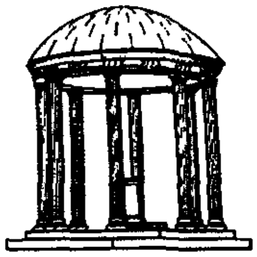
Semantics of Lazy Higher-Order Functional and Logic Programming

TR89-021
(Revision of TR88-004, April 1988)
May 1989

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



Semantics of Lazy Higher-Order Functional and Logic Programming[†]

Status Report

Frank S.K. Silbermann
Bharat Jayaraman

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

Abstract

This paper addresses the semantic and computational issues of set abstraction in functional and logic programming. The main results are:

(i) *Relative set abstraction* can combine a lazy higher-order functional programming with not only first-order Horn logic, but also with a useful subset of higher-order Horn logic. Sets, as well as functions, can be treated as first-class objects.

(ii) *Angelic powerdomains* provide the semantic foundation. These are compatible with lazy evaluation and are well-defined over elements from even non-flat (higher-order) domains.

(iii) A new computation rule, more efficient than the parallel-outermost rule, is developed and shown to be a correct computation rule. (A simple left-most rule is not adequate for this language.)

(iv) Optimizations incorporating ideas from narrowing and resolution greatly improve the efficiency of the interpreter, while maintaining correctness.

† *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

Algorithms are described as a combination of program logic and control [K79]. So that programs may run as efficiently as possible (on conventional sequential machines), users of conventional imperative programming languages express the program control explicitly, leave the program logic implicit (as invariant assertions). Imperative languages can thus be described as *machine oriented.*

In contrast, declarative programming languages are programmer-oriented. Users of declarative languages express the program logic explicitly, leaving much of the control implicit. Based on well-known mathematical theories, declarative languages are often very simple, with semantic descriptions that are both short and elegant. The use of declarative languages has been limited by expensive evaluation procedures, but as the ratio of programmer costs to hardware costs rises, and with programs becoming longer and more complex, declarative languages are becoming ever more attractive. Furthermore, by not overspecifying the order of operations, declarative languages show great potential for implementation on massively parallel hardware. Two of the most popular declarative paradigms are functional and logic programming. The obvious difference is that functional programming is based on function definition and application, whereas logic program define and reason with relations. Each paradigm has its own advantages over the other, which we will describe.

Functional programming offers powerful abstraction tools difficult to incorporate into the logic-programming framework. Infinite data objects such as streams, computed through lazy evaluation, permit one to model input-output within the language proper, as well as concurrent processes (as when separating producers from consumers of data). Higher-order objects (such as functions which take other functions as arguments, and return functions as results), permit the writing of more general-purpose program fragments, increasing reusability. Static scoping provides program modularization. Through the nesting of function application, functional languages can incorporate more control information within the declarative framework than can Horn logic languages. Equivalent logic programs sometimes require a semantically unclear combination of declarative and metalogical constructs. The control information given by function nesting permits efficient deterministic implementations (do not require backtracking). Furthermore, for some functional languages powerful compilation techniqes exist.

From the perspective of functional programming, Predicate-logic programming has its own unique capabilities. Among them are support for constraint reasoning (via unification over first-order terms) and flexible execution moding (non-directionality).

Some problems can be described more naturally using functions. Others are more naturally described using relations. Our goal is a language incorporating the advantages of both functional and logic programming.

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.

We chose to make functional programming rather than Horn logic the basis for our unified declarative language so that simple propagation of objects can be managed without unification or proofs of equality, and so that ordinary functional computations may be performed in the usual way without backtracking. Pure Horn logic programming is based on very few language constructs, functional programming languages are, in contrast, much richer. It should be easier to add the features from a small language into a larger, than vice-versa.

In comparing logic programming with functional programming, logic programming is often described as *relational.* Relations can be expressed by predicates, or alternatively, in terms of set theory. In fact, Horn logic's model-theoretic and fixed-point semantics are described in terms of set theory. We have chosen to supplement a functional programming language with *set abstraction* to express relations. The functional programming paradigm can be enlarged to accept these sets as just another data type.

We are not the first to propose set abstraction as a possible solution to this problem. Darlington [D83] and Robinson [R86] were early advocates of this approach. Nevertheless, their work left several important open problems: In what way does this construct interact with other traditional functional language features, such as infinite and higher-order objects? How can the presence of this feature be reflected in the language's denotational semantics? Will all denotable sets be computable? This paper answers these questions. To our knowledge, these semantic issues have never been rigorously worked out, though Darlingtons recent paper on absolute set abstraction informally sketched an operational procedure for computing some sets [DFP86].

We observe that *relative* set abstraction can *also* provide the needed logic programming capability. We prefer relative set abstraction because it has a more tractable higher-order generalization. Generalizing absolute set abstraction to the higher-order case is thought to require higher-order unification, which is in general undecidable. To achieve the effects of first-order absolute set abstraction using relative set abstraction, one simply replaces each "logical variable" of the absolute set abstraction by a enumeration variable whose generator is the Herbrand universe (i.e. the set of first-order terms). Relative set abstraction's naive generate-test-compute strategy must be improved for solving such abstractions efficiently. We show that these set abstractions generated from the Herbrand universe can be identified, and optimized to provide efficiency comparable to Darlington's procedure.

In this paper, we describe a lazy, statically-scoped, higher-order functional language with set abstraction. In proposing any new language, it is customary to specify not only the syntax and give examples (which we do in Section 3), but also:

a) a standard declarative semantic description (the denotational semantics);

b) a non-standard operational semantics (to show how programs might be executed); and

c) a proof that the standard and the operational semantics describe the same language, i.e. that the operational semantics is correct with respect to the standard semantics.

Section 4 provides a through denotational semantics. Only a few new semantic primitives are needed — the angelic powerdomain constructors and deconstructor, proven to be well-defined and continuous. To obtain a correct operational semantics, we depart from the traditional approach of defining the denotational and an operational semantics separately (and then proving their equivalence). Rather, we derive a correct operational semantics from the denotational semantics in two steps. In Section 5 we show how the denotational equations can be viewed as a program for the interpreter, written in terms of the primitive operations, leaving unspecified only the question of evaluation order. To obtain a correct computation rule for this meta-program, we build upon Vuillemin's work on *safe computation rules* [V74], obtaining an optimized form of parallel-outermost evaluation. That some degree of parallel evaluation is needed for complete evaluation of sets should not be surprising; after all, obtaining a *complete* interpreter for Horn-logic requires breadth-first evaluation. The second step, described in Section 6, is to avoid the default generate-and-test procedure when the Herbrand universe (the set of first-order terms) is recognized as the generator of a relative set abstraction. For this generating set, we partially instantiate the enumerated variable only as needed, using program transformation rules inspired by narrowing in term rewriting systems [R85] and by resolution in logic programming [L87]. We present typical 'optimizations' to the computation procedure to obtain the desired operational semantics.

The next section gives an overview of other attempts to integrate functional and logic programming, including a description of early work in set abstraction.

2. RELATED WORK

Many attempts have been made to combine features of functional and logic programming into a single language (see [BL86] for a recent survey).

2.1 First-order Approaches

One approach is to add features to logic programming, such as unification relative to an equational theory (the equational theory is used to define functions) [K83] [JLM84]. The complexity of the refutation procedure is a difficulty.

A related but simpler approach is the use of equational languages for functional programming, with narrowing to solve constraints posed as equations [GM84]. The equational langage is defined in terms of rewriting (reduction). To solve for logical variables in equations, one reduces the equation via the rewrite rules as much as possible. When the logical variables prevent futher reduction, they are minimally bound so that reduction may continue. When the equality is satisfied, the accumulated bindings provide a solution. For completeness, each time a logical variable is narrowed, one must compute (in parallel) many alternative narrowings.

Narrowing is complete for canonical term rewriting systems. These are equational theories (programs) whose rewrite rules are confluent, and for which all reduction sequences are guarranteed to terminate. The termination requirement rules out functions and relations operating on infinite data structures. Constructor-based equational programming [F84] [JS86] can avoid this problem. The distinction between functions and data constructors permits distinction between equations which define functions and equations which can only be viewed a program properties. The former restricts the left side of a rewrite rule to contain only one functor, placed at the outermost. With such a restriction, narrowing is complete even if the finite termination property does not hold [F84b].

2.2 Higher-order Approaches

Aside from approaches which support no higher-order programming at all [GM84, DP85, YS86], existing approaches fall short in that they either:

(a) require computationally difficult primitives higher-order unification [MN86, R86] (undecidable in worst case), unification relative to an equational theory [GM84],

(b) are not *purely* declarative [SP85, W83, R82, DFP86], or

(c) no denotational (or other declarative) semantics given [L85, SP85, W83, R82, R85, DFP86].

One line of research is higher-order logic programming. This requires a higher-order unification algorithm. However, computationally feasible algorithms do not exist for the general case, so most approaches restrict the higher-order capability to handle only specific subclasses of functions. Miller and Nadathur propose an higher-order extension of Prolog based on Church's typed lambda calculus, using a higher-order unification algorithm. This algorithm may work efficiently when applied to first-order terms, but may be prohibitively expensive when unifying higher-order objects [MN86]. D. H. D. Warren [W83] described a way to encode a some higher-order Horn logic programs within first-order Prolog. In Warren's method, the programmer dedicates special terms to denote the predicates he wishes to pass as arguments. The "higher-order" predicates accept these terms, and calls a small interpreter to apply them. While we consider Warren's encoding a useful Prolog programming technique, as a language extension it violates the referential transparency principle. Syntactic unification of such encodings is inconsistent with their interpretation as higher-order objects, yet Warren's technique permits this.

For first-class higher-order objects, the functional programming paradigm seems more tractable. In traditional functional programming, function arguments replace parameters via one-way substitution, not unification. A better approach might be to incorporate relational programming within the functional framework, rather than vice-versa. Three well-known prototypes are Robinson's *LOGLISP* [R82], Darlingtons extension of *Hope* [D83, DFP86], and Smolka's *Fresh* [SP85].

Smolka begins with a functional language incorporating pattern-matching and adds a Prolog-like capability. The resulting language is very expressive, but it is unclear what would be a meaningful purely declarative subset. The operational semantics given indicates that referential transparency is *not* maintained. No denotational description is offered.

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 calls this approach *absolute set abstraction* (to distinguish it from *relative set abstraction*, discussed later). Absolute set abstraction permits expressions such as

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

to denote define the set of all x satisfying $p(x)$. In this approach, nondeterminism is replaced by set union, and unification is performed to satisfy equations between non-ground objects.

Robinson's language, LOGLISP, attempts to combine LISP and Horn logic through this mechanism. He develops many useful implementation ideas, but as with Smolka, he fails to develop a mathematical justification for his design. Since the base language is LISP, LOGLISP has some higher-order capability, though its use is restricted when accessing the relational features. The evaluation order is applicative, not lazy. Darlington's approach is similar, however his base functional language is lazy, with polymorphic typing. In his recent paper [DFP86], Darlington sketched only a partial and informal operational semantics. To our knowledge, the semantics of set abstraction in functional cum logic programming has (until now) never been rigorously worked out.

The degree with which this construct can exist as a first-class object, and interact freely with other functional language features has been questioned. Both Darlington and Robinson claim that a first-class implementation of absolute set abstraction in a higher-order language would require higher-order unification. Even then, some higher-order programs would be merely unexecutable program specifications. Robinson has criticized existing combinations of higher-order functional programming with first-order relational programming as inelegant [R86]. The goal is to create a purely declarative functional language permitting higher-order relational programming, without arbitrary unorthogonal restrictions on it features. This we do by replacing absolute set abstraction with the semantically simpler ***relative set abstraction.*** A typical relative set abstraction would be an expression of the form:

```
{f(x) : x ∈ M and C(x)}.
```

Here, the generating set '`M`' is provided explicitly, and those elements '`x`' which satisfy the condition '`C`' are used in computing elements of the new set. Compare this to the form of a typical absolute set abstraction:

```
{f(x) : C(x)}.
```

Here, one "solves" the condition '`C`' for suitable values of '`x`', each solution used to compute an element '`f(x)`' of the denoted set.

In principle, the absolute construct is more powerful (this is why its higher-order extension is so problematic). In practice, this is not necessarily true. In the languages of Darlington and Robinson, a logical variable can represent a value only from a special limited domain. This domain consists of the first-order terms, analogous to the *Herbrand universe* in first-order Horn logic. This set of first-order terms, `T`, can easily be expressed via a recursively-defined relative set abstraction. Thus, any first-order absolute set abstraction can easily be expressed as a relative set abstraction. For instance, the example above would be written as:

```
{f(x) : x ∈ T and C(x)}.
```

David Turner pioneered relative set abstraction in KRC [T81]. However, he did not take care to preserve the semantics of true sets. In his languages, sets are implemented as lists, and may be accessed as such, thus adding an implicit ordering on the elements. Turner's evaluation mechanism does not ensure fairness. If computation with the one element of the generator diverges, the next element is never tried. Because of his implementation, Turner's abstractions are now referred to as *list comprehensions* [P87], not sets.

We advocate *true* relative set abstraction. Not only is it as expressive as first-order absolute set abstraction (as shown above), but it can mix freely with higher-order constructs, without requiring arbitrary first-order restrictions. In our system, the set of first-order terms is provided as a (semantically unnecessary but operationally convenient) primitive. In computing a relative set abstraction, only if the variable '`x`' is recognized as being enumerated from the set of first-order terms, is it treated as a logical variable. This special treatment is merely an optimization to the default generate and test mechanism.

3. LANGUAGE DEFINITION

3.1 Syntax and Constructs

In this section we describe the set abstraction construct, and show its use in combining functional and logic programming. As we wish to concentrate of semantic foundations, we instead restrict our consideration to essential features, without providing all the syntactic niceties for programming convenience. For simplicity, we consider neither (polymorphic) typing nor numeric operations.

We refer to this skeleton language as **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
                | λ identifier . 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 ::=  identifier ∈ expr
condition   ::=  expr
```

Enumerations are the syntactic basis for relative set abstraction . Each identifier introduced within the set-clause is associated with a set expression to provide possible values. 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 between higher-order objects is not defined. The result of equating higher-order objects, such as sets or functions, is $\perp$.

3.2 Examples

Functional Programming

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

Higher-order functions and infinite objects can be defined in the usual manner. The `map` example shown above is in curried form.

Set Operations

```
letrec
    crossprod be λ s1 s2. {cons(X,Y) : X∈s1, Y∈s2}
    filter be λ p s. {X : X ∈ s, p(x)}
    intersection be λ s1 s2. {X : X∈s1, Y∈s2, 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 λ list. { [X|Y] : X∈terms, Y∈terms, append(X,Y)=list}
   append be λ l1 l2. if null?(l1) then l2
                      else cons(car(l1), append(cdr(l1),l2)) fi
in
   ...
```

The enumerations **X∈terms, Y∈terms** 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* ∈ *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 '∈'. But we prefer to view the clauses as defining sets, with '∈' 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 ∈ *generating-set*

is equivalent to the conjunction

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

Converting to PowerFuL syntax results in:

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

We have taken the liberty of writing `h,t,y,z ∈ terms` 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 λ v. 'a
     two be λ v. 'b
     three be λ v. 'c
  in
    {F :  F ∈ U({one}, {two}, {three}), 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.

4. DENOTATIONAL SEMANTICS FOR POWERFUL

Denotational semantics has become an essential tool of programming language design. The denotational description provides a deeper understanding of the computational theory being accessed. Also, the convention of using denotational semantics leads us to better language designs, since elegant orthogonal languages with referential transparency have simpler denotational descriptions. By specifying only what is essential, denotational semantics are an especially appropriate choice for the language's standard definition.

4.1 Powerdomains

The denotational semantics of set abstraction requires powerdomain theory. 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.

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 $\lambda x.\ g^{+}(f(x))$. The '$^{+}$' functional is of type

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

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

The larger the set denoted by `f(x)` is, the larger the set denoted by $\mathtt{g}^{+}(\mathtt{f(x)})$ 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.

Rather than letting sets be the implicit result of nondeterminism we make set abstraction an explicit date type. Several well-known powerdomain constructions are available to choose from: the Egli-Milner (or Plotkin), the demonic (or Smythe), or the angelic (or general relational) powerdomain. The key to the correct choice lies in the semantics of Horn-logic programs. According to the model-theoretic and least fixed-point semantics of Horn-logic programming [L87], the programs

```
p(1).
```

and

```
p(1).
p(2) :- p(2).
```

are equivalent in that their least models are identical (the set $\{\mathrm{p}(1)\}$). That is, the presence of non-terminating or failing paths does not prevent one from accepting the results of terminating (successful) paths. To model Horn-logic programs via set abstraction, we note that demonic powerdomains are inapplicable because, for example, $\{1,\perp\}\equiv\{\perp\}$ in this theory; thus, the semantics of the second program above would be ϕ by this theory. Egli-Milner powerdomains make unnecessary distinctions between sets; for example, $\{1,\perp\}\not\equiv\{1\}$, and hence the above two programs would not be semantically equivalent (the Egli-Milner powerdomain might be appropriate if the language had a "set does not contain" predicate, analogous to Prolog's negation by failure). Angelic powerdomains provide the desired semantics for Horn-logic programs; in this theory, for example, $\{1,\perp\}\equiv\{1\}$. Furthermore, angelic power-domains can be constructed for base domains containing higher-order functions and infinite objects [S86], and provide all needed primitives. The details of powerdomain construction are summarized below. For more information, see [S86], [B85], [A82] and [A83].

Building powerdomains from non-flat base domains creates difficult continuity requirements. 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 $\mathtt{A}, \mathtt{B} \subseteq \mathtt{D}$, we say that $\mathtt{A} \sqsubseteq_{\sim} \mathtt{B}$ iff for every $\mathtt{a} \in \mathtt{A}$ and Scott-open set $\mathtt{U} \subseteq \mathtt{D}$, if $\mathtt{a} \in \mathtt{U}$ then there exists a $\mathtt{b} \in \mathtt{B}$ such that $\mathtt{b} \in \mathtt{U}$ also.

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

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

$\_\cup\_$: $\mathcal{P}(\mathtt{D}) \times \mathcal{P}(\mathtt{D}) \mapsto \mathcal{P}(\mathtt{D})$ maps $[\mathtt{A}] \cup [\mathtt{B}]$ to $[\mathtt{A} \cup \mathtt{B}]$.

$^+$: $(\mathtt{D} \mapsto \mathcal{P}(\mathtt{D})) \mapsto (\mathcal{P}(\mathtt{D}) \mapsto \mathcal{P}(\mathtt{D}))$ is $\lambda \mathtt{f}.\lambda[\mathtt{A}].[\cup\{\mathtt{f}(\mathtt{a}) : \mathtt{a} \in \mathtt{A}\}]$.

An example will provide intuition about the use of '$^+$'. Suppose we have a set '$S = \{1, 2, 3\}$', and we wish to create a new set, each element of which is of the form '$f(x)$' where 'x' is in 'S'. Then

'$(\lambda x.\ \{f(x)\})^+(\{1,2,3\}) = \{f(1), f(2), f(3)\}$'.

4.2 Semantic Equations

PowerFuL's domain is the solution to:

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

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.

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

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

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

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

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

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

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

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

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

$\mathcal{E}[\![\mathtt{atomeq?}(\mathit{expr1},\ \mathit{expr2})]\!]\ \rho\ =\ \mathbf{atomeq?}(\mathbf{atom!}(\mathcal{E}[\![\mathit{expr1}]\!]\ \rho), \mathbf{atom!}(\mathcal{E}[\![\mathit{expr2}]\!]\ \rho))$

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

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

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

We can negate a condition.

$\mathcal{E}[\![\mathtt{not}(\mathit{expr})]\!]\ \rho\ =\ \mathbf{not}(\mathbf{bool!}(\mathcal{E}[\![\mathit{expr}]\!]\ \rho))$

We can create conditional expressions:

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

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

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

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

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

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

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 functions through lambda abstraction, and apply functions to their arguments.

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

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.

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

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:

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

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

We can combine the smaller sets to form larger sets:

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

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

$\mathcal{E}([\![\{expr\ :\ id \in genrtr, qualifierlist\}]\!]\ \rho)$

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

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:

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

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

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

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

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

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

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

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. Note the use of the primitive '$^+$' (for distributing elements of a powerdomain to a function) in defining the meaning of the set abstraction construct.

However, a few words must be said about some other novel primitives, here called *coercions*. Coercions are related to the *type-checking* primitives. 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 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 primitives are only defined over portions of the domain D. The boolean operators are only defined over $\mathtt{B}_\perp$; the operations '**left**' and '**right**' assume the arguments to be ordered pairs; function application (β-reduction) is defined only when the left argument is in fact a lambda expression; and only sets can contribute to a set union. Since PowerFuL is an untyped language, we will need a way to coerce inappropriate arguments to an appropriate object. For this purpose, we define five coercion primitives: '**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.

The coercions ensure that primitives handles inappropriate input reasonably. For instance, the union constructor is appropriately applied only to sets. If the argument is something other than as set (perhaps $\perp_D$), then this input is treated as the empty set. This make sense because

A) only sets contain elements — other objects do not;

B) a set is completely defined by the elements it contains; and

C) the empty set is the only set not containing any elements.

Thus, the expression '**U('a, expr)**' denotes a set containing '**'a**', regardless of whether or not '**expr**' can be computed. This is analogous the the set of solutions to a Horn logic program and goal, the elements of which are determined by successful derivations (or refutations), ignoring derivations which fail or diverge. For uniformity, we define analogous coercions to handle similar questions about primitives of other types.

One could avoid the need for such a coercion by replacing each occurrence of '**set**!(*expr*)' with '(**if set**?(*expr*) **then** *expr* **else** ϕ)'. Of course, if '*expr*' diverges, then this expression is also $\perp$. This would equate ϕ (i.e. $\perp_{P(D)}$) with $\perp_D$, coalescing the powerdomain subdomain with the other subdomains. Non-coalesced domains seem to be simpler, so we prefer to distinguish between bottoms of various subdomains. For instance, **set?**(ϕ) = **TRUE**, whereas **set?**$(\bot_D) = \bot_B$ (and with full coalescing, **set?**$(\bot_D)$ would equal $\bot_D$).

Theorem: These coercions are continuous.

Proof: We will prove the continuity of '**set!**'. Consider a sequence of objects from domain 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 $\mathcal{P}(\mathrm{D})$, then for all i, **set!**$(i) = \bot_{\mathcal{P}(D)} = \phi$. Thus,

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

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

The sets denoted by '`bools`', '`atoms`' and '`terms`' are semantically superfluous. The user could create these sets with the other constructs. For instance, each reference to the primitive set '`terms`' could be replaced by:

```
letrec
   bools be U( {TRUE}, {FALSE})
   atoms be U({A_1},..., {A_n})
   terms be U(atoms, bools, {cons(X,Y) : X,Y ∈ terms })
in
   terms).
```

PowerFuL provides these sets as primitives, so the interpreter can recognize them and treat their enumerated variables as logical variables, for greater efficiency. This will be discussed in greater detail later.

5. FROM DENOTATIONAL TO OPERATIONAL SEMANTICS

A programming language's semantics maps the syntax to the semantic domain. When this mapping is described procedurally, then we call it an *operational semantics.* When implemented, this procedure is called an ***interpreter***. We use the word '*metalanguage*' to refer to the language of the interpreter's implementation. The traditional method of defining a language is to give both declarative and 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 language of denotational semantics is itself a kind of declarative *psuedocode.* The operational semantics is usually written in a language closer to the architecture of the intended physical machine, to control execution efficiency. Before using such an operational semantics however, it is nice to know the extent to which the implemented interpreter is equivalent to the mapping declaratively described. Constructing a proof of equivalence can be quite tedious (though less difficult than comparing two different procedural definitions [S77]), and therefore we seek a different approach.

If one extends the mathematical language of the declarative semantics, so it is not a pseudocode, but a programming language in its own right, then the declarative semantics can serve as *both* a definition *and* an implementation. Assuming this declarative metalanguage can be implemented correctly, both views of the denotational semantics (operational and declarative) are equivalent.

To illustrate our approach, we wish to evaluate the expression:

```
car( cons( cons('a,'b), 'a))
```

given the denotational equations for translating syntactic symbols of atoms to real atoms in the semantic domain (differentiated here by the type font):

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

semantic equations for 'cons', 'car' and 'cdr':

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

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

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

and rewrite rules to implement the semantic primitves '**left**' and '**right**':

$\mathbf{left}(<\mathit{1st},\ \mathit{2nd}>) = \mathit{1st}$
$\mathbf{right}(<\mathit{1st},\ \mathit{2nd}>) = \mathit{2nd}$

The denotational equations map syntactic constructs to semantic constructs, and the semantic primitives map semantic objects onto other semantic objects. In this case, both kinds of mappings are defined through rewrite rules. We wish to find the semantic objected denoted by the syntactic expression above. That is, we wish to simplify:

$\mathcal{E}[\![\mathtt{car(cons(cons('a,'b),'a))}]\!]$.

Using the semantic equation for '**car**' expressions as a rewrite rule produces:

$\mathbf{left}\mathcal{E}[\![\mathtt{cons(cons('a,'b),'a)}]\!]$.

We do not yet have enough information to apply the rewrite rule for '**left**', so we must translate more syntax using the semantic equation for '**cons**':

$\mathbf{left}\ < \mathcal{E}[\![\mathtt{cons('a,'b)}]\!],\ \mathcal{E}[\![\mathtt{'a}]\!] >$.

We now have enough information to execute the semantic primitive:

$\mathcal{E}[\![\mathtt{cons('a,'b)}]\!]$.

Further rewriting with the semantic equations produces the final value:

$< \mathbf{'a},\ \mathbf{'b} >$.

Thus, we see that one can sometimes execute a program directly from the denotational semantic equations. In the remainder of this section, we develop this technique. Many of the ideas are based on Vuillemin's pioneering work on correct implementation of recursive programs [V74].

5.1 Least Fixpoints and Safe Computation Rules

Consider a recursive definition of the form:

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

for function 'F', where $\tau[F](\overline{x})$ is a functional over `([D1, ..., Dn)` $\mapsto$ `D]`, expressed by composing a term from:

a) the individual variables $\overline{x} = < x_1,\ x_2,\ \ldots,\ x_n >$;

b) known monotonic functions, called *primitives*; and

c) and the function variable, F.

Theorem (Kleene): There exists for τ a *least fixpoint*, and this fixpoint equals

$$\lim_{i \to \infty} \tau^i(\Omega).$$

If recursive program P consists of such a definition, i.e.:

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

it is generally agreed that the function defined by recursive program P is 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:

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

τ is the functional

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

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

Consider applying the function being defined to some input $\overline{d}$. Let us define a sequence of terms s_0, s_1, s_2, ..., such that the first term s_0 is $F(\overline{d})$, and each term s_{i+1} is computed from s_i by replacing each instance of F in s_i by $\tau(F)$. That is, we expand each occurence of F in the previous term by the recursive definition. Now, let us define a parallel series of terms u_0, u_1, u_2, ..., such that each u_i is computed from the corresponing t_i by replacing each remaining occurence of F by Ω. Clearly, u_i is equal to $\tau^i(\Omega)(\overline{d})$. By continuity,

$$\lim_{i\to\infty} u_i = \lim_{i\to\infty} \tau^i(\Omega)(\overline{d}) = (\lim_{i\to\infty} \tau^i(\Omega))\overline{d} = f_p(\overline{d})$$

Let us define a new series t_i similar to s_i, where $t_0 = s_0 = F(\overline{d})$, but where each t_{i+1} is computed from t_i by expanding only *some* of the occurrences of F in t_i, instead of all.

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

For each t_i, we compute v_i in the same way we computed u_i from s_i.

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

$$\lim_{i\to\infty} v_i \sqsubseteq f_p(\overline{d}).$$

Proof: For any i, $v_i \sqsubseteq u_i$, and therefore

$$\lim_{i\to\infty} v_i \sqsubseteq \lim_{i\to\infty} u_i = f_p(\overline{d}).$$

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

$$\lim_{i\to\infty} v_i \equiv f_p(\overline{d}).$$

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 $F^1, \ldots, F^i$ 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 producing the series v_i is a safe, then

$$f_p(\overline{d}) \sqsubseteq \lim_{i\to\infty} v_i$$

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

$$f_p(\overline{d}) \equiv \lim_{i\to\infty} v_i$$

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.

This leads to a method of applying a function f_P to some object $\overline{d}$, given a recursive definition of f_P. To approximate $f_P(\overline{d})$ to any arbitrary closeness, we simply produce

u_i for some sufficiently large value of i, and then simplify u_i by executing the primitive operations. This assumes that we know how to compute the primitive functions. Unless the execution of primitive functions terminate, we may be faced with the prospect of simultaneously approximating the primitives even as we approximate f_P in terms of them. Vuillemin assumes that every primitive is guarranteed to terminate for every value in its domain. This is stricter than necessary. It is sufficient that every primitive terminate on every value to which it may be applied within a given term v_i, taking into account the range of possible values for $\overline{d}$ and the computation rule chosen. This relaxation will prove important later.

When producing a series of approximations to $f_P(\overline{d})$ in this way, one tends to repeat the same executions of primitives (going from t_i to v_i) over and over. If the primitive operations are implemented via rewrite rules, one can reduce the overcomputation by applying these rewrite rules directly on the terms t_i. Such an rewrite would then carry through automatically in all further approximations, whereas, if one waits to apply it upon v_i, then the it must be repeated again when computing v_{i+1}, etc. Therefore, an extra step is inserted in the computation procedure. To compute t_{i+1} from t_i, we first perform a substitution step, expanding occurrences of F, and then we perform a simplification step, applying rewrite rules from the primitive definitions until no more can be applied. Computing v_i from t_i is as before.

Applying the primitives as their arguments are computed not only is more efficient, it is necessary for termination of the calculation when the result is fully computed. Simplification of primitives (such as simplifying an occurrence of the if/else primitive when the condition has been calculated) may prune branches of the term. If *all* occurrences of F have been pruned, then no further substitution steps will be necessary. Again, we must be wary that for each computation step i, only a finite number of primitive rewrites will be required. Otherwise, we will never get to compute the next approximation.

5.2 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:

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

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.

5.3 Implementing Denotational Semantics

Because we adopted a certain discipline in writing the denotational semantics, listed in Section 4, we can execute the resulting equations as a recursive program. Instead of a single recursive function, we have three mutually-recursive functions, '$\mathcal{E}$', which maps a syntactic expression and an environment to a semantic object, '$\mathcal{F}$' which maps a syntactic expression to a semantic object (without need of the environment), and '$\mathcal{D}$', which maps a syntactic expression and an environment to a new environment. Semantic equations, interpreted as left-to-right rewrite rules, provide both a definition and an execution mechanism.

Note that some semantic equations introduce lambda variables. These may have to be renamed at times to avoid variable capture. However, this is standard practice in executing languages based on lambda calculus. Because functions are written as lambda expressions, β-reduction is treated as a semantic primitive of two arguments, strict in the first. Actually, defining β-reduction as a semantic primitive is dangerous, as there exist lambda-expressions whose simplification will fail to 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.

A fixpoint operator would never terminate, and therefore we do not treat it as a primitive, but rather implement it within the denotational equations, themselves. Creating a denotational description suitable for direct interpretation requires this kind of special care.

To complete the interpreter, we must provide rewrite rules to define the primitive functions, Below is a summary of the PowerFuL primitives.

5.3.1 PowerFuL Semantic Primitives

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.

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:

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

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**: $B_\perp \mapsto B_\perp$, is strict in its only argument. Its simplification rules are:

$$\textbf{not}(\textbf{TRUE}) = \textbf{FALSE}$$
$$\textbf{not}(\textbf{FALSE}) = \textbf{TRUE}$$
$$\textbf{not}(\perp_B) = \perp_B\ .$$

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 $\bot_A$, make up the subdomain $\mathtt{A}_\bot$. For every atom $\mathbf{A}_i$, there is a primitive function $\mathbf{isA}_i\mathbf{?}$: $\mathtt{A}_\bot \mapsto \mathtt{B}_\bot$, strict in its only argument. The simplification rules are:

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

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

$$\begin{aligned}
&\mathbf{atomeq?}(\bot_A,\ \mathit{arg2}) = \bot_B \\
&\mathbf{atomeq?}(\mathit{arg1},\ \bot_A) = \bot_B \\
&\mathbf{atomeq?}(\mathbf{A}_i,\ \mathit{arg2}) = \mathbf{isA}_i\mathtt{?}(\mathit{arg2}) \\
&\mathbf{atomeq?}(\mathit{arg1},\ \mathbf{A}_i) = \mathbf{isA}_i\mathtt{?}(\mathit{arg1}).
\end{aligned}$$

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

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:

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

Powerdomain Input Primitives

The primitive '+' lets us iterate a function of type $\mathtt{D} \mapsto \mathcal{P}(\mathtt{D})$ 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:

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

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

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

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.

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.

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:

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

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

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

and similarly

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

If A is an atom, then

$$\textbf{equal?}(A,\ \mathit{expr}) =$$
$$\textbf{if atom?}(\mathit{exp})\ \textbf{then atomeq?}(A, \textbf{atom!}(\mathit{exp}))\ \textbf{else FALSE fi}$$

and similarly

$$\textbf{equal?}(\mathit{expr},\ A) =$$
$$\textbf{if atom?}(\mathit{exp})\ \textbf{then atomeq?}(\textbf{atom!}(\mathit{expr}), A)\ \textbf{else FALSE fi}$$

If F is a function, then

$$\textbf{equal?}(F,\ \mathit{expr}) = \textbf{if func?}(\mathit{exp})\ \textbf{then}\ \bot_B\ \textbf{else FALSE fi}$$
$$\textbf{equal?}(\mathit{expr},\ F) = \textbf{if func?}(\mathit{exp})\ \textbf{then}\ \bot_B\ \textbf{else FALSE fi}$$

If S is a set, then

$$\textbf{equal?}(S,\ \mathit{expr}) = \textbf{if set?}(\mathit{exp})\ \textbf{then}\ \bot_B\ \textbf{else FALSE fi}$$
$$\textbf{equal?}(\mathit{expr},\ S) = \textbf{if set?}(\mathit{exp})\ \textbf{then}\ \bot_B\ \textbf{else FALSE fi}$$

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) are guaranteed to terminate on any finite input.

Proposition: 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.

5.3.2 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:

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

Expanding the remaining function call yields:

'joe.

5.4 More on Computation Rules

5.4.1 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.

5.4.2 Better Computation Rules

Vuillemin describes some computation rules, each based on a uniform 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 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 safe [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 safe [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

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

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:

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

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

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

This immediately simplifies to:

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

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:

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

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:

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

becomes:

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

which becomes:

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

This, in turn, becomes:

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

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

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

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. The key idea is to only evaluate the body of a function (lambda expression) after the function has been applied (after β-reduction has been performed). Our computation rule does this, so long as the top-level value being computed does not contain an unapplied function as a part.

Theorem: Our operational semantics is sound and complete for any program denoting (at the top level) a value from the domain '`E`':

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

Other elements of domain `D`, such as functions, lists of functions, sets of functions et cetera, can be freely used as *intermediate* values.

Proof: 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. We have also shown that, if the program does not require computing a function for its own sake, but only in the context of an application, then this assumption is valid. The subdomain `E` describes just those objects of `D` not containing (unapplied) functions as parts. Computing these objects does not require evaluation of any lambda expression, except within the context of its application.

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.

6. OPTIMIZATIONS

This section improves the basic operational procedure described in the last chapter, concentrating on the efficiency of set abstraction. More general techniques for improving the efficiency of (pure) functional languages are available [P87], which we will not discuss here.

6.1 Intuition Behind Optimizations

In Section 3, we specified a Horn logic program using '`letrec`' (the feature for creating recursive definitions), set abstraction, the conditional and the equality primitive. Executing this program using PowerFuL's denotational equations as the interpreter would be analogous to using Herbrand's method to solve problems in logic. Herbrand's "generate-and-test" approach is simple but inefficient.

The resolution method avoids blind generation of instantiations, preferring to do as much work as possible on non-ground expressions. In logic programming, a logical variable denotes an element from the set of terms (the Herbrand Universe). In resolution, a logical variable becomes instantiated only to the extent necessary to satisfy the inference rule's equality test. Partial instantiation narrows the set of candidate bindings, without necessarily settling on a single choice. When performed to ensure equality of non-ground terms, partial instantiation is called *unification*, and the substitution implementing the partial instantiation is called a *unifier*.

In PowerFuL, to compute a relative set expression, we normally begin by computing the generator set. Whenever we isolate an expression denoting an element of the generator, a copy of the relative set abstraction is created, with the generator element expression replacing the enumeration parameter. All such instantiations are computed independently. We would like to modify this procedure so that when '`terms`' (analogous to the Herbrand Universe) is the generator set, rather than enumerate its many simple objects we instead treat the enumeration parameter as a logical variable. An enumeration parameter generated by '*atoms*', or '*bools*' can be viewed as a partially-instantiated, or constrained, logical variable.

In Horn logic, each correct answer substitution provides ground bindings for the goal's logical variables. Members of this set can be grouped into families. Within a family, all answer substitutions share common aspects, with the remaining details varying freely. The Herbrand derivation of one member of the family is almost identical to the Herbrand derivation for any other member. For each family of correct answer substitutions, Herbrand's method would derive each member individually, with an infinity of essentially similar derivations. Resolution, however, produces *general computed* answer substitutions, one per family. A general computed answer subsititution only partially instantiates a goal's logical variables, and does so in such a way that for *any* ground completion of the general computed answer substitution would result in a correct answer substitution. The derivation of the general answer subsitution resembles a parameterized Herbrand derivation.

Resolution performs modus ponens inferences on the non-ground clauses directly, rather than first instantiating them. Logical variables become partially instantiated (via a most general unifier) only to the extent necessary to satisfy the inference rule's equality requirement. In a sense, program execution is left unfinished, Though it is easy to extend a most general answer substitution, to produce (ground) correct answer subsititutions, this is not done. Reporting results in the general form is more economical than individually reporting each of the infinite ways in which each most general answer can be extended.

6.2 Optimization Technique

Logical Variable Abstraction

To treat an enumeration parameter as a logical variable, we must recognize that it's generator is the set of first-order terms (or part of this set). An expression of the form:

$(\lambda x.\textit{body})^+\ \mathcal{F}[[\texttt{terms}]]$

is rewritten $\textit{term}(x).\textit{body}$ to indicate that 'x' is to be treated as a logical variable, rather than blindly enumerating its generator. The expressions $\textit{atom}(x).\textit{body}$ and $\textit{bool}(x).\textit{body}$ are constructed analogously. Rather than recomputing the '*body*' for each trivial instantiation, we will evaluate '*body*' in its uninstantiated form, leaving it parameterized by the enumeration variable, computing a parameterized set expression. This parameterized set expression stands for the union of all possible instantiations. We also hope to express results in this compact notation.

We can use parameterized set expressions as generators for other set expressions. Note that:

$(\lambda x.\ \textit{body}_1)^+(\textit{term}(y).\ \textit{body}_2),$

can be rewritten as:

$\textit{term}(y).\ ((\lambda x.\ \textit{body}_1)^+(\textit{body}_2)).$

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

$(\lambda x.\ \textit{body}_1)^+((\lambda y.\ \textit{body}_2)^+(\mathcal{F}[[\texttt{terms}]])),$

and the second is an alternate notation for

$((\lambda x.\ \textit{body}_1)^+(\lambda y.\ \textit{body}_2))^+(\mathcal{F}[[\texttt{terms}]]),$

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

When the body is in the form of a singleton set, we compute the expression within the singleton-constructing brackets. When the body of a general set expression is in the form of a union, we simplify it to be the union of two generalized set expressions. That is,

$\textit{term}(x).(\textit{exp}_1 \cup \textit{exp}_2)$

is simplified to:

$term(x).exp_1 \cup term(x).exp_2$.

Of course, if the body is 'ϕ', then the general set expression denotes a union of empty sets, and so the whole thing can be replaced by a simple empty set. These ideas hold as well for constrained general set expressions.

We can compute a parameterized body, because expansions of recursive function calls (translation from syntax to semantics) will not not depend upon these parameters. There *is* one complication, however: the simplification of primitives. 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. Applicability of a primitive's rewrite rule will often depend upon what kind of term the logical variable stands for. Therefore, we must be able to perform simplifications when primitives are applied to parameters. In each case, one of the following techniques will suffice:

Technique 1: Simple Reduction

Often, implied or stated constraints on the logical variable provide sufficient information, already. In such cases, parameterization does not hinder simplification of the primitive. For instance, given:

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

we can simplify '$\mathbf{func?}(u)$' to '**FALSE**', without knowing the value of 'u', since any value would certainly not be a function. Below is a comprehensive list of similar situations:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

PowerFuL semantic primitives always simplify, given the outermost constructor of any strict argument. If an argument replaces a *nonstrict* parameter, however, its value is irrelevant (at least until something about a strict argument is known). 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.

Technique 2: Splitting by Type

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

Simple reduction suffices when one of these primitives is applied to a logical variable constrained to be an atom, or a boolean. When the variable is enumerated from '`terms`', the simplification chosen depends upon the kind of term. Luckily, we need not consider each ground term individually. The set of terms consists of three subsets, the set of booleans, the set of atoms, and the set of ordered pairs of subterms. For each subset, simple reduction suffices. 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 the first branch of the union, we have partially instantiated the logical variable by constraining it to represent a boolean. In the second branch, we have constrained it to represent an atom. In the third branch, we have constrained it to represent a term which is an ordered pair of subterms. The primitive function simplifies immediately in each subset. We must now compute each branch of the union separately.

This is analogous to the use of most general unifiers in Horn logic resolution. Unification prepares two clauses for modus ponens by instantiating them no more than is necessary to satisfy the equality requirement. One difference is that traditional Horn logic does not use negative information. Horn logic only considers instantiations to make the equality true. In PowerFuL, we are concerned with *all* possible outcomes. Some variations of Horn logic do consider negative information through the use of a dis-equality predicate and negative unifiers [N85] [K84]. We discuss primitives based on equality next.

Technique 3: Splitting on Equality

Equality is strict in each argument, simplifying, as soon as the type of either argument is known, to an '**if**' expression which must know (before all else) the type (boolean, atom, pair, set or function) of the remaining argument. If one argument is a lambda variable enumerated from the set of terms, we first look at the other argument, to avoid splitting the logical variable into three subcases. This 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 equality anyway, no matter what value the logical variable represented. However, if both arguments of the equality predicate are logical variables, we cannot delay them both. We may have an expression of the form:

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

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 it is better to deal with a few large subsets, than an infinity of individual cases. Splitting them into atoms, booleans and ordered pairs does not help. We must recognize that the instantiations fall into two cases: those for which the twe terms are equal, and those for which they are not. 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 '$term(v).\boldsymbol{body_2}$', we are taking the union of instantiations by 'v' in which all possible instantiations of '$\boldsymbol{body_2}$' are identical (since the body no longer depends on 'v'). Clearly, '$term(v).\boldsymbol{body_2}$' can now be replaced by '$\boldsymbol{body_2}$'. (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

$term(u).(\ldots term(v).$**if not(equal?**$(v,\ u)$**)** **then** $(\ldots$**FALSE**$\ldots)$ **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 a dis-equality predicate, defined on terms. His dis-equality 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 dis-equality. 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 dis-equality part of the solution, as a kind of negative unifier. Khabaza describes a way in which this can be done [K84]. In essence, the dis-equality 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 dis-equality 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 dis-equalities 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:

$term(u).(\ldots term(v)u \neq v.(\ldots$**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 dis-equalities result in a constraint which is a conjunction of dis-equalities. 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 dis-equality '$u \neq v$' 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 dis-equalities. As these simple constraints are satisfied, they can be replaced by '**TRUE**'. Those dis-equalities 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 dis-equality of two term variables:

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

can be replaced by

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

For the dis-equality of two atom variables, we have:

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

replaced by

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

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?**'.

Note that we consider the binding of logical variables separate from the definition of the equality primitive itself. Robinson also split unification into these components [R82]. We have generalized the approach to also consider negative unification.

Technique 4: Splitting by **TRUE** *and* **FALSE**

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:

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

6.3 Results

These optimizations avoid blind enumeration of the sets '`terms`', '`atoms`' and '`bools`' when used as relative set abstraction generators. Instead, we treat the enumeration parameter as a logical variable, sometimes constrained. An enumeration variable from the set '`atoms`' is treated as a logical variable carrying the constraint that it can be bound only to an atom. Enumeration variables from the set '`bools`' are handled analogously. Disequality constraints relating two logical variables are also used. With logical variables, one evaluates the generating set (the second argument of '$^+$') only as needed to compute the body (the first argument of '$^+$'). Computing with logical variables and constraints gives the set abstraction facility the efficiency of resolution. The logical variable is merely an *operational* (not semantic) concept, improving the execution efficiency when using these special sets. The default procedure (generate, instantiate, and continue) handles more complicated generators, such as sets of functions, sets of sets, etc.

For an example, suppose we wanted to compute term bindings for '`A`', '`B`' and '`C`' so that lists `[A,B]` and `[B, ['a,C] ]` would be equal. That is, we wish to compute a unifier. The program might be:

`{[A,B,C] : A,B,C ∈ terms, [A,B] = [B, ['a,C] ]}`.

Without the optimizations, the interpreter would produce a tree-like structure, whose internal nodes are the set-union operator, and with one leaf for each possible combination of bindings for terms '`A`', '`B`' and '`C`'. Where the bindings created a unifier, the leaf would be a singleton set containing the list of bindings. Where the the bindings did not form a unifier, the leaf position would be the empty set. The object would indeed by the set of unifiers.

With the optimizations, only a finite union tree would be produced, with a few leaves containing the empty set, and one leaf containing:

$term(u).(\{[['\mathbf{a}, u], ['\mathbf{a}, u], u]\})$

In a sense, with the optimizations, the program produces only the most general unifier. It is conceivable execution of a Horn logic program would mimic the operations of breath-first SLD resolution.

Treating an enumeration parameter as a logical variable is practical because the generating set '`terms`' is so simple in structure. Wherever a logical variable 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. Generators are not limited to these special sets, however.

Three techniques narrow the range of the logical variable. The choice depends upon the primitive being applied, and the constraints already in force. Some primitive simplifications do not require splitting. Some require a two-way split, and others a three-way split. When performing primitive simplifications, it is efficient 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.

We have shown that, for the sake of efficiency, it is sometimes possible to compute (constrained) non-ground set expressions when '`terms`' is a set abstraction generator. Showing that this is always possible requires a systematic look at all the primitives which might operate upon logical variables, to make sure all possibilities are covered. We have already considered these primitives applied to logical variables: '**bool?**', '**atom?**', '**pair?**', '**func?**', '**set?**', '**bool!**', '**atom!**', '**pair!**', '**func!**', '**set!**', '**if**', '**not**', '**isA$_i$?**' (for each atom '**A$_i$**' and '**equal?**'.

The remaining primitives are: '**left**', '**right**', 'β-reduction' and '$^+$'. According to the denotational equations, '**left**' and '**right**' are always applied in conjuction with the coercion '**pair!**'. Since we have already considered logical variables as arguments to '**pair!**', we need no special mechanism for '**left**' and '**right**'. Similarly, the denotational equation for function application applies the coercion '**func!**' to the function position of the application. Since we have already considered logical variables as arguments to '**func!**', we need no special mechanism for β-reduction, either. Analogously, the coercion '**set!**' intercedes between '$+$' and its strict (the second) argument.

Since any set can be used as a generator for defining a new set, we must be able to compute '$+$' when its argument is a non-ground set expression. Let '*head.body*' represent a non-ground set expression, where '*head*' either introduces a logical variable, or perhaps expresses a constraint on existing logical variables (introduced in a more global context). The expression:

$$\mathit{function}^{+}(\mathit{head.body})$$

can be rewritten

$$\mathit{head}.(\mathit{function}^{+}\,\mathit{body}).$$

Eventually, all the logical variable introductions and constraints are peeled off the body, so that the ordinary simplification rules for '$+$' can be applied. The associativity of union assures that these forms are equivalent. Of course, this transformation requires that each logical variable receives a unique name, as in Prolog. Alternatively, where logical variables are differentiated by scope, one would use renaming techniques from lambda-calculus to avoid variable capture.

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.

7. 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. We have achieved all these objectives through a functional language incorporating *set abstraction.* Representative sample programs attest to the power and generality of the language.

A set abstraction construct for both functional and logic programming has long been advocated, but its declarative and operational semantics has not hitherto been fully determined. We showed that first-order *absolute* set abstraction is easily subsumed by *relative* set abstraction, which has a much simpler higher-order extension. Our approach supports higher-order constructs (functions and sets) as first-class objects.

We presented a short denotational description which maps the syntax onto computable (continuous) semantic primitives. 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 set abstraction – an explicit data type in our language.

We derived an operational semantics consistent with the denotational description. To do this, we extended Vuillemin's theory of correct implementation of recursion, and applied the resulting technique to the recursive denotational equations themselves. This methodology requires that denotational equations handle most recursion explicitly (so that the semantic primitives do not provide additional sources of non-termination). We developed a computation rule more efficient than the parallel outermost rule, but correct for this language, as established by a proof of its safety.

Of special interest in logic programming is the set of terms, objects for which identity is synonymous with equality. We showed that, when the set of terms is used as a generating set in a relative set abstraction, the enumeration parameter can be computed as a logical variable, providing the efficiency of absolute set abstraction. In the general case, however, the enumeration parameters are instantiated by the various generator set elements as these elements are computed. Thus, generators need *not* be arbitrarily restricted to contain only first-order types.

Expensive operational mechanisms (e.g. higher-order unification, general theorem-proving and unrestricted narrowing) are often associated with functional/logic programming combinations. Ordinary functional languages avoid these difficulties, propagating higher-order objects via one-way substitution, and defining equality only over first-order objects. By retaining these characteristics, our language avoids such 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.

[F84] L. Fribourg, "Oriented Equational Clauses as a Programming Language." *J. Logic Prog.* **2** (1984) pp. 165-177.

[F84b] L. Fribourg, "A Narrowing Procedure for Theories with Constructors." In *Proceedings of the 7th International Conference on Automated Deduction, LNCS 170* (1984) pp. 259-301.

[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.

[JLM84] J. Jaffar, J.-L. Lassez, M. J. Maher, "A Theory of Complete Logic Programs with Equality," In *J. Logic Prog.*, Vol. 1, pp. 211-223, 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.

[K79] R. A. Kowalski, "Algorithm = Logic + Control," In *Communications of the ACM*, July 1979, pp. 424–435.

[K83] W. A. Kornfeld, "Equality for PROLOG," In *Proceedings of the 8th IJCAI*, 1983, pp. 514–519.

[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.

[R82] J. A. Robinson, E. Sibert, "LOGLISP: An Alternative to PROLOG," Machine Intelligence 10, 1982, pp. 299-314.

[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.