ is immaterial, $\mathcal{I}_\Omega$, like $\mathcal{I}_0$, will be annotated without the environment argument.

Definition 7 (Operational meaning of terms). $[\![t]\!]_{\mathrm{op}} = d$ whenever $t \Longrightarrow^*_{\mathcal{I}} \widehat{d}$; that is, the operational meaning of a term is the value of the base constant to which it simplifies under the operational semantics without the copy rule. If the operational semantics does not halt or does not get to a constant $\widehat{d}$ while computing on t, then $[\![t]\!]_{\mathrm{op}}$ is undefined. By lemma 5, $[\![t]\!]_{\mathrm{op}} = \mathcal{I}_\Omega(t)$.

In general, $[\![A]\!]$ will be used to denote the semantic object to which the syntactic object A corresponds. This semantic object will not always be specified operationally, however.

Definition 8. The *operational meaning* $\mathcal{M}_{\mathcal{I}}(R) : D_{\mathcal{I}}^{\mathrm{ar}(F_m)} \to D_{\mathcal{I}}$ of a recursive program scheme R is a partial function determined by:

$$\mathcal{M}_{\mathcal{I}}(R)(d_1, d_2, \ldots, d_{\mathrm{ar}(F_m)}) = d \quad \text{iff} \quad \mathrm{F}_m(\widehat{d}_1, \widehat{d}_2, \ldots, \widehat{d}_{\mathrm{ar}(F_m)}) \Longrightarrow^*_{\mathcal{I},R} \widehat{d}\ .$$

By fact 4 above, $\widehat{d}$ is unique, so $\mathcal{M}_{\mathcal{I}}(R)$ is well-defined. Note that $d_1, d_2, \ldots, d_{\mathrm{ar}(F_m)}$ is in the domain of $\mathcal{M}_{\mathcal{I}}(R)$ iff the sequence of terms (known as the computation sequence of R on $(\widehat{d}_1, \widehat{d}_2, \ldots, \widehat{d}_{\mathrm{ar}(F_m)})$; see page 18)

$$\mathrm{F}_m(\widehat{d}_1, \widehat{d}_2, \ldots, \widehat{d}_{\mathrm{ar}(F_m)}) \Longrightarrow_{\mathcal{I},R} u_1 \Longrightarrow_{\mathcal{I},R} u_2 \Longrightarrow_{\mathcal{I},R} \cdots$$

is finite, *i.e.*, the evaluation of $\mathrm{F}(\widehat{d}_1, \widehat{d}_2, \ldots, \widehat{d}_{\mathrm{ar}(F_m)})$ terminates.

Lemma 9. $t \not\Longrightarrow$ iff $t = \widehat{d}$ for some d; in other words, if the operational semantics gets "stuck" on a term (can take no $\Longrightarrow$ step), then the term is a base constant.

Proof. We can imagine an operational semantics that gets stuck on terms which are not base constants. We prove that this is impossible for $\Longrightarrow$ by induction on terms.

As noted before, there are only the following forms for a term t.

1. Base constant: $t = \widehat{d}$.
   No rule applies. This fits the desired conclusion.

2. Function variable application: $t = \mathrm{F}(t_1, t_2, \ldots, t_{\mathrm{ar}(F)})$.
   The copy rule applies.

Figure 2-1: The tree representation of the term $\mathrm{F}(x, 3) + \textit{if } \neg \textit{true then } 4 \textit{ else } 5 \textit{ fi}$.

3. Function, predicate, or built-in symbol application: $t = f(t_1, t_2, \ldots, t_{\mathrm{ar}(f)})$.
   If each t_i is a base constant, then the application can be substituted by the value assigned it by the interpretation. If some t_i is not a base constant, then by induction, that term can move under the operational semantics.

4. Conditional statement: $t = \textit{if } b \textit{ then } t_1 \textit{ else } t_2 \textit{ fi}$.
   This case is analogous to the previous one: if b is a Boolean constant, then this term reduces to t_1 or t_2. Otherwise b reduces, by induction. ■

2.3 Substitution and simplification

Definition 10. Suppose $s_i \Longrightarrow_{\mathcal{I},R} s_{i+1}$. If s_{i+1} was obtained from s_i by use of the copy rule (*i.e.*, $s_i \Longrightarrow_R s_{i+1}$), then we call this step a *substitution step.* If s_{i+1} was obtained from s_i by use of some other rule (*i.e.*, $s_i \Longrightarrow_{\mathcal{I}} s_{i+1}$), then we call this step a *simplification step.*

A simplification step always removes an occurrence of a function, predicate, or built-in symbol from a term. A substitution step removes an occurrence of a function variable F_i by replacing the function variable application with the expansion of t_i. A substitution step may also introduce other occurrences of function variables, either by copying occurrences which appear in the arguments to F_i or by creating new ones (those which explicitly appear in t_i).

To make the above a bit more formal, we will define what we mean by an occurrence. Consider a term to be a tree which has function variables and function, predicate, and built-in symbols at the nodes and base constants [and variables] at the leaves. For instance, the tree representation of the term $\mathrm{F}(x, 3) + \textit{if } \neg \textit{true then } 4 \textit{ else } 5 \textit{ fi}$ is shown in figure 2-1.

Definition 11. An *occurrence* in a term is a specification of the path from the term's root to a node or leaf.

The representation of the occurrence does not concern us here; the one that we will use is a sequence of numbers (separated by $\cdot$ and terminated by 0 for notational convenience) which specify which branch to take at each node. For instance, the occurrence of F in the above term would then be just $1{\cdot}0$; the occurrence of *true* would be $2{\cdot}1{\cdot}1{\cdot}0$. The path 0 specifies the root of the tree. We can speak of the term rooted at an occurrence or of the value (the symbol f or function variable F_i, or the term constructor F_i, depending on whether a term or an expression is being represented as a tree) at an occurrence. We will need a concrete representation for an occurrence later, when we formally specify operations on them. For now, when we say “remove an occurrence” or “substitute for an occurrence” we (usually) mean to substitute for or expand the term rooted at that occurrence if we are using the direct representation for terms and to substitute for the term constructor F_i or to apply the function τ_j to a different term constructor than F_i if we are taking the expression view.

[Fix bungled definition.]

We will describe a family of transition relations $\overset{\text{sub}}{\Longrightarrow}_{\mathcal{I},R}$ in terms of substitution and simplification steps; this will specify a new operational semantics *sub* which we will find easier to work with. The subscripts $\mathcal{I}$ and R will be omitted when they are clear from context. The transition relation depends on a *computation rule* (see page 20) which specifies how much work is to be done at each step; we will usually not mention which of the $\overset{\text{sub}}{\Longrightarrow}$ rules in particular we are considering (*i.e.*, which computation rule is operating). Informally, $s_j \overset{\text{sub}}{\Longrightarrow} s_k$ if s_k can be derived from s_j by simultaneously using the copy rule on some occurrences of F_i in s_j and then simplifying wherever possible. Let

$$u_0 \overset{\text{sub}}{\Longrightarrow} u_1 \overset{\text{sub}}{\Longrightarrow} u_2 \overset{\text{sub}}{\Longrightarrow} \ldots$$

be a computation of the new operational semantics, where u_0 is $\mathrm{F}_m(d_1, d_2, \ldots, d_{\mathrm{ar}(F_m)})$. This will be called the *computation sequence* of R on $(\widehat{d}_1, \widehat{d}_2, \ldots, \widehat{d}_{\mathrm{ar}(F_m)})$. (In fact, we will also want to consider computation sequences which start at arbitrary terms; that case is handled identically.) Each u_{i+1} is obtained from u_i by first simultaneously replacing in u_i some occurrences of F by $\tau[F_{1\ldots n}]$ (doing some substitution steps) and then removing, whenever possible, occurrences of function, predicate, and built-in symbols via the simplification rules (doing simplification steps until no more can be done). The formal definition of “simultaneous substitution” is technically cumbersome, so we omit it; the intention should be clear.

Sub simply specifies a particular order for the operations of the original operational semantics and groups many steps of $\Longrightarrow$ as a single step of $\overset{\text{sub}}{\Longrightarrow}$. The specification of *sub* can be formalized by giving an algorithm which specifies how a $\overset{\text{sub}}{\Longrightarrow}$ operation is done. Initially all F's in u, the term upon which a step of $\overset{\text{sub}}{\Longrightarrow}$ is to be performed, are untagged. First perform zero or more sequential substitutions on *untagged* instances of F in u; there is no obligation to perform substitution for an F simply because such a substitution is permissible. Each time that a substitution step is done, all F's in the substituting term are tagged; that is, if $\tau_i[F_{1\ldots n}](\vec{s}_{\mathrm{ar}(F_i)})$ replaces $F_i(\vec{s}_{\mathrm{ar}(F_m)})$, then all F's in the replacing term are tagged (unsubstitutable), even if there were untagged F's in one or more of the s's. When there are no remaining untagged terms or it is decided that no more substitutions will be done (this decision is made by the computation rule), the resulting term is fully simplified by applying simplification rules to every subterm to which they apply, until no more apply. No tagging information is retained from step to step. This constitutes a single $\overset{\text{sub}}{\Longrightarrow}$ step. These rules enforce the restriction that only F's that appeared in u can ever be substituted for, and simultaneous substitution of F's in u is faithfully simulated.

This algorithm provides a specification or sample implementation for $\overset{\text{sub}}{\Longrightarrow}$; this is not necessarily the way that *sub* works. For instance, each step of $\overset{\text{sub}}{\Longrightarrow}$ is atomic, though here several steps of $\Longrightarrow$ were required in order to achieve the effect of a single step of $\overset{\text{sub}}{\Longrightarrow}$. Actually, the substeps used by this method were not really $\Longrightarrow_R$ and $\Longrightarrow_{\mathcal{I}}$ steps, for *internal* expressions could be substituted for or simplified. We cannot show that $\overset{\text{sub}}{\Longrightarrow}$

is precisely equivalent to $\Longrightarrow$, then, for it is not: for the trivial recursive program scheme $\mathrm{F}(x) \Leftarrow \mathrm{F}(x+1)$ and for any $\overset{\text{sub}}{\Longrightarrow}$ relation, the respective computation sequences are

$$\mathrm{F}(0) \Longrightarrow_R \mathrm{F}(0+1) \Longrightarrow_R \mathrm{F}(0+1+1) \Longrightarrow_R \cdots$$
$$\mathrm{F}(0) \overset{\text{sub}}{\Longrightarrow} \mathrm{F}(1) \overset{\text{sub}}{\Longrightarrow} \mathrm{F}(2) \overset{\text{sub}}{\Longrightarrow} \cdots$$

It is the case, however, that some members of the $\overset{\text{sub}}{\Longrightarrow}$ family are equivalent to $\Longrightarrow$ in an important way: they compute to the same value for any term. We will show this for some such $\overset{\text{sub}}{\Longrightarrow}$ rules; in particular, it is true for the one which always does every possible substitution.

Chapter 3

Computation Rules

This chapter introduces computation rules, which control the evaluation of terms by specifying the order in which steps of the operational semantics are taken, or, alternatively, by specifying on which term the operational semantics acts at a particular step. Computation rules were referred to in the previous chapter as something that specified which $\overset{\text{sub}}{\Longrightarrow}$ rule was being used. The syntactic and semantic properties of computation rules will be examined in preparation for the connections made in the next chapter between the operational and denotational meanings of a recursive program scheme.

3.1 Definition and meaning of computation rules

We will first introduce computation rules in a less formal, more intuitive way; shortly we will formalize and sharpen the notion.

From now on the interpretation $\mathcal{I}$ will be left implicit; where it is not mentioned, the usual one is used. For instance, the two symbols C_R and $S_{C,R}$ defined here should really be $C_{\mathcal{I},R}$ and $S_{C,\mathcal{I},R}$ since they depend on the interpretation as well as on the program scheme, but we reject that more unwieldy notation.

Definition 12. A *computation rule* C is an algorithm for selecting in a term some occurrences of F_i to be simultaneously replaced by $\tau_i[F_{1\ldots n}]$ in the next step of the *sub* operational semantics. The substitution $S_{C,R}$ is a mapping from terms to terms in which some occurrences of F_i in the input term are simultaneously replaced by $\tau_i[F_{1\ldots n}]$ (alternatively, occurrences of $\mathrm{F}_i(\vec{s}_{\mathrm{ar}(F_i)})$ in the input term are simultaneously expanded into t_i with occurrences of x_j replaced by s_j) to produce the output term. C_R : term $\rightarrow$ term computes a $\overset{\text{sub}}{\Longrightarrow}$ relation (which particular one is computed depends on C); that is, given a term in a computation sequence, C_R produces the next term.

[Does this make sense? I need to sharpen the distinction between F_i and F_i above.] Note that the term-constructor (F_i) view of function variables is being taken here.

More formally, a computation rule C is a function which takes as input a term and produces as output an ordered set of occurrences of function variables in that term; this choice is used by $S_{C,R}$ and by C_R, which should not be confused with C, which refers to the rule itself (the function that chooses occurrences), not a function that does substitutions. The choice of occurrences chosen by the computation rule to be substituted depends only on the structure of the term, not on the program R. The order of occurrences does not matter so long as it is chosen deterministically; for convenience, we will assume that lexicographic ordering is used. C_R is the function from terms to terms whose output is the simplification of the input term after $\tau_i[F_{1\ldots n}]$ has replaced F_i at each occurrence.

C_R produces a term which is fully simplified, while $S_{C,R}$ only does substitution. For any term A, $S_{C,R}(A) \overset{\text{con}}{\Longrightarrow}_I^* C_R(A)$. $S_{C,R}$ is introduced here (and used, among other places, in the proof of theorem 34) because its syntactic properties are simpler than those of C_R. Actually, S_C could be considered without a particular recursive program scheme R in mind: given a term, it would choose the function variables to substitute for, but it cannot do the substitution (or produce a term as output) since it does not know what to substitute for the selected function variables. We do not consider S_C in that light.

We will only consider "interesting," nonempty substitutions; that is, a substitution must expand some F in the term to which it is applied. Thus, no substitution can be performed on a term which is free of F's. These restrictions prevent a particular case of the operational semantics getting stuck when a substitution meets a term on which it chooses to do no substitutions. The next step would try to do a substitution step on the same term and would also perform the empty substitution. Since this sort of infinite loop is not of interest to us here, we shall avoid it by fiat. It is consolation that, in practice, no one would ever choose to use such a rule. The substituted function may, however, be F_Ω (see page 16), in which case the substitution occurs but does not change the term to which it is applied. Although the effect is similar to that described earlier, this is a different case: if F_Ω is being substituted in, then we want no progress to be made on a particular term, for any computation rule. The computation rules that we prohibit here might not make progress on terms that we considered interesting. We will see later that such a rule is not *safe* (see section 4.1); that is, it would not have been guaranteed to produce the value we wanted for a particular recursive program scheme.

$[\![C_R]\!]$ will denote the semantic function computed by the computation rule C for the recursive program scheme R; it is a partial function that maps $\text{ar}(F_m)$ domain elements to a single domain element. If the computation sequence of C on $(\widehat{d_1}, \widehat{d_2}, \ldots, \widehat{d_k})$ terminates with $\widehat{d}$, then $[\![C_R]\!](d_1, d_2, \ldots, d_k) = d$; if the computation sequence does not terminate, then $[\![C_R]\!](d_1, d_2, \ldots, d_k)$ is undefined.

3.2 Examples of computation rules

Some examples of computation rules are provided here. Similar rules are given in [Man74, p. 375] and [Vui73, p. 33].

Leftmost-innermost ("call-by-value") rule LI**:** Substitute for only the leftmost-innermost occurrence of F (that is, the leftmost occurrence of F which has all its arguments free of F's).

Parallel-innermost rule PI**:** Simultaneously substitute for all the innermost occurrences of F (that is, all occurrences of F whose arguments are all free of F's).

Leftmost ("call-by-name") rule L**:** Substitute for only the leftmost occurrence of F.

Parallel-outermost rule PO**:** Simultaneously substitute for all the outermost occurrences of F (that is, all occurrences of F which do not occur as arguments of other F's).

Free-argument rule FA**:** Simultaneously substitute for all occurrences of F which have at least one argument free of F's.

Full-substitution rule FS**:** Simultaneously substitute for all occurrences of F.

When the above computation rules act on the term F(0,F(1,1))+F(F(2,2),F(3,3)), the underlined occurrences of F will be replaced:

$$\begin{array}{ll}
LI & \mathrm{F}(0,\underline{\mathrm{F}}(1,1))+\mathrm{F}(\mathrm{F}(2,2),\mathrm{F}(3,3)) \\
PI & \mathrm{F}(0,\underline{\mathrm{F}}(1,1))+\mathrm{F}(\underline{\mathrm{F}}(2,2),\underline{\mathrm{F}}(3,3)) \\
L & \underline{\mathrm{F}}(0,\mathrm{F}(1,1))+\mathrm{F}(\mathrm{F}(2,2),\mathrm{F}(3,3)) \\
PO & \underline{\mathrm{F}}(0,\mathrm{F}(1,1))+\underline{\mathrm{F}}(\mathrm{F}(2,2),\mathrm{F}(3,3)) \\
FA & \underline{\mathrm{F}}(0,\underline{\mathrm{F}}(1,1))+\mathrm{F}(\underline{\mathrm{F}}(2,2),\underline{\mathrm{F}}(3,3)) \\
FS & \underline{\mathrm{F}}(0,\underline{\mathrm{F}}(1,1))+\underline{\mathrm{F}}(\underline{\mathrm{F}}(2,2),\underline{\mathrm{F}}(3,3))
\end{array}$$

The full-substitution computation rule can be formally specified as

$$FS(t) = \{o \,|\, \exists i.\mathrm{SymbolAt}(o,t) = \mathrm{F}_i\} \ .$$

FS returns the set of all occurrences o in the input term t such that the symbol at o is a function variable (*i.e.*, the term rooted at o is an application of a function variable). The specification of the rule which acts on expressions which refer to terms is exactly analogous. A slightly less trivial example is that of the parallel-outermost computation rule.

$$\begin{array}{rcl}
PO(c) & = & \{\} \ , \\
PO(f(s_1,s_2,\ldots,s_{\mathrm{ar}(f)})) & = & \{i\cdot o \,|\, o \in PO(s_i)\} \\
 & = & \displaystyle\bigcup_{i\leq \mathrm{ar}(f)} \{i\cdot o \,|\, o \in PO(s_i)\} \ , \\
PO(\mathrm{F}_i(s_1,s_2,\ldots,s_{\mathrm{ar}(F_i)})) & = & \{0\} \ .
\end{array}$$

3.3 A new domain

In this section we introduce a new domain which will be used for the remainder of the thesis. Our previous domain was simply an unordered set of values; the new domain will have more structure and will better model the values that can result from computations; viz., an element which stands for nontermination is added and related to each of the existing terms (none of which are domain-theoretically comparable). We also introduce notation to support this domain and the ideas that appear later.

[This needs work.]

Definition 13 (Ordering of domain elements). $\sqsubseteq$ is a relation between two elements of the same type which denotes "is no more defined than" or "approximates." $\sqsubseteq$ is a partial order; that is, it is reflexive, antisymmetric, and transitive. It is not a total partial order since it is not the case that for any two elements a and b, either $a \sqsubseteq b$ or $b \sqsubseteq a$. In fact, in

$D_{\mathcal{I}}$, no pair of elements is comparable via $\sqsubseteq$; *i.e.*, for no $d \neq d'$ is it the case that $d \sqsubseteq d'$ or $d' \sqsubseteq d$. Concretely, the domain element 3 cannot be said to be any more defined than 2 or than 4. This ordering of domain elements is called the *discrete ordering.* The antisymmetry condition states that for any partial order $\sqsubseteq$ over domain D and any $d, d' \in D$, $d \sqsubseteq d'$ and $d' \sqsubseteq d$ implies that $d = d'$.

Functions are compared elementwise: $f \sqsubseteq g$ iff for all a in the (common) domain of f and g, $f(a) \sqsubseteq g(a)$. $\sqsubset$ is just like $\sqsubseteq$, but it is not reflexive; it can be thought of as denoting "is less defined than." $\sqsupseteq$ and $\sqsupset$ are defined in the obvious way.

$D_\perp$ denotes a domain which is identical to D except that a new element $\perp$ (*bottom*) has been added which is $\sqsubset$ all elements of D. $D_\perp$ is called a *lifted domain.* When a discrete domain is lifted, the resulting domain is called a *flat* domain. In a flat domain, all pairs of elements are incomparable to each other except that each non-$\perp$ element is $\sqsupset$ the new domain element $\perp$. $\perp_D$ is defined to be the least element of D (if one exists); that is, $\perp_D$ is already an element of D; it is the element such that for all $d \in D$, $\perp_D \sqsubseteq d$; if no such d exists, then it is not meaningful to use the notation $\perp_D$. The subscript on $\perp$ will be omitted when it can be determined from context.

A *chain* is a nonempty set of values $x_1, x_2, \ldots, x_n$ such that for all $i < n$, $x_i \sqsubseteq x_{i+1}$. A chain may be infinite or finite. An example of an infinite chain is the family of functions $I_n(y) = \textit{if } y < n \textit{ then } y \textit{ else } \perp \textit{ fi}$, where I_n is the identity function for arguments (numerically) less than n and does not terminate for arguments greater than or equal to n.

For $X \subseteq D$, $\bigsqcup X$ (the *least upper bound* or *lub* of X) denotes the element of D (if it exists) such that for all $x \in X$, $x \sqsubseteq \bigsqcup X$ and for all $d \in D$, if $x \sqsubseteq d$ then $\bigsqcup X \sqsubseteq d$. For a finite chain X, $\bigsqcup X$ is the greatest element of the chain. For some chains (such as the chain of I_n's above), the lub is not in the chain. $\bigsqcup \{I_n \mid n \in \text{Integers}\} = I$, the identity function. [Is this also true for monotone infinite-size chains?]

Definition 14. A function f is *monotonic* or *monotone* if for all a, b in its domain, $a \sqsubseteq b$ implies $f(a) \sqsubseteq f(b)$. In other words, if f is provided with more information (a better-defined input), then its output will not be worse-defined than the output was when the input was less defined. A monotonic function f is *continuous* if for all chains X in the domain of f, $f(\bigsqcup X) = \bigsqcup \{f(x) \mid x \in X\}$.

More complicated orderings than the discrete and flat ones can be imagined for base domains, but this thesis will consider only discrete and flat domains. Most orderings on nonbase domains (for instance, the ordering on $D_{\mathcal{I}} \to D_{\mathcal{I}}$, which contains infinite chains, as shown above) are more complicated. The technical complications of dealing with more complicated base domains are considerable, and many of the theorems in this thesis do not hold for interpretations with such domains. In any event, flat domains model the domains of computation values accurately, as we shall see.

The notions of lifting and comparison of definedness present a natural framework for talking about the meaning of nontermination. In $(D_{\mathcal{I}})_\perp$, the least element will stand for nontermination or undefinedness and the other domain elements will stand for themselves as they appear in $D_{\mathcal{I}}$. Partial functions with range $D_{\mathcal{I}}$ (such as $\mathcal{M}_{\mathcal{I}}(R)$ and $[\![C_R]\!]$) can be considered to be total functions whose value is $\perp_{(D_{\mathcal{I}})_\perp}$, the least element of $(D_{\mathcal{I}})_\perp$, when the partial version of the function would have been undefined. Functions with domain

$D_{\mathcal{I}}$ can likewise be extended so that their input domains are also $(D_{\mathcal{I}})_\perp$; whenever any argument is $\perp_{(D_{\mathcal{I}})_\perp}$, the result is $\perp_{(D_{\mathcal{I}})_\perp}$. The resulting total functions are strict in all of their arguments; such functions are called *naturally extended.* From now on we will use the domain $\mathcal{D} = (D_{\mathcal{I}})_\perp$ in preference to $D_{\mathcal{I}}$ and take this extended view of the symbols. $\perp$ will refer to $\perp_{\mathcal{D}}$ unless otherwise specified. (This use of $\perp$ is sometimes annotated ω; in this thesis, ω is reserved for the least transfinite number.)

The intuitive motivation for $\sqsubseteq$ remains valid: the meaning of a term that is undefined over $D_{\mathcal{I}}$ is $\sqsubseteq$ the meaning of one that means a base constant, but we cannot say that the meaning of the term 3 (that is, the number 3) is any more or less defined than the number 2.

It is an important technical detail that no constant denotes $\perp$;; $\perp$ is the value of a nonterminating computation of base type, and while we can specify terms that mean $\perp$, there is no $\widehat{\perp}$. For every $k > 0$ there is an $\Omega_k : \mathcal{D}^k \to \mathcal{D}$, the least function of its type. $\Omega_k(x_1, x_2, \ldots, x_k) = \perp$ independent of the values of the x_i's. The subscript of Ω will typically be omitted. It was Ω that motivated the name of F_Ω.

3.4 Monotonic fixed points

The following section shows that the least fixed point of a monotonic functional can be determined without the use of the least upper bound operator, which is not necessarily valid for infinite chains in monotone models.

We will require several definitions and notations for the proof.

First, we will define the ordinals in the following manner:

$$\begin{array}{rcll} 0 & = & \{\} & \text{(the empty set)} \\ n & = & \{k \mid k < n\} & \\ \omega & = & \{n \mid n \text{ an integer}\} & \end{array}$$

In general, $\beta + 1 = \beta \cup \{\beta\}$; this holds for $\beta = \omega$ as well. There are three types of ordinals, then: zero, successor ordinals $\delta = \beta + 1$ for some β, and limit ordinals which are neither of these. Limit ordinals are typically gotten to by use of the least upper bound operator.

The *ordinal* of a set is the largest cardinal such that there is a bijection between the set and the cardinal (considered as a set, as above). Another way to state this is that for any set A, $\mathrm{ord}(A, \leq) =$ the unique α such that $(A, \leq) \equiv (\alpha, \in)$.

The *depth* of a domain D is the ordinal of the longest chain in D and is denoted $\lfloor D \rfloor$. (We cannot define the depth of D to be just the length of the longest chain in D because that is not well-defined for infinitely long chains.) $|D|$, the cardinality of D, is an upper bound on $\lfloor D \rfloor$. The depth of a discrete domain (such as $D_{\mathcal{I}}$) is 1 and the depth of a flat domain (such as $\mathcal{D}$) is 2.

Because the following theorems are general, $\perp_D$ is used in preference to $\perp$, which is $\perp_{\mathcal{D}}$.

Definition 15. The notation $g^\beta(x)$ will be used to denote β applications of g to x.

$$g^\beta(x) = \begin{cases} x & \text{if } \beta = 0 \\ g(g^\delta(x)) & \text{if } \beta = \delta + 1 \text{ for some } \gamma \\ \bigsqcup_{\delta < \beta} g^\delta(x) & \text{if } \beta \text{ is a limit ordinal} \end{cases}$$

Fact 16. $g^{\beta}(x) = \bigsqcup \{g(g^{\delta}(x)) \mid \delta < \beta\}$ for all β. Actually this definition of $g^{\beta}(x)$ has one more application of g at limit ordinals than does the other definition, but that won't make any difference in the proofs because a fixed point will have been reached by then anyway. We will let $H(g, \beta)$ denote $\{g(g^{\delta}(x)) \mid \delta < \beta\}$.

Lemma 17. $g^{\beta}(x)$ is monotone in x; that is, for all $\beta \in \mathrm{ORD}$ and all $a \sqsubseteq b$, $g^{\beta}(a) \sqsubseteq g^{\beta}(b)$.

Proof. By easy transfinite induction. ■

Fact 18. For function spaces from domains of finite depth to any domains, monotonicity coincides with continuity.

We see from this fact that in the domains which we address, monotonic functions are also continuous. Thus, it would be valid to use the method outlined above for continuous functions to get the least fixed point of one of the monotonic functions in which we are interested. It is the goal of this thesis to show adequacy *without* resorting to continuity, however; since the functions are specified as monotonic, it is desirable to perform the proof entirely in the monotone frame.

Definition 19. A *fixed point* of $g : D \to D$ is a $d \in D$ such that $g(d) = d$. μg denotes the least fixed point of the function g; that is, for all fixed points d of g, $\mu g \sqsubseteq d$.

Theorem 20. Every monotone function $g : D \to D$ has a least fixed point $\mu g = g^{\lfloor D \rfloor}(\bot_D)$

To prove theorem 20, we need the following claim:

Claim 21. For all n, $\{g^{i}(\bot_D) \mid i \in \mathrm{ORD}\}$ is a well-ordered chain of ordinal no greater than $\lfloor D \rfloor$, where ORD is the class of ordinal numbers.

Proof of claim 21. First we will show that is monotone in β; *i.e.*, if $\beta \leq \beta'$ then $g^{\beta}(x) \leq g^{\beta}(x)$. $\beta \leq \beta'$ implies that $H(g, \beta) \subseteq H(g, \beta)$, where $H(g, \beta)$ is as defined in fact 16. The fact implies that $\bigsqcup H(g, \beta) \leq \bigsqcup H(g, \beta)$, so $\{g^{i}(\bot_D) \mid i \in \mathrm{ORD}\}$ is a chain; it is clearly well-ordered.

The chain cannot have length greater than $\lfloor D \rfloor$ because (by definition) no chain in D has length greater than $\lfloor D \rfloor$. ■

The proof of claim 21 can also be done by induction on the cases of definition 15, but that proof is longer and less elegant.

We can now describe how the least fixed point of a continuous function g is determined: the least upper bound is taken some number of times. Let p be least upper bound of the chain $\{g^{i}(\bot_D) \mid i \in \mathrm{ORD}\}$ (from claim 21). If p is not a fixpoint of g, then compute p, the least upper bound of the chain $\{g^{i}(p) \mid i \in \mathrm{ORD}\}$. The least fixed point is the fixed point reached by this method, and it is guaranteed to be reached.

Proof of theorem 20. First we will note that some $g^{\beta}(\bot_D)$ is a fixed point of g; then we will show that $g^{\lfloor D \rfloor}(\bot_D)$ is such a fixed point; finally we will prove that $g^{\lfloor D \rfloor}(\bot_D)$ is the least fixed point.

1. $\exists \beta \in \mathrm{ORD}. g^{\beta}(\perp) = g^{\beta+1}(\perp)$

   Suppose that this were not the case. Then all of the elements of $\{g^{\gamma}(\perp) \mid \gamma < \beta\}$ are distinct for any β. For $\beta =$ a cardinal greater than $|D|$, $\{g^{\gamma}(\perp) \mid \gamma < \beta\}$ has more elements than $|D|$, which cannot be the case since all of its elements are also members of $|D|$.

2. Let β be the least ordinal such that $g^{\beta}(\perp) = g^{\beta+1}(\perp)$. $g^{\beta}(\perp)$ is a fixed point of g.

   Obviously $\beta \leq \lfloor D \rfloor$. For all $\beta' \geq \beta$, $g^{\beta}(\perp) = g^{\beta'}(\perp)$ because, as noted in the proof of claim 21, g^{β} is monotonic in β.

3. $\mu g = g^{\beta}(\perp_D)$

   Let p be a fixed point of g. $\perp \sqsubseteq p$. By lemma 17, $g^{\beta}(\perp) \sqsubseteq g^{\beta}(p)$. Since p was an arbitrary fixed point of g, $g^{\beta}(\perp_D)$ is the least fixed point of g.

Since $\lfloor D \rfloor > \beta$, $g^{\lfloor D \rfloor}(\perp_D) = g^{\beta}(\perp_D)$. So $g^{\lfloor D \rfloor}(\perp_D)$ is μg, the least fixed point of g. ■

Let $\mathcal{D}_{F_i} = \mathcal{D}^{\mathrm{ar}(F_i)} \to \mathcal{D}$ and $\mathcal{D}_{\tau} = \langle \mathcal{D}_{F_1}, \mathcal{D}_{F_2}, \ldots, \mathcal{D}_{F_n} \rangle$. These semantic domains roughly correspond to the syntactic domains D_{F_i} and D_{τ}.

Definition 22 (Denotational meaning of τ). $[\![\tau_i]\!] : \mathcal{D}_{\tau} \to \mathcal{D}_{F_i}$ is the semantic function which corresponds to the syntactic function τ_i.

$$[\![\tau_i]\!][g_1, g_2, \ldots, g_n](d_1, d_2, \ldots, d_{\mathrm{ar}(F_i)}) = \mathcal{I}((t_i)^{\vec{d}_k}_{\vec{x}_k})(\cdot[\mathrm{F}_j \mapsto g_j])$$

where each g has type $\mathcal{D}_{F_i}$. The environment $\cdot$ is immaterial since the F_j's are the only free variables in $(t_i)^{\vec{d}_k}_{\vec{x}_k}$.

$[\![\tau]\!] : \mathcal{D}_{\tau} \to \mathcal{D}_{\tau}$, the denotational meaning of τ, is a tuple of the $[\![\tau_i]\!]$'s. $[\![\tau_i^m]\!]$ is defined similarly.

Corollary 23. The monotone functional $[\![\tau]\!]$ over $\mathcal{D}_{\tau}$ has a least fixed point

$$\mu[\![\tau]\!] = [\![\tau]\!]^{\gamma}[\Omega, \Omega, \ldots, \Omega]$$

for some sufficiently large ordinal γ.

$\mu[\![\tau]\!]$ is $\langle \mu_1[\![\tau]\!], \mu_2[\![\tau]\!], \ldots, \mu_n[\![\tau]\!] \rangle$, where $\mu_i[\![\tau]\!]$ denotes i th element of $\mu[\![\tau]\!]$.

This corollary follows at once from theorem 20. It only remains to determine how large γ is; that is, how large $\lfloor \mathcal{D}_{\tau} \rfloor$ is.

The number of elements of the domain over which τ operates ($\mathcal{D}_{\tau}$ or $\langle \mathcal{D}_{F_1}, \mathcal{D}_{F_2}, \ldots, \mathcal{D}_{F_n} \rangle$) is the product of the number of elements in the component domains.

While the theoretical limit for the size of $\mathcal{D}_{F_i}$ is

$$|\mathcal{D}|^{(|\mathcal{D}|^{\mathrm{ar}(F_i)})}$$

elements (functions from $\mathcal{D}^{\mathrm{ar}(F_i)}$ to $\mathcal{D}$), its actual size is smaller. A theoretical upper bound for γ is

$$\prod_{i=1}^{n} |\mathcal{D}|^{(|\mathcal{D}|^{\mathrm{ar}(F_i)})} \leq |\mathcal{D}|^{n(|\mathcal{D}|^{\max\{\mathrm{ar}(F_i)\}})}$$

This approximation is coarse because it does not make use of any information about the structure of the domain or about the functions themselves. There are fewer total functions than all (*i.e.*, partial; every function is a partial function since every total function is also a partial function) functions and no more continuous functions than monotone functions; we are only interested in total monotone functions. However, this approximation is as close as we need to get, for it gives us an upper bound of how many times τ must be applied before obtaining the least fixed point; it can never hurt to apply the functional too many times. It is worth noting that this proof works even if the size of the domain is transfinite, as will usually be the case. γ will be a transfinite number computed by transfinite multiplication and exponentiation; it is possible to create a chain made up of a transfinite number of elements, so the proof goes through similarly in the transfinite case.

The consequence of the finite depth of this domain is that fact 18 can be used to justify the use of $\bigsqcup$ in the definition of g^i on page 24: since the domains in question are of finite depth, the least fixed point of any subset of the domain is in the subset.

Definition 24. The *denotational meaning* f_R of a recursive program scheme R is the m^{th} element of the least fixed point of $[\![\tau]\!]$; that is, it is $\mu_m[\![\tau]\!]$. Like $\mathcal{M}_{\mathcal{I}}(R)$, f_R is a partial function from $D_{\mathcal{I}}^{\operatorname{ar}(F_m)}$ to $D_{\mathcal{I}}$ (or a function from $D_{\mathcal{I}}^{\operatorname{ar}(F_m)}$ to $\mathcal{D}$ which sometimes takes on the value $\perp$).

3.5 Incorrect computation rules

The major thrust of this thesis is to make connections between the operational and denotational meanings of programs. Let us show by example that some of the computation rules are incorrect in that they do not compute the fixed point. For these computation rules, $[\![C_R]\!] \neq f_R$.

Consider the following recursive program scheme [Mor71]:

$$\mathrm{F}(x,y) \Leftarrow \mathit{if}\ x = 0\ \mathit{then}\ 0\ \mathit{else}\ \mathrm{F}(x-1, \mathrm{F}(x,y))\ \mathit{fi}$$

Its least fixed point is the two-argument constant zero function, but if F(1,0) is computed using the *LI* or *PI* rules, the computation runs forever.

The computation sequences for *LI* and *PI* happen to be the same; this sequence is illustrated below. The function variables that are about to be expanded are underlined.

$$\underline{\mathrm{F}}(1,0) \Longrightarrow \mathrm{F}(0,\underline{\mathrm{F}}(1,0)) \Longrightarrow \mathrm{F}(0,\mathrm{F}(0,\underline{\mathrm{F}}(1,0))) \Longrightarrow \cdots$$

This is not a reason for never using *LI* or *PI*; it just points out that the semantics that we have chosen does not accurately model those computation rules (or, conversely, that the computation rules are not successful at evaluating to the values which the semantics assigns to some terms).

In section 4.1, page 35, we will present a condition called *safety* which, if true of a computation rule, guarantees that the rule computes the least fixed point of a recursive program scheme.

3.6 Syntactic properties of computation rules

This section proves a number of ancillary theorems about computation rules and their associated substitutions. This all leads up to theorems which relate the operational meanings of programs under various computation rules to each other and to their denotational meanings.

Definition 25. A *fixpoint computation rule* is a computation rule which computes the least fixed point of the function to which it is applied; that is, $[\![C_R]\!] = f_R$.

Let u be an arbitrary term in the computation sequence of recursive program scheme R on $(c_1, c_2, \ldots, c_{\mathrm{ar}(F_m)})$; suppose it contains j occurrences $\mathrm{F}^1, \ldots, \mathrm{F}^j$ of function variables. The superscripts serve only to distinguish the individual occurrences of function variables and do not necessarily indicate any ordering on the occurrences (for instance, their order of appearance in u).

Let C be a computation rule, and suppose that it chooses to substitute for $\mathrm{F}^1, \ldots, \mathrm{F}^i$ in u. Then u will be written $\alpha_C(F^{1\ldots i}, F^{i+1\ldots j})(c_1, c_2, \ldots, c_{\mathrm{ar}(F_m)})$ to distinguish the two sorts of occurrences of F in its body: those which will be substituted in the next step and those which will not. This notation will also be used to specify substitutions; for instance, $\alpha_C(G, H)(c_1, c_2, \ldots, c_{\mathrm{ar}(F_m)})$ is the term resulting when $F^1, \ldots, F^i$ in u are replaced by G and $F^{i+1}, \ldots, F^j$ in u are replaced by H. The distinction between substitution for function variables and expansion of applications has been blurred by the use of term constructing functions for describing terms. The substitutions are of term constructors for term constructors; the effect on the term denoted by the expression is expansion. While the division of F's into the two groups in α_C is usually according to whether a substitution rule selects them, it may be done arbitrarily as well, so that substitutions on some other basis can be specified. $\alpha_C(F^{1\ldots i}, F^{i+1\ldots j})$ is a function of type $\mathcal{D}_{F_m}$; the particular term to which $\alpha_C(F^{1\ldots i}, F^{i+1\ldots j})(c_1, c_2, \ldots, c_{\mathrm{ar}(F_m)})$ corresponds will be specified for each use of the α_C notation and will sometimes be indicated by context. If no computation rule subscript appears on α, that indicates that no computation rule was necessarily used to determine the division of function variable arguments in the term.

Let us formalize α_C by giving its type. It takes as inputs two ordered sets of term constructors (a term constructor is a function from term^k to term) and returns a term constructor. When we specify only a single term constructor instead of a set of constructors, we mean for that argument to α_C to be the ordered set of the appropriate size all of whose elements are the specified constructor.

α_C is specialized for a particular term and computation rule; we can think about a more general function A which is an α constructor. $\mathrm{A} : C \to \text{term} \to (tcs_1 \times tcs_1) \to \text{term}$ is a function which, given a computation rule, a term, and two sets of term constructors, returns the original term with those instances of function variables chosen by C replaced by the elements of tcs_1 and those which are not selected by C replaced by elements of tcs_2. This can be fairly simply implemented.

In the following definition (and elsewhere), $\circ$ denotes functional composition; $(f \circ g)(x) = f(g(x))$.

Definition 26. The *computation path* for computation rule C on program R and input $(c_1, c_2, \ldots, c_{\mathrm{ar}(F_m)})$ is the series of terms $S^i_{C,R}(c_1, c_2, \ldots, c_{\mathrm{ar}(F_m)})$ where $S^i_{C,R} : \text{term}^{\mathrm{ar}(F_m)} \to$ term is defined inductively: $S^0_{C,R} = F_m$ and $S^{i+1}_{C,R} = S_{C,R} \circ S^i_{C,R}$ for all $i \geq 0$.

To get a better feel for what $S^i_{C,R}$ is, let us work through its definition. The first few such functions are:

$$S^0_{C,R} = F_m = \tau_m[F_{1\ldots n}] = \alpha_C(F^{1\ldots i}, F^{i+1\ldots j})$$
$$S^1_{C,R} = S_{C,R} \circ S^0_{C,R} = \alpha_C(\tau_{l_{1\ldots i}}[F_{1\ldots n}], F^{i+1\ldots j}) = \alpha_C(\underbrace{\alpha_C(F^{1\ldots i'_k}, F^{i'_k+1\ldots j'_k})}_{\text{for } k=1 \text{ to } i}, F^{i+1\ldots j})$$
$$\vdots$$

where $l_{1\ldots i}$ are the subscripts on $F^{1\ldots i}$ and τ_{l_k} has j'_k instances of function variables, i'_k of which are substituted for by the second step of the computation of C on R.

In other words, each $S^i_{C,R}$ is the function which, given a vector of input terms $\vec{c}_{\mathrm{ar}(F_m)}$, returns the term which would result after i instances of simultaneous application of the substitution rule to $\mathrm{F}(\vec{c}_{\mathrm{ar}(F_m)})$ under the direction of computation rule C. At each step of the computation path some occurrences of F_j are replaced by $\tau_j[F_{1\ldots n}]$. $S^i_{C,R}(c_1, c_2, \ldots, c_{\mathrm{ar}(F_m)})$ is not the same as the i th term in the computation sequence of $(c_1, c_2, \ldots, c_{\mathrm{ar}(F_m)})$ because $S_{C,R}$ does no simplification; $S^i_{C,R}$ and C^i_R have operationally and denotationally identical meanings, however. This follows from the Church-Rosser property (fact 4): it does not matter when simplifications are done, for the operational semantics will determine the same meaning for a term regardless of the order in which its operations are performed. The order in which steps of $\Longrightarrow$ (substitution and simplification steps) are taken may, however, affect the number of them required in order to drive a term to base type. While $S^i_{C,R}$ is not quite $(S_{C,R})^i$, it is close: $S^i_{C,R} = (S_{C,R})^i \circ S^0_{C,R}$.

Definition 27 (ordering of terms). For terms A and B, $A \sqsubseteq_t B$ if A and B are the same except that wherever A contains a function variable applied to arguments, B may contain some expansion of that application. Stated inductively, this definition says

- For all terms A, $A \sqsubseteq_t A$.
- For all function variables F_i and terms s_j, $F_i(\vec{s}_{\mathrm{ar}(F_i)}) \sqsubseteq_t \tau_i[F_{1\ldots n}](\vec{s}_{\mathrm{ar}(F_i)})$.
- For all function variables and function, predicate, and built-in symbols ϕ, $\phi(s_1, s_2, \ldots, s_{\mathrm{ar}(\phi)}) \sqsubseteq_t \phi(s'_1, s'_2, \ldots, s'_{\mathrm{ar}(\phi)})$ if for all i, $s_i \sqsubseteq_t s'_i$.

$\sqsubseteq_t$ is monotonic and transitive.

Fact 28. For *any* separation of function variables (that is, regardless of the criteria used for separating the occurrences of function variables into the two ordered sets demanded by α) in a term $t = \alpha(F^{1\ldots i}, F^{i+1\ldots j})(c_1, c_2, \ldots, c_{\mathrm{ar}(F_m)})$, α is monotonic in each of its occurrences of function variables; for instance, $G \sqsubseteq G'$ implies $\alpha(G, H) \sqsubseteq \alpha(G', H)$ and $\alpha(H, G) \sqsubseteq \alpha(H, G')$ for all G, G', and H of the appropriate type.

Lemma 29. Expanding a term (substituting for it) causes it to become strictly greater under $\sqsubseteq_t$. Thus, the computation path for any computation rule C is a chain.

Proof. This follows from fact 28 and definitions 26 and 27:
$i \leq j$ implies $S^i_{C,R}(c_1, c_2, \ldots, c_{\mathrm{ar}(F_m)}) \sqsubseteq_t S^j_{C,R}(c_1, c_2, \ldots, c_{\mathrm{ar}(F_m)})$. ■

Lemma 30. If $A \sqsubseteq_t B$, then $S_{FS,R}(A) \sqsubseteq_t S_{FS,R}(B)$.

Proof. We will consider the occurrences of F_i's in A. We only need to consider such terms because for every other sort of term (for constants and for function, predicate, and built-in symbol applications), $S_{FS,R}$ is the identity function.

To each such occurrence there corresponds in B either a matching occurrence of F_i or an occurrence of $\tau_i[F_{1\ldots n}]$, possibly with some of its F's expanded as well. There are two cases, depending on what term in B an occurrence of $\mathrm{F}_i(\vec{s}_{\mathrm{ar}(\mathrm{F}_i)})$ in A corresponds to.

1. An occurrence of $F_i(\vec{s}'_{\mathrm{ar}(F_i)})$ matches, where $s_j \sqsubseteq s'_j$. These subterms of A and B will expand identically, and in the corresponding slots of the expanded terms will be $\tau_i[F_{1\ldots n}](\vec{s}_{\mathrm{ar}(\mathrm{F}_i)})$ and $\tau_i[F_{1\ldots n}](\vec{s}'_{\mathrm{ar}(F_i)})$; these terms fall under the $\sqsubseteq_t$ relation.

2. An expansion B' of $F_i(\vec{s}'_{\mathrm{ar}(F_i)})$ matches. Let A' be the single expansion of $F_i(\vec{s}'_{\mathrm{ar}(F_i)})$ in A resulting from the application of $S_{FS,R}$. $A' \sqsubseteq_t B'$, for B' is simply A', perhaps with further expansions. Let B'' be the expansion of B' performed by one full substitution step. $B' \sqsubseteq_t B''$, so by transitivity, $A' \sqsubseteq_t B''$. ■

It is *not* true in general that $A \sqsubseteq_t B \Rightarrow S_{C,R}(A) \sqsubseteq_t S_{C,R}(B)$. Consider the computation rule O, which substitutes for the first, third, fifth, etc. occurrences (*i.e.*, the odd occurrences) of F in a term, acting on a recursive program scheme which includes the equation $\mathrm{F}(x,y) \Leftarrow \mathrm{H}(\mathrm{H}(y))$. Let $A = \mathrm{F}(\mathrm{F}(c_1,c_2),\mathrm{F}(c_1,c_2))$ and $B = \mathrm{F}(\mathrm{H}(\mathrm{H}(c_1)),\mathrm{F}(c_1,c_2))$.
Then $A \sqsubseteq_t B$ but $S_{O,R}(A) \not\sqsubseteq_t S_{O,R}(B)$:

$$\begin{array}{rcl} \mathrm{F}(\mathrm{F}(c_1,c_2),\mathrm{F}(c_1,c_2)) & \sqsubseteq_t & \mathrm{F}(\mathrm{H}(\mathrm{H}(c_1)),\mathrm{F}(c_1,c_2)) \\ \mathrm{H}(\mathrm{H}(\mathrm{H}(\mathrm{H}(c_2)))) & \not\sqsubseteq_t & \mathrm{H}(\mathrm{H}(\mathrm{F}(c_1,c_2))) \end{array}$$

O seems odd since it uses global information in deciding which occurrences of F to expand; this is something that none of the sample computation rules given on page 21 do. Nevertheless, O is a valid computation rule. In fact, because it is a safe computation rule (see section 4.1 on page 35), use of it computes the same value for any input as does the denotational meaning of the program, which indicates that it is a reasonable rule to use.

The reason that this substitution failed to preserve the ordering on terms was that while the same rule was being applied to the two terms, the substitution did not necessarily affect corresponding subterms in the same way. "Corresponding subterms" are those which occupy the same relative positions in their respective terms [*i.e.*, are located by the same occurrence]; if $A \sqsubseteq_t B$, then a subterm of A is $\sqsubseteq_t$ the corresponding subterm of B. Let us consider the case of two corresponding subterms $A' = \mathrm{F}_i(s_1, s_2, \ldots, s_{\mathrm{ar}(F_i)})$ and B' in A and B, respectively. [Why restriction on form of A': well, this does not work without it...] If they are not the same term (*i.e.*, if $A' \sqsubset_t B'$), then regardless of the substitutions performed on A and B, the terms replacing A' and B' fall under the $\sqsubseteq_t$ relation: the former is $\mathrm{F}_i(s_1, s_2, \ldots, s_{\mathrm{ar}(F_i)})$ or $\tau_i[F_{1\ldots n}](s_1, s_2, \ldots, s_{\mathrm{ar}(F_i)})$ and the latter is $\sqsupseteq_t \tau_i[F_{1\ldots n}](s_1, s_2, \ldots, s_{\mathrm{ar}(F_i)})$ since $B' \sqsupseteq_t \tau_i[F_{1\ldots n}](s_1, s_2, \ldots, s_{\mathrm{ar}(F_i)})$. If, on the other hand, $A' = B'$, then difficulties may arise. No complications are encountered if both, neither, or only B' is substituted for; the bad case is when $A' = B'$ and only A' is expanded for; then for these subterms the $\sqsubseteq_t$ relation becomes $\sqsupseteq_t$ and $S_{C,R}(A) \sqsubseteq_t S_{C,R}(B)$ fails.

Why would a computation rule choose to substitute for an F in A but not the corresponding one in $B \sqsupseteq_t A$? This can only be the case if the computation rule uses some criterion other than the containing and contained expressions (the environment or context of the F in question). [[Do I want to say this?] Of the computation rules introduced on page 21, PO depends on the containing expressions and LI, PI, and FA depend on the contained expressions. There is no nontrivial condition upon which the substitutions done by rules FS and R depend, and L never makes the problematic substitution [say why].] Since the expressions correspond, their containing expressions are identical, and because $A' = B'$, their contained expressions are also the same. So some external factor (such as the overall structure of the term) must be at work if F in A' is to be substituted for but the F in B' is not.

A related topic is the appropriateness of our choice of F_Ω, which always reduces to itself (see page 16). We can now see why it was an appropriate choice for the nonterminating function in our recursive program schemes; it was initially introduced without justification. We can imagine other syntactic analogs for Ω which denote nontermination of a computation; for instance, we might use some other function such as $\mathrm{F}_r(s) \Leftarrow \mathrm{F}_r(\mathrm{F}_r(s))$, or we might use a distinguished non-base constant that does not reduce to anything. Both of these alternatives are unacceptable, however, not because of how they reduce (for neither reduces to a base constant), but because of how they may cause other terms around them to reduce. If the distinguished nontermination element did not reduce to anything under $\Longrightarrow_{\mathcal{I},R}$, then a substitution rule acting on a term containing it would have to choose some other (non-nontermination) element to reduce; that might have an effect on the computation. (F_Ω is like a nontermination element that *does* reduce, but to itself.) We could not use F_r or some other nontermination element that reduces to something besides itself (though a pair of elements such that each reduced to the other would be acceptable) because that too might change the behavior of the computation rule on other terms. The odd computation rule O (page 30) is an example of a computation rule for which whether an F_i is substituted for depends on more than just the context (the ancestors and children) of the F_i in question. It is essential, then, that the syntactic analog to Ω can get substituted for by computation rules and that it does not affect what *other* function variables are expanded.

Theorem 31. For all terms A and B in the computation path of C on R, if $A \sqsubseteq_t B$, then $S_{C,R}(A) \sqsubseteq_t S_{C,R}(B)$.

Proof. Consider $A' = S_{C,R}(A)$. If $A = B$, then the result is obvious. If $A \neq B$, then $A' \sqsubseteq_t B$. $B \sqsubseteq_t S_{C,R}(B)$, so $A' \sqsubseteq_t S_{C,R}(B)$. ∎

This theorem is even stronger than its statement above; theorem 31 is true for any $A \sqsubseteq_t B$ where B is in a computation path which contains A (but which does not necessarily start at $\mathrm{F}_m(\widehat{d}_1, \widehat{d}_2, \ldots, \widehat{d}_{\mathrm{ar}(F_m)})$). This reformulation of the theorem allows "computation sequences" which start at arbitrary terms (terms that might not be possible to compute to from any input vector of base constants) rather than requiring the sequence to start at $\mathrm{F}_n(c_1, c_2, \ldots, c_{\mathrm{ar}(F_n)})$. This corresponds to the alternate definition of computation sequence which allows computation sequences to start at arbitrary terms.

Lemma 32. Let A be a term, R be a recursive program scheme, C be a computation rule, and FS be the full-substitution rule. $S_{C,R}(A) \sqsubseteq_t S_{FS,R}(A)$.

Proof. $S_{C,R}(A)$ and $S_{FS,R}(A)$ will be identical except that in some places in which $S_{C,R}(A)$ has $\mathrm{F}_i(\widehat{d}_1, \widehat{d}_2, \ldots, \widehat{d}_{\mathrm{ar}(F_i)})$, $S_{FS,R}(A)$ has $\tau[F_{1\ldots n}](\widehat{d}_1, \widehat{d}_2, \ldots, \widehat{d}_{\mathrm{ar}(F_i)})$. These are precisely the instances of F in A not expanded by $S_{C,R}$. By the definition of $\sqsubseteq_t$, $S_{C,R}(A) \sqsubseteq_t S_{FS,R}(A)$. ■

3.7 Semantic properties of computation rules

Lemma 33. $A \sqsubseteq_t B$ implies $[\![A]\!]_{\mathrm{op}} \sqsubseteq [\![B]\!]_{\mathrm{op}}$.

It is not the case that $A \sqsubseteq_t B$ implies $[\![A]\!]_{\mathrm{op}} = [\![B]\!]_{\mathrm{op}}$ because B can be an expansion of A achieved through some expansion of the copy rule. For instance, consider the program scheme $\mathrm{F}_1(x_1) \Leftarrow$ *if* $x = 0$ *then* 0 *else* $\mathrm{F}_1(x_1 - 1)$ *fi*. $\mathrm{F}_1(0) \sqsubseteq_t$ *if* $0 = 0$ *then* 0 *else* $\mathrm{F}(-1)$ *fi* and $[\![\mathrm{F}_1(0)]\!]_{\mathrm{op}} = \bot \sqsubseteq [\![\textit{if } 0 = 0 \textit{ then } 0 \textit{ else } \mathrm{F}(-1) \textit{ fi}]\!]_{\mathrm{op}} = 0$.

Proof. Suppose that $A \sqsubseteq_t B$; we will show that $[\![A]\!]_{\mathrm{op}} \sqsubseteq [\![B]\!]_{\mathrm{op}}$. If $[\![A]\!]_{\mathrm{op}} = \bot$, the result is immediate.

We now consider the case of $[\![A]\!]_{\mathrm{op}} = d$. Simplification steps have no effect on F's; on the other hand, function symbols whose arguments are free of F's (the only kind that $\Longrightarrow_{\mathcal{I}}$ can make progress on) are not affected by the copy rule. So the same simplification steps that took A to $\widehat{d}$ can take B to $\widehat{d}$; simplifications of subterms not appearing in A are immaterial. [Further, if we start doing simplification steps on B, then since it is finite size, we are sure to eventually get around to doing the simplification steps that reduced A to $\widehat{d}$.] Thus, $[\![A]\!]_{\mathrm{op}} = d$ implies that $[\![B]\!]_{\mathrm{op}} = d$. ■

Theorem 34. For any program R, any computation rule C, and the full-substitution computation rule FS, $[\![C_R]\!] \sqsubseteq [\![FS_R]\!]$.

Proof. We will show by induction on i that for every i, $S^i_{C,R} \sqsubseteq S^i_{FS,R}$; that is, the computation path for C is elementwise less defined than that of FS. This implies the theorem via lemma 33.

Basis: $S^0_{C,R} \sqsubseteq S^0_{FS,R}$ because $S^0_{C,R} = S^0_{FS,R} = F_m$.

Induction step: Assume $S^i_{C,R} \sqsubseteq S^i_{FS,R}$ and show that $S^{i+1}_{C,R} \sqsubseteq S^{i+1}_{FS,R}$.

The F's substituted by $S_{C,R}$ may depend on its argument, while $S_{FS,R}$ always substitutes for all F's in its argument. By lemma 32, $S_{C,R} \circ S^i_{C,R} \sqsubseteq S_{FS,R} \circ S^i_{C,R}$. By lemmas 32 and 30, $S_{FS,R} \circ S^i_{C,R} \sqsubseteq S_{FS,R} \circ S^i_{FS,R}$. By transitivity of $\sqsubseteq$, $S_{C,R} \circ S^i_{C,R} \sqsubseteq S_{FS,R} \circ S^i_{FS,R}$. This implies that $S^{i+1}_{C,R} \sqsubseteq S^{i+1}_{FS,R}$. ■

3.8 Adequacy for the full-substitution computation rule

This section partially describes the relationship between $[\![C_R]\!]$, the function computed by computation rule C, and the fixpoint f_R. We will show that $[\![C_R]\!] \sqsubseteq f_R$ and, further, that $[\![FS_R]\!] = f_R$. [Also see section 4.3, Adequacy.]

[Complete this proof.]

Lemma 35. For all finite $j \geq 1$, $S^j_{FS,R} = \tau^j_m[F_{1\ldots n}]$.

Proof. The proof is by induction.
Base case: $j = 1$.
$FS_R^1 = F_m$
$\tau_m^1[F_{1\ldots n}] = F_m$
Inductive case: assume $FS_R^j = \tau_m^j[F_{1\ldots n}]$. $\tau^j[F_{1\ldots n}]$ is a tuple of n functions, the i^{th} of which ($\tau_i^j[F_{1\ldots n}]$), given $\vec{s}_{\mathrm{ar(F}_i)}$ as its arguments, returns a term corresponding to the j^{th} expansion of the term $\mathrm{F}_i(\vec{s}_{\mathrm{ar(F}_i)})$.

$S_{FS,R}^{j+1}(\vec{s}_{\mathrm{ar(F}_m)}) = S_{FS,R}(S_{FS,R}^j(\vec{s}_{\mathrm{ar(F}_m)}))$ is the term resulting from expanding once each function in $S_{FS,R}^j(\vec{s}_{\mathrm{ar(F}_m)})$, the j^{th} expansion of $\mathrm{F}_m(\vec{s}_{\mathrm{ar(F}_m)})$.

$\tau_m^{j+1}[F_{1\ldots n}](\vec{s}_{\mathrm{ar(F}_m)}) = \tau_m(\tau^j[F_{1\ldots n}])(\vec{s}_{\mathrm{ar(F}_m)})$ is the term resulting from replacing each instance of F_i in $\tau_m[F_{1\ldots n}]$ with $\tau_i^j[F_{1\ldots n}]$, the j^{th} expansion of F_i. [cannot really have an expansion of a variable, sorta]

If full expansion is done, then there is no difference between first expanding once, then expanding j times and first expanding j times, then expanding once. In other words, at this step $S_{FS,R}$ substitutes into an expression which has already been expanded j times, while τ substitutes j^{th} generation expansion functions into a simple expression. ■

Say why only works for finite j: cannot make a term of infinite size, cannot run OS for an infinite number of steps.

Theorem 36. For all finite $j \geq 1$,

$$[\![\tau_m^j[F_{1\ldots n}](\widehat{d}_1, \widehat{d}_2, \ldots, \widehat{d}_{\mathrm{ar}(F_m)})]\!]_{\mathrm{op}} = [\![\tau_m]\!]^j[\Omega, \Omega, \ldots, \Omega](d_1, d_2, \ldots, d_{\mathrm{ar}(F_m)})$$

Proof. We will prove this theorem by induction on j simultaneously for each m. Let $k = \mathrm{ar}(F_m)$.
Base case: $j = 1$.

$$\begin{aligned}
[\![\tau_m[F_{1\ldots n}](\widehat{d}_1, \widehat{d}_2, \ldots, \widehat{d}_k)]\!]_{\mathrm{op}} &= [\![(t_m)_{\vec{x}_k}^{\vec{\widehat{d}}_k}]\!]_{\mathrm{op}} \\
&= \mathcal{I}_\Omega((t_m)_{\vec{x}_k}^{\vec{\widehat{d}}_k})
\end{aligned}$$

$$\begin{aligned}
[\![\tau_m]\!][\Omega, \Omega, \ldots, \Omega](d_1, d_2, \ldots, d_k) &= \mathcal{I}((t_m)_{\vec{x}_k}^{\vec{\widehat{d}}_k})(\cdot[\mathrm{F}_i \mapsto \Omega]) \\
&= \mathcal{I}_\Omega((t_m)_{\vec{x}_k}^{\vec{\widehat{d}}_k})
\end{aligned}$$

Inductive case: Assume $[\![\tau_i^j[F_{1\ldots n}](d_1, d_2, \ldots, d_{\mathrm{ar}(F_i)})]\!]_{\mathrm{op}} = [\![\tau_i]\!]^j[\Omega, \Omega, \ldots, \Omega](d_1, d_2, \ldots, d_{\mathrm{ar}(F_i)})$ and let $g_i = [\![\tau_i]\!][\Omega, \Omega, \ldots, \Omega]$.

$$\begin{aligned}
[\![\tau_m^{j+1}[F_{1\ldots n}](\widehat{d}_1, \widehat{d}_2, \ldots, \widehat{d}_k)]\!]_{\mathrm{op}} &= [\![\tau_m(\tau^j[F_{1\ldots n}])(\widehat{d}_1, \widehat{d}_2, \ldots, \widehat{d}_k)]\!]_{\mathrm{op}} \\
&= [\![(\,(t_m)_{\tau_i^j[F_{1\ldots n}](\vec{s}_k)}^{\mathrm{F}_i(\vec{s}_k)})_{\vec{x}_k}^{\vec{\widehat{d}}_k}]\!]_{\mathrm{op}} \\
&= [\![(\,(t_m)_{\vec{x}_k}^{\vec{\widehat{d}}_k})_{\tau_i^j[F_{1\ldots n}](\vec{s}_k)}^{\mathrm{F}_i(\vec{s}_k)}]\!]_{\mathrm{op}}
\end{aligned}$$

$$
\begin{aligned}
&= \llbracket \left((t_m)^{\hat{\vec{d}}_k}_{\vec{x}_k} \right)^{\mathrm{F}_i(\vec{s}_k)}_{\llbracket \tau_i^j [F_{1\ldots n}](\vec{s}_k) \rrbracket_{\mathrm{op}}} \rrbracket_{\mathrm{op}} \\
&= \mathcal{I}((t_m)^{\hat{\vec{d}}_k}_{\vec{x}_k})(\cdot[\mathrm{F}_i \mapsto g_i^j])
\end{aligned}
$$

$$
\begin{aligned}
\llbracket \tau_m \rrbracket^{j+1}[\Omega, \Omega, \ldots, \Omega](d_1, d_2, \ldots, d_k) &= \llbracket \tau_m \rrbracket(\llbracket \tau \rrbracket^j [\Omega, \Omega, \ldots, \Omega])(d_1, d_2, \ldots, d_k) \\
&= \llbracket \tau_m \rrbracket[g_1^j, \ldots, g_n^j](d_1, d_2, \ldots, d_k) \\
&= \mathcal{I}((t_m)^{\hat{\vec{d}}_k}_{\vec{x}_k})(\cdot[\mathrm{F}_i \mapsto g_i^j]) \quad \blacksquare
\end{aligned}
$$

[Add lemma to justify the step above which nests ops.]

Theorem 37. The full-substitution computation rule is a fixpoint computation rule.

We would like to prove this theorem by making a connection between the operational and denotational semantics for the full-substitution computation rule; this connection, known as *adequacy*, will be described more generally in section 4.3. Unfortunately, the connection is difficult to make in the monotone case.

From corollary 23, the fixed point in question is $\mu\llbracket\tau\rrbracket = \llbracket\tau\rrbracket^\gamma[\Omega, \Omega, \ldots, \Omega]$ for some sufficiently large ordinal γ. From lemma 35 and theorem 36 we know that for finite $j \geq 1$, $\llbracket S^j_{FS,R}(d_1, d_2, \ldots, d_{\mathrm{ar}(F_m)}) \rrbracket_{\mathrm{op}} = \llbracket \tau_m \rrbracket^j [\Omega, \Omega, \ldots, \Omega](d_1, d_2, \ldots, d_{\mathrm{ar}(F_m)})$. All that remains is to close the gap between the finite j and the possibly transfinite γ.

It turns out that while γ is transfinite, it is not very transfinite; that is, $\gamma = \omega$, the first transfinite number. There is not any obvious *a priori* reason that this must be the case. We can imagine monotone functions from D to D which we would have to apply more than ω times to $\perp_D$ in order to reach a fixed point [Give an example here; in fact, it would be best if it had the type of τ.], but no τ that can be defined via a recursive program scheme is one of them. Every such τ is both monotone and continuous. τ does not "fully use" its type; the restricted rules for creating recursive program schemes ensure that all functions computed by them are first-order. Since there is no functional abstraction and no higher-order abstraction permitted, all arguments and results are of base type. That the computed functions are also monotone is dependent upon the model used in interpreting them. Another of these properties is that $\tau[\Omega, \Omega, \ldots, \Omega]$'s fixed point is reached after only ω applications of it to $\perp$.

Chapter 4

Semantical Connections

This chapter investigates the relation between operational and denotational semantics which has already been alluded to. We will show that for a certain class of computation rules, the computed function is the same as the least fixed point of τ; that is, the denotational and operational meanings are the same.

4.1 Safety for computation rules

Safety is a condition on computation rules which implies that the rule is a fixpoint computation rule. First, however, let us note that no computation rule is more defined than the least fixed point of the program's τ. Since safety will imply the converse, it will follow that a safe computation rule computes the fixed point.

Corollary 38. Every computation rule yields a value which is $\sqsubseteq$ the fixpoint. In other words, for any computation rule C, $[\![C_R]\!] \sqsubseteq \mu[\![\tau]\!]$.

Proof. This corollary follows at once from theorems 34 and 37.

We could also make an argument from corollary 23 which built upon the argument in section 3.8: given a finite amount of time, the operational semantics can only run for a finite number of steps. Getting to the fixed point, however, may require γ, which is greater than any finite number, applications of $[\![\tau]\!]$. $[\![C_R]\!] \sqsubseteq \mu[\![\tau]\!]$ follows by monotonicity. ■

Let $[\![\alpha_C]\!]$ be the semantic function corresponding to the syntactic α_C.

$$[\![\alpha_C]\!](g^{1\ldots i}, g^{i+1\ldots j})(d_1, d_2, \ldots, d_{\mathrm{ar}(F_m)}) = \mathcal{I}(\alpha_C(F^{1\ldots i}, F^{i+1\ldots j})(\widehat{d}_1, \widehat{d}_2, \ldots, \widehat{d}_{\mathrm{ar}(F_m)}))(\cdot[\mathrm{F}_i \mapsto g_i])$$

Note that the g's have type $\mathcal{D}_{F_i}$, not D_{F_i}.

Let f^k be the fixed point of the semantic function corresponding to F^k in $\alpha_C(F^{1\ldots i}, F^{i+1\ldots j})$. $f^k = \mu_{l_k}[\![\tau]\!]$ where l_k is the subscript of F^k; in other words, this is the semantical meaning of F^k.

Definition 39. A *safe computation rule* C is one for which, for any recursive program scheme R and any term $u = \alpha_C(F^{1\ldots i}, F^{i+1\ldots j})(\widehat{d}_1, \widehat{d}_2, \ldots, \widehat{d}_{\mathrm{ar}(F_m)})$ in the computation sequence of C_R on $\widehat{d}_1, \widehat{d}_2, \ldots, \widehat{d}_{\mathrm{ar}(F_m)}$, $[\![\alpha_C]\!](\Omega, \Omega) = [\![\alpha_C]\!](\Omega, f^{i+1\ldots j})$.

In other words, if the occurrences F^K of function variables chosen by C at this step were not replaced by some expansion of their bodies but were instead replaced by (the syntactic analog of) Ω, then it would not matter how little (Ω) or how much (f^k) information was known about the nonsubstituted functions. We would get the same result if the function variables *not* substituted at this step were replaced by the least function, by their (respective) fixed points, or by anything in between.

Let us consider the operational meaning of the term u of definition 39, immediately above. $[\![u]\!]_{\text{op}}$ is either a base constant (the computation terminates) or $\perp$ (it does not).

If the computation terminates even though none of the functions substituted at this step do, then it is clear that those particular terms do not matter. The computation returns the same value regardless of their value, because $\Omega \sqsubseteq g$ for any $g : \mathcal{D}_{F_i}$ (in particular, $\Omega \sqsubseteq f^k$ for any k), so $[\![\alpha]\!](\Omega, \Omega) \sqsubseteq [\![\alpha]\!](\Omega, f^{i+1 \ldots j})$ by monotonicity. However, there is no value which is strictly more defined than a base value in $\mathcal{D}$, so

$$\begin{gathered}[\![\alpha]\!](\Omega,\Omega)(d_1, d_2, \ldots, d_{\operatorname{ar}(F_m)}) = d \sqsubseteq [\![\alpha]\!](\Omega, g^{i+1 \ldots j})(d_1, d_2, \ldots, d_{\operatorname{ar}(F_m)}) \\ \text{implies} \quad [\![\alpha]\!](\Omega, g^{i+1 \ldots j})(d_1, d_2, \ldots, d_{\operatorname{ar}(F_m)}) = d\end{gathered}$$

The monotonicity argument holds for any subset of the F's in a term, so in the case that $[\![\alpha_c]\!](\Omega,\Omega)(\widehat{d_1}, \widehat{d_2}, \ldots, \widehat{d}_{\operatorname{ar}(F_m)}) = d$, it does not matter which F's are substituted. This is obvious since the substituted and non-substituted functions are treated alike in $[\![\alpha]\!](\Omega,\Omega)$. Since $[\![\alpha_C]\!](\Omega,\Omega)(d_1, d_2, \ldots, d_{\operatorname{ar}(F_m)}) = [\![u]\!]_{\text{op}} = d$ and u is fully simplified (because it is a member of a computation sequence); $u = \widehat{d}$.

If, on the other hand, the computation does not terminate given that the substituted instances do not terminate, then it does not matter how much is known about the non-substituted instances. That $[\![\alpha]\!](\Omega,\Omega)(d_1, d_2, \ldots, d_{\operatorname{ar}(F_m)}) = \perp$ simply means that $\alpha(F^{1 \ldots i}, F^{i+1 \ldots j})(d_1, d_2, \ldots, d_{\operatorname{ar}(F_m)})$ does not simplify to a base constant; the use of any substitution rule in evaluation guarantees that the term means $\perp$ since no substituted function terminates. Although $[\![\alpha]\!](\Omega, g) = [\![\alpha]\!](\Omega,\Omega) = \Omega$, $[\![\alpha]\!](g, \Omega)$ may be better-defined than $[\![\alpha]\!](\Omega,\Omega)$.

A simpler way to think of this is to use the following lemma:

Lemma 40. For any ordered set G of functions of type $\mathcal{D}_{F_k}$ and for any term $u = \alpha(F^{1 \ldots i}, F^{i+1 \ldots j})(d_1, d_2, \ldots, d_{\operatorname{ar}(F_m)})$ in the computation path of safe computation rule C,

$$[\![\alpha]\!](\Omega, G)(d_1, d_2, \ldots, d_{\operatorname{ar}(F_m)}) = [\![\alpha]\!](\Omega,\Omega)(d_1, d_2, \ldots, d_{\operatorname{ar}(F_m)}) = [\![u]\!]_{\text{op}} \ .$$

Proof. The proof follows the reasoning of the previous paragraph. ■

Another way to think of this is via "expressions of interest." An expressions of interest is, intuitively, one that matters with respect to the value of the computation: if it were changed, then the value of the computation could. For now, the expressions of interest of u are the expressions that $[\![u]\!]_{\text{op}}$ is strict in. A rule that expands no expressions of interest is nonsafe. [I do not think that this will be quite true after parallel-or — think about consequences.] The definition of expression of interest will not change, but we will see that the strictness reformulation is not quite right.

It is easy to see that the only subexpressions of interest will be those resulting from an expansion of an expression of interest: the expansion of a noninteresting expression is noninteresting, and so are all of its subterms. [reword] This means that we can't expand a noninteresting term and get an interesting term which we might spend all of our time evaluating.

A related fact will be shown by claim 42.

Definition 41. A substitution for $F^{1\ldots i}$ is a *safe substitution* if, for a fully simplified term $u = \alpha(F^{1\ldots i}, F^{i+1\ldots j})(\widehat{d}_1, \widehat{d}_2, \ldots, \widehat{d}_{\mathrm{ar}(F_m)})$,

$$[\![\alpha]\!](\Omega, f^{i+1\ldots j})(d_1, d_2, \ldots, d_{\mathrm{ar}(F_m)}) = \bot$$

Note that safety on computation rules and safety on substitutions are two different properties, though they are strongly linked. We will show that a computation rule which uses safe substitutions is safe.

Claim 42. A safe substitution on a term $u = f(y_1, y_2, \ldots, y_{\mathrm{ar}(f)})$ is a safe substitution on at least one y_k.

Proof. Suppose the computation rule C with associated substitution rule $S_{C,R}$ is used on $f(y_1, y_2, \ldots, y_{\mathrm{ar}(f)})$. $S_{C',R}(y_k)$ will denote the term resulting from substitutions performed on y_k when $S_{C,R}$ is applied to u, which are not necessarily the same substitutions as would be performed on y_k if $S_{C,R}$ were directly applied to y_k. For instance, if the leftmost (call-by-name) computation rule L is used on the term $f(\mathrm{F}_1(\widehat{d}_1), \mathrm{F}_2(\mathrm{F}_3(\widehat{d}_2)))$, then $S_{L',R}$ for $y_2 = \mathrm{F}_2(\mathrm{F}_3(\widehat{d}_2))$ does no substitutions, though F_1 is substituted for in $y_1 = \mathrm{F}_1(\widehat{d}_1)$; were $S_{L,R}$ applied to y_2 directly, then F_2 would be substituted for.

Let

$$u = \alpha_C(F^{1\ldots i}, F^{i+1\ldots j})(\widehat{d}_1, \widehat{d}_2, \ldots, \widehat{d}_{\mathrm{ar}(F_m)}) \ ,$$
$$y_k = \alpha_{C'}(F^{1\ldots i'}, F^{i'+1\ldots j'})(\widehat{d}_1, \widehat{d}_2, \ldots, \widehat{d}_{\mathrm{ar}(F_m)}) \ .$$

If the substitution is safe, then $[\![\alpha_C]\!](\Omega, f^{i+1\ldots j})(d_1, d_2, \ldots, d_{\mathrm{ar}(F_m)}) = \bot$; *i.e.*, $[\![u]\!]_{\mathrm{op}} = \bot$.

If for all k, $[\![\alpha_{C'}]\!](\Omega, f^{i'+1\ldots j'})(d''_1, d''_2, \ldots, d''_{\mathrm{ar}(f)}) = d'_k$, then the substitution $S_{C',R}$ was not safe on any y_k; in this case, $[\![u]\!]_{\mathrm{op}} = f(d'_1, d'_2, \ldots, d'_{\mathrm{ar}(f)}) = d$ since all of its arguments are fully evaluated and thus $S_{C,R}$ was not safe on u.

If for some k, $[\![\alpha_{C'}]\!](\Omega, f^{i'+1\ldots j'})(d''_1, d''_2, \ldots, d''_{\mathrm{ar}(f)}) = \bot$, then $S_{C',R}$ was safe on y_k and $[\![u]\!]_{\mathrm{op}} = \bot$ since f is strict in its arguments. These two alternatives are exhaustive. ■

[Is this clear?]

Claim 42 is quite intuitive; this paragraph is a highly informal description of one case that might be disturbing. The only subtlety is that if $y_k = \ldots \mathrm{F}_i(\ldots \mathrm{F}_j \ldots) \ldots$, then $S_{C',R}$ might expand the F_j instead of the F_i. If $S_{C,R}$ is safe and the computation of F_j does not halt, then neither will F_i since $S_{C',R}$ will never get around to expanding F_i (it will never finish evaluating F_j). If F_j does halt, then (since we know that F_i must not be substituted for lest the evaluation of y_k halt) some F_l in F_i which does not halt must be substituted for; *i.e.*, F_j was the wrong y' in y_k to be considering, if y_k is the correct (nonterminating subterm of u to be considering. If no such y' exists then F_i may be expanded if there are no other function symbols in y_k, so $S_{C',R}$ is not safe on y_k.

Theorem 43. Any computation rule which performs only safe substitutions is safe.

Proof. This follows from lemma 40 and the definitions of safety for computation rules and for substitutions. ■

Lemma 44. A safe computation rule uses only safe substitutions.

Proof. Suppose that C is a safe computation rule which at some step makes a nonsafe substitution for $u_k = \alpha_C(F^{1...i}, F^{i+1...j})(d_1, d_2, \ldots, d_{\mathrm{ar}(F_m)})$. Then for some d,

$$[\![\alpha_C]\!](\Omega, f_p^{i+1...j})(d_1, d_2, \ldots, d_{\mathrm{ar}(F_m)}) = d\,.$$

Lemma 40 implies that

$$[\![\alpha_C]\!](\Omega, \Omega)(d_1, d_2, \ldots, d_{\mathrm{ar}(F_m)}) = \bot\ .$$

$[\![\alpha_C]\!](\Omega, f_p^{i+1...j}) \neq [\![\alpha_C]\!](\Omega, \Omega)$ contradicts the assumption that C is a safe computation rule. ■

4.2 Safety implies fixpoint computation

Theorem 45. Any safe computation rule is a fixpoint computation rule.

Proof. We will prove this theorem by considering some safe computation rule C and the full-substitution computation rule FS and showing that they compute the same values. The main idea of this proof is that any rule eventually (by step m) does as much work as the full-substitution rule did by step n. That part of the proof is by contradiction: if they do not compute the same values, then C is not safe. We know from theorem 37 that FS computes the least fixed point, so it will follow that C also computes the least fixed point and so is a fixpoint computation rule.

Suppose $C \neq FS$, so for some recursive program scheme R there is a vector of base (nonbottom) constants $\vec{d}_{\mathrm{ar}(F_m)} = d_1, d_2, \ldots, d_{\mathrm{ar}(F_m)}$ such that $[\![C_R]\!](\vec{d}_{\mathrm{ar}(F_m)}) \neq [\![FS_R]\!](\vec{d}_{\mathrm{ar}(F_m)})$. By theorem 34, $[\![C_R]\!] \sqsubseteq [\![FS_R]\!]$. Therefore, $[\![C_R]\!](\vec{d}_{\mathrm{ar}(F_m)}) = \bot$ and $[\![FS_R]\!](\vec{d}_{\mathrm{ar}(F_m)}) = d$ for some nonbottom d. Since $[\![C_R]\!](\vec{d}_{\mathrm{ar}(F_m)}) = \bot$, there are two possibilities for the computation sequence of C on $(\widehat{d}_1, \widehat{d}_2, \ldots, \widehat{d}_{\mathrm{ar}(F_m)})$: some term in it means $\bot$, or it is infinite. (The input vector had to be composed of nonbottom elements so that we could speak of the computation sequence on $(\widehat{d}_1, \widehat{d}_2, \ldots, \widehat{d}_{\mathrm{ar}(F_m)})$.) Each of these alternatives is inconsistent.

If some term means $\bot$, the contradiction is derived immediately. Consider replacing each term in the sequence by the appropriate application of $[\![\alpha_C]\!](f^{1...i}, f^{i+1...j})$, where f^k is the fixed point of F^k as defined on page 35. Each term is now equivalent to its predecessor in the sequence, while the first term means $F_m(\vec{d}_{\mathrm{ar}(F_m)}) = [\![C_R]\!](\vec{d}_{\mathrm{ar}(F_m)})$ and the last means $\bot$. This contradicts the assumption that C is safe.

The remaining case is that the computation sequence of C is infinite.

Let the *level* of an instance of F be its level or depth of its nesting with applications of τ_i; for instance, in $\tau_i[F_1, \tau_j[\tau_k[F_1, F_2], F_2]]$ the F's have level 1, 3, 3, and 2, respectively. Whenever $\tau_i[F_{1...n}]$ is substituted for a F_i of level n, the resulting occurrences of F are all of level $n+1$. This view of a term does not say anything about its syntactic structure, only its logical structure.

Let x be the number such that FS^x_R computes the fixed point of R; that is, for any input vector $\vec{d}_{\mathrm{ar}(F_m)}$, $f_R(\vec{d}_{\mathrm{ar}(F_m)}) = d$ iff $FS^x_R(\mathrm{F}_m(\vec{\hat{d}}_{\mathrm{ar}(F_m)})) = \hat{d}$. We will show that for some y, $FS^x_R \sqsubseteq C^y_R$. There are only finitely many instances of F of depth $\leq n$ in $C^k_R(\mathrm{F}_m(\vec{\hat{d}}_{\mathrm{ar}(F_m)}))$. Let y be the smallest k such that at step k, no instances of F of depth $\leq x$ are substituted for. In other words, in $C^y_R(\mathrm{F}_m(\vec{\hat{d}}_{\mathrm{ar}(F_m)})) = \alpha_C(F^{1\ldots i}, F^{i+1\ldots j})(\vec{\hat{d}}_{\mathrm{ar}(F_m)})$, each of $\mathrm{F}^{1\ldots i}$ is of depth $> x$. Now consider the α_C computed for the initial term of the computation sequence, $\mathrm{F}_m(\vec{\hat{d}}_{\mathrm{ar}(F_m)})$. The term $\alpha_C(F_\Omega, \tau^x_{l_{i+1\ldots j}}[F_{1\ldots n}])(\hat{d}_1, \hat{d}_2, \ldots, \hat{d}_{\mathrm{ar}(F_m)})$ has no instances of F_Ω of depth $\leq x$, so

$$\tau^x[F_{1\ldots n}] \sqsubseteq \alpha(F_\Omega, \tau^x_{l_{i+1\ldots j}}[F_{1\ldots n}])\,.$$

This implies that

$$[\![\tau^x]\!][F_{1\ldots n}] \sqsubseteq [\![\alpha_C]\!](\Omega, \tau^x_{l_{i+1\ldots j}}[F_{1\ldots n}]) \sqsubseteq [\![\alpha_C]\!](\Omega, f^{i+1\ldots j}) \sqsubseteq f_m\,.$$

Since $[\![\tau^x_{l_i}]\!][F_{1\ldots n}] \sqsubseteq f^i$ and α is monotonic with respect to all of its arguments,

$$[\![\alpha_C]\!](\Omega, [\![\tau^x_{l_{i+1\ldots j}}]\!][F_{1\ldots n}]) \sqsubseteq \alpha_C(\Omega, f^{i+1\ldots j})\,.$$

This implies by transitivity that

$$[\![\tau^x]\!][F_{1\ldots n}] \sqsubseteq [\![\alpha_C]\!](\Omega, f^{i+1\ldots j})\,.$$

But $[\![\tau^x_m]\!][F_{1\ldots n}] \neq \Omega$, so $[\![\alpha_C]\!](\Omega, f^{i+1\ldots j}) \neq \Omega$. This contradicts the assumption that C is a safe computation rule. ■

While safety is a sufficient condition for a computation rule to be a fixpoint computation rule, it is not a necessary condition. Consider a computation rule that alternates at each step between safe and unsafe substitutions. This rule is a fixpoint computation rule, for it will eventually compute the correct answer. If a safe computation rule which used only the safe substitutions took x steps to complete, then this unsafe rule will take less than $2n$ steps to complete.

In fact, we know that every computation which halts does so in some finite number of steps; further, there is an upper limit on the number of steps a safe computation rule can take in determining the value of a particular term. If we could guarantee that that a computation rule made at least that number of safe substitutions, then we would know that the computation rule computed the fixed point. This limit is finite for any term but unbounded in general.

Claim 46. A computation rule is a fixpoint computation rule iff it makes safe substitutions infinitely often.

Proof. $(\Leftarrow)$ Above.

$(\Rightarrow)$ For any finite number x there is a term such that FS takes x steps to compute its value. Any other computation rule will take at least as many steps to determine its value. If a computation rule does not make safe substitutions infinitely often, then there is some finite y such that it does not make that many safe substitutions on any term. Then the computation rule does not compute the correct value of the term and so is not a fixpoint computation rule. ■

The definition we really wanted for safety on computation rules, then is the weaker definition of doing safe substitutions infinitely often rather than at every step. Unfortunately, that definition is considerably harder to work with than the one that was given, so we shall stick with the first formulation.

4.3 Adequacy

Definition 47. *Adequacy* is a condition on models which relates the operational and denotational meanings of a term. In an adequate semantics, a term of base type means d iff it evaluates computationally to the numeral $\widehat{d}$; *i.e.*,

$$[\![\mathrm{F}_m(\widehat{d_1}, \widehat{d_2}, \ldots, \widehat{d}_{\mathrm{ar}(F_m)})]\!]_{\mathrm{op}} = \widehat{d} \quad \text{iff} \quad [\![R]\!]_{\mathrm{sem}}(d_1, d_2, \ldots, d_{\mathrm{ar}(F_m)}) = d$$

where $[\![R]\!]_{\mathrm{sem}}$ is the denotational meaning of the program (and of F_m).

An adequate semantics may make more distinctions than those definable by contexts in the language; another condition, full abstraction, makes fewer distinctions. The definition can be stated as

$$[\![\mathrm{F}_m(\widehat{d_1}, \widehat{d_2}, \ldots, \widehat{d}_{\mathrm{ar}(F_m)})]\!]_{\mathrm{op}} = \widehat{3} \quad \text{iff} \quad [\![R]\!]_{\mathrm{sem}}(d_1, d_2, \ldots, d_{\mathrm{ar}(F_m)}) = 3$$

for a particular base constant without loss of generality, since we only need make t' which is $t \pm \widehat{k}$ for some k to extend the result to any element of our numeric base domain.

Theorem 48. The monotone model is adequate for any safe computation rule.

We have already done enough work to prove the main theorem of this thesis. Recall that the least fixed point *is* the denotational meaning; then adequacy follows immediately from theorem 45 and we see that the computational and denotational rules match up.

Acknowledgments

I thank Albert Meyer for his support, not only in supervising this thesis but also in introducing me to the field of programming language semantics. He is an inexhaustible fount of wisdom and advice, and I have rarely met anyone who can tell me so kindly that I'm stupid.

I also thank Bard Bloom and Jon Riecke, who were not only always willing to answer my questions but also provided many useful comments on a draft of this thesis, correcting errors of fact and of exposition.

Bibliography

[Bar81] Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic*. North-Holland, 1981. Revised Edition, 1984.

[End77] Herbert B. Enderton. *Elements of Set Theory*. Academic Press, 1977.

[LS87] Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification*. Wiley-Teubner Series in Computer Science. John Wiley and Sons, 1987. Second Edition.

[Man74] Zohar Manna. *Mathematical Theory of Computation*. McGraw-Hill, 1974.

[Mey88] Albert R. Meyer. Semantical paradigms: Notes for an invited lecture, with two appendices by Stavros Cosmodakis. Technical Report MIT/LCS/TM353, MIT Lab. for Comp. Sci., July 1988.

[Mor71] J. H. Morris. Another recursion induction principle. *Communications of the ACM*, 14(5), 1971.

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–257, 1977.

[RV80] Jean-Claude Raoult and Jean Vuillemin. Operational and semantic equivalence between recursive programs. *Journal of the ACM*, 27:772–796, 1980.

[Sch88] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Wm. C. Brown, 1988.

[Tai75] William W. Tait. A realizability interpretation of the theory of species. In R. Parikh, editor, *Logic Colloqium, '73*, volume 453 of *Lect. Notes in Math*, pages 22–37. Springer-Verlag, 1975.

[Vel] Dan Velleman. Manuscript of Jan. 20, 1987 on the relation between monotone and continuous models.

[Vui73] Jean Vuillemin. Proof techniques for recursive programs. note de travail, Institut de Recherche d'Informatique et d'Automatique, Domaine de Voluceau, Rocquencourt, 78150 – Le Chesnay, June 1973.