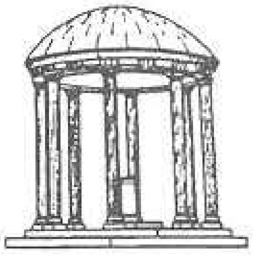
Practical Higher-Order Functional and Logic Programming Based on Lambda-Calculus and Set-Abstraction

TR88-004

January 1988
(Revised December 1988)

Frank S. K. Silbermann
Bharat Jayaraman

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175



Practical Higher-order Functional and Logic Programming based on Lambda Calculus and Set Abstraction†

Status Report

Frank S.K. Silbermann
Bharat Jayaraman

Department of Computer Science
University of North Carolina at Chapel Hill
Chapel Hill, NC 27514

Tel: (919) 962-1764
Arpanet: bj@cs.unc.edu

Abstract

This dissertation addresses the unification of functional and logic programming. We concentrate on the role of set abstraction in this integration. The proposed approach combines the expressive power of lazy higher-order functional programming with not only first-order Horn logic, but also a useful subset of higher-order Horn logic. Set abstractions take the place of Horn logic predicates. Like functions, they are first-class objects. The denotational semantics of traditional functional programming languages is based on an extended lambda calculus. Set abstractions are added to the semantic domain via angelic powerdomains, which are compatible with lazy evaluation and are well-defined over even non-flat (higher-order) domains. From the denotational equations we derive an equivalent operational semantics. A new computation rule more efficient than the parallel-outermost rule is developed and proven to be a correct computation rule for this type of language. Concepts from resolution are generalized to provide efficient computation of set abstractions. The implementation avoids computationally explosive primitives such as higher-order unification or general theorem-proving.

† *This research was supported by grant DCR-8603609 from the National Science Foundation and contract 00014-86-K-0680 from the Office of Navel Research.*

1. INTRODUCTION AND RELATED WORK

Declarative languages are noted for their simple and elegant semantics, being based on well-known mathematical theories. The use of declarative languages has been limited by expensive evaluation procedures, but as the ratio of programmer costs to hardware costs rises, with programs becoming longer and more complex, declarative languages are becoming ever more attractive. Furthermore, declarative languages show potential for great efficiency if implemented on massively parallel hardware.

Two of the most popular declarative paradigms are functional and logic programming. Most functional programming languages are built around the lambda calculus, and most logic programming languages are based on a subset of first-order predicate logic, called Horn clauses. From the perspective of predicate-logic programming, functional programming offers three important additional capabilities: infinite data objects, higher-order objects, and directional (non-backtrackable) execution. From the perspective of functional programming, the unique capabilities of predicate-logic programming are its support for constraint reasoning, via unification over first-order terms, and flexible execution moding (non-directionality).

Many attempts have been made to combine the advantages of functional and logic programming into a single language (see [BL86] for a recent survey). Existing approaches fall short in that they either

(a) support no higher-order programming at all [GM84, DP85, YS86], or

(b) require computationally difficult primitives higher-order unification [MN86, R86] (undecidable in worst case), unification relative to an equational theory [GM84], or general theorem proving [MMW84], or

(c) have no extensional declarative semantics [SP85, W83], or

(d) have no identifiable purely functional subset [R85, L85].

We seek a declarative language with simple semantics (including referential transparency), reasonable higher-order capabilities, with the potential for efficient execution. Backtracking should not be used where simple rewriting is sufficient, and the interpreter should not rely on potentially explosive primitives, such as higher-order unification or unification relative to an equational theory. In our approach, we have chosen to supplement a functional programming language with *set abstraction*. We believe that functional programming is a better basis for a unified declarative language than Horn logic, because

a) simple propagation of objects can be managed without unification or proofs of equality;

b) ordinary functional computations not involving set abstraction may be performed in the usual way without backtracking; and

c) since the semantics of Horn logic programming is described in the language of set-theory, it may be more natural to deal directly with sets as objects.

An early attempt at set abstraction was David Turner's KRC [T81]. In this language, programmers may define operations on sets (*relative set abstraction*), encoded as *list comprehensions* [P87], implicitly invoking a backtracking mechanism. This feature allows some problems to be expressed in a manner analogous to logic programming style. However, these list comprehensions may be used in ways inconsistent with their interpretation as sets. If one chooses to view list comprehensions as abstraction, one must admit that referential transparency is violated. Furthermore, many Horn logic programs have no obvious translation into this language.

Darlington [D83] and Robinson [R86] were the first advocates of adding logic programming capability to functional programming through set abstraction. Robinson suggests that a functional language should have a construct denoting the complete set of solutions to a Horn logic program, and that the user be able to build functions accepting such sets as arguments. Darlington called his version *absolute set abstraction*, to distinguish it from Turner's notation. Absolute set abstraction permits expressions such as

$\{x : p(x)\}$,

to denote define the set of all x satisfying $p(x)$. The work of Darlington and Robinson work leave open several interesting problems. To our knowledge, the semantics of set abstraction in functional cum logic programming have never been rigorously worked out. The degree with which this construct can freely interact with other traditional functional language features (such as lazy evaluation, first-class higher-order objects, etc.) has been questioned. In his recent paper [DFP86], Darlington sketched only a partial and informal operational semantics.

Both Darlington and Robinson claim that extending their approach to a higher-order language would require higher-order unification, which is known to be undecidable. Furthermore, some higher-order programs in Darlington's language are unexecutable. Darlington believes these inexecutable programs serve as useful program specifications. Robinson has criticized existing combinations of higher-order functional programming with first-order relational programming as inelegant [R86]. The challenge is to create a language permitting higher-order relational programming, without arbitrary unorthogonal restrictions on usage of higher-order objects.

Our work solves these problems by using results from the theory of power-domains to formalize a lazy higher-order functional language with *true* relative set abstraction. Our approach is motivated by the observation that the nondeterminism in logic programming is essentially 'angelic nondeterminism'. This is fortunate, because the theory of power-domains for angelic nondeterminism extends even for base domains containing higher-order functions and infinite objects. A sugared lambda-calculus language can be augmented with set abstraction using only a few new simple primitives. These primitives are nothing more than the basic powerdomain constructors and deconstructor, proven to be well-defined and continuous (and easily implemented). This approach avoids the problems associated with higher-order first-class objects; for example, higher-order unification is not needed to evaluate this language.

In proposing a new language, one must:

a) specify the syntax and provide examples;

b) define its declarative semantics (what programs mean);

c) define its operational semantics (how programs are executed); and

d) prove that the denotational semantics and the operational semantics describe the same language.

Our goal is a thorough understanding of the semantic and operational aspects of this language. In Section 2, we describe (through formal syntax and program examples) a new language to subsume both Horn logic programming and higher-order lazy functional programming. In this languages, functions and set abstractions are first-class objects. In Section 3, we review the basics of angelic powerdomain theory, the semantic basis for relative set abstraction. This permits us to provide the language's denotational semantics, mapping the features of the language to well-understood semantic primitives in a way which maintains referential transparency.

In Section 4, we show that by imposing constraints on the definitions of semantic primitives, the denotational semantics for a functional language can be written so that a provably equivalent operational semantics is automatically implied. In other words, the denotational equations simultaneously provide both the declarative meaning of programs in the language, and an operational procedure for language interpretation. These results are used to provide an operational semantics mechanism for correctly interpreting programs in the new language. In Section 5, we improve the effiency of this procedure, for certain kinds of set abstractions. When the set of "first-order terms" is the generator of a relative set abstraction, we calculate members not through a simple generate and filter mechanism (the default evaluation mechanism for relative set abstractions), but rather through a set of program transformation rules, inspired by theories of narrowing in term rewriting systems, and resolution in logic programming. Since these transformations are used only for certain types of set abstractions, we avoid the problems associated with unrestricted narrowing.

Section 6 summarizes our accomplishments, in both general and specific terms.

2. LANGUAGE DEFINITION

2.1 Syntax and Constructs

In this section we describe a language synthesizing logic and functional programming through set abstraction. We call this language **PowerFuL**, because it uses angelic **Power**domains to unite **Fu**nctional and **L**ogic programming. A PowerFuL program is an expression to be evaluated. The syntax is:

```
expr         ::=   (expr) | atom
                 | cons(expr, expr) | car(expr) | cdr(expr)
                 | atomeq?(expr,expr) | null?(expr) | expr = expr
                 | bool?(expr) | atom?(expr) | pair?(expr) | func?(expr) | set?(expr)
                 | if expr then expr else expr fi | not(expr)
                 | identifier
                 | L identifiers . expr
                 | expr(expr, ..., expr)
                 | letrec identifier be expr, ..., identifier be expr in expr
                 | phi | atoms | terms | set-clause | U(set-clause, set-clause)

set-clause   ::=   {expr :  qualifierlist}
qualifier    ::=   enumeration | condition
```

enumeration ::= *expr* -> *identifier*
condition ::= *expr*

An enumeration is a relative set abstraction. For each identifier introduced within the set-clause, a set is provided to generate possible values. We prefer relative set abstraction because it has a simpler higher-order generalization. To simulate ***absolute*** set abstraction, simply declare that each "logical variable" takes its values from the set of first-order terms (this set, **`terms`**, is provided as a primitive, though it could be defined in terms of the other constructs).

The scope of the enumerated identifier contains the principal expression (left of the ':'), and also all qualifiers to the right of its introduction. In case of name conflict, an identifier takes its value from the latest definition (innermost scope). In any case, the scope of an enumerated identifier never reaches beyond the set-clause of its introduction.

Lists may be written in the `[...]` notation, e.g. `['apple, 'orange, 'grape]` as a syntactic sugar. Similarly, expressions of the form U(set_1, ..., set_n) are syntactic sugar for a nesting of binary unions. Furthermore, when the list of qualifiers is empty one may omit the ':'.

As is required for full referential transparency (extensionality), equality is not defined over higher-order objects. The result of equating higher-order objects, such as sets or functions, is $\perp$ (even without set abstraction, correctly implementing a higher-order equality predicate would require solving the halting problem). For simplicity, we omit discussion of integers and their operations in this paper, although they can easily be added to the language.

2.2 Examples

Functional Programming

```
letrec
   append be L l1 l2. if null?(l1) then l2
                      else cons(car(l1), append(cdr(l1),l2)) fi
   map be L f.L l.if null?(l) then []
                      else cons(f(car(l)), map(f,cdr(l)))fi
   infinite be cons('a, infinite)
in
```

```
    ...
```

First-order and higher-order functions, as well as infinite objects can be defined in the usual manner. The map example shown above is in curried form.

Set Operations

```
letrec
     crossprod be L s1 s2. {cons(X,Y) : s1->X, s2->Y}
     filter be L p s. {X : s -> X, p(x)}
     intersection be L s1 s2. {X : s1->X, s2->Y, X=Y}
in
    ...
```

The operations `crossprod` and `filter` are similar to those in Miranda [T85]. Note that one *cannot* define an operation to compute the size of a set, nor can one test whether a value is or is not a member. Such operations, analogous to Prolog's meta-logical features, would not be continuous on our domains; furthermore, they are not needed to obtain the declarative capabilities of logic programming.

Logic Programming

```
letrec
    split be L list. { [X|Y] : terms->X, terms->Y, append(X,Y)=list}
    append be L l1 l2. if null?(l1) then l2
                          else cons(car(l1), append(cdr(l1),l2))fi
in
    ...
```

The enumerations `terms->X, terms->Y` in `split` are needed because the set-abstraction is relative, not absolute. For efficiency, an operation such as append might be compiled in different ways corresponding to whether or not it was used within a set-abstraction.

To demonstrate that any first-order Horn logic program can be mechanically converted into PowerFuL, consider the semantics of Horn logic programming. The universe of discourse is taken to be the Herbrand Universe (this corresponds to our set `terms`, the set of terms). A predicate symbol gets its meaning from the *set* of ground instantiations in the Herbrand model (those instantiations implied true by the program clauses).

we could write our logic programs in terms of sets, instead of predicates. A predicate which is true for certain tuples of terms becomes a set which includes just those tuples of terms as members. Where a conventional Prolog program asserts *P(tuple)*, we could equivalently assert that *tuple* $\in$ *P*, *P* now referring to a set.

Consider the following program and goal, written in Prolog syntax [CM81].

```
app([], Y, Y).
app([H|T], Y, [H|Z]) :- app(T, Y, Z).
rev([], []).
rev([H|T], Z) :- rev(T, Y), app(Y, [H], Z).
?- rev(L, [a, b, c]).
```

In the style oriented towards sets, we would write:

```
[ [], Y, Y] ∈ app
[ [H|T], Y, [H|Z] ] ∈ app :- [T, Y, Z] ∈ app
[ [], [] ] ∈ rev
[ [H|T], Y] ∈ rev :- [T, Z] ∈ rev, [Z, [H], Y] ∈ app
?- [X, [a, b, c] ] ∈ rev
```

In one sense, all we have done is create a new Prolog program defining the predicate '$\in$'. But we prefer to view the clauses as defining sets, with '$\in$' taken as a mathematical primitive. With this second viewpoint, translation to PowerFuL is straightforward. Logical variables represent enumeration variables implicitly generated from the set of terms, corresponding to '`terms`'. Furthermore, it is easy to see that

term $\in$ *generating-set*

is equivalent to the conjunction

generating-set `-> New-enum-var, New-enum-var =` *term.*

Converting to PowerFuL syntax results in:

```
letrec
   app be U( { [ [],L,L ] : terms->L},
             {[ [H|T], Y, [H|Z] ] : terms->H,T,Y,Z,
                                    app->W, W=[T,Y,Z]})
   rev be U( {[ [], [] ] },
             {[ [H|T], Z] : terms->H,T,Y,Z, rev->V, app->W,
                V = [T, Y], W = [Y, [H], Z]})
in
    { L : terms -> L, rev -> V, V = [L, ['a, 'b, 'c]] }
```

We have taken the liberty of writing `terms -> h,t,y,z` instead of four separate enumerations.

The PowerFuL program uses sets to express Prolog predicates, which the Prolog program used to express functions. With so many layers of indirection, it is no wonder this PowerFuL version is ugly. But this is to be expected from a mechanical translation. A better PowerFuL style would be to use Lisp-like functions where functions are intended, and sets only where necessary. Still, this technique of Horn logic to PowerFuL conversion demonstrates that we have indeed captured the full expressive power of Horn logic.

Higher-order Functional and Horn logic programming

```
letrec
    one be L v. 'a
    two be L v. 'b
    three be L v. 'c
in
   {F :  U({one}, {two}, {three}) -> F, map(F)(['x, 'y, 'z]) = ['c, 'c, 'c]}
```

The result of the above set-abstraction is the set `{three}`. In this example, the generator set for `F`, `U({one}, {two}, {three})` is first enumerated to obtain a function which is then passed on to `map`. Those functions which satisfy the equality condition are kept in the resulting set, while the others are screened out.

3. DENOTATIONAL SEMANTICS FOR POWERFUL

3.1 Powerdomains

Set abstraction gives functional programming its logic programming capability. In this section, we describe angelic powerdomains, the semantic basis for set abstraction. Intuitively, given a domain D, each element of domain D's powerdomain $\mathcal{P}(D)$ is to be viewed as a set of elements from D. In theory it is a bit more complicated. The details of powerdomain construction are summarized below. For more information, see [S86], [B85], [A82] and [A83].

$\mathcal{P}$(D) is the angelic powerdomain constructed from domain D. Powerdomain theory was developed to describe the behavior non-deterministic calculations. The original application was operating system modelling, where results depend on the random timing of events, as well as on the values of the inputs. Suppose a procedure accepts an element of domain D, and based on this element produces another element in D, nondeterministically choosing from a number of possibilities. We say that the set of possibilities, as subset of D, is a member of $\mathcal{P}$(D). Such a procedure is therefore of type D $\mapsto$ $\mathcal{P}$(D). Computation approximates this set by non-deterministically returning a member.

Suppose f and g are non-deterministic computations performed in sequence, first f and then g. For each possible output of f, g defines a set of possible results. The union of these sets contains all possible results of the sequence. We express this sequencing of non-deterministic functions by λx. g^+(f(x)). The '+' functional is of type

$(D \mapsto \mathcal{P}(D)) \mapsto (\mathcal{P}(D) \mapsto \mathcal{P}(D))$,

defined as $\lambda \mathtt{f}.\lambda \mathtt{set}. \cup \{\mathtt{f(x)} : \mathtt{x} \in \mathtt{set}\}$.

The larger the set denoted by f(x) is, the larger the set denoted by g^+(f(x)) will be, and the larger the likelyhood that the complete sequence can terminate with any correct result. One powerdomain construction ensures that larger sets are considered more defined than their subsets, the empty set being least-defined. Special problems must be considered when building powerdomains of non-flat base domains. A set becomes more defined in two completely different ways: individual set elements can be made more defined according to the partial order of the base domain, or the more-defined elements can be added to the set. Thus, the same information can be combined in different ways to create sets that are distinct, yet computationally equivalent. So theoretically one works not with sets, but with equivalence classes of sets. This should not be too disconcerting. Even in mathematics, a set has not single canonical representation, and equivalent set expressions can be gotten by permuting the ordering of of elements.

Definition: The symbol $\sqsubseteq_\sim$, pronounced 'less defined than or equivalent to', is a relation between sets. For $A, B \subseteq D$, we say that $A \sqsubseteq_\sim B$ iff for every $a \in A$ and Scott-open set $U \subseteq D$, if $a \in U$ then there exists a $b \in B$ such that $b \in U$ also.

Definition: We say $A \approx B$ iff both $A \sqsubseteq_\sim B$ and $B \sqsubseteq_\sim A$. We denote the equivalence class containing A as [A]. This class contains all sets $B \subseteq D$ such that $A \approx B$. We define the partial order on equivalence classes as: $[A] \sqsubseteq [B]$ iff $A \sqsubseteq_\sim B$. For domain D, the powerdomain of D, written $\mathcal{P}$(D), is the set of equivalence classes, each member of an equivalence class being a subset of D.

Theorem (Schmidt [S86]): The following operations are well-defined and continuous:

ϕ: $\mathcal{P}$(D) denotes [{}]. This is the least element.

{_}: D $\mapsto$ $\mathcal{P}$(D) maps d $\in$ D to [{d}].

_∪_: $\mathcal{P}$(D)×$\mathcal{P}$(D)$\mapsto$$\mathcal{P}$(D) maps [A] ∪ [B] to [A ∪ B].

$^+$: (D$\mapsto$$\mathcal{P}$(D)) $\mapsto$ ($\mathcal{P}$(D)$\mapsto$$\mathcal{P}$(D)) is λf.λ[A].[∪{f(a) : a ∈ A}].

3.2 Semantic Equations

PowerFuL's domain is the solution to:

$$D = (B_{\perp_B} + A_{\perp_A} + D\times D + D\mapsto D + \mathcal{P}(D))_{\perp_D},$$

where 'B' refers to the booleans, and 'A' to a finite set of atoms.

PowerFuL is a functional programming language, so we present its semantics in the denotational style usual for such languages [S77]. Our convention to differentiate language constructs from semantic primitives is to write the primitives in **boldface**. Language constructs are in `teletype`. Variables in rewrite rules will be *italicized*.

In the definitions below, the semantic function $\mathcal{E}$ maps general expressions to denotable values. The equations for most expressions are the conventional ones for a typical lazy higher-order functional language. The environment, ρ, maps identifiers to denotable values, and belongs to the domain [Id$\mapsto$D]. The semantic equations for set-abstractions provide the novelty. For simplicity, the semantic equations ignore simple syntactic sugars.

Many of PowerFuL's denotational equations are similar to those of any typical lazy functional language. For instance, for each syntactic atom (represented by $\mathtt{A}_i$) in a program, we assume the existance of an atomic object in the semantic domain (represented by $\mathbf{A}_i$).

$\mathcal{E}[\![\mathtt{A}_i]\!]\ \rho\ =\ \mathbf{A}_i$

We can group objects into ordered pairs to create lists and binary trees.

$\mathcal{E}[\![\mathtt{cons}(\mathit{expr1},\ \mathit{expr2})]\!]\ \rho = <(\mathcal{E}[\![\mathit{expr1}]\!]\ \rho)\ ,\ (\mathcal{E}[\![\mathit{expr2}]\!]\ \rho>)$

$\mathcal{E}[\![\mathtt{car}(\mathit{expr})]\!]\ \rho\ =\ \mathbf{left}(\ \mathbf{pair!}(\mathcal{E}[\![\mathit{expr}]\!]\ \rho))$

$\mathcal{E}[\![\mathtt{cdr}(\mathit{expr})]\!]\ \rho\ =\ \mathbf{right}(\mathbf{pair!}(\mathcal{E}[\![\mathit{expr}]\!]\ \rho))$

$\mathcal{E}[\![\mathtt{bool?}(\mathit{expr})]\!]\ \rho\ =\ \mathbf{bool?}(\mathcal{E}[\![\mathit{expr}]\!]\ \rho)$

$\mathcal{E}[\![\texttt{atom?}(expr)]\!]\ \rho\ =\ \textbf{atom?}(\mathcal{E}[\![expr]\!]\ \rho)$

$\mathcal{E}[\![\texttt{pair?}(expr)]\!]\ \rho\ =\ \textbf{pair?}(\mathcal{E}[\![expr]\!]\ \rho)$

$\mathcal{E}[\![\texttt{func?}(expr)]\!]\ \rho\ =\ \textbf{func?}(\mathcal{E}[\![expr]\!]\ \rho)$

$\mathcal{E}[\![\texttt{set?}(expr)]\!]\ \rho\ =\ \textbf{set?}(\mathcal{E}[\![expr]\!]\ \rho)$

Testing atoms for equality relies on the primitive definition of the atoms.

$\mathcal{E}[\![\texttt{atomeq?}(expr1,\ expr2)]\!]\ \rho\ =\ \textbf{atomeq?}(\textbf{atom!}(\mathcal{E}[\![expr1]\!]\ \rho), \textbf{atom!}(\mathcal{E}[\![expr2]\!]\ \rho))$

$\mathcal{E}[\![(expr1\ =\ expr2)]\!]\ \rho\ =\ \textbf{equal?}((\mathcal{E}[\![expr1]\!]\ \rho), (\mathcal{E}[\![expr2]\!]\ \rho))$

A conventional sugar tests whether a "list" is empty (whether the object equals the atom "nil").

$\mathcal{E}[\![\texttt{null?}(expr)]\!]\ \rho\ =\ \textbf{if}(\textbf{atom?}(\mathcal{E}[\![expr]\!]\ \rho)\textbf{then is'nill?}([\![expr]\!]\ \rho)\textbf{else FALSE fi})$

We can negate a condition.

$\mathcal{E}[\![\texttt{not}(expr)]\!]\ \rho\ =\ \textbf{not}(\textbf{bool!}(\mathcal{E}[\![expr]\!]\ \rho))$

We can add new identifiers to the environment, and later look up their meaning.

$\mathcal{E}[\![\texttt{letrec}\ defs\ \texttt{in}\ expression]\!]\ \rho\ =\ \mathcal{E}[\![expression]\!]\ (\mathcal{D}[\![defs]\!]\ \rho)$

$\mathcal{E}[\![identifier]\!]\ \rho\ =\ \rho(identifier)$

$\mathcal{D}[\![id\ \texttt{be}\ expr]\!]\ \rho\ =\ \rho[(\mathcal{F}[\![\texttt{fix}]\!]\ )(\lambda\ X.\ (\mathcal{E}[\![expr]\!]\ \rho[X/id]))/id]$

$\mathcal{D}[\![id\ \texttt{be}\ expr, defs]\!]\ \rho\ =\ (\mathcal{D}[\![defs]\!]\ \rho)\ [(\mathcal{F}[\![\texttt{fix}]\!]\ )(\lambda\ X.\ (\mathcal{E}[\![expr]\!]\ (\mathcal{D}[\![defs]\!]\ \rho[X/id])))/id]$

Rather than treat the fixpoint operator as a primitive, we define `fix` in the semantic equations. For reasons to become apparent later, we wish to have only one source of potentially unbounded recursion, and we wish that source to be the semantic equations themselves.

$\mathcal{F}[\![\texttt{fix}]\!]\ =\ \lambda f.\ f((\mathcal{F}[\![\texttt{fix}]\!]\ )(f))$

We can create conditional expressions:

$\mathcal{E}[\![\texttt{if}(expr1,\ expr2,\ expr3\ )]\!]\ \rho\ =\ \textbf{if}(\textbf{bool!}(\mathcal{E}[\![expr1]\!]\ \rho),\ (\mathcal{E}[\![expr2]\!]\ \rho),\ (\mathcal{E}[\![expr3]\!]\ \rho))$

We can create functions through lambda abstraction, and apply functions to their arguments.

$\mathcal{E}[\![\texttt{L}\ id.\ expr]\!]\ \rho\ =\ \lambda\ x.\ (\mathcal{E}[\![expr]\!]\ \rho[x/id])$

In the above equation, we considered only functions of one argument. A function of multiple arguments can be considered syntactic sugar either for a curried function, or for a function whose single argument is an ordered sequence, or list.

$\mathcal{E}[\![\mathit{expr1}\ \mathit{expr2}]\!]\ \rho\ =\ \textbf{func!}(\mathcal{E}[\![\mathit{expr1}]\!]\ \rho)(\mathcal{E}[\![\mathit{expr2}]\!]\ \rho)$

Empty sets and singleton sets form the building blocks. We can union smaller sets to form larger sets, and via a relative set abstraction we can transform elements of one set to create another. We can denote the empty set explicitly:

$\mathcal{E}[\![\texttt{phi}]\!]\ \rho = \phi$

We can create a singleton set from a base domain:

$\mathcal{E}[\![\{\mathit{expr}\ \ :\}]\!]\ \rho = \{\mathcal{E}[\![\mathit{expr}]\!]\ \rho\}$

We can choose to include only those elements meeting a specified condition:

$\mathcal{E}([\![\{\mathit{expr}\ \ :\ \ \mathit{condition}, \mathit{qualifierlist}\}]\!]\ \rho)$

$\quad = \textbf{set!}(\textbf{if}\ \mathcal{E}[\![\mathit{condition}]\!]\ \rho\ \textbf{then}\ \mathcal{E}[\![\{\mathit{expr}\ \ :\ \ \mathit{qualifierlist}\}]\!]\ \rho\ \textbf{else}\ \phi\textbf{fi})$

We can combine the smaller sets to form larger sets:

$\mathcal{E}[\![\texttt{U}(\mathit{expr}_1, \mathit{expr}_2)]\!]\ \rho = \textbf{set!}([\![\mathit{expr}_1]\!]\ \rho) \cup \textbf{set!}([\![\mathit{expr}_2]\!]\ \rho)$

We can build a set based on the elements included in some another set. The '+' operator was defined for this purpose:

$\mathcal{E}([\![\{\mathit{expr}\ \ :\ \ \mathit{genrtr}\texttt{->}\mathit{id}, \mathit{qualifierlist}\}]\!]\ \rho)$

$\quad = (\lambda\ X.\ \mathcal{E}[\![\{\mathit{expr}\ \ :\ \ \mathit{qualifierlist}\}]\!]\ \rho[X/\mathit{id}])^{+}(\textbf{set!}(\mathcal{E}[\![\mathit{genrtr}]\!]\ \rho))$

The sets `bools`, `atoms` and `terms` may be viewed as syntactic sugars, since the user *could* program these using the previously given constructs. In that sense, their presence adds nothing to the expressive power of the language. Nevertheless, providing them in the syntax permits important optimizations through run-time program transformation (discussed later). Thus we have:

$\mathcal{E}[\![\texttt{bools}]\!]\ \rho = \mathcal{F}[\![\texttt{bools}]\!]$

$\mathcal{F}[\![\texttt{bools}]\!] = \{\mathrm{TRUE}\} \cup \{\mathrm{FALSE}\}$

$\mathcal{E}[\![\texttt{atoms}]\!]\ \rho = \mathcal{F}[\![\texttt{atoms}]\!]$

$\mathcal{F}[\![\texttt{atoms}]\!] = \cup(\{\mathrm{A}_1\},\ \ldots,\ \{\mathrm{A}_n\})$

$\mathcal{E}[\![\texttt{terms}]\!]\ \rho = \mathcal{F}[\![\texttt{terms}]\!]$

$\mathcal{F}[\![\texttt{terms}]\!] =$

$\cup(\mathcal{F}[\![\texttt{bools}]\!], \mathcal{F}[\![\texttt{atoms}]\!], (\lambda s.((\lambda t.\{< s, t >\})^{+}(\mathcal{F}[\![\texttt{terms}]\!])))^{+}(\mathcal{F}[\![\texttt{terms}]\!])$

The functions $\mathcal{E}$, $\mathcal{D}$ and $\mathcal{F}$ are mutually recursive. Their meaning is the least fixed point of the recursive definition. This fixed-point exists because we have combined continuous primitives with continuous combinators. Most of these primitives are fairly standard, and will be described in a later section.

However, a few words must be said about some of the novel primitives. The primitive '+' for distributing elements of a powerdomain to a function has already been discussed. The other novel primitives are the *coersions*, related to the *type-checking* primitives. They are described below.

PowerFuL is basically an untyped language. For limited run-time type-checking, we rely on these primitive semantic functions over $\texttt{D} \mapsto \texttt{B}_{\perp}$: **atom?**, **bool?**, **pair?**, **func?** and **set?**.

For instance, **func?** returns **TRUE** if the argument is a primitive function or a lambda expression, **FALSE** if the argument is an atom, an ordered pair or a set. The only other possibility is $\perp_D$, so **func?**$(\perp_D)$ rewrites to $\perp_B$. The other type-checking functions are defined analogously.

Most of our primitives are defined over only portions of the domain D. The boolean operators are defined only over $\texttt{B}_{\perp}$. Only ordered pairs have left and right sides. Function application is defined only when the left argument is in fact a function. Only sets can contribute to a set union. Since PowerFuL is an untyped language, we will need a way to coerce arguments to the appropriate type. One way is to use the type-checking primitives in conjuction with typed-if primitives. We find it simpler to define five primitive coercions. They are: **bool!**, **atom!**, **pair!**, **func!** and **set!**.

The function **bool!**: $\texttt{D} \mapsto \texttt{B}_{\perp}$ maps *arg* to itself if *arg* is a member of $\texttt{B}_{\perp}$, and to $\perp_B$ otherwise.

The function **atom!**: $\texttt{D} \mapsto \texttt{A}_{\perp}$ maps *arg* to itself if *arg* is a member of $\texttt{A}_{\perp}$, and to $\perp_A$ otherwise.

The function **pair!**: $\texttt{D} \mapsto \texttt{D}\times\texttt{D}$ maps *arg* to itself if *arg* is a member of $\texttt{D}\times\texttt{D}$, and to $\perp_{D\times D}$ (that is, $< \perp_D, \perp_D >$) otherwise.

The function **func!**: $\texttt{D} \mapsto [\texttt{D}\mapsto\texttt{D}]$ maps *arg* to itself if *arg* is a member of $\texttt{D}\mapsto\texttt{D}$ and to $\perp_{D\mapsto D}$ (that is, $\lambda\texttt{x}.\ \ \perp_D$) otherwise.

The function **set!**: $\mathtt{D} \mapsto \mathcal{P}(\mathtt{D})$. maps *arg* to itself if *arg* is a member of $\mathcal{P}(\mathtt{D})$ and to $\perp_{\mathcal{P}(D)}$ (that is, ϕ) otherwise.

Theorem: These coercions are continuous.

Proof: We will prove the continuity of '**set!**'. Consider a sequence of objects from domain $\mathtt{D}$, $t_0, t_1, t_2, \ldots$, such that for $i < j$, $t_i \sqsubseteq t_j$. If there is no i such that t_i is in $\mathcal{P}(\mathtt{D})$, then for all i, $\mathbf{set!}(i) = \perp_{\mathcal{P}(D)} = \phi$. Thus,

$$\lim_{i\to\infty} \mathbf{set!}(t_i) \;=\; \mathbf{set!}(\lim_{i\to\infty} t_i) \;=\; \perp_{\mathcal{P}(D)} \;=\; \phi.$$

If there *is* an i such that t_i is in $\mathcal{P}(\mathtt{D})$, then let t_k be the first one. That is, for all i, if t_i is in $\mathcal{P}(\mathtt{D})$, then $t_k \sqsubseteq t_i$. Then for all $i < k$, $t_i = \perp_D$, and $\mathbf{set!}(t_i) = \mathbf{set!}(\perp_D) = \perp_{\mathcal{P}(D)=\phi}$. For all $i \geq k$, and $\mathbf{set!}(t_i) = t_i$. Therefore,

$$\lim_{i\to\infty} \mathbf{set!}(t_i) \;=\; \lim_{k\to\infty} \mathbf{set!}(t_i) \;=\; \mathbf{set!}(\lim_{k\to\infty} t_i) \;=\; \mathbf{set!}(\lim_{i\to\infty} t_i).$$

Hence, '**set!**' is continuous.

Proof of the continuity of the other coercions is left to the reader.

4. FROM DENOTATIONAL TO OPERATIONAL SEMANTICS

The design of a new programming language must contain a concise yet precise description of what language constructs mean. This description is usually provided by the denotational semantics. Though the designer is not responsible for creating an efficient and practical compiler, he should show that the language *can* be efficiently implemented. Ususally, a separate operational semantics connsisting of a system of rewrite rules is given, with rules closely related to the denotational equations. The designer is then obligated to prove that the denotational and operational semantics are in some sense equivalent.

We favor a different approach. The set of denotational equations resembles a functional program. In fact, a denotational semantics can be viewed as the program for an interpreter, written in a declarative (functional) psuedocode. In this section, we show that if we adopt a certain discipline for writing denotational semantics, the resulting equations are not merely pseudocode, but an actual program in a functional meta-language whose correct implementation is well understood. The denotational equations thus have two equivalent interpretations: they are the declarative description of the object language's semantics, and they form a program to interpret programs written in the object language. This idea is not new. Peyton Jones [P87] describes systems for automatically generating compilers from a language's denotational semantics.

In detailing our method, we rely on results from the literature, some of which are reviewed in this section. Also in the section we show how these results can be adapted to meet our needs. The next section is based on Vuillemin [V74].

4.1 Least Fixpoints and Safe Computation Rules

Consider a recursive definition of the form

$$F(\overline{x}) \Leftarrow \tau[F](\overline{x})$$

where $\tau[F](\overline{x})$ is a functional over `([D₁, ..., Dₙ)` $\mapsto$ `D]`, expressed by composition of known monotonic functions, called *primitives*, and the function variable F, constructing a term from these and the individual variables $\overline{x} =< x_1, x_2, \ldots, x_n >$. It is generally agreed that the function defined by a recursive program P

$$P: \; F(\overline{x}) \Leftarrow \tau[F](\overline{x})$$

is f_τ, the *least fixpoint* of τ. We denote this fixpoint by f_P.

For example, suppose that '`*`' (multiplication), '`-`' (subtraction), '`=`' (equality) and '`if`' (if/then/else/fi) are primitive functions. Given a program P:

$$P: \; fact(x) \Leftarrow \texttt{if}((x=0), 1, x \times fact(x-1)),$$

τ is the functional

$$\lambda f.\, \texttt{if}((x=0), 1, x \times f(x-1))$$

and $fact$ is the name of recursive function, represented by F in the schema. The fixpoint of this functional is the factorial function.

Definition: A *canonical simplification* is defined by a set of simplification rules, that have have the following properties:

(a) for each term t there is a unique term t' into which t simplifies, where t' cannot be further simplified;

(b) the set of simplification rules for a primitive is consistent with the primitive's interpretation (intended meaning); and

(c) the simplification set for a primitive is complete, i.e., it completely specifies the primitive.

Rewrite rules for standard simplifications are terminating and confluent. Let us assume that all the primitive functions are defined by canonical simplifications. We permit no primitives of zero arity. Each constant is a unique data constructor; collectively, they make up the *flat domain* of atoms.

Given primitives defined as standard simplifications, and a recursive program defining a function, we can compute the function for a given value of $\overline{d}$. During computation, the interpreter constructs a sequence of terms t_0, t_1, t_2, ..., called the *computation sequence of F for $\overline{d}$*. The first term t_0 is $F(\overline{d})$. For each $i > 0$, the term t_{i+1} is obtained from t_i in two steps, by *substitution* and *simplification*. In the substitution stage, some occurrences of F in t_i are each replaced by $\tau[F]$.

Definition: A *computation rule* C tells us which occurrences of $F(\overline{e})$ should be replaced by $\tau[F](\overline{e})$ in each step.

In the simplification stage, primitive functions are rewritten according to the simplification rules, wherever possible, until no further rewritings can be made. For each term t_i, we construct $\hat{t_i}$ by replacing with Ω (the undefined function) all remaining occurrences of F, and then simplifying. The computed value $C_P(\overline{d})$ is limit of the sequence $\{\hat{t_i}\}$. We wish our simplification rules to be confluent, so that the order of simplifications performed be irrelevant, and so that the primitive function they represent be well-defined. We want the simplification rules to terminate, so that each computation step require a finite amount of work.

Theorem (Cadiou [V74]): For any computation rule C,

$$C_P \sqsubseteq f_P.$$

That is, the computed function C_p approximates the fixpoint f_P.

Definition: A computation rule is said to be a *fixpoint computation rule* if for every recursive program P, for all $\overline{d}$ in the relevant domain,

$$C_P(\overline{d}) \equiv f_P(\overline{d}).$$

We need to give a condition which, if satisfied, will imply that a computation rule is a fixpoint rule.

Definition: A *substitution step* is a computation step in which some of the recursive function calls in a term are expanded.

Definition: For a substitution step, let F^1, ..., F^i be the occurrences of the recursive function expanded in the term, and let $F^{i+1}, \ldots, F^k$ be the occurrences not expanded. Compare the result obtained by replacing replaced $F^1, \ldots, F^i$ each by Ω, and $F^{i+1}, \ldots, F^k$ each by f_P, with the result obtained by replacing $F^1, \ldots, F^i, F^{i+1}, \ldots, F^k$ each by Ω. If the results are equal, then we say the substitution step is a *safe substitution step.*

Intuitively, a safe substitution is one which performs enough essential work. That is, if this work were never done, then all other work would be irrelevant. If enough essential work is performed in each step, then every essential piece of work will eventually be done.

Definition: A computation rule is *safe* if it provides for only safe substitution steps.

Theorem (Vuillemin [V74]): If the computation rule used in computing C_P is a safe, then

$$f_P \sqsubseteq C_P,$$

Using Cadiou's theorem and Vuillemin's theorem, then for any safe computation rule,

$$C_P \equiv f_P,$$

and therefore *all safe computation rules are fixpoint rules.*

Theorem (Vuillemin [V74]): The parallel outermost rule (replace all outermost occurences of F simultaneously) is a safe rule.

Therefore the sequence of computed values $\hat{t_0}, \hat{t_1}, \hat{t_2}, \ldots,$ using the parallel outermost computation rule produces arbritrarily good finite approximations to the denoted value. If a term t_k contains no occurrences of the symbol F, then the sequence is finite, and t_k is itself the denoted value.

4.2 Relaxing the Notation

It is sometimes more convenient to specify a recursive function via equations, rather than the notation of lambda abstraction. Consider the *append* function, which can be written:

$$P: \quad append(x,y) \Leftarrow \texttt{if}(\texttt{null?}(x), y, <\texttt{car}(x), append(\texttt{cdr}(x),\ y)>).$$

For this program P, τ is the functional

$$\lambda f.\ \texttt{if}(\texttt{null?}(x), y, <\texttt{car}(x), f(\texttt{cdr}(x),\ y)>.$$

Alternatively, we can define append by these equations:

$$append([\,],\ y) \;=\; y$$

$$append(<h, t>, y) = <h, append(t, y)>$$

The equations handle mutually exclusive cases. A function defined through lambda-abstraction is applied using β-reduction. To apply a function defined by a set of equations, one finds the equation which matches the format of the argument, replaces the equation variables in the right-hand side with the parts of the arguments matching them on the left. In this case, our functional is

$$\lambda f. \{f([\,], y) = y; \quad f(<h, t>, y) = <h, f(t, y)>\}$$

Despite the new notation, and its associated mechanics for function application, the same theorems hold as before.

We can also permit a system of mutually-recursive functions. If the equations define two functions, $g(x)$ and $h(x)$, they can be viewed as a single function $f(w, x)$, where the first argument to f tells whether the rules for function g or h are to be used. Vuillemin notes that the extension of his results to a set of mutually-recursive functions is straightforward.

4.3 Implementing Denotational Semantics

The semantics describes the meaning a programming language's syntax. After describing the syntactic and semantic domains, the designer defines the function which maps syntactic objects into the semantic domain. When the function is described procedurally, then we call it the *operational semantics.* If implemented, this function is called the *interpreter.* We use the word '*metalanguage*' to refer to the language in which the implementation is written.

It is common practice to provide *both* a declarative semantics (written in a declarative language or psuedocode) and a non-declarative operational semantics. The declarative semantics becomes the official definition of the language, as it is simpler and easier to understand. In it one describes the mapping desired via a well-understood mathematical theory (recursive function theory for functional programming, predicate logic for logic programming). The operational semantics can be optimized to execute faster, since it can be written in a language closer to the architecture of the physical machine. Before using such an operational semantics however, it is considered proper to describe (and prove) the extent to which the function implemented equals the function described declaratively.

If the language of the declarative semantics is not just a mathematical language, but a programming language in its own right, then the declarative semantics can serve as *both* a definition *and* an implementation. Assuming the metalanguage can be implemented correctly, both views are equivalent.

A language's *denotational semantics* is a declarative semantics written in a metalanguage consisting of a system of recursive equations. Each semantic equation handles a different catagory of syntactic expression. Using care, we can write the denotational equations in such a way that all primitive functions qualify as standard simplifications. Vuillemin's results (described above) then provide a correct implementation of the metalanguage. In this way, the denotational semantics provides simultaneously a declarative description of the language and an equivalent interpreter. This interpreter does not provide the most efficient operational semantics possible, but it is guaranteed to correspond completely to the declarative semantics.

Below is the denotational definition of a simple, higher-order functional language. The language has '`cons`' for constructing lists, '`L`' for constructing lambda expressions (functions), and '`letrec`' for creating recursively-defined objects. With the primitives properly defined, the denotational definition also serves as a program in Vuillemin's metalanguage. We then execute this meta-program, using a simple object-language program as input. We use boldface for semantic primitives, typewriter font for syntactic primitives.

The semantic domain, `D`, is the solution to:

$$\mathtt{D} = (\mathtt{B}_{\perp_B} + \mathtt{A}_{\perp_A} + \mathtt{D}\times\mathtt{D} + \mathtt{D}\mapsto\mathtt{D})_{\perp_D}.$$

where '`B`' refers to the booleans, and '`A`' to a finite set of atoms.

$$\mathcal{E}[\![\mathtt{A}_i]\!]\ \rho = \mathbf{A}_i$$

$$\mathcal{E}[\![\mathtt{car}(\mathit{expr})]\!]\ \rho = \mathbf{left}(\mathbf{pair!}(\mathcal{E}[\![\mathit{expr}]\!]\ \rho))$$

$$\mathcal{E}[\![\mathtt{cdr}(\mathit{expr})]\!]\ \rho = \mathbf{right}(\mathbf{pair!}(\mathcal{E}[\![\mathit{expr}]\!]\ \rho))$$

$$\mathcal{E}[\![\mathtt{cons}(\mathit{expr1},\ \ \mathit{expr2})]\!]\ \rho = <(\mathcal{E}[\![\mathit{expr1}]\!]\ \rho)\ ,\ (\mathcal{E}[\![\mathit{expr2}]\!]\ \rho>)$$

$$\mathcal{E}[\![\mathtt{atom?}(\mathit{expr})]\!]\ \rho = \mathbf{atom?}(\mathcal{E}[\![\mathit{expr}]\!]\ \rho)$$

$$\mathcal{E}[\![\mathtt{pair?}(\mathit{expr})]\!]\ \rho = \mathbf{pair?}(\mathcal{E}[\![\mathit{expr}]\!]\ \rho)$$

$$\mathcal{E}[\![\mathtt{null?}(\mathit{expr})]\!]\ \rho = \mathbf{if}(\mathbf{atom?}(\mathcal{E}[\![\mathit{expr}]\!]\ \rho)\mathbf{then\ is'nil?}([\![\mathit{expr}]\!]\ \rho)\mathbf{else\ FALSE\ fi})$$

$$\mathcal{E}[\![\mathtt{if}(\mathit{expr1},\ \mathit{expr2},\ \mathit{expr3}\ )]\!]\ \rho = \mathbf{if}((\mathcal{E}[\![\mathit{expr1}]\!]\ \rho),\ (\mathcal{E}[\![\mathit{expr2}]\!]\ \rho),\ (\mathcal{E}[\![\mathit{expr3}]\!]\ \rho))$$

$$\mathcal{E}[\![\mathtt{letrec}\ \mathit{defs}\ \mathtt{in}\ \mathit{expression}]\!]\ \rho = \mathcal{E}[\![\mathit{expression}]\!]\ (\mathcal{D}[\![\mathit{defs}]\!]\ \rho)$$

$$\mathcal{E}[\![\mathit{identifier}]\!]\ \rho = \rho(\mathit{identifier})$$

$\mathcal{D}[\![id \texttt{ be } expr]\!]\ \rho\ =\ \rho[(\mathcal{F}[\![\texttt{fix}]\!])(\lambda\ X.\ (\mathcal{E}[\![expr]\!]\ \rho[X/id]))/id]$

$\mathcal{D}[\![id \texttt{ be } expr, defs]\!]\ \rho\ =\ (\mathcal{D}[\![defs]\!]\ \rho)\ [(\mathcal{F}[\![\texttt{fix}]\!])(\lambda\ X.\ (\mathcal{E}[\![expr]\!]\ (\mathcal{D}[\![defs]\!]\ \rho[X/id])))/id]$

$\mathcal{F}[\![\texttt{fix}]\!]\ =\ \lambda f.\ f((\mathcal{F}[\![\texttt{fix}]\!])(f))$

$\mathcal{E}[\![\texttt{L}\ id.\ expr]\!]\ \rho\ =\ \lambda\ x.\ (\mathcal{E}[\![expr]\!]\ \rho[x/id])$

$\mathcal{E}[\![expr1\ expr2]\!]\ \rho\ =\ \textbf{func!}(\mathcal{E}[\![expr1]\!]\ \rho)(\mathcal{E}[\![expr2]\!]\ \rho)$

Note that the function '$\mathcal{E}$' ususally has two parameters, a syntax expression (within the brackets) and an environment. The same is true for the '$\mathcal{D}$'. Note that the fixpoint operator would never qualify as a standard simplification, as it does not terminate. Therefore, rather than treat it as a primitive, we chose to define the fixpoint operator in the denotational equations. To complete the semantic description, we must define the primitive functions **left**, **right**, **if then else fi**, **isA**$_i$, **atom!**, **pair!**, **func!**, **atom?** and **pair?**. Equations, interpreted as left-to-right rewrite rules, provide both a definition and an execution mechanism. The primitives **left** and **right** are defined on the subdomain D×D, the ordered pairs.

left(<*1st*, *2nd*>) = *1st*
right(<*1st*, *2nd*>) = *2nd*

The only boolean primitive used in this small language is the conditional.

if(**TRUE**, *arg2*, *arg3*) = *arg2*
if(**FALSE**, *arg2*, *arg3*) = *arg3*
if($\bot_B$, *arg2*, *arg3*) = $\bot_D$

For every atom $\mathbf{A}_i$, there is a primitive function '**isA**$_i$**?**'. The rules are

isA$_i$**?**($\bot_A$) = $\bot_B$
isA$_i$**?**($\mathbf{A}_i$) = **TRUE**
isA$_i$**?**($\mathbf{A}_j$) = **FALSE** for $i \neq j$

Since this is an untyped language, the user may request a primitive operation be performed on data from an inappropriate subset of the domain. One solution is to expand the definitions of primitives, explicitly stating their result ($\bot$) for all the ways this can occur. We find it easier to define what we call *coercions*, one for each major subset of the domain. That way, we need not repeat the entire list of subdomains on which a primitive is undefined, for each primitive designed to operate on a specific subdomain.

A coercion transforms inappropriate objects to $\perp_x$, (where x is the appropriate subtype), but acts as the identity function if the object is appropriate.

For instance, if A is from the subdomain $\mathtt{B}_\perp$, then **bool!**(A) equals A, otherwise, **bool!**(A) equals $\perp_B$. Similarly, if A is from the subdomain $\mathtt{D}\times\mathtt{D}$, then **pair!**$(A)$ equals A, itself; otherwise **pair!**(A) equals $< \perp_D, \perp_D >$, (which is $\perp_{D\times D}$). The coercions for atoms and functions is analogous. The semantic equations associate these coercions with the use of primitive functions where necessary. In the section giving the denotational semantics of PowerFuL, we proved that these coercions are continious.

Finally, we provide predicates **atom?** and **pair?**. If A is an element of $\mathtt{A}_{\perp_A}$, then **atom?**(A) equals **TRUE**. If A is a member of one of the other subdomains, then **atom?**(A) equals **FALSE**. However, **atom?**$(\perp_D)$ equals $\perp_B$. Other type predicates are defined analogously.

We avoided making the fixpoint operator a semantic primitive. Such a primitive could never be defined as a standard simplification, as it does not terminate. Instead, the recursive defintion of the fixpoint operator was built into the denotational equations themselves. Creating a denotational description suitable for direct interpretation requires such special care.

Some semantic equations introduce lambda variables. These may have to be renamed at times to avoid variable capture. This, however, is standard practice in executing languages based on lambda calculus. Because functions are written as lambda expressions (primitive functions are an exception), β-reduction is treated as a primitive, strict in its first argument. Actually, defining β-reduction as a primitive is dangerous, as it does not necessarily qualify as a standard simplification. For some lambda expressions, β-reduction will not terminate. For the time being, we will assume that in every computation step, the β-reductions will terminate. Later, we will discuss conditions under which this assumption is valid.

4.3.1 Executing a Program

Let us execute (translate into the semantic domain) the object program:

```
letrec
   inf be cons('joe, inf)
in
   car(inf).
```

We start with an empty environment, so the initial input is:

$\mathcal{E}[\![\texttt{letrec inf be cons('joe,inf) in car(inf)}]\!]\ []$.

Expanding the outermost call yields:

$\mathcal{E}[\![\texttt{car(inf)}]\!](\mathcal{D}[\![\texttt{inf be cons('joe,inf)}]\!]\ [])$.

There are still no simplifications to be performed, so we again expand the outermost function call, yielding:

$\mathbf{left}(\mathbf{pair!}(\mathcal{E}[\![\texttt{inf}]\!](\mathcal{D}[\![\texttt{inf be cons('joe,inf)}]\!]\ [])))$.

Expanding the outermost function call yields:

$\mathbf{left}(\mathbf{pair!}((\mathcal{D}[\![\texttt{inf be cons('joe,inf)}]\!]\ [])\texttt{inf}))$,

and then:

$\mathbf{left}(\mathbf{pair!}([((\mathcal{F}[\![\texttt{fix}]\!])\lambda X.\ (\mathcal{E}[\![\texttt{cons('joe,inf)}]\!]\ []\ [X/\texttt{inf}]))/\texttt{inf}]\texttt{inf}))$.

Note that when introducing new lambda variables, one must be careful to standardize variables apart (rename bound variables to as not to confuse them with pre-existing lambda variables). Simplfiying (applying the environment) yields

$\mathbf{left}(\mathbf{pair!}((\mathcal{F}[\![\texttt{fix}]\!])(\lambda X.(\mathcal{E}[\![\texttt{cons('joe,inf)}]\!]\ [X/\texttt{inf}]))))$.

Expanding the outermost call yields:

$\mathbf{left}(\mathbf{pair!}((\lambda F.\ F((\mathcal{F}[\![\texttt{fix}]\!])F))(\lambda X.\ (\mathcal{E}[\![\texttt{cons('joe,inf)}]\!]\ [X/\texttt{inf}]))))$.

Performing β-reduction yields:

$\mathbf{left}(\mathbf{pair!}((\lambda X.\ (\mathcal{E}[\![\texttt{cons('joe,inf)}]\!]\ [X/\texttt{inf}]))$

$((\mathcal{F}[\![\texttt{fix}]\!])(\lambda X.\ (\mathcal{E}[\![\texttt{cons('joe,inf)}]\!]\ [X/\texttt{inf}])))))$.

Performing another β-reduction yields:

$\mathbf{left}(\mathbf{pair!}(\mathcal{E}[\![\texttt{cons('joe,inf)}]\!]\ \rho))$,

where ρ is:

$[((\mathcal{F}[\![\texttt{fix}]\!])(\lambda Y.(\mathcal{E}[\![\texttt{cons('joe,inf)}]\!]\ [Y/\texttt{inf}])))/\texttt{inf}]$.

Expanding the outermost function call yields:

$\mathbf{car}(\mathbf{pair!}(<(\mathcal{E}[\![\texttt{'joe}]\!]\ \rho),\ (\mathcal{E}[\![\texttt{inf}]\!]\ \rho)>))$.

This simplifies to:

$\mathcal{E}[\![\texttt{'joe}]\!]\ \rho$.

Expanding the remaining function call yields:

'**joe**.

4.4 Alternative Computation Rules

4.4.1 Motivation

Vuillemin[V74] proves that for a language with strict primitives and flat domain (except the if/else, which is strict in its first argument), leftmost reduction is safe. However, many interesting languages do not meet these criteria. Consider the problem of non-flat domains. Suppose we have built a hierarchical domain using a sequence constructor, in our case '$<,>$' the ordered pair constructor. Suppose we are trying to compute an ordered pair, of which both elements are infinite lists. If we evaluating this object as a top-most goal using the left-most rule, no part of the right side would ever be computed.

Nevertheless, one would like to limit computation to a primitive's strict arguments, rather than to rewrite function occurences in all arguments (even non-strict arguments) simultaneously. It is not always necessary to expand all outermost function calls in every step. Consider the if/then/else primitive, which is strict in just one of its arguments. We would prefer to evaluate the strict argument first, postponing evaluation of non-strict arguments, which may never be needed. Even if the primitive is strict in all its arguments, we may prefer to concentrate on just one at a time. If evaluation of the chosen strict argument fails to terminate (effectively computing the $\perp$ of the appropriate domain or subdomain), then the primitive expression as a whole denotes $\perp$, and the values of the other arguments do not matter. If evaluation does produce a non-bottom value, the primitive may be able to simplify immediately.

For a higher-order language this evaluation strategy is not always safe. Consider the unlikely (but valid) example in which we are computing *an unapplied function* as a topmost goal. Assume *exp1* denotes an infinite list, and the interpreter is asked to evaluate the function

$\lambda f.(\mathbf{if} f(expr_1)\mathbf{then} exp_2 \mathbf{else} exp_3)$.

The **if** primitive is strict in the first argument. Though evaluation of $f(expr_1)$ fails to terminate, we cannot say that this application denotes bottom; its value depends on the hypothetical value symbolized by the lambda variable. Since the lambda expression is not being applied to any argument here, the first argument of '**if**' will not reduce to an element of the semantic domain. It remains as a parameterized description of a domain element. If the computed value is to be equivalent to the fixpoint definition in this circumstance, we must evaluate all three arguments of the 'if' expression simultaneously. To use leftmost evaluation with higher-order language, we must be content to evaluate a function *only* within the context of its application. It is not sufficient if we wish to compute a function for its own sake, where we cannot rely on evaluating the first argument and then reducing.

Most functional languages *are* higher order, and many interesting ones *do* permit infinite lists. Yet, these are often implemented with a *leftmost* computation rule. This works so long as:

a) all primitives are strict in the first argument (this is ususally true);

b) one is only concerned with computing finite objects (computations for which the parallel-outermost rule terminates), though parts of infinite objects may be used during the computation.

If the parallel outermost rule fails to terminate, then so will any other rule, and one might not care whether two non-terminating calculations are approaching the same limit. This compromise is inadequate for set abstraction. Sets can be infinite; even finite sets may contain non-terminating (but empty) branches. In such cases, computation of the set will never terminate.

Since the elements of a set are not ordered, one cannot isolate a finite subset (analogous to taking a prefix of an infinite list), nor direct one's reference to the 'first' element of the set. At least, one cannot do this within the programming language. Yet, even when computation of the set never terminates, certain elements of the set might be computed within only a finite amount of computation. The user would certainly like to see those elements, as they are computed. We must have a reasonable way to compute a non-terminating goal. Prolog provides a precedent for this. A Prolog program with goal denotes a (possibly infinite) set of correct answer substitutions. Rather than waiting for the entire set to be computed, the system suspends and turns control over to the user every time another member of this set is computed. To evaluate a (possibly infinite) set, the system must provide the user with a series of finite approximations. This could be done interactively, with the system suspending each time a new element is ready for output, resuming at the option of the user.

Definition: An infinite object is *computable* if it is the limit of an infinite series of finite objects.

Definition: *Completeness* for such an interpreter means that any finite member of the denoted set will eventually be computed, even if only by providing an infinite series of finite approximations.

Sequential Prolog interpreters are not complete in this regard, but, in principle, complete (breadth first) Prolog interpreters could be built. Perhaps in an interactive implementation of language with set abstractions, the programmer will be able to direct where in the set expression the computational effort should be concentrated. Analogous to online-debugger commands, such features pertain to the meta-linguistic environment, not to the language itself, so we will not consider these details any further.

4.4.2 Better Computation Rules

Vuillemin describes some computation rules, each based on a single type of substitution step. Choosing the substitution step depending on the form of the expression can provide greater efficiency without sacrificing safety. We describe a new safe computation rule below. It uses the parallel-outermost substitution step as a last resort, but seeks a more selective step when circumstances permit. The computation rule is recursively defined, in that in each substitution step, the recursive function calls chosen for function substituion, depends on those chosen by the computation rule applied to each subexpression individually.

We consider four separate cases: when the expression is a recursive function call (not a primitive or constructor); when the expression is headed by either a data constructor; when the expression is headed by a primitive function occurring within the context of a lambda expression; and when the expression is headed by a primitive function not within the context of a lambda expression (where we need not consider the presence of unbound lambda variables).

Lemma: If the expression is a recursive function call, expanding only the main (single outermost) function call is a safe substitution.

Proof: This is a parallel outermost substitution step, a substitution already proven to be save [V74].

Lemma: If the expression is headed by a data constructor, and if the set of function calls chosen to be expanded is the union of sets computed by applying a safe computation rule individually to each argument, then this is a safe substitution.

Proof: By induction on the height of the term. If the substitution steps calculated for each subterm are safe, then the safety-defining equality holds individually for each argument, and therefore must also hold for the expression as a whole.

Lemma: If the expression is headed by a primitive function occuring within the context of a lambda expression (so that unbound lambda variables may appear in the arguments), then choosing to expand all outermost function calls (parallel outermost) is a safe substitution.

Proof: This is a parallel outermost substitution step, a substitution already proven to be save [V74].

Lemma: Suppose the expression is headed by a primitive function not within the context of a lambda expression, representing a parallel operation not strict in any of its arguments individually. In that case, expanding the function calls in the union of sets computed by applying a safe computation rule individually to each argument is a safe substitution.

Proof: By induction on the height of the term. If the substitution steps calculated for each subterm are safe, then the safety-defining equality holds individually for each argument, and therefore must also hold for the expression as a whole. One example of a primitive not strict in either argument would be the "parallel-AND" primitive, which evaluates to **TRUE** if either argument is true, even if the other argument diverges.

Lemma: Suppose the expression is headed by a primitive function not within the context of a lambda expression, representing an operation strict in at least one of its arguments. Then let *Arg* be any of the arguments in which the primitive is strict, and let *Set* be a set of function calls chosen by a safe computation rule applied to *Arg*. Then any substitution step chosing all the occurrences in *Set* is a safe computation step for that expression.

Proof: Because *Set* was chosen by applying a safe computation rule to *arg*, replacing these recursive function calls by Ω (and the remaining calls by the recursive function fixpoint) will give the save result in *arg* as if we had replaced *all arg*'s function calls by Ω. Either this result is $\perp$, or we already knew the outermost constructor of *arg*. But, we cannot have already known the outermost constructor, or the primitive function would already have simplified. Therefore it is $\perp$. Since the primitive function is strict in that argument, it too evaluates to $\perp$. Thus, the safety equation holds for the primitive function expression, too. Note that if the primitive is strict in several arguments, this computation rule gives us a choice of substitution steps.

These cases are all the possibilities. We must now prove the computation rule is safe.

Theorem: A computation rule which chooses from among the above substitution steps depending upon the situation is safe.

Proof: A safe computation rule is, by definition, one which uses only save computation steps. All the substitution steps described above were proven safe.

The main advantage of this approach over simple parallel outermost is that, when a primitive is strict in an argument, and does not occur within the context of a lambda expression, we need look only in the strict argument for function calls to expand. This gives us some of the computational advantages of the leftmost (outermost) rule, without sacrificing safety.

Irrespective of the need to compute infinite lists (and later sets), some may argue that, there is never any good reason to compute an unapplied function, nor any list structure containing such a function as an element. If one wishes to learn about a function, one can apply it on any number of arguments. Therefore, we only ask that our operational semantics be correct when computing objects from the domain 'E', where

$$\mathtt{E} = (\mathtt{B}_{\perp_B} + \mathtt{A}_{\perp_A} + \mathtt{E}\times\mathtt{E})_{\perp}.$$

Though we will use functions as objects in defining objects in domain 'E', these functions will be either applied or ignored; they will never be included as part of the final answer. With this limitation, we need never compute an object within the context of a lambda expression. The body of a lambda expression needs not be evaluated until after application (β-reduction). Consider an application of the form:

$$(\mathcal{E}[\![\mathtt{Lx.}\ body]\!]\rho_1)(\mathcal{E}[\![arg]\!]\rho_2).$$

Since a β-reduction is strict in the first argument, we only expand the first outermost occurrance of $\mathcal{E}$:

$$(\lambda\ y.\ \mathcal{E}[\![body]\!]\rho_1\ [y/\mathtt{x}])(\mathcal{E}[\![arg]\!]\rho_2).$$

This immediately simplifies to:

$$\mathcal{E}[\![body]\!]\ \rho_1[(\mathcal{E}[\![arg]\!]\rho_2)/\mathtt{x}].$$

The expression '*body*' no longer occurs within the context of a lambda expression. So long as the outermost expression being computed denotes an element of $\mathcal{E}$, we need never compute anything within the context of a lambda expression.

Earlier, we commented that β-reduction of lambda expressions does not always terminate. This can only happen when a lambda expression is applied to another lambda expression, so that one β-reduction enables more. Consider the evaluation of:

$$\textbf{func!}(\mathcal{E}[\![\mathbf{Lx.xx}]\!]\ \rho)(\mathcal{E}[\![\mathbf{Lx.xx}]\!]\ \rho)$$

If we simplify both arguments of this β-reduction simultaniously, we eventually get:

$$(\lambda\ y.yy)(\lambda\ y.yy),$$

a synonym for $\perp$, whose β-reduction will never terminate. We do not want non-termination to be expressed this way. This expression may be only a small piece of the main expression, and we do not want endless simplification to prevent computation of the other parts. When we use the new computation rule, delaying computation of a function until it is needed, the evaluation proceeds in a more orderly fashion:

$$\textbf{func!}(\lambda y.\ \mathcal{E}[\![\mathbf{xx}]\!]\ \rho[y/\mathrm{x}])(\mathcal{E}[\![\mathbf{Lx.xx}]\!]\ \rho)$$

becomes:

$$(\mathcal{E}[\![\mathbf{xx}]\!]\ \rho[(\mathcal{E}[\![\mathbf{Lx.xx}]\!]\rho)\ /\ \mathbf{x}]),$$

which becomes:

$$\textbf{func!}((\mathcal{E}[\![\mathbf{x}]\!]\rho[(\mathcal{E}[\![\mathbf{Lx.xx}]\!]\rho)\ /\ \mathrm{x}])(\mathcal{E}[\![\mathbf{x}]\!]\rho[(\mathcal{E}[\![\mathbf{Lx.xx}]\!]\rho)/\mathbf{x}]).$$

This, in turn, becomes:

$$\textbf{func!}(\mathcal{E}[\![\mathbf{Lx.xx}]\!]\rho)(\mathcal{E}[\![\mathbf{x}]\!]\rho[(\mathcal{E}[\![\mathbf{Lx.xx}]\!]\rho)/\mathbf{x}]).$$

To see that this is getting nowhere, let us do an expansion at E[[x]]:

$$\textbf{func!}(\mathcal{E}[\![\mathbf{Lx.xx}]\!]\rho)(\mathcal{E}[\![\mathbf{Lx.xx}]\!]\rho),$$

which is exactly what we started with. Therefore, all partial computations will simplify to $\perp$, yet because we never evaluate a function until its application, in no computation step need we deal with an infinity of simplifications.

We proved the correctness of our computation rule under the assumption that within in each computation step, only a finite number of simplifications will be available, and then showed that, provided we use this computation rule, the assumption is correct. This completes the circle.

Further optimizations are needed to make the implementation efficient. If each optimization maintains correctness, then the resulting efficient operational semantics will also be correct with respect to the normative denotational description. Much research has already been done on techniques to implement lazy functional languages (see [P87]), and we will not discuss these techniques here. When proposing a language, it is good to show that it *can* be correctly implemented, at least theoretically. We have shown that if the denotational semantics is written carefully, so that all semantic primitives can be viewed as standard simplifications, and one correct implementation is automatically available.

4.5 Implementing PowerFuL

A previous section gave the denotational semantics for PowerFuL. In writing the denotational equations, we took great care to ensure that all primitive functions (except β-reduction) could be defined as qualified as standard simplifications. The denotational equations themselves can be interpreted as a functional program, serving as both a declarative description of the language and an equivalent operational semantics, simultaneously. The computation rule described in Chapter 3 is used, and it is assumed that programs will compute (at the top level) values from the domain 'E', such that:

$$E = (B_{\perp_B} + A_{\perp_A} + E \times E + \mathcal{P}(E))_{\perp}.$$

As in the simpler language used in demonstrating the methodology, functions are only computed in the context of application.

Below is a summary of the PowerFuL primitives.

4.5.1 Function Application

In the semantic equations we treat explicitly only functions of one argument. A multi-argument function can be thought of as syntactic sugar for a curried functions, or for a function taking a sequence as its argument. Application is essentially β-reduction of the lambda calculus. An application is strict in its first argument, the function to be applied.

4.5.2 Boolean Input Primitives

In the semantic domain we use the conditional **if**: $B_\perp \times D \times D \mapsto D$. This primitive is strict in the first argument. The equations defining **if** are:

$$\text{if}(\text{TRUE},\ arg2,\ arg3) = arg2$$
$$\text{if}(\text{FALSE},\ arg2,\ arg3) = arg3$$
$$\text{if}(\perp_B,\ arg2,\ arg3) = \perp_D\ .$$

In both the syntactic and the semantic domains, we shall feel free to express nested conditionals using common sugars such as "if/then/elseif/then/else/fi."

Negation, called **not**: $\mathtt{B}_\perp \mapsto \mathtt{B}_\perp$, is strict in its only argument. Its simplification rules are:

$$\begin{aligned}&\textbf{not}(\textbf{TRUE}) = \textbf{FALSE}\\&\textbf{not}(\textbf{FALSE}) = \textbf{TRUE}\\&\textbf{not}(\perp_B) = \perp_B\ .\end{aligned}$$

4.5.2 Atomic Input Primitives

We assume that (for each program) there is a finite set of atoms (which always includes `'nil`). For each such atom A_i in the syntax there exists a corresponding semantic primitive $\mathbf{A}_i$. These primitives, together with $\perp_A$, make up the subdomain $\mathtt{A}_\perp$. For every atom $\mathbf{A}_i$, there is a primitive function '$\mathbf{isA}_i\mathbf{?}$: $\mathtt{A}_\perp \mapsto \mathtt{B}_\perp$', strict in its only argument. The simplification rules are:

$$\begin{aligned}&\mathbf{isA}_i\mathbf{?}(\perp_A) = \perp_B\\&\mathbf{isA}_i\mathbf{?}(\mathbf{A}_i) = \textbf{TRUE}\\&\mathbf{isA}_i\mathbf{?}(\mathbf{A}_j) = \textbf{FALSE}\quad \text{for } i \neq j\end{aligned}$$

Also provided is **atomeq?**: $\mathtt{A}_\perp \times \mathtt{A}_\perp \rightarrow \mathtt{B}_\perp$, to compare atoms for equality. Strict in both arguments, the simplification rules are:

$$\begin{aligned}&\textbf{atomeq?}(\perp_A,\ \mathit{arg2}) = \perp_B\\&\textbf{atomeq?}(\mathit{arg1},\ \perp_A) = \perp_B\\&\textbf{atomeq?}(\mathbf{A}_i,\ \mathit{arg2}) = \mathbf{isA}_i\mathbf{?}(\mathit{arg2})\\&\textbf{atomeq?}(\mathit{arg1},\ \mathbf{A}_i) = \mathbf{isA}_i\mathbf{?}(\mathit{arg1}).\end{aligned}$$

Note that the third and fourth rules are actually rule schemas, instantiated by each atom $\mathbf{A}_i$.

4.5.3 List Primitives

The primitive functions **left** and **right**, of type $\mathtt{D}\times\mathtt{D} \mapsto \mathtt{D}$, are strict in the single arguments. The simplification rules are:

$$\begin{aligned}&\textbf{left}(<\mathit{1st},\ \mathit{2nd}>) = \mathit{1st}\\&\textbf{right}(<\mathit{1st},\ \mathit{2nd}>) = \mathit{2nd}\ .\end{aligned}$$

4.5.4 Powerdomain Input Primitives

The primitive '+' lets us iterate a function of type `D -> ` $\mathcal{P}$`(D)` over the elements of an input set, combining the results via union into a single new set. It is strict in the second argument. We can define '+' recursively via the rules:

$$\begin{aligned}
&\mathtt{F}^{+}(\phi) = \phi \\
&\mathtt{F}^{+}(\{\mathtt{Expr}\}) = \mathtt{F(Expr)} \\
&\mathtt{F}^{+}(\mathtt{Set}_1 \cup \mathtt{Set}_2) = (\mathtt{F}^{+}(\mathtt{Set}_1) \cup \mathtt{F}^{+}(\mathtt{Set}_2))
\end{aligned}$$

Theoretically, it is also strict in the first argument, since

$$(\perp_{D \mapsto \mathcal{P}(D)})^{+}(set) = \perp_{\mathcal{P}(D)}$$

However, we will ignore this strictness in the operational semantics, as the simplification rules for '+' require knowledge about the second argument.

Theorem: Though '+' is defined recursively, simplifications during computation must terminate.

Proof: Each recursion goes deeper into the union-tree, and, at any stage of computation, such a set will have been computed only to finite depth.

4.5.5 Run-time Type-checking and Coercions

PowerFuL is basically an untyped language. For limited run-time type-checking, we rely on these primitive semantic functions over $\mathtt{D} \mapsto \mathtt{B}_{\perp}$: **atom?**, **bool?**, **pair?**, **func?** and **set?**.

For instance, **func?** returns **TRUE** if the argument is a primitive function or a lambda expression, **FALSE** if the argument is an atom, an ordered pair or a set. The only other possibility is $\perp_D$, so **func?**$(\perp_D)$ rewrites to $\perp_B$. The other type-checking functions are defined analogously.

Most of our primitives are defined over only portions of the domain `D`. The boolean operators are defined only over $\mathtt{B}_{\perp}$. Only ordered pairs have left and right sides. Function application is defined only when the left argument is in fact a function. Only sets can contribute to a set union. Since PowerFuL is an untyped language, we will need a way to coerce arguments to the appropriate type. One way is to use the type-checking primitives in conjuction with typed-if primitives. We find it simpler to define five primitive coercions. They are: **bool!**, **atom!**, **pair!**, **func!** and **set!**.

The function **bool!**: $\mathtt{D} \mapsto \mathtt{B}_\perp$ maps *arg* to itself if *arg* is a member of $\mathtt{B}_\perp$, and to $\perp_B$ otherwise.

The function **atom!**: $\mathtt{D} \mapsto \mathtt{A}_\perp$ maps *arg* to itself if *arg* is a member of $\mathtt{A}_\perp$, and to $\perp_A$ otherwise.

The function **pair!**: $\mathtt{D} \mapsto \mathtt{D}\times\mathtt{D}$ maps *arg* to itself if *arg* is a member of $\mathtt{D}\times\mathtt{D}$, and to $\perp_{D\times D}$ (that is, $<\perp_D, \perp_D>$) otherwise.

The function **func!**: $\mathtt{D} \mapsto [\mathtt{D}\mapsto\mathtt{D}]$ maps *arg* to itself if *arg* is a member of $\mathtt{D}\mapsto\mathtt{D}$ and to $\perp_{D\mapsto D}$ (that is, $\lambda\mathbf{x}.\ \ \perp_D$) otherwise.

The function **set!**: $\mathtt{D} \mapsto \mathcal{P}(\mathtt{D})$. maps *arg* to itself if *arg* is a member of $\mathcal{P}(\mathtt{D})$ and to $\perp_{\mathcal{P}(D)}$ (that is, ϕ) otherwise.

4.5.6 Equality

A first-order object is one whose meaning is identified with its syntactic structure. First-order objects are equal iff they are identical. Equality is the same as identity. They include atoms, booleans, and nested ordered pairs whose leaves are atoms and booleans. Given access to the **atomeq** primitive, the user could write his own equality predicate to test first-order objects for equality. Nevertheless, defining equality as a primitive strict in both arguments frees the interpreter to choose which argument to evaluate first. This can be important when computing certain types of set expressions, as will be seen in a later section. Simplification rules are explained below:

$$\textbf{equal?}(\perp,\ arg2) = \perp_B$$
$$\textbf{equal?}(arg1,\ \perp) = \perp_B\ .$$

If we know anything at all either argument, we know whether it is a member of $\mathtt{B}_\perp$ (a boolean), $\mathtt{A}_\perp$ (an atom), $\mathtt{D}\times\mathtt{D}$ (an ordered pair), $\mathtt{D}\mapsto\mathtt{D}$ (a function) or $\mathcal{P}(\mathtt{D})$ (a set). As soon as we know this about one of the arguments, we can apply one of the following equalities.

If B is known to be a boolean, then

$$\textbf{equal?}(B,\ expr) =$$
$$\textbf{if bool?}(exp)\ \textbf{then if}(B, \textbf{bool!}(exp), \textbf{not}(\textbf{bool!}(exp)))\ \textbf{else FALSE fi}$$

and similarly

$$\textbf{equal?}(expr,\ B) =$$

if **bool?**(exp) **then** **if**(**bool!**($expr$), B, **not**(B)) **else** **FALSE** **fi**

If A is an atom, then

equal?(A, $expr$) =
  if **atom?**(exp) **then** **atomeq?**(A, **atom!**(exp)) **else** **FALSE** **fi**

and similarly

equal?($expr$, A) =
  if **atom?**(exp) **then** **atomeq?**(**atom!**($expr$), A) **else** **FALSE** **fi**

If F is a function, then

equal?(F, $expr$) = **if** **func?**(exp) **then** $\perp_B$ **else** **FALSE** **fi**
equal?($expr$, F) = **if** **func?**(exp) **then** $\perp_B$ **else** **FALSE** **fi**

If S is a set, then

equal?(S, $expr$) = **if** **set?**(exp) **then** $\perp_B$ **else** **FALSE** **fi**
equal?($expr$, S) = **if** **set?**(exp) **then** $\perp_B$ **else** **FALSE** **fi**

If P is an ordered pair, then

```
equal?(P, expr) =
        if not(pair?(exp)) then FALSE
        elseif not(equal?(left(P), left(pair!(exp)))) then FALSE
        else equal?(right(P), right(pair!(exp)))fi
```

and similarly

```
equal?(expr, P) =
        if not(pair?(exp)) then FALSE
        elseif not(equal?(left(pair!(exp)), left(P))) then FALSE
        else equal?(right(pair!(exp)), right(P))fi
```

Theorem: Though this primitive is defined recursively, its application is bound to terminate.

Proof: Each recursion goes deeper into the ordered-pair tree, and at any stage of computation, only a finite portion of any object is available for the primitives to act upon.

Proposition: All our primitives are continuous, and all (except for β-reduction) satisfy Vuillemin's criteria of standard simplifications. If the primitive is strict in one of its arguments, and if the outermost data constructor of that argument is already computed, then the primitive can simplify immediately.

5. OPTIMIZATIONS

This section describes some of the ways the basic operational procedure can be improved. The main thrust will be towards improving the efficiency of set abstraction computation. More general techniques for improving the efficiency of (pure) functional languages are available [P87], but we will not discuss them here.

5.1 Intuition Behind Optimizations

Consider a Horn logic interpreter constructed directly from the fixpoint semantics. The set of all possible instantiations for a logical variable is the *Herbrand universe.* This set is analogous to the set '`terms`' in PowerFuL. It consists of all finite objects which can be built from constructors and atoms. A Horn logic *program* consists of a set of *program clauses* and a *goal clause.* Each program clause represents an infinite set of *ground clauses*, each produced by instantiating the clause's logical variables with elements from the set of terms (Herbrand universe). For instance, if a program clause '*progclause*' has logical variables '`A`' and '`B`', then the program clause represents the set of ground clauses:

$(\lambda \mathtt{A}.\ (\lambda \mathtt{B}.\ \{\mathit{progclause}\})^{+}\, \mathit{HerbrandUniverse})^{+}\, \mathit{HerbrandUniverse}.$

In Horn logic, this is usually written:

$\forall \mathtt{A}. \forall \mathtt{B}. \mathit{progclause},$

often with the quantifiers omitted, but understood. Similarly, in PowerFuL we choose to use the notation:

$\mathit{term}(x).\mathit{body}$

to mean

$(\lambda x.\mathit{body})^{+}\ \mathcal{F}[[\mathtt{terms}]],$

and similarly $\mathit{atom}(x).\mathit{body}$ and $\mathit{bool}(x).\mathit{body}$ for $(\lambda x.\mathit{body})^{+}\ \mathcal{F}[[\mathtt{atoms}]]$, and $(\lambda x.\mathit{body})^{+}\ \mathcal{F}[[\mathtt{bools}]]$, respectively.

In Horn logic, each ground clause is of the form:

P_0 :- P_1, ..., P_n.

Each 'P_i' stands for a ground predicate. The ground clause means, "If ground predicates P_1 through P_n are true, then so is P_0." The predicates P_1 through P_n are referred to as the *body* of the clause; predicate P_0 is referred to as the *head*. When there are no predicates in the body, then the ground clause asserts the head predicate (P_0). Using the set of ground clauses represented by the non-ground program clauses, and *modus ponens* inferences, we can derive many other ground clauses implied by the program. Modus ponens states that, given two ground clauses of the form:

P_0 :- P_1, ..., P_n,

and

Q_0 :- Q_1, ..., Q_m,

if P_1 equals Q_1, then the two clauses imply a third clause:

P_0 :- Q_1, ..., Q_m, P_2,..., P_n.

Some of the ground clauses derived from the program will have no body. In these cases, the program implies that the predicate in the clause head is true. The set of ground predicates implied by the progrm is called the *Herbrand model*. *Herbrand's method* [Q82] is a procedure for computing members of the Herbrand model by doing modus ponens inferences on ground instantiations of the program clauses. This method is closely related to the fixpoint semantics of Horn logic programming. The goal clause is a conjunction of non-ground predicates. An *answer substitution* maps an element from the Herbrand Universe to each logical variable of the goal clause. An answer solution is *correct* if, when applied to the goal clause, it results in a conjuction of ground predicates, each of which is in the Herbrand model. Program execution computes members of the set of correct answer substitutions.

In Section 2, we showed how a Horn logic program could be specified in PowerFuL, using only '`letrec`' (the feature for creating recursive definitions), set abstraction, the conditional and the equality primitive. If we executed this program directly using PowerFuL's denotational equations as the interpreter, the execution would be analogous to solving a problem in Horn logic using Herbrand's method. This "generate-and-test" approach is easy to understand, but inefficient.

Often, members of the set of correct answer substitutions can be grouped into families. Each answer within the family has certain common elements, with the remaining details varying freely. The derivation of one member of the family is almost identical derivation for any other member. Deriving each member of the family individually produces an infinity of essentially similar derivations. This leads to the idea of a *general* answer substitution. A general answer subsititution would instantiate a goal's logical variables only partially, in such a way that any completion of the instantiation would result in a correct answer substitution. The derivation of the general answer subsitution resembles the derivation of any member of the family, but parameterized by those parts to be left uninstantiated.

The resolution method is a technique for deriving general answer substitutions. In resolution, one attempts to perform modus ponens inferences using the non-ground clauses directly, rather than instantiating them first. Logical variables become instantiated only to the extent necessary to satisfy the inference rule's equality test. This partial instantiation to ensure equality of non-ground predicates is called *unification*, and the substitution is called a *unifier*. If a non-ground clause represents a set of ground instantiations, then application of the unifier narrows the the set of ground instantiations to those which could participate in a modus ponens inference with an instantiation of the other clause.

A resolution derivation produces an answer substitution parameterized by unbound logical variables, called a *most general computed answer substitiotion*, each unbound logical variable representing a term from the Herbrand universe. Any ground instantiation of the most general computed answer substitution would be a correct answer substitution. Applying any of these ground instantiations to each line of the derivation would produce a traditional derivation for the associated (ground) correct answer substitution. Clearly, a most general computed answer substitution summarizes in compact form an entire family of correct correct answer substitutions. Each resolution step represents a modus ponens inference in an infinity of concrete fully instantiated derivations. Though it is easy to instantiate a most general answer substitution, to generate all the correct answer subsitituions within the family, this is never done. Reporting results in the general form is much more economical than outputting individually all the (infinite) ways in which each most general answer can be extended. Theoretically, the calculation of correct answer substitutions is left unfinished. Since the remaining work is trivial, we are willing to put up with that.

5.2 Strategy

In PowerFuL, we compute relative set expressions by computing the generator set.

Each time we identify elements of this set, we replace the associated parameter in the remainder of the expression. We then computed the instantiated expression, once for each possible instantiation. We would like to modify this procedure when we know that the generating set is simply the set of terms (or a subset):

$term(x).body$,

$atom(x).body$,

or

$bool(x).body$.

Rather than recomputing the '*body*' for each trivial instantiation, we will evaluate '*body*' in its uninstantiated form, leaving it parameterized by the enumeration variable. The goal is to compute a single parameterized set expression which stands for the union of all possible instantiations. We can do this because expansions of recursive function calls (translation from syntax to semantics) does not depend upon these parameters. Not only is this a more compact form for expressing final results, but we can freely use such parameterized set expressions as generators for other set expressions, since:

$(\lambda x.\ body_1)^+(term(y).\ body_2)$,

can be rewritten as:

$term(y).\ ((\lambda x.\ body_1)^+(body_2))$.

This is because the first expression is an alternate notation for:

$(\lambda x.\ body_1)^+((\lambda y.\ body_2)^+(\mathcal{F}[[\texttt{terms}]]))$,

and the second is an alternate notation for

$((\lambda x.\ body_1)^+(\lambda y.\ body_2))^+(\mathcal{F}[[\texttt{terms}]])$,

and these are equal, due to the associativity of set union.

To compute the parameterized body of:

$term(x).body$

we must consider one possible complication: the simplification of primitives. Primitive simplification *may* depend upon which term is being represented by the parameter. For instance, during a simplification stage of computation of the body, we may find a subexpression of the form $p(x)$, where 'p' is a semantic primitive, and 'x' is a parameter representing an arbitrary term. Were an actual term provided, the primitive might simplify immediately. Therefore, we must have a way to perform simplifications when primitives are applied to parameters.

Often, this presents no problem, as the same simplification would be performed regardless of which term the parameter might symbolize. In such cases, parameterization does not hinder simplification of the primitive. For instance, in

$\mathit{term}(u).(\ldots \mathbf{func?}(u) \ldots)$,

we can simplify '**func?**(u)' to '**FALSE**', since the primitive would so simplify for any object that 'u' might represent. Below is a list of similar cases.

$\mathit{term}(u).(\ldots \mathbf{func?}(u) \ldots) \rightarrow \mathit{term}(u).(\ldots \mathbf{FALSE} \ldots)$

$\mathit{term}(u).(\ldots \mathbf{func!}(u) \ldots) \rightarrow \mathit{term}(u).(\ldots \perp_{D \mapsto D} \ldots)$

$\mathit{atom}(u).(\ldots \mathbf{func?}(u) \ldots) \rightarrow \mathit{atom}(u).(\ldots \mathbf{FALSE} \ldots)$

$\mathit{atom}(u).(\ldots \mathbf{func!}(u) \ldots) \rightarrow \mathit{atom}(u).(\ldots \perp_{D \mapsto D} \ldots)$

$\mathit{bool}(u).(\ldots \mathbf{func?}(u) \ldots) \rightarrow \mathit{bool}(u).(\ldots \mathbf{FALSE} \ldots)$

$\mathit{bool}(u).(\ldots \mathbf{func!}(u) \ldots) \rightarrow \mathit{bool}(u).(\ldots \perp_{D \mapsto D} \ldots)$

$\mathit{term}(u).(\ldots \mathbf{set?}(u) \ldots) \rightarrow \mathit{term}(u).(\ldots \mathbf{FALSE} \ldots)$

$\mathit{term}(u).(\ldots \mathbf{set!}(u) \ldots) \rightarrow \mathit{term}(u).(\ldots \phi \ldots)$

$\mathit{atom}(u).(\ldots \mathbf{set?}(u) \ldots) \rightarrow \mathit{atom}(u).(\ldots \mathbf{FALSE} \ldots)$

$\mathit{atom}(u).(\ldots \mathbf{set!}(u) \ldots) \rightarrow \mathit{atom}(u).(\ldots \phi \ldots)$

$\mathit{bool}(u).(\ldots \mathbf{set?}(u) \ldots) \rightarrow \mathit{bool}(u).(\ldots \mathbf{FALSE} \ldots)$

$\mathit{bool}(u).(\ldots \mathbf{set!}(u) \ldots) \rightarrow \mathit{bool}(u).(\ldots \phi \ldots)$

$\mathit{atom}(u).(\ldots \mathbf{bool?}(u) \ldots) \rightarrow \mathit{atom}(u).(\ldots \mathbf{FALSE} \ldots)$

$\mathit{atom}(u).(\ldots \mathbf{bool!}(u) \ldots) \rightarrow \mathit{atom}(u).(\ldots \perp_B \ldots)$

$\mathit{bool}(u).(\ldots \mathbf{bool?}(u) \ldots) \rightarrow \mathit{bool}(u).(\ldots \mathbf{TRUE} \ldots)$

$\mathit{bool}(u).(\ldots \mathbf{bool!}(u) \ldots) \rightarrow \mathit{bool}(u).(\ldots u \ldots)$

$\mathit{atom}(u).(\ldots \mathbf{atom?}(u) \ldots) \rightarrow \mathit{atom}(u).(\ldots \mathbf{TRUE} \ldots)$

$\mathit{atom}(u).(\ldots \mathbf{atom!}(u) \ldots) \rightarrow \mathit{atom}(u).(\ldots u \ldots)$

$\mathit{bool}(u).(\ldots \mathbf{atom?}(u) \ldots) \rightarrow \mathit{bool}(u).(\ldots \mathbf{FALSE} \ldots)$

$\mathit{bool}(u).(\ldots \mathbf{atom!}(u) \ldots) \rightarrow \mathit{bool}(u).(\ldots \perp_A \ldots)$

$atom(u).(\ldots\mathbf{pair?}(u)\ldots) \rightarrow atom(u).(\ldots\mathbf{FALSE}\ldots)$

$atom(u).(\ldots\mathbf{pair!}(u)\ldots) \rightarrow atom(u).(\ldots\bot_{D\times D}\ldots)$

$bool(u).(\ldots\mathbf{pair?}(u)\ldots) \rightarrow bool(u).(\ldots\mathbf{FALSE}\ldots)$

$bool(u).(\ldots\mathbf{pair!}(u)\ldots) \rightarrow bool(u).(\ldots\bot_{D\times D}\ldots)$

$bool(u).(\ldots\mathbf{equal?}(u,\ ex)\ldots) \rightarrow \mathbf{if}(\ \mathbf{bool?}(ex),\ \mathbf{if}(u, ex, \mathbf{not}(\mathbf{bool!}(ex))),\ \mathbf{FALSE})$

$bool(u).(\ldots\mathbf{equal?}(ex,\ u)\ldots) \rightarrow \mathbf{if}(\ \mathbf{bool?}(ex),\ \mathbf{if}(\mathbf{bool!}(ex), u, \mathbf{not}(u)),\ \mathbf{FALSE})$

$atom(u).(\ldots\mathbf{equal?}(u,\ ex)\ldots) \rightarrow \mathbf{if}(\mathbf{atom?}(ex), \mathbf{atomeq?}(u, \mathbf{atom!}(ex)), \mathbf{FALSE})$

$atom(u).(\ldots\mathbf{equal?}(ex,\ u)\ldots) \rightarrow \mathbf{if}(\mathbf{atom?}(ex), \mathbf{atomeq?}(\mathbf{atom!}(ex), u), \mathbf{FALSE})$

$term(u).(\ldots\mathbf{equal?}(u,\ u)\ldots) \rightarrow term(u).(\ldots\mathbf{TRUE}\ldots)$

$atom(u).(\ldots\mathbf{atomeq?}(u,\ u)\ldots) \rightarrow atom(u).(\ldots\mathbf{TRUE}\ldots).$

Note also that if a parameter is a nonstrict argument of a primitive, the primitive might not simplifify, anyway, no matter what term were substituted. For example, consider

$term(u).(\ldots\mathbf{if}((\mathcal{E}[[exp_1]]\ \rho)u, exp_2)\ldots).$

The primitive '**if**' is strict in its first argument, and would not simplify at this time, regardless of what term might replace 'u'. In such a case, leaving the body parameterized by 'u' is acceptable.

5.3 Partial Instantiation for Type Primitives

Suppose the primitive applied to the parameter is one of these four: '**bool?**', '**bool!**', **atom?**, **atom!**, **pair?** or **pair!**.

We have already described that these could simplify immediately when applied to a parameter symbolizing a boolean or an atom. When the variable is enumerated from '`terms`', the simplification chosen depends upon *which* term. Luckily, we do not need to consider each possibility, individually. The set of terms consists of just three subsets, namely the set of booleans, the set of atoms, and the set of ordered pairs, such that both left and right members of the pair are terms. For each of these subsets, each of the primitives above simplifies in a uniform manner. Let '*prim*' represent one of these four primitives. An expression of the form

$term(u).(\ldots prim(u)\ldots)$

is an alternate notation for

$(\ldots prim(u)\ldots)^{+}(\mathcal{F}[[\texttt{terms}]])$

Since '+' is strict in the second argument, it must be correct to rewrite the argument to:

$(\mathcal{F}[[\texttt{bools}]])$

$\cup\ (\mathcal{F}[[\texttt{atoms}]])$

$\cup\ (term(u).term(v).\ <u,v>)$.

Distributing $(\ldots prim(u)\ldots)$ over the union yields:

$(\ldots prim(u)\ldots)^{+}(\mathcal{F}[[\texttt{bools}]])$

$\cup\ (\ldots prim(u)\ldots)^{+}(\mathcal{F}[[\texttt{atoms}]])$

$\cup\ (\ldots prim(u)\ldots)^{+}(term(v).term(w).\ <v,w>)$.

This is equivalent to:

$bool(u).(\ldots prim(u)\ldots)$

$\cup\ atom(u).(\ldots prim(u)\ldots)$

$\cup\ term(v).term(w).(\ldots prim(u)\ldots)[<v,w>/u]$.

In all three subsets, the primitive function simplifies immediately. For each branch of the union, we can continue evaluating the body. For each branch, we have partially instantiated the parameter 'u'. In the first branch, we have instantiated it to represent a boolean. In the second branch, we have instantiated it to represent an atom. In the third branch, we have instantiated it to represent a term which is an ordered pair. This is analogous to the use of most general unifiers in Horn logic resolution. Resolution prepares two clauses for modus ponens by applying a unifier that is *most general*. It instantiates the clauses no more than necessary to satisfy the equality requirement.

One difference is that in Horn logic does not use negative information, so we are only concerned with instantiations to make the equality true. In PowerFuL, we are concerned with *all* possible outcomes. Actually, some variations of Horn logic do consider negative information through the use of an inequality predicate and negative unifiers [N85] [K84]. We discuss primitives based on equality in the next section.

5.4 Simplifying Equalities

Equality is strict in both arguments, and simplifies whenever the type of either argument is known. However, it simplifies to an expression which, to compute any value, must know (before all else) whether the remaining argument is an atom, boolean, ordered pair, set or function. If one argument is a lambda variable enumerated from the set of terms, it is a minor modification to delay this trivial simplification until we know the subdomain (bool, atom, pair, set or function) of other argument. That way, the preliminary computation of the other argument need be done only once, rather than once for each of the three subsets comprising '`terms`'. The worst that could happen is that computation of the other argument might diverge. In that case, we would never be able to compute the predicate anyway, whether or not we performed the trivial simplification on the basis of the term variable.

If both arguments of the equality predicate are parameters, we cannot delay them both. Suppose we have an expression of the form:

$term(u).(\ldots term(v).(\ldots \mathbf{equal?}(v,\ u)\ldots))$

(remember that '$\mathbf{equal?}(u,\ u)$' immediately simplifies to '**TRUE**). Theoretically, one could break this into an infinity of special cases, in each case u and v each being replaced by an element of the set of terms. For some combinations the predicate '**equal?**' would simplify to '**TRUE**', and '**FALSE**' for other combinations. This could also have been done with the primitives described earlier, but we preferred to split the enumerated set into subsets, so that for each subset the predicated simplified the same way. That way we could deal with whole categories of terms simultaneously. Here we have two sets to enumerate, and splitting them into atoms, booleans and ordered pairs does not work. The subset handling the cases in which the two terms are equal can be summarized by replacing all occurrences of 'v' with occurrences of 'u':

$term(u).(\ldots term(v).(\ldots \mathbf{equal?}(v,\ u)\ldots)\ [v/u]).$

This then simplies directly to

$term(u).(\ldots.(\ldots \mathbf{TRUE}\ldots)\ [v/u]).$

It is easy to see that this is the case. Since there are no more occurances of 'v' in the body of '$term(v).body_2$', we are taking the union of instantiations by 'v' in which all possible instantiations of '$body_2$' are identical (since the body no longer depends on 'v'). Clearly, '$term(v).body_2$' can now be replaced by '$body_2$'. (In fact, this simplification can be performed whenever a body does not depend on the enumerating variable. For instance, the expression '$term(x).\phi$' can certainly be replaced by 'ϕ'.)

We also need to summarize the cases when u and v are *not* equal. This could be summarized by

$$term(u).(\ldots term(v).\textbf{if not(equal?}(v,\ u)) \textbf{ then } (\ldots \textbf{FALSE} \ldots) \textbf{ else } \phi).$$

This summarizes the elements of the set for which which the two terms 'u' and 'v' are *not equal.* Is there a way to compute this further, without trying individually all possible combinations of unequal terms?

Lee Naish [N85] proposes for Prolog an inequality predicate, defined on terms. His inequality predicate would fail when two terms are identical, succeed when two terms are identical, and delay when two terms are unifiable, but not identical. In the last case, the other subgoals would be executed first, until the values of logic variables have been instantiated enough to prove either the terms' equality or their inequality. If all other subgoals succeed, without instantiating the variables enough, Naish's Prolog gives an error message. This is not ideal behavior, since unequal instantiations can certainly be computed. A better alternative would be to make the inequality part of the solution, as a kind of negative unifier. Khabaza describes a way in which this can be done [K84]. In essence, the inequality becomes part of the general solution. Specific ground solutions can be generated from the general solutions by instantiating logical variables in all possible ways *subject to the inequality constraint.* Constraint logic programming [JL87] sets another precedent for this approach. We accept general non-ground solutions, because it yields great efficiency, and because replacing term variables by arbitrary ground terms is such a trivial operation. Requiring such term enumerations to satisfy a few inequalities adds little to the complexity of the output, and makes it more compact.

To express such a constraint, we could write the above subset as:

$$term(u).(\ldots term(v) u \neq v.(\ldots \textbf{FALSE} \ldots)).$$

We have simplified the equality predicate by splitting into two expresions: one expression representing the cases for which the equality holds, and the other expression representing the cases for which it is false, without the need to consider every case individually.

Solving subsequent inequalities result in a constraint which is a conjunction of inequalities. If the satisfaction of other predicates cause 'u' and 'v' to become refined into the ordered pairs, '$< u_1, u_2 >$' and '$< v_1, v_2 >$', respectively, then the inequality '$u \neq v$' will become '$< u_1, u_2 > \neq < v_1, v_2 >$', which simplifies to '$or(u_1 \neq v_1,\ u_2 \neq v_2)$'. In general, the total constraint will be an and/or tree of simple inequalities. As these simple constraints are satisfied, they can be replaced by '**TRUE**'. Those inequalities which become unsatisfiable can be replaced by '**FALSE**', leading to further simplifications of the and/or tree. If the whole tree simplifies to '**FALSE**', then we are enumerating an empty set, and the whole expression within can be replaced by ϕ. Similar techniques are used for the predicates '**atomeq?**' and '**isA$_i$?**'.

We summarize the optimizations relating to equality below. For the inequality of two term variables:

$term(u).(\ldots term(v)\ldots constraint.(\ldots \mathbf{equal?}(v,\ u)\ldots))$

can be replaced by

$term(u).(\ldots constraint(\ldots \mathbf{equal?}(v,\ u)\ldots)\ [v/u])$

$\cup\ term(u).(\ldots term(v)\mathrm{and}(constraint, (u \neq v)).(\ldots \mathbf{FALSE}\ldots)).$

For the inequality of two atom variables, we have:

$atom(u).(\ldots atom(v)\ldots constraint.(\ldots \mathbf{atomeq?}(v,\ u)\ldots))$

replaced by

$atom(u).(\ldots constraint(\ldots \mathbf{atomeq?}(v,\ u)\ldots)\ [v/u])$

$\cup\ atom(u).(\ldots atom(v)\mathrm{and}(constraint, (u \neq v)).(\ldots \mathbf{FALSE}\ldots)).$

When comparing an atom variable to a specific atom we have:

$atom(u).(\ldots constraint.(\ldots \mathbf{isA}_i?(u)\ldots))$

(where '**A$_i$?**' is a particular atom), is replaced by

$(\ldots constraint(\ldots \mathbf{isA}_i?(u)\ldots)\ [u/\mathbf{A}_i]).$

$\cup\ atom(u).(\ldots \mathbf{and}(constraint, (u \neq \mathbf{A}_i)).(\ldots \mathbf{FALSE}\ldots)).$

Note that as soon as the substitutions are performed, the predicates in question will be ready to simplify, using optimizations described earlier.

These optimizations are of course symmetrical in the order of arguments to '**equal?**' and '**atomeq?**'.

5.5 Boolean Primitives

When faced with an expression of the form

$bool(u).\ body,$

and within '*body*' is an occurrence of '**not**(u)' or '**if**$(u,\ exp_1,\ exp_2)$', then simplification requires the specific value 'u' represents. Since the set of booleans is very small, the default evaluation of '+' is good enough. The default evaluation (enumerate '`bools`' first) results in this step:

$$bool(u).exp \quad \rightarrow \quad (exp)^{+}\{\text{TRUE}\} \cup (exp)^{+}\{\text{FALSE}\}.$$

5.6 Putting It All Together

Using the procedure set forth in this chapter, it is often possible to avoid full enumeration from the special sets '`terms`', '`atoms`' and '`bools`'. Where a relative set expression is enumerated by the set of terms, we treat the enumeration parameter as a logical variable. An enumeration variable from the set '`atoms`' is simply a logical variable carrying the constraint that it must be bound to an atom, and analogously for enumeration variables from '`bools`'. Inequality constraints relating two logical variables are also handled. Compuation with such 'logical variables' and constraints gives the set abstraction facility the power of logic programming, even some of the expressiveness of constraint logic programming.

Yet, a relative set abstraction is not limited to using '`terms`' as a generating set. Semantically, the set '`terms`' is no different from any other set the user might construct, and theoretically could be executed the same way. The logical variable is an operational concept which improves the execution efficiency when using the set of terms. With logical variables, one evaluates the generating set (the second argument of '+') only as needed to compute the body (the first argument of '+').

This mechanism is practical because the generating set '`terms`' is so simple in structure. Wherever a logical variable (a term parameter) is the argument of a primitive function, and the primitive function needs more information about its argument to execute, the generating set is divided into a few subsets, thereby dividing the whole expression into subsets. In each subset, the range of the logical variable is narrowed enough that the primitive has enough information to execute.

There are three ways to narrow the range of the logical variable. The choice depends upon the primitive being applied, and the constraints already in force. If the logical variable is constrained to be a boolean, and the primitive a boolean operations, then the expression divides into two cases, one case in which the logical variable must be bound to '**TRUE**', the other in which it must be bound to '**FALSE**'. If the primitive function compares two logical variables for equality (or an atom variable with an atom), then the expression divides into two cases, one case in which the equality holds, and the other case in which, for all instantiations, the equality is constrained not to hold. For all other primitives needing more information about a logical variable, it is enought to know whether the variable refers to an atom, a boolean, or an ordered pair of terms. In this case, the generating set is narrowed into precisely these three subsets.

For efficiency, it is a good idea to do first those simplifications which do not split the computation into subcases, then those which split into two subcases and save for last those requiring a three-way split. Eventally, we are left with subsets described in a general way, parameterized by variables representing arbitrary terms, atoms or booleans.

Soundness Theorem: If t_i is a partially computed parameterized set expression, and t_i' is an approximation produced by setting all unevaluated function calls ($\mathcal{D}$, $\mathcal{E}$ or $\mathcal{F}$) to $\perp$, then for every instantiation σ replacing parameters with terms satisfying the constraints, $t_i'\sigma$ approximates a subset of $\lim_{i \to \infty} t_i$'.

Proof: The theorem is true because of the meaing of a parameterized expression (in terms of '$^+$'), and the fact that all steps in a parameterized derivation replace expressions by equals.

Completeness Theorem: A parameterized derivation computes (at least implicitly) all members of the set.

Proof: This theorem is true because when dividing a parameterized expression into cases (for the purpose of simplifying a primitive), every possible instantiation of logical variables (parameters) which satisfies the constraints is a possible instantiation of one of the subcases. No possible instantiation is ever lost.

6. CONCLUSIONS

Proponents of declarative programming languages have long called for the combination of functional and logic programming styles into a single declarative language. Most difficult has been the problem of maintaining functions (and other higher-order constructions) as first-class objects, without losing referential transparency and practical efficiency. Our work contains a solution to this problem. We designed a small, elegant, orthogonal language to meets the objectives. Representative sample programs attest to the power and generality of the language.

A short, but thorough denotational description maps the syntax on to well-understood semantic primitives. All primitives are continuous (computable). Of special interest is the novel use of angelic powerdomains. Although powerdomain theory was developed to described non-deterministic languages, we use powerdomains to provide the semantics for an explicit data type, —relative set abstraction.

To derive an operational semantics, we extended Vuillemin's theory of correct implementation of recursion to accept rewrite rules as a notation for describing recursive functions. We developed a computation rule more efficient than the parallel outermost rule, but a correct computation rule, nonetheless, as established by a proof of its safety. With these developments, the denotational equations themselves serve as an interpreter of the new language. Use of this methodology requires that denotational equations handle most recursion explicitly, as primitive functions must always terminate.

Of special interest in logic programming is the set of terms (objects for which identity is synonymous with equality). When the set of terms is used as a generating set in a relative set abstraction, we showed that, for greater operational efficiency, the enumeration parameter can be treated as a logical variable. In the general case, however, the enumeration parameters are instantiated by the various generator set elements, as these elements are computed. Thus, we *need not* arbitrarily restrict generator sets to contain first-order types.

Expensive operational mechanisms (e.g. higher-order unification, general theorem-proving and unrestricted narrowing) are often associated with higher-order functional and logic programming combinations. Ordinary higher-order functional languages avoid these difficulties, as they propagate higher-order objects via one-way substitution, and they define equality only over first-order objects. By retaining these characteristics, our approach also avoids computationally difficult primitives. We have shown that logic programming can be combined with higher-order lazy functional programming in a way that is not only aesthetically pleasing, but also operationally feasible.

References

[A82] S. Abramsky, "On Semantic Foundations for Applicative Multiprogramming," In *LNCS 154: Proc. 10th ICALP*, Springer, Berlin, 1982, pp. 1-14.

[A83] S. Abramsky, "Experiments, Powerdomains, and Fully Abstract Models for Applicative Multiprogramming," In *LNCS 158: Foundations of Computation Theory*, Springer, Berlin, 1983, pp. 1-13.

[B85] M. Broy, "Extensional Behavior of Concurrent, Nondeterministic, and Communicating Systems," In *Control-flow and Data-flow Concepts of Distributed Programming*, Springer-Verlag, 1985, pp. 229-276.

[BL86] M. Bellia and G. Levi, "The Relation between Logic and Functional Languages: A Survey," In *J. of Logic Programming*, vol. 3, pp.217-236, 1986.

[CM81] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.

[DP85] N. Dershowitz and D. A. Plaisted, "Applicative Programming *cum* Logic Programming," In *1985 Symp. on Logic Programming*, Boston, MA, July 1985, pp. 54–66.

[D83] J. Darlington, "Unification of Functional and Logic Programming," unpublished manuscript, 1983.

[DFP86] J. Darlington, A.J. Field, and H. Pull, "Unification of Functional and Logic Languages," In DeGroot and Lindstrom (eds.), *Logic Programming, Relations, Functions and Equations*, pp. 37-70, Prentice-Hall, 1986.

[GM84] J. A. Goguen and J. Meseguer, "Equality, Types, Modules, and (Why Not?) Generics for Logic Programming," *J. Logic Prog.*, Vol. 2, pp. 179–210, 1984.

[JL87] J. Jaffar, J.-L. Lassez, "Constraint Logic Programming," In *14th ACM POPL*, pp. 111-119, Munich, 1987.

[JS86] B. Jayaraman and F.S.K. Silbermann, "Equations, Sets, and Reduction Semantics for Functional and Logic Programming," In *1986 ACM Conf. on LISP and Functional Programming*, Boston, MA, Aug. 1986, pp. 320-331.

[K84] T. Khabaza, "Negation as Failure and Parallelism." In *Internatl. Symp. Logic Programming, IEEE*, Atlantic City 1984, pp. 70–75.

[L85] G. Lindstrom, "Functional Programming and the Logical Variable," In *12th ACM Symp. on Princ. of Prog. Langs.*, New Orleans, LA, Jan. 1985, pp. 266–280.

[M65] J. McCarthy, et al, "LISP 1.5 Programmer's Manual," MIT Press, Cambridge, Mass., 1965.

[M74] Z. Manna,, "Mathematical Theory of Computation," McGraw-Hill Inc., New York, 1974.

[MMW84] Y. Malachi, Z. Manna, and R. Waldinger, "TABLOG: The Deductive-Tableau Programming Language," In *ACM Symp. on LISP and Functional Programming*, Austin, TX, Aug. 1984, pp. 323–330.

[MN86] D. Miller and G. Nadathur, "Higher-Order Logic Programming," In *Third International Conference on Logic Programming*, London, July 1986, 448-462.

[N85] L. Naish, "Negation and Control in Prolog," Doctoral Dissertation, University of Melbourne, 1985.

[P87] S.L. Peyton Jones, "The Implementation of Functional Programming Languages," Prentice-Hall, 1987.

[R85] U. S. Reddy, "Narrowing as the Operational Semantics of Functional Languages," In *1985 Symp. on Logic Programming*, Boston, MA, July 1985, pp. 138–151.

[R86] J. A. Robinson, "The Future of Logic Programming," IFIP Proceedings, Ireland, 1986.

[S77] J. E. Stoy, "Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory," MIT Press, Cambridge, Mass., 1977.

[S86] D. A. Schmidt, "Denotational Semantics: A Methodology for Language Development," Allyn and Bacon, Inc., Newton, Mass., 1986.

[SP85] G. Smolka and P. Panangaden, "A Higher-order Language with Unification and Multiple Results," Tech. Report TR 85-685, Cornell University, May 1985.

[T81] D. A. Turner, "The semantic elegance of applicative languages," In *ACM Symp. on Func. Prog. and Comp. Arch.*, New Hampshire, October, 1981, pp. 85-92.

[V74] J. Vuillemin, "Correct and Optimal Implementations of Recursion in a Simple Programming Language" Journal of Computer and System Sciences **9**, 1974, 332-354.

[W83] D. H. D. Warren, "Higher-order Extensions of Prolog: Are they needed?" Machine Intelligence **10**, 1982, 441-454.

[YS86] J-H. You and P. A. Subrahmanyam, "Equational Logic Programming: an Extension to Equational Programming," In *13th ACM Symp. on Princ. of*

Prog. Langs., St. Petersburg, FL, 1986, pp. 209-218.