J. Functional Programming 2 (3): 273–321, July 1992 © 1992 Cambridge University Press

A domain-theoretic approach to functional and logic programming

FRANK S. K. SILBERMANN
Department of Computer Science, School of Engineering, 301 Stanley Thomas Hall, Tulane University, New Orleans, LA 70118, USA
(fs@cs.tulane.edu)

BHARAT JAYARAMAN
Department of Computer Science, State University of New York at Buffalo, Buffalo, NY 14260, USA
(bharat@cs.buffalo.edu)

Abstract

The integration of functional and logic programming languages has been a topic of great interest in the last decade. Many proposals have been made, yet none is completely satisfactory especially in the context of higher order functions and lazy evaluation. This paper addresses these shortcomings via a new approach: *domain theory* as a common basis for functional and logic programming. Our integrated language remains essentially within the functional paradigm. The logic programming capability is provided by *set abstraction* (via Zermelo–Frankel set notation), using the Herbrand universe as a set abstraction generator, but for efficiency reasons our proposed evaluation procedure treats this generator's enumeration parameter as a logical variable. The language is defined in terms of (computable) domain-theoretic constructions and primitives, using the lower (or angelic) powerdomain to model the set abstraction facility. The result is a simple, elegant and purely declarative language that successfully combines the most important features of both pure functional programming and pure Horn logic programming. Referential transparency with respect to the underlying mathematical model is maintained throughout. An implicitly correct operational semantics is obtained by direct execution of the denotational semantic definition, modified suitably to permit logical variables whenever the Herbrand universe is being generated within a set abstraction. Completeness of the operational semantics requires a form of parallel evaluation, rather than the more familiar left-most rule.

Capsule review

This paper's aim is to allow the expressiveness of both logic and lazy functional programming styles to coexist within a single language. Cleanliness and correctness are the guiding principles, and this contrasts refreshingly with much of the previous work in this area.

Rather than attempting to add logical variables directly, sets are used to provide logical expressiveness. The authors demonstrate the relationship between such sets and the more usual predicate approach (found in Prolog, for example). The semantics used for sets does not

† This research was supported by grants DCR-8603609 and CCR-9004357 from the National Science Foundation, contract 00014-86-K-0680 from the Office of Naval Research, the Tulane Committee on Research and contract LEQSF RD-A-19 of the Louisiana Board of Regents.

interfere with the standard lazy functional semantics, and so their inclusion is cleanly achieved. Perhaps the most interesting and original view developed in the paper is that logical variables ought not to be considered a feature of the programming language, but rather that they arise as an implementation optimisation. The viability of this view is demonstrated by translating the denotational semantics into an operational semantics.

Consistent with their intention of providing a clean approach is the recognition that some degree of parallel evaluation is required to achieve completeness. The authors have also quite correctly resisted the temptation to provide an abundant repertoire of set operations, and have instead provided only those required for the purpose in hand. This parsimony means that referential transparency may be retained.

---

1 Integration of functional and logic programming

On the surface, functional programming and logic programming seem to be two quite distinct ways to describe computation. In functional programming, programs are function definitions, and computations are expressed as the definition and application of functions, whereas in logic programming, computations are expressed as the search for values satisfying (logical) constraints. We should clarify at the outset that, by logic programming, we mean programming with first-order relational Horn clauses (van Emden and Kowalski, 1976) (or Horn logic for short), the declarative basis for Prolog. A Horn logic program presents a set of relational definite clauses over a domain of first-order terms, along with a goal clause. Some problems can more naturally be described in terms of functions, others in terms of predicates. However, both capabilities are desirable, and hence the integration of these two paradigms has been a topic of great interest in the last decade (see DeGroot and Lindstrom (1986) and Bellia and Levi (1986) for two good surveys).

Functional programs contain much more control knowledge than logic programs, due to the deterministic nature of functions and the uni-directional nature of function reduction (or application). This property is the key to obtaining more efficient implementations of functional languages. (Often, deterministic computation in Prolog requires a complex and sometimes semantically unclear metalogical annotation of the program.) The functional paradigm also offers lazy evaluation and higher-order functions, whose uses for program modularisation are well-known (Hughes, 1990). On the other hand, logic programs can sometimes be more abstract than functional programs, since less control knowledge need be embedded in the program. The price for this increased abstraction is that a non-deterministic search must be conducted to obtain solutions to the logical constraints.

An integrated language must account for all of the above features. Although many proposals have been made, none is completely satisfactory especially in the context of higher order functions and lazy evaluation. This paper addresses these shortcomings via a new approach: *domain theory* (see Stoy (1977) for an introduction). Domain theory was devised by Scott to justify the popular interpretation of recursive lambda expressions as functions. Since most functional programming is based upon this interpretation, we consider domain theory of fundamental importance in the theory of functional programming. From the perspective of domain theory, a functional language is essentially a shorthand notation for a restricted and computable subset of domain-theoretic concepts. Domain theory can also be used to understand logic programs, since the fixed-point semantics of Horn logic programming is given in domain-theoretic terms (Lloyd, 1987). Thus domain theory can be used as the foundation for an integrated language.

1.1 Design philosophy

There are two roles that denotational semantics may play in the creation of a programming language: the *descriptive* role and the *prescriptive* role. The more traditional descriptive role of denotational semantics is to concisely describe a language whose design was motivated by operational efficiency and generality, rather than clarity and simplicity. The prescriptive approach (Ashcroft and Wadge, 1982) uses the domain-theoretic concepts as building blocks for the designer. If only computable domain-theoretic primitives are used, a simple operational semantics results as a by-product of the design process. The prescriptive approach aims for simplicity and elegance, perhaps at the expense of generality and efficiency. The goals of the prescriptive approach to denotational semantics and the goals of functional programming are thus in perfect agreement, so in this work, we conscientiously follow the prescriptive approach.

Just as there are two approaches to the use of denotational semantics, there are two approaches to functional programming. The purists demand that every feature in the language be consistent with the basic paradigm. This ensures that the mathematical structure of the paradigm may be used in reasoning about programs. This greatly aids writing, debugging and reasoning formally about programs. The pragmatists, on the other hand, recognise that practical programming tasks may require that the rules be broken occasionally. Language designers from both camps have the same goal – a language which is pure, elegant and practical. Nevertheless, they approach this goal from different directions. The pragmatists seek to control and tame the dangers of existing meta-logical and extra-logical features, while the purists try to design pure languages of increased power. This work is presented from the purists' perspective. In judging our proposed language's expressiveness, one should not compare it to Lisp (McCarthy *et al.*, 1965) or Prolog (Clocksin and Mellish, 1981), but rather to their pure subsets in which all metalogical and extra-logical features are absent. The combination of Lisp and Prolog has been well-studied (Robinson and Sibert, 1982; Smolka and Panangaden, 1985). Our objective is to show that purely declarative versions of these languages may be subsumed by a single purely declarative language.

1.2 Approach

Our approach is to incorporate the logic programming capability within the functional framework via *set abstraction* (Zermelo–Frankel set theory). We showed (Silbermann and Jayaraman, 1989) that *relative set abstraction* adds to a lazy, higher-order functional programming the expressiveness of first-order logic programming. Sets, as well as functions, can be treated as first-class objects. We prefer relative set abstraction to the absolute set abstraction construct (Darlington *et al.*, 1986) because the former combines better with higher-order constructs. Essentially, we require every *logical variable* in the absolute set construct to have its generator-set explicitly specified as the Herbrand universe of terms, in order to obtain the corresponding relative set construct. We showed (Silbermann and Jayaraman, 1989) that the non-flat *lower powerdomain* (sometimes called *angelic*) was the appropriate domain construction for incorporating these sets. Thus our proposed language is defined in terms of (computable) domain-theoretic constructors and primitives, using power-domains to model the set abstraction facility. The result is a simple, elegant and purely declarative language that successfully combines the most important features of both pure functional programming and pure Horn logic programming. Referential transparency with respect to the underlying mathematical model is maintained throughout. The language embodying these ideas is called *PowerFuL*, because *Power*domains were used for integrating *Fu*nctional and *L*ogic programming.

The key concepts of PowerFuL will be demonstrated via a sparse notation, admittedly without many modern syntactic niceties. In particular, we have considered neither clausal-style function definition via pattern-matching, static polymorphic type checking, nor arithmetic. Many modern functional languages *simulate* equational reasoning via a syntax resembling clausal equational function definition, implemented by left-to-right pattern matching (Hudak, 1989). Because the format of this syntactic variant distracts from the language's true semantics, we do not discuss it. Nevertheless, its implementation is well understood, and can easily be added to our language in the manner described in Peyton Jones (1987). There is considerable interest in strongly typed logic programming languages, and we see no incompatibility between functional and logic programming in this area. However, the development and use of type systems in logic programming lags behind functional languages, for which many polymorphic type systems have been studied (Hudak, 1989). Until there is a consensus on appropriate type systems for logic programming, we prefer to deal with an untyped language augmented with a modest set of run-time type-checking primitives.

Our approach to the operational semantics follows the idea of semantics-based translation. That is, denotational semantics is treated not only as a means of language description, but also as an *implementation* language (Pleban, 1984; Schmidt, 1986). This approach is based on the observation that reduction of the program syntax by the operational semantics parallels the simplification of the corresponding mathematical expressions in the semantic domain, and hence the reduction of the syntax will not be significantly more efficient than simplification of the denoted semantic expression. That is, a second semantics based on reduction rules for program syntax would be redundant. Our approach involves compiling the syntax to the domain-theoretic notation, and simplifying the resulting mathematical expression via equality axioms provided for each semantic primitive. The equality axioms for each semantic primitive are used as left-to-right reduction rules, applied using the *leftmost* computation rule (so that reduction will terminate, if possible).

For logic programming, the standard operational semantics (resolution) is much more efficient than direct execution of the fixed point semantics (Lloyd, 1987). For efficient execution of PowerFuL's logic programming component, the semantics-driven evaluation procedure is modified by the incorporation of runtime program transformations. These transformations result in certain set enumeration parameters (those enumerated by the first-order Herbrand universe) being treated in ways analogous to the treatment of logical variables in resolution derivations (Lloyd, 1987). Completeness of the operational semantics requires a degree of parallel evaluation, rather than the more familiar left-most rule.

1.3 Outline of paper

Before introducing PowerFuL, we first present in section 2 a higher-order functional language called LispLike, in domain-theoretic terms. In so doing, we clearly delineate those aspects of our language design that are commonplace in functional programming from those that are new in this proposal. Section 3 extends the language of section 2 with a set abstraction construct that meshes well with the higher-order capability. Section 4 discusses two critical issues relating to the operational semantics: The first issue is that quite reasonable and correct logic programs may be inherently nonterminating, and therefore call for a fair evaluation strategy in the underlying computation rule. The second issue is that the execution procedure, when the Herbrand universe is used as a set abstraction generator, must employ generalised derivations, using *logical variables* to represent arbitrary elements of the Herbrand universe, rather than working from ground instantiations. The principle is analogous to Robinson's *resolution method* (van Emden and Kowalski 1976; Lloyd, 1987) in Horn logic programming. Section 5 compares our accomplishments with those of related work, and suggests avenues of further research.

2 Lisp-like functional programming

We describe a purely declarative, untyped, lazy, higher-order functional programming language. Being untyped, it resembles a purely declarative subset of Lisp. Therefore, we will refer to this language as *LispLike*. Its set of first-class objects includes booleans, atoms, ordered pairs of first-class objects, and functions over the set of first-class objects.

Before beginning a formal analysis, let us demonstrate the style and look of the language with a few simple function definitions.

```
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 'nil
                 else cons(f(car(l)), map(f,cdr(l))) fi
   infinite be cons('a, infinite)
in
   ...
```

The map example shown above is in curried form. Higher-order functions and infinite objects are defined in a conventional manner.

The remainder of this section is organised as follows. Section 2.1 will review the mathematical primitives from domain theory that will be used. The formal definition of Lisplike follows in section 2.2. Section 2.3 discusses implementation via direct execution of the denotational semantics. Section 2.4 discusses use of the denotational semantics in equational reasoning.

2.1 Review of semantic primitives

We assume the reader is familiar with elementary domain theory, specifically with concepts and notations such as *bottom* ($\perp$), *approximation* ($\sqsubseteq$), *pointed complete partial order* (i.e. *domain*), *functional*, *least fixed point*, $\equiv$, *strictness*, *chain*, *least upperbound* ($\sqcup$), *monotonicity*, *continuity*, the interpretation of a recursive definition as the least fixed point of the associated functional, and Scott's inverse limit construction (Scott, 1982; Schmidt, 1986; Gunter and Scott, 1990) for recursive domain specifications. The basic notation and theory may be found in Schmidt (1986).

This section lists the primitive domains, domain constructors and operations which may be used by a simple functional programming language. Each primitive operation is described via equations which, when used as left-to-right rewrite rules, implement that operation.

To distinguish the notation of domain theory from programming language syntax, our convention will be to write programming language syntax in `teletype`, the primitive semantic functions in **boldface**; metavariables in rules, axioms and other equations will be *italicised*. Definitions and theorems in this section were taken from Schmidt (1986).

2.1.2 Elementary domain constructors

The basic kinds of elements in LispLike are booleans, atoms, ordered pairs and functions. The relevant domains, domain constructors and operations are standard in the literature, so proofs of continuity will be omitted. (The interested reader may find proofs in Schmidt (1986), and other writings on denotational semantics.) Each primitive function can be implemented via its associate axioms of equality, if interpreted as left-to-right rewrite rules. Note that for each primitive, a simplification rule applies as soon as the outermost constructor of any strict argument is available.

In our notation, B refers to the discrete domain containing **TRUE**, **FALSE** and $\perp$ (sometimes written as $\perp_{\mathrm{B}}$ to prevent confusion with bottom elements of other domains).

For any domain D, we have the primitive function **if**: $\mathrm{B} \times D \times D \to D$, implemented via:

$$\mathbf{if}(\mathbf{TRUE}, arg2, arg3) = arg2$$
$$\mathbf{if}(\mathbf{FALSE}, arg2, arg3) = arg3$$
$$\mathbf{if}(\perp, arg2, arg3) = \perp.$$

The conditional is strict only in its first argument. For clarity, when writing nested conditionals, we shall feel free to express this primitive using the alternative **if** ... **then** ... **else** ... **fi** notation.

From the syntax of each program, we determine a finite set of atoms, which denote atomic values $\mathbf{A}_1$ through $\mathbf{A}_n$. With the addition of a bottom element $\perp$ these elements make up the domain of atoms, $\mathtt{A}$.

Provided monotonicity is maintained, a function over the atoms can be defined via a set of equations, one for each element of the domain. For example, for each atom $\mathbf{A}_i$ one can define a strict primitive function $\mathbf{isA}_i\mathbf{?}$: $\mathtt{A} \to \mathtt{B}$:

$$\mathbf{isA}_i\mathbf{?}(\perp_{\mathtt{A}}) = \perp_{\mathtt{B}}$$
$$\mathbf{isA}_i\mathbf{?}(\mathbf{A}_i) = \mathbf{TRUE}$$
$$\mathbf{isA}_i\mathbf{?}(\mathbf{A}_j) = \mathbf{FALSE} \text{ for } i \neq j.$$

For instance, the primitive **is'nil?** tests whether an atom is equal to **'nil**. Note that the last equation is actually an equation schema, with one equation for each combination of $i \neq j$.

Given domains D_1 and D_2, we can construct domain $D_1 \times D_2$, whose elements are ordered pairs of elements from their respective domains. Associated with this construction are two strict primitives: **left**: $D_1 \times D_2 \to D_1$, and **right**: $D_1 \times D_2 \to D_2$. Each primitive is defined by a single simplification rule:

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

Given domains D_1 and D_2, we can create a domain $D_1 \to D_2$, the continuous function space from D_1 to D_2. Elements of $D_1 \to D_2$ are created as follows: if e is an expression containing occurrences of an identifier x, such that whenever a value $a \in D_1$ replaces the free occurrences of x in e, the value $[a/x]e \in D_2$ results, then $\lambda x.e$ is an element of $D_1 \to D_2$.

The associated primitive operation is function application, written simply as one argument beside the other: $(D_1 \to D_2) \times D_1 \to D_2$, which takes a function $f \in D_1 \to D_2$ and an element $a \in D_1$, and produces $f(a) \in D_2$. This operation associates to the left. Whereas other primitives are implemented via rewrite rules, function application is implemented via *beta-reduction*. (The details of beta-reduction may be found in Schmidt (1986) or any basic text on the lambda calculus, and will not be given here). Function application is strict in the left argument, in the sense that $(\perp_{D \to D})\,a$ equals $\perp_D$ for any second argument a in $\mathtt{D}$. Beta-reduction of an application may be performed as soon as the abstraction variable x is identified, even where the body e is not yet fully simplified. Thus, function application behaves in a way analogous to our other primitive operations, simplifying as soon as the 'outermost constructor' is known.

A *functional* is a continuous function whose mapping is of the form $D \to D$ (same domain D for input and output). Often domain D is itself a function space.

Theorem 2.1 (Schmidt, 1986, p. 114)
For any domain D, the least fixed point of a continuous functional $F: D \to D$ exists and is defined to be $\mathbf{fix}F = \bigsqcup_{i \geqslant 0}\{F^i(\perp)\}$, *where $F^i = F \circ F \circ \ldots \circ F$, i times. The fixed point operator* **fix** *is implemented via the axiom below*:

$$\mathbf{fix}(F) = F(\mathbf{fix}(F)).$$

Given domains $D_1, \ldots, D_n$, the domain $D_1 + \ldots + D_n$ is a collection of elements of the form d_{D_i}, where $d \in D_i$.

If *elmnt* is an element of $D_1 + \ldots + D_n$, and if, for $1 \leqslant i \leqslant n$, e_i: D whenever free occurrences x_i in e_i are replaced by elements in D_i, then the following is a value in D:

$$\begin{aligned} &\textbf{cases } \mathit{elmnt} \textbf{ of is}D_1(x_1) \rightarrow e_1 \\ &\qquad [] \ \textbf{is}D_2(x_2) \rightarrow e_2 \\ &\qquad \ldots \\ &\qquad [] \ \textbf{is}D_n(x_n) \rightarrow e_n \\ &\textbf{end}. \end{aligned}$$

If *elmnt* is replaced by any element of form a_{D_i}, then the **cases** expression above equals $[a/x_i]e_i$. If *elmnt* is replaced by $\perp$, then the expression equals $\perp_D$. When the case is the top-level construct in a function definition, we often use an equational notation with one equation for each subdomain, plus one equation mapping $\perp$ to $\perp$. For instance, Stoy (1977) defines two operations over disjoint sums via this mechanism.

Definition 2.1
For each $1 \leqslant i \leqslant n$, operator $\_|D_i$: $(D_1 + \ldots + D_n) \rightarrow D_i$ (read as the ***retracts*** *from domain $D_1 + \ldots + D_n$ to D_i) is defined as follows:*

$$\begin{aligned} \perp | D_i &= \perp_{D_i} \\ X_{D_i} | D_i &= X \\ X_{D_j} | D_i &= \perp_{D_i} \textit{for } i \neq j. \end{aligned}$$

Definition 2.2
For each $1 \leqslant i \leqslant n$ operator $\_\varepsilon D_i$: $(D_1 + \ldots + D_n) \rightarrow \mathtt{B}$ is defined as follows:

$$\begin{aligned} \perp \varepsilon D_i &= \perp_{\mathtt{B}} \\ X_{D_i} \varepsilon D_i &= \textbf{TRUE} \\ X_{D_j} \varepsilon D_i &= \textbf{FALSE} \textit{ for } i \neq j. \end{aligned}$$

These operations can be used for run-time typechecking.

2.1.3 A recursive domain for functional programming

The domain we propose for definition of a Lisp-like functional programming language is the solution to the following recursive equation:

$$\mathtt{D} = \mathtt{B} + \mathtt{A} + \mathtt{D} \times \mathtt{D} + \mathtt{D} \rightarrow \mathtt{D},$$

where B refers to the booleans, and A to a finite set of atoms. That is, LispLike's domain contains booleans, atoms, ordered pairs of smaller elements (to create lists and trees) and continuous functions. At the top level, D is a disjoint sum of four subdomains: booleans, atoms, ordered pairs from D and functions over D. To further compact the notation for subdomain tags, we rename subdomains $\mathtt{B}$, $\mathtt{A}$, $\mathtt{D}\times\mathtt{D}$ and $\mathtt{D}\rightarrow\mathtt{D}$ as b, a, p and f (for *b*ooleans, *a*toms, ordered *p*airs and *f*unctions, respectively). Similarly, we will write the retracts $\_|\mathtt{B}$, $\_|\mathtt{A}$, $\_|\mathtt{D}\times\mathtt{D}$ and $\_|\mathtt{D}\rightarrow\mathtt{D}$ as $\_|b$, $\_|a$, $\_|p$ and $\_|f$, respectively. The type predicates $\_\varepsilon\mathtt{B}$, $\_\varepsilon\mathtt{A}$, $\_\varepsilon\mathtt{D}\times\mathtt{D}$ and $\_\varepsilon\mathtt{D}\rightarrow\mathtt{D}$ will be written as $\_\varepsilon b$, $\_\varepsilon a$, $\_\varepsilon p$ and $\_\varepsilon f$, respectively.

2.1.4 Equality

A programmer will frequently wish to compare elements for equality. Unfortunately, strong equality ($\equiv$) is not generally computable over pointed complete partial orders. For example, consider the predicate $P\colon\mathtt{B}\rightarrow\mathtt{B}$, where P is defined as $\lambda x.x\equiv\mathbf{TRUE}$. We have $\perp\sqsubseteq\mathbf{TRUE}$, yet $P(\perp)\not\sqsubseteq P(\mathbf{TRUE})$, demonstrating that $\equiv$ is not monotonic. Remember that $\perp$ often represents the value of an inherently nonterminating computation. The undecidability of the *halting problem* prevents us from always recognising when an argument equals $\perp$. Instead, we must rely on a strict approximation to $\equiv$. The function $\lambda x.\lambda y.\mathbf{if}(x,y,\mathbf{not}(y))$, where **not** is defined as $\lambda x.\mathbf{if}(x,\mathbf{FALSE},\mathbf{TRUE})$, *is* monotonic (and continuous). This kind of approximation is called *weak equality*.

We would like to define a form of weak equality over $\mathtt{D}$. Unfortunately, there does not seem to be any reasonable continuous approximation to equality over $\mathtt{D}\rightarrow\mathtt{D}$. Equality over $\mathtt{D}\times\mathtt{D}$ can be approximated to the extent that equality can be approximated over $\mathtt{D}$ (two ordered pairs are equal iff their corresponding elements are equal). We present the following approximation of equality over $\mathtt{D}$, via a system of mutually-recursive functions, again relying on rewrite rule notation in lieu of using the case statement. The first step is the determination of the first argument's source.

$$\begin{aligned}
&\mathbf{DeqD}?(\perp_{\mathtt{D}}, obj) = \perp_b\\
&\mathbf{DeqD}?(boolean_b, obj) = \mathbf{DeqB}?(obj, boolean)\\
&\mathbf{DeqD}?(atom_a, obj) = \mathbf{DeqA}?(obj, atom)\\
&\mathbf{DeqD}?(pair_p, obj) = \mathbf{DeqP}?(obj, pair)\\
&\mathbf{DeqD}?(function_f, obj) = \mathbf{DeqF}?(obj, function).
\end{aligned}$$

After stripping the tag from the first argument, control is passed to a second function which checks the tag of the other argument, before any further evaluation. These functions, $\mathbf{DeqB}?\colon\mathtt{D}\times\mathtt{B}\rightarrow\mathtt{B}$, $\mathbf{DeqA}?\colon\mathtt{D}\times\mathtt{A}\rightarrow\mathtt{B}$, $\mathbf{DeqP}?\colon\mathtt{D}\times(\mathtt{D}\times\mathtt{D})\rightarrow\mathtt{B}$ and $\mathbf{DeqF}?\colon\mathtt{D}\times(\mathtt{D}\rightarrow\mathtt{D})\rightarrow\mathtt{B}$ return **FALSE** if the other tag disagrees, and otherwise continues the examination, if appropriate. These special functions are given below:

The definition of $\mathbf{DeqB}?\colon\mathtt{D}\times\mathtt{B}\rightarrow\mathtt{B}$ is as follows.

$$\begin{aligned}
&\mathbf{DeqB}?(\perp_{\mathtt{D}}, boolean) = \perp_b\\
&\mathbf{DeqB}?((b_1)_b, b_2) = \mathbf{if}(b_1, b_2, \mathbf{if}(b_2, \mathbf{FALSE}, \mathbf{TRUE}))\\
&\mathbf{DeqB}?(atom_a, boolean) = \mathbf{FALSE}\\
&\mathbf{DeqB}?(pair_p, boolean) = \mathbf{FALSE}\\
&\mathbf{DeqB}?(function_f, boolean) = \mathbf{FALSE}.
\end{aligned}$$

Given a function **AeqA**?: $A \times A \to B$, which compares two atoms for equality (and which will be presented later), the definition of **DeqA**?: $D \times A \to B$ is as follows:

$$\begin{aligned}
&\textbf{DeqA}?(\bot_D, atom) = \bot_b\\
&\textbf{DeqA}?(boolean_b, atom) = \textbf{FALSE}\\
&\textbf{DeqA}?((atom_1)_a, atom_2) = \textbf{AeqA}?(atom_1, atom_2)\\
&\textbf{DeqA}?(pair_p, atom) = \textbf{FALSE}\\
&\textbf{DeqA}?(function_f, atom) = \textbf{FALSE}.
\end{aligned}$$

We can define the function **AeqA**?: $A \times A \to B$ to compare two atoms for equality:

$$\begin{aligned}
&\textbf{AeqA}?(\bot_a, atom) = \bot_b\\
&\textbf{AeqA}?(\mathbf{A}_1, atom) = \mathbf{isA}_1(atom)\\
&\dots\\
&\textbf{AeqA}?(\mathbf{A}_n, atom) = \mathbf{isA}_n(atom).
\end{aligned}$$

We define function **DeqP**?: $D \times (D \times D) \to B$ as follows (note that this function contains a recursive call to **DeqD**?):

$$\begin{aligned}
&\textbf{DeqP}?(\bot_D, pair) = \bot_b\\
&\textbf{DeqP}?(boolean_b, pair) = \textbf{FALSE}\\
&\textbf{DeqP}?(atom_a, pair) = \textbf{FALSE}\\
&\textbf{DeqP}?((p_1)_p, p_2) = \textbf{if}(\textbf{DeqD}?(\textbf{left}(p_1), \textbf{left}(p_2)), \textbf{DeqD}?(\textbf{right}(p_1), \textbf{right}(p_2)), \textbf{FALSE})\\
&\textbf{DeqP}?(function_f, pair) = \textbf{FALSE}.
\end{aligned}$$

Function **DeqF**?: $D \times (D \to D) \to B$ is trivial. It returns **FALSE** if the tag of the first argument disagrees with the tag of the second argument, and $\bot_b$ otherwise. The details are:

$$\begin{aligned}
&\textbf{DeqF}?(\bot_D, function) = \bot_b\\
&\textbf{DeqF}?(boolean_b, function) = \textbf{FALSE}\\
&\textbf{DeqF}?(atom_a, function) = \textbf{FALSE}\\
&\textbf{DeqF}?(pair_p, function) = \textbf{FALSE}\\
&\textbf{DeqF}?((f_1)_f, f_2) = \bot_b.
\end{aligned}$$

Note that the functions defined by these mutually recursive specifications are correctly implemented by mutually recursive sets of rewrite axioms in such a way that any application of one of these functions can be simplified as soon as the outermost constructor of the left-most argument becomes known.

Theorem 2.2
The function **AeqA**? *is symmetric in its two arguments; i.e. for any* $a_1, a_2 \in A$, $\textbf{AeqA}?(a_1, a_2) \equiv \textbf{AeqA}?(a_2, a_1)$.

Proof
Since the domain A has a finite number of elements, this can be verified by enumerating all possibilities. □

Theorem 2.3
The function **DeqD**? *is symmetric in its two arguments; i.e. for any* $d_1, d_2 \in \mathrm{D}$, **DeqD**?$(d_1, d_2) \equiv$ **DeqD**?(d_2, d_1).

Proof Sketch
Replace the set of equations for each function with a single case statement. In the body of **DeqD**?, replace the function calls by their defining case statements. Take the resulting recursive definition of **DeqD**? and abstract out the function name, writing it instead as the least fixed point of the associated functional. One can then prove its symmetry via *fixed point induction* (sometimes called *computational induction*; see Manna (1974) for a good introduction). □

Fixed point induction is one of several techniques which denotational semantics provides for proving properties of programs. Though this is a valuable benefit of our approach to functional programming, the details of these techniques is beyond the scope of this paper.

Though the functions used to build the approximation to equality could have been defined by composing other more primitive semantic operators, we prefer to implement them directly via simplification equations (the same way the true primitives are implemented – the motivation for this should become apparent in section 4.2). When mentioning 'semantic primitives' in later sections, we will have these 'less primitive' functions in mind as well.

2.2 *Formal definition*

A LispLike program is an expression to be evaluated. The syntax is:

```
expr  ::=  (expr) | TRUE | FALSE | A_1 | ... | A_n
           | cons(expr, expr) | car(expr) | cdr(expr)
           | if(expr, expr, expr) | null?(expr)
           | bool?(expr) |atom?(expr) | pair?(expr)
           | func?(expr)
           | expr = expr
           | identifier | let identifier be expr in expr
           | letrec identifier be expr, ..., identifier be expr in expr
           | λ identifier . expr
           | expr expr.
```

These constructs have close analogs in other functional languages, so only a brief explanation of the denotational equations is required.

A LispLike program is a syntactic expression, built up via the composition of subexpressions, as shown in the above BNF. Every legal expression denotes an object in domain D, and this denotation gives meaning to the expressions. The meaning of an expression is defined in terms of the denotations of its subexpressions.

The equations below define the semantic function $\mathscr{E}$, which maps each syntactic *expression* to an object in domain D. In each defining equation, the environment ρ associates pre-defined syntactic identifiers with other elements in domain D (ρ is therefore in `[Id→D]`).

We present the semantic equations for the various constructs in the order of their appearance in the BNF:

- Parentheses provide clarity and override the default left-associativity:

$$\mathscr{E}[\![(expr)]\!]\,\rho = \mathscr{E}[\![expr]\!]\,\rho.$$

- For each syntactic atom (represented by $\mathtt{A}_i$) in a program, we assume the existence of an atomic object (represented by $\mathbf{A}_i$) in the domain of atoms A, and therefore, with the appropriate tag, in D. In practice, an initial quote distinguishes an atom from an identifier:

$$\mathscr{E}[\![\mathrm{A}_i]\!]\,\rho = (\mathbf{A}_i)_a.$$

- The `cons` specifies an ordered pair, useful for creating lists and binary trees. We will permit lists to be written in the `[...]` notation, e.g. `['apple, 'orange, 'grape]`, but this is only an abbreviation for the nested-pair representation using `cons`:

$$\mathscr{E}[\![\mathtt{cons}(expr1, expr2)]\!]\,\rho = <(\mathscr{E}[\![expr1]\!]\,\rho), (\mathscr{E}[\![expr2]\!]\,\rho>_p.$$

- To select a pair's left and right elements, `car` and `cdr` are used. Though these are useful only when applied to an ordered pair, because LispLike is untyped they may be applied to any legal expression. Applications of `car` and `cdr` to inappropriate sorts will be undefined, i.e. they will denote $\perp_{\mathrm{D}}$. The denotational equation for a `car` expression could have been written via the **cases** operation:

$$\begin{aligned}\mathscr{E}[\![\mathtt{car}(expr)]\!]\,\rho = \mathbf{cases}\ \mathscr{E}[\![expr]\!]\ \mathbf{of}\ &\mathbf{is}\mathrm{B}(bool) \to \perp\\ [\,]\ &\mathbf{is}\mathrm{A}(atom) \to \perp\\ [\,]\ &\mathbf{is}\mathrm{D}\times\mathrm{D}(pair) \to \mathbf{left}(pair)\\ [\,]\ &\mathbf{is}\mathrm{D}\to\mathrm{D}(func) \to \perp\\ \mathbf{end}.&\end{aligned}$$

The **cases** statement would remove the tag and apply the relevant primitive to the ordered pair. If the argument were from any of the other subdomains of D, (or if the argument is $\perp_{\mathrm{D}}$), the result would be $\perp_{\mathrm{D}}$.

Because the definition of 'cdr' would require a similar case analysis, we prefer to abstract out the case analysis via retracts. The retract $\_|p$ strips the tag that was added when the ordered-pair was inserted from D × D into D. If the argument had been inserted into D from some other domain, $\_|p$ returns the undefined ordered pair (so that selection of either side yields $\perp_{\mathrm{D}}$). Though a real implementation might offer an error message in such a case, we believe that, within the declarative paradigm, such messages are best viewed as commentaries on the computation process, e.g. as supplementary output from a rudimentary symbolic debugger invoked by default. If the programmer wishes to trap and handle such useless applications, he is free to test the argument *before* applying the operator:

$$\mathscr{E}[\![\mathrm{car}(expr)]\!]\,\rho = \mathbf{left}((\mathscr{E}[\![expr]\!]\,\rho)\,|\,p)$$
$$\mathscr{E}[\![\mathrm{cdr}(expr)]\!]\,\rho = \mathbf{right}((\mathscr{E}[\![expr]\!]\,\rho)\,|\,p).$$

• Equations for the booleans resemble those for the atoms. Included are some basic boolean functions:

$$\mathscr{E}[\![\text{TRUE}]\!]\,\rho = \mathbf{TRUE}_b$$
$$\mathscr{E}[\![\text{FALSE}]\!]\,\rho = \mathbf{FALSE}_b$$
$$\mathscr{E}[\![\text{not}(expr)]\!]\,\rho = (\mathbf{if}((\mathscr{E}[\![expr]\!]\,\rho) \mid b, \mathbf{FALSE}, \mathbf{TRUE}))_b$$
$$\mathscr{E}[\![\text{if}(expr1, expr2, expr3)]\!]\,\rho = \mathbf{if}((\mathscr{E}[\![expr1]\!]\,\rho) \mid b, \mathscr{E}[\![expr2]\!]\,\rho, \mathscr{E}[\![expr3]\!]\,\rho).$$

The conditional, `if`(*condition*, *expr2*, *expr3*), may also be written as `if` *condition* `then` *expr2* `else` *expr3* `fi`.

• The programmer can test whether an expression denotes a boolean, an atom, an ordered pair, or a function:

$$\mathscr{E}[\![\text{bool?}(expr)]\!]\,\rho = ((\mathscr{E}[\![expr]\!]\,\rho)\,\varepsilon b)_b$$
$$\mathscr{E}[\![\text{atom?}(expr)]\!]\,\rho = ((\mathscr{E}[\![expr]\!]\,\rho)\,\varepsilon a)_b$$
$$\mathscr{E}[\![\text{pair?}(expr)]\!]\,\rho = ((\mathscr{E}[\![expr]\!]\,\rho)\,\varepsilon p)_b$$
$$\mathscr{E}[\![\text{func?}(expr)]\!]\,\rho = ((\mathscr{E}[\![expr]\!]\,\rho)\,\varepsilon f)_b.$$

• In some cases, expressions can be compared for equality. The equality predicate can compare booleans, atoms, and provided it can compare the respective subtrees, ordered pairs. It does not attempt to compare functions for equality (if you try, it returns $\perp_b$). The condition `null?` tests whether its argument equals the atom `'nil`:

$$\mathscr{E}[\![(expr1 = expr2)]\!]\,\rho = (\mathbf{DeqD}?(\mathscr{E}[\![expr1]\!]\,\rho, \mathscr{E}[\![expr2]\!]\,\rho))_b$$
$$\mathscr{E}[\![\texttt{null?}(expr)]\!]\,\rho = \mathscr{E}[\![\texttt{'nil} = expr]\!]\,\rho.$$

In a recursive higher-order language, the comparison of two functions for equality is not generally computable. Our equality predicate therefore denotes $\perp_b$ when applied to two functions. A typed language might prevent equality from being applied to functions in the first place. For instance, ML distinguishes *equality types* from other types which may contain functions (Milner, 1984). Lisp (McCarthy, 1965), using the `equal` predicate, and Scheme (Abelson and Sussman, 1985), using the `equal?` predicate, compare the syntax of functional expressions for equality. To maintain referential transparency, however, it is important to distinguish equality of syntax from semantic equality.

The user is free to use `cons` and atoms to define his own notation for function definition and application, and even an interpreter to evaluate 'function' applications. If this is done, then = could be used to compare such expressions for syntactic equality. However, to be able to reason about higher-order programs, it is nice to have functions directly supported by the denotational semantics, even if true function equality cannot be made available.

• We can look up identifiers in the environment, and also create new bindings. The `let` construct establishes the static scope of an identifier, and associates a value to it. The `letrec` establishes a static scope for one or more identifiers whose values are defined via mutual recursion. A program is invalid if it contains a reference to an identifier outside its scope. The semantic function $\mathscr{D}$ creates a new environment by extending the current environment whenever a new identifier definition is encountered. The application of an environment to an identifier denotes the object in D which the environment associates with that identifier:

$$\mathcal{E}[\![\mathit{identifier}]\!]\,\rho = \rho(\mathit{identifier})$$
$$\mathcal{E}[\![\texttt{let}\ \mathit{id}\ \texttt{be}\ \mathit{expr}_1\ \texttt{in}\ \mathit{expr}_2]\!]\,\rho = \mathcal{E}[\![\mathit{expr}_2]\!]\,\rho[\mathcal{E}[\![\mathit{expr}_1]\!]\,\rho/\mathit{id}]$$
$$\mathcal{E}[\![\texttt{letrec}\ \mathit{defs}\ \texttt{in}\ \mathit{expression}]\!]\,\rho = \mathcal{E}[\![\mathit{expression}]\!]\,(\mathcal{D}[\![\mathit{defs}]\!]\,\rho)$$
$$\mathcal{D}[\![\mathit{id}\ \texttt{be}\ \mathit{expr}]\!]\,\rho = \rho[\mathbf{fix}(\lambda X.(\mathcal{E}[\![\mathit{expr}]\!]\,\rho[X/\mathit{id}]))/\mathit{id}]$$
$$\mathcal{D}[\![\mathit{id}\ \texttt{be}\ \mathit{expr},\ \mathit{defs}]\!]\,\rho = \mathcal{D}[\![\mathit{defs}]\!]\,(\rho[\mathbf{fix}(\lambda X.(\mathcal{E}[\![\mathit{expr}]\!]\,(\mathcal{D}[\![\mathit{defs}]\!]\,\rho[X/\mathit{id}])))/\mathit{id}]).$$

• We can create functions through lamba abstraction, and apply functions to their arguments. A function parameter is represented by an identifier:

$$\mathcal{E}[\![\lambda\,\mathit{id}.\mathit{expr}]\!]\,\rho = (\lambda x.(\mathcal{E}[\![\mathit{expr}]\!]\,\rho[x/\mathit{id}]))_f$$
$$\mathcal{E}[\![\mathit{expr}_1\ \mathit{expr}_2]\!]\,\rho = ((\mathcal{E}[\![\mathit{expr}_1]\!]\,\rho)\,|f)\,(\mathcal{E}[\![\mathit{expr}_2]\!]\,\rho).$$

In the above equations, 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 a list.

2.3 A sample program execution

Consider the following program:

```
letrec append be λ x y. if null?(x) then y else
                                cons(car(x), append(cd(x), y))
    in append ['g, 'h] ['a, 'b].
```

Translating from syntax to semantics, we get:

```
𝓔 ⟦ letrec append be λ x. λ y. if null?(x) then
                           y else cons(car(x), append (cdr(x), y))
    in append cons('g,cons('h,'nil)) cons('a,cons
    ('b,nil)) ⟧ []
 = 𝓔 ⟦ append cons ('g,cons('h,nil))
                                     cons('a,cons('b,'nil)) ⟧ ρ₁
```

where ρ_1 is

```
𝓓 ⟦ append be λ x. λ y. if null?(x) then y else
                                 cons(car(x), append(cdr(x), y)) ⟧ [].
```

Before continuing further, let us determine the value of environment ρ_1.

```
𝓓 ⟦ append be λ x. λ y. if null?(x) then y else
                                 cons(car(x), append(cdr(x), y)) ⟧ []
= [fixλZ.(𝓔 ⟦ λ x. λ y.
    if null?(x) then y else cons (car(x), append
                                 (cdr(x), y)) ⟧ [Z/append] / append]
```

. . .

$$= [\mathbf{fix}\lambda Z.(\lambda X.(\lambda Y.\ \mathbf{IF}((\mathbf{DeqD}?(\text{'}\mathbf{nil}_a, X)_b)|b,\ Y\ \langle\ \mathbf{LEFT}(X|p), (\mathcal{E}[\![\texttt{append(cdr(x))}]\!][\texttt{Z/append}][X/\texttt{x}][Y/\texttt{y}])_f\ Y\rangle_p))_f)_f/\texttt{append}]$$

$$= [\mathbf{fix}\lambda Z.(\lambda X.(\lambda Y.\ \mathbf{IF}((\mathbf{DeqD}?(\text{'}\mathbf{nil}_a, X)_b)|b,\ Y,\ \langle\ \mathbf{LEFT}(X|p), ((Z|f)\mathbf{RIGHT}(X|p))|f\ Y\rangle_p))_f)_f/\texttt{append}].$$

Before using this environment to evaluate other syntax, we may wish to simplify it further. This is dangerous because simplification of λ-expressions in the language of denotational semantics does not always terminate. However, we need not worry about non-termination at this stage so long as we exclude beta-reduction and fixed point expansion from our simplifications.

$$[\mathbf{fix}\lambda Z.(\lambda X.(\lambda Y.\ \mathbf{IF}(\mathbf{AeqD}?(X, \text{'}\mathbf{nil}),\ Y,\ \langle\mathbf{LEFT}(X|p), ((Z|f)\,\mathbf{RIGHT}(X|p))|f\ Y\rangle_p))_f)_f/\texttt{append}].$$

All references to ρ_1 in the following will refer to the above environment. We now continue with our previous development.

$$\mathcal{E}[\![\ \texttt{append cons('g,cons('h,'nil)) cons('a,cons('b,'nil))}\ ]\!]\rho_1$$
$$= (\mathcal{E}[\![\ \texttt{append cons('g,cons('h,nil))}]\!]\rho_1)|f\ \ \mathcal{E}[\![\ \texttt{cons('a,cons('b,cons ('b,'nil))}]\!]\rho_1$$
$$= ((\mathcal{E}[\![\ \texttt{append}]\!]\rho_1)|f\mathcal{E}[\![\ \texttt{cons('g,cons('h,'nil))}]\!]\rho_1)|f\ \ \mathcal{E}[\![\ \texttt{cons('a,cons('b,'nill))}\ ]\!]\rho_1$$
$$=((\mathbf{fix}\lambda Z.(\lambda X.(\lambda Y.\ \mathbf{IF}(\mathbf{DeqA}?(X, \text{'}\mathbf{nil}),\ Y,\ \langle\ \mathbf{LEFT}(X|p),\ ((Z|f)\,\mathbf{RIGHT}(X|p))|f\ Y\rangle_p))_f)_f)|f\langle\text{'}\mathbf{g}_a, \langle\text{'}\mathbf{h}_a, \text{'}\mathbf{nil}_a\rangle_p\rangle_p)|f\langle\text{'}\mathbf{a}_a,\ \langle\text{'}\mathbf{b}_a, \text{'}\mathbf{nil}_a\rangle_p\rangle_p.$$

We have completed the compilation into our domain-theoretic notation. What remains is the interpretation of this notation via leftmost simplification of the redexes. We begin by rewriting the **fix** expression. Everytime we need to reduce an expression of the form

$$\mathbf{fix}\ \lambda Z.body$$

we replace it with

$$(\lambda X.(\lambda Y.\mathbf{IF}(\mathbf{DeqA}?(X, \text{'}\mathbf{nil}),\ Y, \langle\ \mathbf{LEFT}(X|p), ((\mathbf{fix}\lambda Z.body|f)\ \mathbf{RIGHT}(X|p))|f\ Y\rangle_p))_f)_f.$$

Expanding the **fix** operator, we obtain:

$$\begin{array}{l}
(((\lambda X.(\lambda Y. \\
\quad \mathbf{IF}(\mathbf{DeqA}?(X, \mathbf{'nil}), \\
\qquad Y, \\
\qquad \langle \mathbf{LEFT}(X \mid p), ((\mathbf{fix}\lambda Z.\mathit{body} \mid f), \\
\qquad\quad \mathbf{RIGHT}(X \mid p)) \mid f\; Y\rangle_p))_f)_f) \mid f \langle \mathbf{'g}_a, \langle \mathbf{'h}_a, \mathbf{'nil}_a\rangle_p\rangle_p) \mid f \langle \mathbf{'a}_a, \langle \mathbf{'b}_a, \mathbf{'nil}_a\rangle_p\rangle_p \\
= ((\lambda X.(\lambda Y. \\
\quad \mathbf{IF}(\mathbf{DeqA}?(X, \mathbf{'nil}), \\
\qquad Y, \\
\qquad \langle \mathbf{LEFT}(X \mid p), \\
\qquad\quad ((\mathbf{fix}\lambda Z.\mathit{body} \mid f)\, \mathbf{RIGHT}(X \mid p)) \mid f\; Y\rangle_p))_f) \langle \mathbf{'g}_a, \langle \mathbf{'h}_a, \mathbf{'nil}_a\rangle_p\rangle_p) \mid f \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \langle \mathbf{'a}_a, \langle \mathbf{'b}_a, \mathbf{'nil}_a\rangle_p\rangle_p.
\end{array}$$

The next two leftmost operations are beta-reduction followed by simplification of a retract, yielding:

$$\begin{array}{l}
(\lambda Y.\mathbf{IF}(\mathbf{DeqA}?(\langle \mathbf{'g}_a, \langle \mathbf{'h}_a, \mathbf{'nil}_a\rangle_p\rangle_p, \mathbf{'nil}), \\
\quad Y, \\
\quad \langle \mathbf{LEFT}(\langle \mathbf{'g}_a, \langle \mathbf{'h}_a, \mathbf{'nil}_a\rangle_p\rangle_p \mid p), \\
\qquad ((\mathbf{fix}\lambda Z.\mathit{body} \mid f)\, \mathbf{RIGHT}(\langle \mathbf{'g}_a, \langle \mathbf{'h}_a, \mathbf{'nil}_a\rangle_p\rangle_p \mid p)) \mid f\; Y\rangle_p)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \langle \mathbf{'a}_a, \langle \mathbf{'b}_a, \mathbf{'nil}_a\rangle_p\rangle_p.
\end{array}$$

Performing beta-reduction again:

$$\begin{array}{l}
\mathbf{IF}(\lambda \mathbf{DeqA}?(\langle \mathbf{'g}_a, \langle \mathbf{'h}_a, \mathbf{'nil}_a\rangle_p\rangle_p, \mathbf{'nil}), \\
\quad \langle \mathbf{'a}_a, \langle \mathbf{'b}_a, \mathbf{'nil}_a\rangle_p\rangle_p, \\
\quad \langle \mathbf{LEFT}(\langle \mathbf{'g}_a, \langle \mathbf{'h}_a, \mathbf{'nil}_a\rangle_p\rangle_p \mid p), \\
\qquad ((\mathbf{fix}\lambda Z.\mathit{body} \mid f)\, \mathbf{RIGHT}(\langle \mathbf{'g}_a, \langle \mathbf{'h}_a, \mathbf{'nil}_a\rangle_p\rangle_p \mid p)) \mid f \langle \mathbf{'a}_a, \langle \mathbf{'b}_a, \mathbf{'nil}_a\rangle_p\rangle_p\rangle_p) \\
= \mathbf{IF}(\mathbf{FALSE}, \\
\quad \langle \mathbf{'a}_a, \langle \mathbf{'b}_a, \mathbf{'nil}_a\rangle_p\rangle_p, \\
\quad \langle \mathbf{LEFT}(\langle \mathbf{'g}_a, \langle \mathbf{'h}_a, \mathbf{'nil}_a\rangle_p\rangle_p \mid p), \\
\qquad ((\mathbf{fix}\lambda Z.\mathit{body} \mid f)\, \mathbf{RIGHT}(\langle \mathbf{'g}_a, \langle \mathbf{'h}_a, \mathbf{'nil}_a\rangle_p\rangle_p \mid p)) \mid f \langle \mathbf{'a}_a, \langle \mathbf{'b}_a, \mathbf{'nil}_a\rangle_p\rangle_p\rangle_p) \\
= \langle \mathbf{LEFT}(\langle \mathbf{'g}_a, \langle \mathbf{'h}_a, \mathbf{'nil}_a\rangle_p\rangle_p \mid p), \\
\quad ((\mathbf{fix}\lambda Z.\mathit{body} \mid f)\, \mathbf{RIGHT}(\langle \mathbf{'g}_a, \langle \mathbf{'h}_a, \mathbf{'nil}_a\rangle_p\rangle_p \mid p)) \mid f \langle \mathbf{'a}_a, \langle \mathbf{'b}_a, \mathbf{'nil}_a\rangle_p\rangle_p\rangle_p \\
= \langle \mathbf{'g}_a, ((\mathbf{fix}\lambda Z.\mathit{body} \mid f)\, \mathbf{RIGHT}(\langle \mathbf{'g}_a, \langle \mathbf{'h}_a, \mathbf{'nil}_a\rangle_p\rangle_p \mid p)) \mid f \langle \mathbf{'a}_a, \langle \mathbf{'b}_a, \mathbf{'nil}_a\rangle_p\rangle_p\rangle_p.
\end{array}$$

Continued leftmost simplification will eventually produce:

$$\langle \mathbf{'g}_a, \langle \mathbf{'h}_a, \langle \mathbf{'a}_a, \langle \mathbf{'b}_a, \mathbf{'nil}_a\rangle_p\rangle_p\rangle_p\rangle_p.$$

2.4 Equational reasoning in LispLike

Equivalence of expressions is based upon mathematical equality in the domain of the denoted semantic objects. One consequence of having infinite objects in the domain (objects such as infinite lists and functions with infinite traces) is that no algorithm exists to decide the equality of two arbitrary elements. (Equational reasoning is easier in weaker systems, and some have used this fact as an argument against higher-order objects (Goguen, 1988). Nevertheless, one can sometimes prove the equality of functions and thereby prove theorems about the language and demonstrate the correctness of program transformations. Earlier, we proved that the function corresponding to '=' is symmetric in its arguments. Below is another example.

Theorem 2.4
The LispLike expression

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

is a Y-combinator, i.e. for any LispLike expression g, Y g and g (Y g) both denote the same element of LispLike's semantic domain.

Proof
Y has no free identifiers, so we can ignore its environment. For any environment ρ,

$$\mathscr{E}[\![Y]\!]\,\rho = (\lambda F.((\lambda X.F|f((X|f)\,X))_f)\,|f(\lambda X.F|f((X|f)\,X))_f)_f.$$

This simplifies to:

$$(\lambda F.(\lambda X.F|f((X|f)\,X))\,(\lambda X.F|f((X|f)\,X))_f)_f.$$

Let G be the semantic object denoted by g. Therefore, $Y g$ denotes:

$$((\lambda F.(\lambda X.F|f((X|f)\,X))\,(\lambda X.F|f\,((X|f)\,X))_f)_f)\,|f\,G.$$

This simplifies to

$$(\lambda F.(\lambda X.F|f((X|f)\,X))\,(\lambda X.F|f((X|f)\,X))_f)\,G.$$

Beta reduction produces

$$(\lambda X.G|f((X|f)\,X))\,(\lambda X.G|f((X|f)\,X))_f. \tag{1}$$

If the above is what $Y g$ denotes, then $g\,(Y g)$ must certainly denote:

$$(G|f)((\lambda X.(G|f)((X|f)\,X))\,(\lambda X.(G|f)((X|f)\,X))_f). \tag{2}$$

Beta reduction of (1) produces

$$G|f(((\lambda X.G|f((X|f)\,X))_f|f)\,(\lambda X.G|f((X|f)\,X))_f).$$

Simplifying the second occurrence of $|f$ yields:

$$G|f((\lambda X.G|f((X|f)\,X))\,(\lambda X.G|f((X|f)\,X))_f).$$

But that is exactly what $g\,(Y g)$ was shown to denote in line (2) above. Therefore, $Y g$ and $g\,(Y g)$ denote the same object, and are considered equal. □

3 The PowerFuL programming language

This section extends LispLike with a relative set abstraction construct based on lower powerdomain theory, hence the name *PowerFuL* (*Power*domains for *Fu*nctional and *L*ogic programming). To understand why such a construct provides the expressive power of first-order Horn logic, we must view logic programming from another perspective: a set of Horn logic clauses defines a set of recursively enumerable predicates whose arguments are elements of the program's *Herbrand universe*. The Herbrand universe is a set of finite, fully-defined first order terms, containing a finite number of atomic values, and trees built by nesting a variety of ordered-sequence constructors called *functors*, with the atomic values as leaves. A LispLike program is capable of denoting the elements of a Herbrand universe, provided one is satisfied with `cons` as the sole constructor – experience with Lisp testifies to its generality and power.

In order to define predicates over this universe of terms, we note that a Horn logic predicate is a recursively-enumerable relation whose attributes represent elements of the Herbrand universe. That is, such a relation can be viewed as a (possibly infinite) set of *tuples* of elements from the Herbrand universe. A tuple is an ordered sequence whose length is determined by the arity of the predicate. A LispLike program is capable of representing any such tuple as a list (built using `cons`). What Lisplike lacks is the ability to denote recursively-enumerable sets. A relative set abstraction construct provides this missing capability.

The BNF of PowerFuL appends the following equations to the BNF of LispLike:

expr	::=	`set?`(*expr*) | U(*expr*, *expr*) | {*expr*: *qualifier* *}
qualifier	::=	*condition* | *enumeration*
condition	::=	*expr*
enumeration	::=	*identifier* ∈ *expr*.

This syntax adds the recursively enumerable set as a first-class datatype. If the condition in a set clause is unsatisfied, the clause represents ∅. Enumerations in set clauses permit us to build new sets from old. When there are no qualifiers to satisfy, the set-clause indicates a singleton set, and the ':' is usually omitted. Below are a few short program fragments to demonstrate the use of set clauses:

```
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}
   splits be λ l. U(
            {cons('nil, l)},
            {cons( cons(car(l),car(S)), cdr(S) ) :
            S ∈ splits(cdr(l)) })
in
 ...
```

The function `crosspod` creates a set of ordered pairs; in each pair, the left element is from the first set, and the right element is from the second. The function `filter` takes a set and creates a subset, including only those elements which can satisfy the predicate. The elements of `intersection`(set_1, set_2) are those objects which

PowerFuL can determine are in both set_1 and set_2. (Note that the result will not contain any higher-order objects, since PowerFuL cannot verify the equality of such objects). The function `splits` takes a list, and computes a set, each element of which splits the input list into two shorter lists.

Ordered pairs may contain sets as elements, functions may take sets as parameters and return sets as results, and sets may contain functions, or even other sets. The following example demonstrates the use of a set which contains functions:

```
letrec
   map be λ f.λ l.if null?(l) then 'nil
                                  else cons(f(car(l)),
                                  map(f,cdr(l)))fi
   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]}.
```

As the generator set for `F`, `U({one}, {two}, {three})`, is enumerated, each element is tested to see whether it produces list `['c, 'c, 'c]` when mapped across the elements of `['x, 'y, 'z]`. The resulting set will contain those functions from the generator which satisfy the condition. The value of this set-abstraction is therefore `{three}`.

To demonstrate logic programming in PowerFuL, one merely translates assertions about the truth of predicates into statements about set-membership. Where a definite clause program asserts *P*(*tuple*), we could equivalently assert that *tuple* ∈ *P*, where *P* now refers to a set, and *tuple* is represented as a list. For example, consider the following program and goal, written in Prolog syntax (Clocksin and Mellish, 1981):

```
      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]).
```

The logical variables in each program clause are (implicitly) universally quantified over the Herbrand universe. With a syntax more oriented towards sets, one might instead 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.
```

To make the universal quantification of logic variables more explicit (and thus come closer to the syntactic style of PowerFuL), one might write the following (where hu stands for 'Herbrand universe'):

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

Here, we have taken the liberty of writing h, t, y, z ∈ hu instead of four separate enumerations. We have used mutually-recursive definite clauses to define sets (instead of predicates). We could call this paradigm *set clause programming*, but it is really just another syntax for relational clauses. It becomes obvious that the predicates/sets are defined via relative set abstraction and the Herbrand universe. Furthermore, the Herbrand universe itself can be defined via relative set abstraction.

Let the booleans TRUE and FALSE and the set of atoms A_i be the zero-arity constructors. Suppose we are willing to live with cons as the single nonzero-arity constructor. Then the Herbrand universe, hu, can be defined in PowerFuL by the following:

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

Therefore, the equivalent PowerFuL program is:

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

   app be U( { [ [],L,L ] : L ∈ hus},
             {[ [H | T], Y, [H | Z] ] : H,T,Y,Z ∈ hu,
                                  W ∈ app, W=[T,Y,Z]})
   rev be U( {[ [], [] ]},
             {[ [H | T], Z] : H,T,Y,Z ∈ hu, V ∈ rev, W ∈ app,
                                  V = [T, Y], W = [Y, [H], Z]})

in
   { L : L ∈ hu, V ∈ rev, V = [L, ['a, 'b, 'c]] }.
```

This technique demonstrates that any first-order relational clause program may be easily translated into PowerFuL, and that we have indeed captured the full expressive power of definite clauses.

In the last example, the PowerFuL program used sets to express definite clause relations, which in turn were used to define functions. With so many levels of translation, it is no wonder that the resulting PowerFuL version is ugly! A better PowerFuL style would be to use functions where functions are intended, and sets only where necessary.

```
letrec
   bools be U( {TRUE}, {FALSE})
   atoms be U({A₁}, ..., {Aₙ})
   hu be U(atoms, bools, {cons(X,Y) : X,Y ∈ hu})

   append be λ l1 l2. if null?(l1) then l2
                      else cons(car(l1), append(cdr(l1), l2)) fi

   reverse be λ l. if null?(l) then []
                   else append( reverse( cdr(l) ), [car(l)]) fi

in
   { L : L ∈ hu, V ∈ rev, V = [L, ['a, 'b, 'c]] }.
```

A formal definition of PowerFuL requires new semantic primitives. To be specific, we need a domain, each of whose elements represents a set of elements drawn from another domain. Such domains, called *powerdomains*, are reviewed in the next section. The formal definition of PowerFuL follows in section 3.2. Section 3.3 demonstrates the computation of sets via direct execution of the denotational semantics. Section 3.4 discusses use of the denotational semantics in equational reasoning about sets.

3.1 Review of powerdomain primitives

Section 2.1 discussed the definition of a cross-product domain from two simpler base domains; what remains to be discussed is the definition of a domain, each of whose elements represents a subset of a simpler base domain. This is the essence of *powerdomain* theory. This section reviews the basics of powerdomain theory, so that our functional programming domain may be extended to include sets. The next section will use this enhanced domain as the semantic basis of a functional programming language incorporating logic programming capability.

3.1.1 Varieties of powerdomains

Several versions of powerdomains have been proposed. These differ primarily according to the interpretation given to a computation that is divergent or otherwise undefined, and therefore disagree on the partial ordering over sets. The *demonic* powerdomain (Main, 1987) is used when program correctness requires that *all* possible computation paths be successful. Computation can then be viewed as the process of paring out that which *cannot* result. Therefore, the least defined element, $\perp_{\mathscr{P}(D)}$, of the demonic powerdomain $\mathscr{P}(D)$ is equated with the set of *all* elements in domain D. If set A is a subset of set B, one considers set A to be *more defined* than set B. This powerdomain also goes by the names *Smyth* powerdomain and *upper* powerdomain. The *angelic* powerdomain (Main, 1987) is the dual of the demonic. Angelic powerdomains are appropriate for describing whatever successful outcomes *might* result from a non-deterministic program's execution. Finitely failed and non-terminating paths add no new information. This powerdomain is also called the *Hoare powerdomain* and the *lower* powerdomain. The Egli–Milner powerdomain (Main, 1987) was the first powerdomain developed. It combines aspects of both the demonic and angelic powerdomains. A partially-defined set under this partial order combines information both about that which the set is known to contain *and* that which it is known not to contain. Because it demands preservation of all possible information about the computation, it is very difficult to work with.

The fixed-point semantics of first-order Horn logic programming (Lloyd 1987; van Emden and Kowalski, 1976) can be described in terms of powerdomains. A Horn logic program defines a functional τ whose least fixed point is the set of predicate statements whose truth is implied by the program (thus demonstrating a direct connection between the fixed point semantics and the *model-theoretic* semantics). Computation of this set begins with the empty set; predicate statements are added to the set as proofs for their inclusion are found. If all derivations fail or diverge, the set of ground predicates remains empty. The addition of new failing and diverging paths may reduce the efficiency of the computation, but will not alter the final result. All elements of the Herbrand universe are fully defined, so the partial order over sets of elements reduces to the subset relation; the empty set is the $\perp$ element. These considerations imply that a very simple angelic powerdomain is being used.

When using the angelic powerdomain to model deterministic computation of sets, rather than non-deterministic computation of single values, the preferred name is *lower powerdomain*. Before incorporating sets via lower powerdomains into our functional programming domain, let us examine its properties in more detail.

3.1.2 Properties of lower powerdomains

With the lower powerdomain's partial order, a set becomes more defined both by adding new elements and by increasing the definition (according to the partial order of the base domain) of pre-existing elements.

Conventional domain theory assumes the continuity of operators. To ensure the continuity of the operator which creates a singleton set in $\mathscr{P}(D)$ from a single element in domain D, for any chain of elements $t_i \in D$, $\{\bigsqcup_i \{t_i\} \equiv \bigsqcup_i \{t_i\}$. We can find a partial order with the properties we desire only if we are willing to work with *equivalence classes* of sets. This is no problem, because under the assumption of angelic non-determinism, many distinct sets are computationally equivalent (and thus may be treated as being equal). The singleton set $\{t_i\}$ will therefore actually represent the equivalence class of sets to which that singleton set belongs.

To define the partial order over (equivalence classes of) sets, a few definitions are needed:

Definition 3.1 (Schmidt, 1986, p. 295)
A Scott-topology upon a domain D is a collection of subsets of D known as **open sets**. *A set $U \subseteq D$ is open on the Scott-topology iff:*

(1) *U is closed upwards, that is, for every $d_2 \in D$, if there exists a $d_1 \in U$ such that $d_1 \sqsubseteq d_2$, then $d_2 \in U$;*
(2) *If $d \in U$ is the least upper bound of a chain C in D, then some $c \in C$ is in U.*

The definition of open set just given is meant to capture the idea of a *computable property*.

Definition 3.2 (Schmidt, 1986, p. 296)
The symbol $\sqsubseteq_{\sim}$, pronounced 'less defined than or equivalent to', is a relation between sets. For $A, B \subseteq D$, we say that $A \sqsubseteq_{\sim} B$ iff for every $a \in A$ and open set $U \subseteq D$, if $a \in U$ then there exists a $b \in B$ such that $b \in U$ also.

Intuitively, $A \sqsubseteq_{\sim} B$ means that for every computable property held by any element of set A, there is an element in set B which also has the property.

Definition 3.3 (Schmidt, 1986, p. 296)
We say $A \approx B$ iff both $A \sqsubseteq_{\sim} B$ and $B \sqsubseteq_{\sim} A$.

Intuitively, two sets are equivalent (under the angelic assumption) if no element of one set has a computable property unless some element of the other set also has it. We denote the equivalence class containing A as $[A]$. This class contains all sets $B \subseteq D$ such that $A \approx B$.

Definition 3.4 (Schmidt, 1986, p. 297)
We define the partial order on equivalence classes as: $[A] \sqsubseteq [B]$ iff $A \sqsubseteq_{\sim} B$. For domain D, the powerdomain of D, written $\mathscr{P}(D)$, is the set of equivalence classes, each member of an equivalence class being a subset of D.

Definition 3.1 (Schmidt, 1986, p. 297)
The following operations are continuous:

$$\{\_\}: D \to \mathscr{P}(D) \text{ maps } d \in D \text{ to } [\{d\}].$$
$$\_ \cup \_: \mathscr{P}(D) \times \mathscr{P}(D) \to \mathscr{P}(D) \text{ maps } [A] \cup [B] \text{ to } [A \cup B].$$
$$^{+}: (D_1 \to \mathscr{P}(D_2)) \to (\mathscr{P}(D_1) \to \mathscr{P}(D_2)) \text{ is } \lambda f.\lambda[A].[\cup \{f(a): a \in A\}].$$

To motivate the $^{+}$ primitive, suppose an operator f accepts an element of d_1 of domain D_1 and, based on this element, implicitly defines a set of possible results, each of which is in domain D_2. Suppose that operator f then produces an element of this set, non-deterministically choosing from among the possibilities. The application $f(d_1)$ is said to denote the set of possible results, a subset of domain D_2. This subset is represented by a member of domain $\mathscr{P}(D_2)$ (the powerdomain of D_2). The operator is therefore in $D_1 \to \mathscr{P}(D_2)$. Suppose two non-deterministic operations f and g are composed, i.e. that we perform non-deterministic operation g upon input x, yielding output y, and then perform non-deterministic operation f upon y. The set of possible results is the union over y (where y is any element in the set denoted by $g(x)$) of the sets denoted by $f(y)$. We may express composition of f and g applied to x as $f^+g(x)$. The larger the set denoted by $g(x)$, the larger will be the set denoted by $f^+(g(x))$. Thus, $^+$ is interpreted as

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

The operator $^+$ may be used to distribute a function over the elements of a set. Suppose we have a set $S=\{1,2,3\}$, and we wish to create a new set such that for each element x is in S, $f(x)$ is contained in the new set. We may write

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

which reduces to

$$\{f(1),f(2),f(3)\}.$$

Our powerdomain construction differs slightly from that which is usually found in the literature. The pioneers of powerdomain theory were primarily concerned with binary non-determinism. One of the two paths would always be chosen, so every set of possible results would contain at least one value (even if that value were $\perp$). The least defined angelic powerdomain element is therefore $\{\perp\}$. To represent sets and relations directly, the lower powerdomain is lifted by a new bottom element, the empty set $\varnothing$. The empty set is needed to represent an empty relation, e.g. a predicate that is false on all arguments, or an empty set of bindings to a Horn logic goal. It is clear that $\{\perp\}$ is inadequate for this purpose. Should $^+$ be used to distribute a function over an empty set of solutions, the result, too, should be empty. But, consider what happens when the function to be distributed is a constant function: $(\lambda x.\{'a\})^+\{\perp_D\}$ reduces to $(\lambda x.\{'a\}),\perp_D$, which is $\{'a\}$. In other words, $\{\perp_D\}$ contains an element with the computable property of being an element of D (a trivial property held by *all* elements of the base domain, represented by the Scott-open set that is D itself), whereas no computable property is manifested in any of the elements of $\varnothing$.

It is clear from the partial order given in Definitions 3.2–4 that $\varnothing$, if included in the powerdomain, would be strictly less defined than any other element, including $\{\perp\}$. Though it may be reasonable to interpret the completely undefined set as empty, it seems less intuitive that the empty set should be considered undefined. With lower powerdomains, sets are assumed empty until proven otherwise. When summarising the result of a partial evaluation, we approximate an uncomputed set with $\varnothing$ to indicate that we have no evidence as of yet that the set contains anything (the *possibility* of finding elements in the future is not necessarily ruled out). When partial evaluation yields $\{\perp\}$, then at least we know that the set does contain something, even though we know nothing about it. Seen in this light, $\varnothing$ carries less information than $\{\perp_D\}$.

Another variation is that we use a noncurried variation of '$^+$': $((D\to\mathscr{P}(D))\times\mathscr{P}(D))\to\mathscr{P}(D)$. The following axioms can be used to implement $^+$:

$$F^+(\varnothing)=\varnothing$$
$$F^+(\{Expr\})=F(Expr)$$
$$F^+(Set_1\cup Set_2)=(F^+(Set_1)\cup F^+(Set_2)).$$

This primitive is not quite strict in the first argument, since

$$(\perp_{D\to\mathscr{P}(D)})^{+}(set)$$

produces $\{\perp\}$ (assuming *set* is non-empty), which, though less defined than most powerdomain elements, is more defined than $\perp_{\mathscr{P}(D)} = \varnothing$. It is, however, strict in the second parameter. Therefore, when computing using a 'leftmost' computation rule, one should consider the second argument to be 'leftmost'. A simplification axiom is applicable whenever the input set's outermost constructor is known.

It is not essential to understand every detail about the construction of powerdomains. What is important is to choose the appropriate version, and to use only those operators known to be continuous under that construct's partial order.

3.1.3 A recursive domain for functional and logic programming

The domain D for PowerFuL may contain 'sets' of elements from D, including sets of functions, and sets of sets.

We shall enhance LispLike with a set abstraction feature whose semantics are based on the lower powerdomain. The new language, will be based on the domain which satisfies the following recursive domain equation:

$$\mathtt{D} = \mathtt{B} + \mathtt{A} + \mathtt{D}\times\mathtt{D} + \mathtt{D}\to\mathtt{D} + \mathscr{P}(\mathtt{D}).$$

As before, the domain contains booleans, atoms, ordered pairs of smaller elements (to create lists and trees) and continuous functions. The difference is that this new domain also includes a subdomain of sets (elements of lower powerdomains). To de-emphasise the tags, we will replace the subdomain tag $\__{\mathscr{P}(D)}$ with $\__s$. Similarly, we will write the retract $\_|\mathscr{P}(\mathtt{D})$, as $\_|s$. The type predicate $\_\varepsilon\mathscr{P}(\mathtt{D})$ will be written as $\_\varepsilon s$. Here, s stands for *s*et.

The weak equality predicate will be extended by including **DeqS**?: $\mathtt{D}\times\mathscr{P}(\mathtt{D})\to\mathtt{B}$:

$$\begin{aligned}
&\mathbf{DeqS}?(\perp_{\mathtt{D}}, set) = \perp_b\\
&\mathbf{DeqS}?(booleans_b, set) = \mathbf{FALSE}\\
&\mathbf{DeqS}?(atom_a, set) = \mathbf{FALSE}\\
&\mathbf{DeqS}?(pair_p, set) = \mathbf{FALSE}\\
&\mathbf{DeqS}?(function_f, set) = \mathbf{FALSE}\\
&\mathbf{DeqS}?((s_1)_s, s_2) = \perp_b.
\end{aligned}$$

The following additional axioms are needed to define and implement **DeqD**?:

$$\begin{aligned}
&\mathbf{DeqD}?(set_s, obj) = \mathbf{DeqS}?(obj, set)\\
&\mathbf{DeqB}?(set_s, boolean) = \mathbf{FALSE}\\
&\mathbf{DeqA}?(set_s, atom) = \mathbf{FALSE}\\
&\mathbf{DeqP}?(set_s, pair) = \mathbf{FALSE}\\
&\mathbf{DeqF}?(set_s, function) = \mathbf{FALSE}.
\end{aligned}$$

The symmetry of **DeqD**? (see Theorem 2.3) is unaffected.

3.2 Formal definition

This section provides denotational equations for the new syntactic constructs, to be added to the denotational equations given for LispLike.

- First, we add another type-checking predicate

$$\mathscr{E}[\![\text{set?}(expr)]\!]\,\rho = ((\mathscr{E}[\![expr]\!]\,\rho)\,\varepsilon s)_b.$$

- To build sets, the user begins with singleton sets, each constructed from an element of the powerdomain's base domain. The union operation builds larger sets from smaller ones. Expressions of the form $\text{U}(set_1, \ldots, set_n)$ are syntactic sugar for a nesting of binary unions:

$$\mathscr{E}[\![\text{U}(expr_1, expr_2)]\!]\,\rho = (((\mathscr{E}[\![expr_1]\!]\,\rho)\,|\,s) \cup ((\mathscr{E}[\![expr_2]\!]\,\rho)\,|\,s))_s$$
$$\mathscr{E}[\![\{expr :\}]\!]\,\rho = (\{\mathscr{E}[\![expr]\!]\,\rho\})_s.$$

- To build a new set out of an old one, we can filter out those elements not meeting a specified condition by using a condition as a qualifier. Unless the condition evaluates to **TRUE**, the expression denotes the empty set:

$$\mathscr{E}[\![\{expr\colon condition,\ qualifierlist\}]\!]\,\rho$$
$$= (\textbf{if } (\mathscr{E}[\![condition]\!]\,\rho)\,|\,b \textbf{ then } (\mathscr{E}[\![\{expr\colon qualifierlist\}]\!]\,\rho)\,|\,s \textbf{ else } \varnothing \textbf{ fi})_s.$$

- Enumerations are the syntactic basis for relative set abstraction. An enumeration parameter is associated with a set expression which provides possible instantiations. The scope of the *enumerated* identifier includes the principal expression (left of the ':'), and also all qualifiers to the right of its introduction. (The scope of an enumerated identifier never reaches beyond the set-clause of its introduction. In case of name conflict, i.e. when the identifiers on the left-hand sides of two enumeration clauses have the same name, an occurrence of the identifier refers to the enumeration parameter of the innermost scope.)

$$\mathscr{E}[\![\{expr\colon id \in genrtr,\ qualifierlist\}]\!]\,\rho$$
$$= (\lambda\, X.\,((\mathscr{E}[\![\{expr\colon qualifierlist\}]\!]\,\rho[X/id])\,|\,s)^{+}\,(\mathscr{E}[\![genrtr]\!]\,\rho)\,|\,s)_s.$$

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

3.3 Program execution

To compute with conceptually-infinite objects, whether these objects be recursive functions, recursively-defined infinite lists, or recursively-defined infinite sets (e.g. hu), an outermost computation rule is needed. Because most common primitives are strict in their leftmost argument, the *leftmost* computation rule is traditional. For expressions in our functional notation, leftmost evaluation will produce a *normal form* (a fully-simplified equivalent expression) if one exists. If the expression equals anything other than $\perp_D$, leftmost evaluation will produce the *weak head-normal* form (Peyton Jones, 1987), i.e. an expression which, though not necessarily fully simplified, exposes the outermost constructor.

For our use, the definition of *leftmost* must be modified slightly. The primitive function '${}^{+}$': $((\mathrm{D} \to \mathscr{P}(\mathrm{D})) \times \mathscr{P}(\mathrm{D})) \to \mathscr{P}(\mathrm{D})$ is strict in its *second* argument, not the first. To distribute a function over the elements of a set, it must first evaluate its second argument (the set), at least until a singleton set constructor or union constructor is encountered (upon which the '${}^{+}$' simplifies). Therefore, we have defined the second argument of '${}^{+}$' as being leftmost.

Sets are represented as binary union trees whose leaves are singleton sets. Syntactically, this is similar to the representation of `cons` trees. The '${}^{+}$' operator distributes a function across all the leaves of the union tree; its operational behaviour is analogous to an easily constructed LISP functional which would apply an input function to all leaves of a `cons` tree. The leftmost rule should therefore retain the same completeness properties. The sufficiency of these completeness properties will be discussed in section 4. For now, however, we must be content with a sample execution to trace the use of the semantics operation '${}^{+}$' to define and implement relative set abstraction. Consider the following syntax:

```
let setOfFuncts be U({λ x.car(x)}, {λ x.cdr(x)})
    in let setOfPairs be U({cons('a, 'b)}, {cons('c, 'd)})
        in {g(x) : g ∈ setOfFuncts, x ∈ setOfPairs}.
```

Let us translate this to the semantic domain. Translating the '`let`' constructs yields:

$$\mathscr{E}[\![\texttt{\{g(x) : g} \in \texttt{setOfFuncts, x} \in \texttt{setOfPairs\}}]\!] \rho$$

where ρ maps `setOfFuncts` to

$$((\{\lambda X.\mathbf{left}(X|p))_f\}_s|s) \cup (\{(\lambda X.\mathbf{right}(X|p))_f\}_s|s))_s$$

which we will refer to as VAL1 for convenience, and maps `setOfPairs` to

$$((\{\langle '\mathbf{a}_a, '\mathbf{b}_a\rangle_p\}_s|s) \cup (\{\langle '\mathbf{c}_a, '\mathbf{d}_a\rangle_p\}_s|s))_s,$$

which we will refer to as VAL2.

Further translation of the main expression yields:

$$(\lambda G.((\mathscr{E}[\![\texttt{\{g(x) : x}\in\texttt{setOfPairs\}}]\!]\rho[G/\texttt{g}])|s)^{+}(\mathscr{E}[\![\texttt{setOfFuncts}]\!]\rho)|s)_s,$$

or

$$(\lambda G.((\mathscr{E}[\![\texttt{\{g(x) : x}\in\texttt{setOfPairs\}}]\!]\rho[G/\texttt{g}])|s)^{+}(VAL1)|s)_s.$$

Further translation yields

$$(\lambda G.(((\lambda X.((\mathscr{E}[\![\texttt{\{g(x) : \}}]\!]\rho[G/\texttt{g}][X/x])|s)^{+}(\mathscr{E}[\![\texttt{setOfPairs}]\!]$$
$$\rho[G/\texttt{g}][X/x])|s)_s)|s)^{+}(VAL1)|s)_s,$$

or

$$(\lambda G.(((\lambda X.((\mathscr{E}[\![\texttt{\{g(x) : \}}]\!]\rho[G/\texttt{g}][X/x])|s)^{+}(\mathrm{VAL2})|s)_s)|s)^{+}(\mathrm{VAL1})|s)_s.$$

Completing the translation to the semantic domain yields:

$$(\lambda G.(((\lambda X.\{(G|f)X\}_s)|s)^{+}(\mathrm{VAL2})|s)_s)|s)^{+}(\mathrm{VAL1})|s)_s.$$

Simplifying the retracts, VAL1 becomes

$$(\{(\lambda X.\mathbf{left}(X|p))_f\} \cup \{(\lambda X.\mathbf{right}(X|p))_f\})_s,$$

VAL2 becomes

$$(\{\langle '\mathbf{a}_a, '\mathbf{b}_a\rangle_p\} \cup \{\langle '\mathbf{c}_a, '\mathbf{d}_a\rangle_p\})_s,$$

and the main expression becomes

$$(\lambda G.((\lambda X.\{(G|f)\,X\})^+(\{\langle '\mathbf{a}_a, '\mathbf{b}_a\rangle_p\} \cup \{\langle '\mathbf{c}_a, '\mathbf{d}_a\rangle_p\}))^+ \\ (\{(\lambda X.\mathbf{left}(X|p))_f\} \cup \{(\lambda X.\mathbf{right}(X|p))_f\}))_s.$$

Simplifying the outermost occurrence of $^+$ yields:

$$((\lambda G.(\lambda X.\{(G|f)\,X\})^+(\{\langle '\mathbf{a}_a, '\mathbf{b}_a\rangle_p\} \cup \{\langle '\mathbf{c}_a, '\mathbf{d}_a\rangle_p\}))^+\{(\lambda X.\mathbf{left}(X|p))_f\} \cup \\ (\lambda G.((\lambda X.\{(G|f)\,X\})^+(\{\langle '\mathbf{a}_a, '\mathbf{b}_a\rangle_p\} \cup \{\langle '\mathbf{c}_a, '\mathbf{d}_a\rangle_p\})))^+\{(\lambda X.\mathbf{right}(X|p))_f\})_s.$$

Simplifying each of the two outermost occurrences of $^+$ yields:

$$((\lambda G.((\lambda X.\{(G|f)\,X\})^+(\{\langle '\mathbf{a}_a, '\mathbf{b}_a\rangle_p\} \cup \{\langle '\mathbf{c}_a, '\mathbf{d}_a\rangle_p\})))\,(\lambda X.\mathbf{left}(X|p))_f \cup \\ (\lambda G.((\lambda X.\{(G|f)\,X\})^+(\{\langle '\mathbf{a}_a, '\mathbf{b}_a\rangle_p\} \cup \{\langle '\mathbf{c}_a, '\mathbf{d}_a\rangle_p\})))\,(\lambda X.\mathbf{right}(X|p))_f)_s.$$

Further reduction yields:

$$(((\lambda X.\{(\lambda X.\mathbf{left}(X|p))\,X\})^+(\{\langle '\mathbf{a}_a, '\mathbf{b}_a\rangle_p\} \cup \{\langle '\mathbf{c}_a, '\mathbf{d}_a\rangle_p\})) \cup \\ ((\lambda X.\{(\lambda X.\mathbf{RIGHT}(X|p))\,X\})^+(\{\langle '\mathbf{a}_a, '\mathbf{b}_a\rangle_p\} \cup \{\langle '\mathbf{c}_a, '\mathbf{d}_a\rangle_p\})))_s.$$

Continuing in this vein yields:

$$(((\lambda X.\{(\lambda X.\mathbf{left}(X|p))\,X\})\,\langle '\mathbf{a}_a, '\mathbf{b}_a\rangle_p \cup (\lambda X.\{(\lambda X.\mathbf{left}(X|p))\,X\})\,\langle '\mathbf{c}_a, '\mathbf{d}_a\rangle_p) \cup \\ ((\lambda X.\{(\lambda X.\mathbf{right}(X|p))\,X\})\,\langle '\mathbf{a}_a, '\mathbf{b}_a\rangle_p \cup (\lambda X.\{(\lambda X.\mathbf{right} \\ (X|p))\,X\})\,\langle '\mathbf{c}_a, '\mathbf{d}_a\rangle_p))_s.$$

Additional beta-reductions produce:

$$((\{\mathbf{left}(\langle '\mathbf{a}_a, '\mathbf{b}_a\rangle_p|p)\} \cup \{\mathbf{left}(\langle '\mathbf{c}_a, '\mathbf{d}_a\rangle_p|p)\}) \cup (\{\mathbf{right}(\langle '\mathbf{a}_a, '\mathbf{b}_a\rangle_p|p)\} \cup \\ \{\mathbf{right}(\langle '\mathbf{c}_a, '\mathbf{d}_a\rangle_p|p)\}))_s.$$

Final simplification of retracts and ordered pair selectors yields:

$$((\{'A_a\} \cup \{'\mathbf{c}_a\}) \cup (\{'\mathbf{b}_a\} \cup \{'\mathbf{d}_a\}))_s,$$

or less formally:

$$(\{'\mathbf{a}_a, '\mathbf{c}_a, '\mathbf{b}_a, '\mathbf{d}_a\})_s.$$

3.4 Equational reasoning

Equivalence of PowerFuL expressions is based upon mathematical equality in the semantic domain of the denoted objects. To prove the correctness of program transformations, one first translates the syntax of each expression to the semantic domain. There, the equality may be proved using mathematical reasoning. Below are simple examples of theorems which are easily proved:

Theorem 3.2 (in the semantic domain)

$$(\lambda x.\{x\})^+ S = S.$$

Proof
(By definition of + and set union.)

Corollary (in the syntactic domain)

$$\{x : x \in \{y\}\} = \{y\}.$$

In general, one can demonstrate that, if PowerFuL expression S denotes an element of the form set_s, then S and $\{ z : z \in S\}$ denote the same element.

Theorem 3.3 (in the semantic domain)

$$f^{+}(S_1 \cup S_2) = f^{+}(S_1) \cup f^{+}(S_2).$$

Proof
(By definition of + and set union.)

Corollary (in the syntactic domain)

$$\{f(x) : x \in S_1 \cup S_2\} = \{f(x) : x \in S_1\} \cup \{f(x) : x \in S_2\}.$$

3.5 Discussion

In the implementation of Horn logic programming (Lloyd, 1987), procedure activation depends upon the equality of the arguments of a subgoal with the respective arguments of a corresponding clause head. It has generally been assumed that a language combining logic programming with higher-order functional programming would require that higher-order objects be tested for equality. The term usually used is *higher-order unification*, a procedure capable not only of verifying the equality of two objects, but also of finding bindings for parameters so as to *make* two expressions equal.

The reader may be disappointed that no construct is provided to compute equality over functions or over sets. In higher-order functional programming, higher-order equality is not continuous, and therefore undecidable; in first-order Horn logic there are no higher-order objects to be compared. Perhaps, then, it is reasonable to omit this operation from a language combining the two paradigms. Nevertheless, a great deal can be done with higher-order objects in functional programming which does not depend on equality testing. In the logic programming framework, where parameter passing itself is based on unification, incorporation of higher-order objects would be more difficult. This is why our synthesis is essentially functional in nature.

We would emphasise that our objective was not to provide a rich set of operations on sets; rather, we view set abstraction as merely a vehicle for integrating functional and (definite clause) logic programming. Given this goal, it is not *necessary* to have operations such as finding the cardinality of a set or deciding whether a value is or is not a member. As noted earlier, such operations would be analogous to Prolog's *metalogical* operations, e.g. negation-by-failure. A possible line of further work would be to incorporate such capabilities in an extended language, and examine their semantic and operational implications. Semantic analysis of this feature, analogous to Lloyd's analysis of negation-by-failure in Horn logic (Lloyd, 1987), would probably require use of the Egli–Milner powerdomain (Schmidt, 1986), which records both the properties which can be found among a set's elements and the properties which cannot be found among them. Noting that a complete and efficient implementation of negation-by-failure has yet to be found, serious obstacles to that line of research can be expected. Fortunately, the need for negation-by-failure will often be avoided through the use of the if/then/else construct and the type-checking primitives (Naish, 1985).

We have seen that conventionally-defined functions, such as `append`, can be used, not only in the usual way, but also within a set abstraction to compute the set of inputs which would yield the desired output. For efficiency, such a function might be compiled differently for use within and without set abstractions. PowerFuL distinguishes between usages where backtracking might be needed, and where it is not (i.e. whether or not the usage falls within a set abstraction). Actually, a theoretically-complete or *fair* implementation must compute elements of sets in *parallel*, not via backtracking, as we shall see in the next section. This distinction is important for obtaining a more efficient implementation than is possible with all functions defined as relations.

4 Operational semantics

Section 2 showed that a functional language could be executed by direct implementation of the denotational semantics, usually using a leftmost computation rule. The example in section 3 demonstrated the use of this method in computing set abstractions. This section discusses modifications to this method which must be considered in the implementation of PowerFuL.

4.1 Non-termination and computation rules

Most implementors are concerned with computation of expressions denoting objects that are finite and fully defined. Lazy languages are *demand-driven*; they compute only those finite pieces of infinite objects that are specifically requested (e.g. when *car* and *cdr* are applied to an infinite list so as to isolate a specific finite portion). In contrast to an infinite list built by *ordered* pairing, sets have no implicit ordering; the binary set union creates *un*ordered pairs of subsets. The analogy between `cons` trees and set-union trees breaks down when we realise that there is no referentially transparent way to restrict computation to a union's left or right subset – the designation of left and right are arbitrary and without meaning – and therefore there is no operation in the language to reduce an infinite set to any specific element. The user cannot ask for the 'first' element; any operation performed on one element must be performed on all.

Because sets may be infinite, we cannot wait until computation is complete before reporting our results. Even in ordinary Horn logic, a program can denote an infinite set of possible solutions, as in the following example:

```
            app([], Y, Y).
```

```
app([H | T], Y, [H | Z]) :- app(T, Y, Z).
?- app([1,2], X, Y), app(X, [1,2],Y).
```

Even though standard Prolog's search strategy is not complete for Horn logic, Prolog does make a serious effort to compute infinite sets. Rather than waiting for the entire set to be computed (which cannot happen), the system suspends and turns control over to the user each time another member of this set is computed. With each new solution, the user has a better approximation to the complete set. In the execution of a program denoting an infinite set, it is up to the user to terminate computation when the approximation becomes adequate.

Even when the set of solutions is finite, one cannot assume that a complete search of the solution space will terminate. The derivation space may contain non-productive branches which diverge. Nevertheless, a complete breadth-first Horn logic interpreter can be built. This requires the parallel computation of a union's subsets. In essence, we are saying that both subsets of a binary union are equally and simultaneously 'leftmost'. With nested unions, there can be quite a lot of parallel computation. To be sure, one can evaluate one side of the union first, but then evaluation of the other side would depend on termination of the former. This latter strategy, analogous to Prolog's depth-first search strategy, might be adopted for the sake of efficiency, but complete evaluation is at least possible in principle. Perhaps in an interactive implementation, the programmer will be able to direct the computational effort to a specific portion of the set expression. As the desire for referential transparency prevents such commands from being part of the language per se, they could be provided in the meta-linguistic environment, analogous to online-debugger commands.

4.1.1 Complete computation rule

Because computation of logic programs with infinite solution sets will not terminate, we must understand the meaning of non-terminating programs and their computation. Our approach is analogous to Vuillemin's study of systems of first-order mutually-recursive functions (Vuillemin, 1974), and also to Wadsworth's theory of ω-normal forms in the pure untyped lambda calculus (Wadsworth, 1976). When programs are apt to denote infinite objects, a complete implementation will produce a sequence of increasingly-better finite approximations to the denoted object, such that the least upper bound of this (possibly infinite) chain of finite approximations is in fact the denoted object. In this way, operational completeness is extended to consider non-terminating computations.

Program execution was demonstrated in the examples of sections 2.3 and 3.3. Once the denotational equations have translated the program syntax to the semantic domain, execution refers to the simplification of the resulting semantic expression via the axioms provided with each semantic primitive. (These axioms were provided in sections 2.1.2, 2.1.4 and 3.1.2). Each simplification is the result of an axiom being used as a left-to-right rewrite rule. We say that a semantic expression is *simplifiable* if any of these axioms is applicable, i.e. if it contains redexes. A single application of an axiom to a redex is called a *simplification*. If a semantic expression is not simplifiable, then we say the expression is *fully computed*, or alternatively, in *normal form*. Not every expression has a normal form – it may represent an object that is infinite, or perhaps not fully defined.

Though execution proceeds one simplification at a time, let us group the simplifications into *computation steps*. Each computation step simplifies some or all of an expression's redexes, but to limit the amount of work done in a single computation step; any redexes that result from simplifications in the current step will be left for later computation step. The *computation rule* decides which of an expression's redexes will be simplified in the next computation step. To aid further discussion of program execution, the following definitions provide a compact notation for these and other concepts.

Definition 4.1
If t_1 and t_2 are two expressions in our functional notation, we say that $t_1 \rightarrow t_2$ if t_2 can be produced from t_1 via a single computation step (i.e. by simplifying some or all of t_1's redexes, but leaving any new redexes).

Definition 4.2
A computation is a sequence of zero or more computation steps. We say that t_n is computed from t_0, written as $t_0 \rightarrow^* t_n$, if t_n can be produced from t_0 via a finite number (possibly zero) of '$\rightarrow$' steps. When computation steps are performed according to computation rule c, we write $t_1 \rightarrow_c t_2$ to indicate a single step, and $t_1 \rightarrow_c^* t_n$ to represent a finite computation of zero or more steps using computation rule c.

Definition 4.3.
Given an expression t, we define the current approximation of t, written $t[\perp]$, to be the result obtained by replacing each redex with the bottom element of the reducible primitive's output domain.

Theorem 4.1
For any expression t in the functional notation, $t[\perp] \sqsubseteq t$.

Proof
By monotonicity of the primitives in t. □

Corollary
If t is a fully computed expression in the functional notation, then $t \equiv t[\perp]$.

Proof
From the definition of 'fully computed', t and $t[\perp]$ are identical expressions. □

Given any expression t, we can seek approximations better than $t[\perp]$ by performing simplifications *before* approximating.

Definition 4.4
*For any expressions t_1 and t_n such that $t_1 \to^* t_n$, we say that $t_n[\bot]$ is a **finite approximation** of t_1.*

Since all our operations are continuous, we may equate an expression with the least upper bound of all possible finite approximations, i.e. for any expression t in the functional notation, $t = \bigsqcup_{t \to^* t_i} t_i[\bot]$. Note that if t has a normal form t_n, then t_n is, in fact, this least upper bound. This equation generalises the notion of normal form even to consider expressions for which simplification does not terminate.

Definition 4.5
*For any expression t and computation rule c, the **computed result** is $\bigsqcup_{t \to_c^* t_i} t_i[\bot]$.*

Theorem 4.2
For any expression t in the functional notation, and any computation rule c, the computed result approximates the true result; i.e. $\bigsqcup_{t \to_c^* t_i} t_i[\bot] \sqsubseteq \bigsqcup_{t \to^* t_i} t_i[\bot]$.

Proof
Because $\{t_i[\bot] : t \to_c^* t_i\} \subseteq \{t_i[\bot] : t \to^* t_i\}$. □

Definition 4.6
*A computation rule is said to be **complete** if the computed result does not merely approximate the true result, but actually equals it.*

Theorem 4.3
A computation rule is complete for term t if, for every derivation $t \to^* t_i$ there exists a derivation $t \to_c^* t_j$ such that $t_i[\bot] \sqsubseteq t_j[\bot]$.

Proof
The condition of the theorem implies that the true result approximates the computed result. Since the converse is also true, the computed result must equal the true result. □

4.1.2 Safe computation rules and completeness

To determine which computation rules are likely to be complete, Vuillemin developed the concept of *safety*.

Definition 4.7
For any expression t and computation rule c, let $t[c \leftarrow \bot]$ be the result of replacing every redex that would have been chosen by rule c with the bottom value of the reducible primitive's output domain.

Thus, $t[c \leftarrow \bot]$, i.e. $\bigsqcup_{t[c \leftarrow \bot] \rightarrow^* t_i} t_i[\bot]$, is taken to represent an upper bound on the quality of our approximation of t, given that none of the chosen simplifications ever get performed.

Definition 4.8
*A **safe computation step** is any computation step (i.e. a step using any rule c) where $t[c \leftarrow \bot] = t[\bot]$. A **safe computation rule** is one which performs only safe computation steps.*

Among the simplifications performed in any safe computation step are those so critical that no improvement of the current approximation would be possible were they forever omitted. If any improvement to the current approximation is possible, then a safe computation step will perform work necessary toward that end. Since any finite approximation requires only a finite amount of work (by definition), it should be possible to equal or better *any* finite approximation via a sufficient number of safe computation steps. Therefore, any safe computation rule is assumed to be complete (Vuillemin (1974) did in fact prove that safe computation rules are complete, albeit for a weaker system). We have not yet found a more rigorous proof that safety implies completeness for this denotational framework, but we believe it to be true (such a result would belong to the general literature on domain theory, and thus would be beyond the scope of this paper, in any case). We shall now discuss safe computation rules for the functional notation into which PowerFuL programs are compiled.

Definition 4.9
*The **parallel outermost rule** chooses all outermost redexes in each computation step.*

Theorem 4.4
The parallel outermost rule is a safe rule.

Proof
For any expression t, under the parallel outermost rule, all non-outermost simplification opportunities will disappear from $t[c \leftarrow \bot]$. □

It should not be necessary to simplify *all* outermost redexes in every computation step. Consider an expression headed by the conditional primitive (**if**). It seems reasonable to restrict computation to the first argument, the condition, and postpone evaluation of the other two arguments until we know which one is needed. Indeed, for *any* primitive, we would like to limit computation to the leftmost argument in which that primitive is strict, until the primitive itself is reducible. If the outermost constructor of this argument never appears, then this argument must be $\bot$. Because the primitive in question is strict in this argument, the entire expression headed by the primitive must equal $\bot$ (in which case, the values of its other arguments are irrelevant). If the outermost constructor of the first strict argument *does* appear so that the primitive may simplify, then evaluation of the other arguments has been delayed only temporarily.

The above reasoning may not apply if the primitive application occurs within the *body* of a function. Evaluation of the leftmost argument may be impeded by the presence of unbound lambda variables. However, if reduction is limited to outermost redexes, beta reduction would bind the lambda variable before redexes within the function body are even considered. The presence of unbound lambda variables only becomes a problem when computing an unapplied function for its own sake (an unusual situation indeed), in which case we simply use the full parallel-outermost rule when selecting redexes within the function body.

In each computation step, the *modified parallel-outermost* computation rule therefore simplifies all outermost redexes *except* that, in searching for redexes among the arguments of a primitive not contained within a function body, it limits consideration to redexes occurring in the primitive's leftmost strict argument. This modified parallel-outermost rule can be proven safe via structural induction on the height of the expression.

Assuming that the value being computed occurs in domain:

$$\mathtt{E} = (\mathtt{B} + \mathtt{A} + \mathtt{E}\times\mathtt{E} + \mathscr{P}(\mathtt{E}))_{\perp},$$

(i.e. the value being computed is neither an unapplied function, nor a structure containing an unapplied function), then parallel evaluation (i.e. multiple simplifications within one computation step) will only occur when computing the subsets of a union (as we already guessed) and the arguments of an ordered pair. (The latter also makes sense – purely leftmost computation of a pair of infinite lists would never begin the second list.)

4.2 Optimisations

This section modifies the reduction operational semantics so as to make logic programming (i.e. the use of hu as a set abstraction generator) practical. The standard way of computing a relative set expression is to enumerate the elements of the generator set and reinstantiate the body as each element of the generator set appears. Consider the following expression:

```
letrec
   append be λ l1 l2. if null?(l1) then l2
                      else cons (car(l1), append(cdr(l1),l2)) fi
in {X : X ∈ hu, append(X, ['a, 'b]) = ['c, 'd, 'a, 'b]}.
```

The set denoted by hu is analogous to Horn logic's Herbrand universe of first-order terms. Evaluation of this set would result in a union tree whose leaves are each a singleton set containing one element. As it is created, it is absorbed into the development of the union tree of elements for the set abstraction as a whole. Each leaf in the final result is a value that would be denoted by

```
letrec
   append be λ l1 l2. if null?(l1) then l2
                      else cons (car(l1), append(cdr(l1),l2)) fi
```

in

{ *term* : append(*term*, ['a, 'b]) = ['c, 'd, 'a, 'b]}

where *term* represents one element of hu. This is accomplished via the simplification rules for '$^+$' (the semantic primitive which gives meaning to our set abstraction construct). Each leaf singleton set is easily computed; for values of *term* not satisfying the equality, the subset reduces to $\varnothing$. This is the general method for computing relative set abstractions, and it works for any generator the user can define.

However, when hu is the generator set, this conceptually simple 'generate-and-test' approach is wasteful. Such an implementation is analogous to a Horn logic interpreter which first enumerates ground bindings for the goal's logical variables, then for each enumeration, attempts to find a derivation using ground instantiations of the program clauses. Rather than instantiating blindly, and then trying to complete a derivation, the resolution method (van Emden and Kowalski, 1976; Lloyd, 1987) avoids redundant work by interleaving instantiation and derivation stages. In resolution, a logical variable becomes instantiated only to the extent necessary to satisfy the inference rule's equality test. Partial instantiation of two or more non-ground terms to ensure their equality is called *unification*. The substitution computed to implement such a partial instantiation is called a *unifier*. The result of a resolution derivation is a *most general computed* answer substitution. This is a partial binding of the logical variables in the goal for which every ground extension is a *correct answer substitution* (a substitution that would instantiate the goal in such a way as to be implied by the program clauses).

The derivation of the general answer substitution resembles a parameterised ground derivation. That is, for every ground extension of the computed answer substitution, there is a direct proof of the instantiated goal that can be viewed as an instantiation of the resolution derivation. The need for an infinity of essentially similar derivations is thus avoided. Though it is easy to extend a most general answer substitution, to produce (ground) correct answer substitutions, in practice this is not done, and in a sense, the solution is left unfinished. Reporting results via most general computed answer substitutions is more economical than individually reporting each of the infinite ways in which each most general solution can be extended.

To gain the efficiency of the resolution method, an analogous approach must be taken in the implementation of PowerFuL, to treat enumeration parameters of hu as logical variables, rather than instantiating them with each individual member of this set. To recognise when this case occurs, we make hu a built-in syntactic primitive.

4.2.1 *First-order universe*

This section makes the set hu, the user-defined Herbrand universe, a syntactic primitive in PowerFuL. First, we must add the following line to the BNF:

expr ::= hu.

Every syntactic construct must have a denotational equation to give it meaning.

Hence the following:

$$\mathscr{E}[\![\mathrm{hu}]\!]\,\rho = \mathbf{hu}_s.$$

The definitions below lead to the definition of the set **hu**.

Definition 4.10
The set **bools**$:\mathscr{P}(\mathrm{B})$ *is defined to be* $\{\mathbf{TRUE}\} \cup \{\mathbf{FALSE}\}$.

Definition 4.11
Given that domain A *contains atoms* $\mathbf{A}_1 \dots \mathbf{A}_n$, *the set* **atoms**$:\mathscr{P}(\mathrm{A})$ *is defined to be* $\{\mathbf{A}_1\} \cup \dots \cup \{\mathbf{A}_n\}$.

Definition 4.12
Given the above definitions, we recursively define the set **hu**$:\mathscr{P}(\mathrm{D})$, *as follows*:

$$\mathbf{hu} = (\lambda x.\{x_b\})^+(\mathbf{bools}) \cup (\lambda x.\{x_a\})^+(\mathbf{atoms}) \cup (\lambda x.(\lambda y.\{<x,y>_p\})^+\mathbf{hu})^+\mathbf{hu}.$$

The first subset of **hu** is therefore $\{\mathbf{TRUE}_b, \mathbf{FALSE}_b\}$, the second subset is $\{(\mathbf{A}_1)_a, \dots, (\mathbf{A}_n)_a\}$, and the third contains all pairs of the form $<x,y>_p$ where x and y are also elements of **hu**.

4.2.2 Logical variable abstraction

Because logical variables in a Horn logic clause are universally quantified over the Herbrand universe, a program clause actually represents the set of all its possible ground instantiations. Analogously, an expression of the form

$$(\lambda x.\mathit{body})^+\,\mathbf{hu},$$

represents the union of all possible sets '$\mathit{body}\sigma$', where σ is a substitution replacing x with any element of **hu**. The brute force way of computing this would begin by reducing **hu** into its component subsets, atoms, booleans and ordered pairs, as per Definition 4.11, whereupon, '$^+$' could be simplified, distributing the abstraction body over each subset, and eventually applying it to each ground term as it appears. Instead, such an expression is rewritten to

$$\mathit{hu}(x).\mathit{body},$$

indicating our intention to treat x as a logical variable *constrained* to denote a value from **hu**. Though Horn logic variables are *implicitly* constrained to represent values from the Herbrand universe, our constraints are made explicit. This is necessary because other kinds of constraints are also considered. For instance, the expressions $\mathit{atom}(x).\mathit{body}$ and $\mathit{bool}(x).\mathit{body}$ are analogously constructed from $(\lambda x.\mathit{body})^+$ **atoms** and $(\lambda x.\mathit{body})^+$ **bools**, respectively. Later, we will allow logical variables to be further constrained by inequalities. Thus, a parameterised set expression has two parts, a set of constraints which introduce logical variables and delimit their scope, and a parameterised body. Instead of recomputing *body* for each trivial instantiation, we will evaluate *body* in its parameterised form, which compactly denotes the union of all instantiations satisfying the constraints. A fully-computed set will still be represented as a union tree whose leaves are singleton sets (or sometimes ϕ), but we now permit the singleton sets to be in parameterised form, analogous to Horn logic's non-ground computed answer substantiations. So that singleton sets will be parameterised individually, a parameterised union is broken up into the union of two parameterised subsets. Because of the commutativity and associativity of set union, an expression of the form

$$constraints.(exp_1 \cup exp_2)$$

can be rewritten as

$$constraints.exp_1 \cup constraints.exp_2.$$

The '$^+$' operation is the only primitive function which is strict in an argument from a powerdomain. We need a new simplification rule permitting '$^+$' to accept a parameterised singleton set as an argument. Thus, an expression of the form

$$(\lambda x.body)^+(constraints.\{element\}),$$

will be rewritten as

$$constraints.((\lambda x.body)^+\{element\}).$$

This transformation is valid due to the associativity and commutativity of set union.

Finally, a parameterised empty set, such as *constraints*. ϕ, will be simplified to ϕ.

It is not enough that '$^+$' accept parameterised sets – we must also ensure that the presence of the parameters themselves will not impede simplification of the basic operators. When evaluating a parameterised set expression, one strives to compute the parameterised body just as though every logical variable had been replaced by any arbitrary value satisfying the constraints. Whenever the outermost constructor of a primitive's leftmost argument is available, that expression is simplifiable. A primitive must *remain* simplifiable if a logical variable appears as its leftmost argument instead.

4.2.3 Simplifying primitives applied to logical variables

The presence of a logical variable in place of a fully computed element of **hu** presents no special problems, until it appears as the leftmost argument of a primitive we wish to simplify. Were the logical variable replaced by any fully-computed value that satisfied the variable's constraints, the primitive would be simplifiable. If the computation rule would indeed choose to perform this simplification at this time, then presence of a logical variable could impede simplification of the primitive. In such a case, simplification of the primitive may require narrowing the range of values a logical variable may assume (by a case analysis of the possibilities).

We have introduced logical variables which represent elements of B, A and D. Therefore, we must consider operators whose leftmost parameters represent arguments from any of these domains. We describe each case using the notation:

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

Here, a logical variable u is introduced by a constraint, and occurs as the argument of *prim* somewhere in the body. The primitive's position in the expression is relevant only to the extent that we assume the computation rule chooses *prim* to simplify in the current computation step.

The only primitive with leftmost argument of domain B is **if**. When a logical variable of domain B is its leftmost argument, we simplify split into two subsets. That is, an expression of the form

$$bool(B).(\ldots \mathbf{if}(B, exp_2, exp_3)\ldots)$$

is rewritten as the union of two subsets, each with the logical variable constraint removed, as follows:

$$(\ldots \mathbf{if}(B, exp_2, exp_3)\ldots)[\mathbf{TRUE}/B]$$
$$\cup\ (\ldots \mathbf{if}(B, exp_2, exp_3)\ldots)[\mathbf{FALSE}/B].$$

The primitives strict in an argument of domain A are **AeqA**? and, for each atom $\mathbf{A}_i$, $\mathbf{isA}_i$?. Let us consider an expression of the form

$$atom(A).(\ldots \mathbf{isA}_i(\mathrm{A})\ldots).$$

Given n atoms in the program, we might split this into n subsets, analogous to the treatment of boolean variables. This would be inefficient in a program with many atoms. Indeed, we divide into two cases, the first where the logical variable *is* assumed to be $\mathbf{A}_i$ (so the primitive can simplify to **TRUE**), and the other where it is assumed to be something else (so the primitive can simplify to **FALSE**). The first case eliminates the logical variable, binding it to $\mathbf{A}_i$, and the other case generates an *inequality constraint*:

$$(\ldots \mathbf{isA}_i?(A)\ldots)[A/\mathbf{A}_i])$$
$$\cup\ atom(A).(A \neq \mathbf{A}_i).(\ldots \mathbf{FALSE}\ldots).$$

This is the basic idea, but there are a few exceptions to keep in mind. Had the set expression *already* contained the constraint $(A \neq \mathbf{A}_i)$, we could have simplified the $\mathbf{isA}_i$? directly, without splitting into subsets, knowing that the result would be false for any possible instantiation.

A possible complication of using inequality constraints with atoms is that, with a sufficient number of inequality constraints, the constraints may become unsatisfiable. When the constraints are unsatisfiable, the parameterised set expression is equivalent to ϕ, and further computation of the body is wasted effort. Although this problem is unlikely to arise in a program containing many atoms, a concerned implementer of PowerFuL might seek to insert tests to ensure satisfiability.

Handling of **AeqA**? is only a bit trickier. If only the first argument is a logical variable, a simple remedy is to swap the two arguments (by Theorem 2.3, **AeqA**? is symmetric in its arguments), and thus delay the need to examine the logical variable. Even if both arguments are logical variables, their equality may already be known. If there is already an inequality constraint between them, then the **AeqA**? expression simplifies to **FALSE**; if both arguments are the same variable, then it simplifies to **TRUE**. Otherwise, one must split into subsets, analogous to the split for $\mathbf{isA}_i$?. An expression of the form

$$atom(X).(\ldots atom(Y).(\ldots \mathbf{AeqA}?(Y,X)\ldots))$$

is replaced with the union of subset

$$atom(X).(\ldots \mathbf{Aeq}?(Y,X)\ldots)[X/Y]$$

(in which the resulting **atom**?(X,X) simplifies to **TRUE**) with subset

$$atom(X).(\ldots atom(Y).X \neq Y.(\ldots \mathbf{AeqA}?(Y,X)\ldots)),$$

(in which **atom**?(Y,X) may be simplified to **FALSE**). Note that the order of arguments to **AeqA**? is irrelevant. The second expression will reduce to $\varnothing$ should X and Y later become bound to the same atom, or to one another.

The primitives with a leftmost parameter of domain D are **DeqD**? (and its supporting primitives **DeqB**?, **DeqA**?, **DeqP**?, **DeqF**?, and **DeqS**?), the type checking primitives (**bool**?, **atom**?, **pair**?, **func**? and **set**?), and the retracts $\_|b$, $\_|a$, $\_|p$, $\_|f$ and $\_|s$).

The handling of **DeqD**? is analogous to that just described for **AeqA**?, except that we need no longer worry about unsatisfiability from too many constraints (since its variables range over an infinite set). A parameterised expression reduces to ϕ when both sides of an inequality become identical. An inequality constraint may be dropped when the two sides are no longer unifiable.

Simplification of the remaining primitives depends on the tag of the leftmost argument. The primitives **func**?, **set**?, $\_|f$ and $\_|s$, **DeqF** and **DeqS** each simplify uniformly over every object that a logical variable may represent (because logical variables *never* denote functions or sets). Thus, no instantiation is necessary in the following simplifications:

$$hu(D).(\ldots \mathbf{func}?(D)\ldots) \to hu(D).(\ldots \mathbf{FALSE}\ldots)$$
$$hu(D).(\ldots \mathbf{func}!(D)\ldots) \to hu(D).(\ldots \Omega\ldots)$$
$$hu(D).(\ldots \mathbf{set}?(D)\ldots) \to hu(D).(\ldots \mathbf{FALSE}\ldots)$$
$$hu(D).(\ldots \mathbf{set}!(D)\ldots) \to hu(D).(\ldots \phi\ldots).$$
$$hu(D).(\ldots \mathbf{DeqF}?(D, function)\ldots) \to hu(D).(\ldots \mathbf{FALSE}\ldots).$$
$$hu(D).(\ldots \mathbf{DeqS}?(D, set)\ldots) \to hu(D).(\ldots \mathbf{FALSE}\ldots).$$

The other primitives (**bool**?, **atom**?, **pair**?, $\_|b$, $\_|a$, $\_|p$, **DeqB**?, **DeqA**? and **DeqP**?) require case analysis. The simplification rule depends upon the subdomain of the (leftmost) argument. If the argument is a logical variable, the three alternatives, which must be considered separately, are boolean, atom, and ordered pair. We divide the parameterised set expression into three subsets, each of which instantiate the logical variable only to the extent of exposing the tag (so that the ordinary reduction rule will apply).

Let *prim* represent any of these primitives. An expression of the form

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

(for primitives with arity 2, the second argument is 'understood') is replaced by:

$$\begin{aligned}&bool(w).(\ldots prim(u)\ldots)[w_b/u]\\ \cup\ &atom(w).(\ldots prim(u)\ldots)[w_a/u]\\ \cup\ &hu(v).hu(w).(\ldots prim(u)\ldots)[<v,w>_p/u].\end{aligned}$$

In the first branch of the union, the logical variable is partly instantiated to represent a boolean; in the second branch, to represent an atom; and in the third branch, to represent an ordered pair of subterms. All references to the old logical variable are replaced by references to the new variable(s). In each subset, the tag is now available, permitting *prim* to simplify using an ordinary rewrite rule.

4.2.4 Example

Consider the unification of two non-ground first-order terms: `[A|B]` and `[C|'d]`. We seek replacements for logical variables `A`, `B` and `C` such that `[A|B] = [C|'d]`.

To calculate the set of unifiers, one might execute the PowerFuL program:

```
{[A,B,C] : A,B,C ∈ hu, [A | B]=[C | 'd]}
```

Each unifier will be represented by a linked-list, whose elements are the respective bindings for `A`, `B` and `C`. Removing some syntactic sugars, the denoted object is defined as

$$\mathscr{E}[\![\ \{\ \texttt{cons(A,cons(B, cons(C,'nil)))}\ :\ \texttt{A} \in \texttt{hu},\ \texttt{B} \in \texttt{hu},\ \texttt{C} \in \texttt{hu},\ \texttt{cons(A,B)=cons(C,'d)}\ \}\ ]\!]\ \Lambda.$$

Compiling this into the semantic domain and simplifying retracts yields:

$$(\lambda A.\lambda B.\lambda C.(\mathbf{if\ DeqD?}(\langle A,B\rangle_p,\langle C,'\mathbf{d}\rangle_p),\{\langle A,\langle B,\langle C,'\mathbf{nil}_a\rangle_p\rangle_p\rangle_p\},\phi)^+hu^+hu^+hu)_s.$$

If not for the optimisations, this program would produce a set in the form of an infinite union tree. Each leaf in the tree would represent one possible assignment of values to `A`, `B` and `C`. For those assignments which unify $\langle A,B\rangle_p$ with $\langle C,'\mathbf{d}\rangle_p$, the leaf would be the singleton set $\{\langle A,\langle B,\langle C,'\mathbf{nil}_a\rangle_p\rangle_p\rangle_p\}$. At leaves corresponding to assignments which are not unifiers, the value would be ϕ.

Instead, we treat A, B, and C as logical variables:

$$(term(A).term(B).term(C).(\mathbf{if\ DeqD?}(\langle A,B\rangle_p,\langle C,'\mathbf{d}\rangle_p),\\ \{\langle A,\langle B,\langle C,'\mathbf{nil}_a\rangle_p\rangle_p\rangle_p\},\varnothing))_s.$$

Simplifying the equality yields

$$(term(A).term(B).term(C).(\mathbf{if(DeqP?}(\langle C,'\mathbf{d}_a\rangle_p,\langle A,B\rangle),\\ \{\langle A,\langle B,\langle C,'\mathbf{nil}_a\rangle_p\rangle_p\rangle_p\},\phi)))_s,$$

which reduces to:

$$\begin{aligned}(term(A).term(B).term(C).(&\mathbf{if(if(DeqD?}(C, A), \mathbf{if(DeqD?}('\mathbf{d}_a, B),\\ &\mathbf{TRUE, FALSE) FALSE}),\\ &\{\langle A, \langle B, \langle C, '\mathbf{nil}_a\rangle_p\rangle_p\rangle_p\},\\ &\phi)))_s.\end{aligned}$$

We should reduce the leftmost occurrence of '**DeqD**?'. This splits into two subsets, one which assumes the equality of C and A, and one which assumes their inequality. The subset corresponding to the inequality of C and A is:

$$\begin{aligned}(term(A).term(B).term(C).C \neq A.(&\mathbf{if(if(FALSE, if(DeqD?}('\mathbf{d}_a, B),\\ &\mathbf{TRUE, FALSE) FALSE},\\ &\{\langle A, \langle B, \langle C, '\mathbf{nil}_a\rangle_p\rangle_p\rangle_p\},\\ &\phi)))_s.\end{aligned}$$

The above simplifies to:

$$(term(A).term(B).term(C).C \neq A.\phi)_s,$$

or, simply ϕ.

The subset which assumes the equality of A and C is:

$$\begin{aligned}&(term(A).term(B).(\mathbf{if(if(DeqD?}(A, A), \mathbf{if(DeqD?}('\mathbf{d}_a, B), \mathbf{TRUE, FALSE) FALSE}),\\ &\qquad\{\langle A, \langle B, \langle C, '\mathbf{nil}_a\rangle_p\rangle_p\rangle_p\},\\ &\qquad\phi)))_s,\end{aligned}$$

which simplifies to

$$\begin{aligned}&(term(A).term(B).\mathbf{if(if(DeqA?}(B, '\mathbf{d}), \mathbf{TRUE, FALSE})\\ &\qquad\qquad \{\langle A, \langle B, \langle C, '\mathbf{nil}_a\rangle_p\rangle_p\rangle_p\}, \phi))_s.\end{aligned}$$

To simplify **DeqA**?, we must consider three possibilities for logical variable B – i.e. that it is either an atom, a boolean or an ordered pair. Clearly, the three subsets corresponding to the second and third option will both simplify to ϕ. The subset which instantiates B to be an atom (D_a) is:

$$\begin{aligned}&(term(A).atom(D).\mathbf{if(if(DeqA?}(D_a, '\mathbf{d}), \mathbf{TRUE, FALSE})\\ &\qquad\qquad \{\langle A, \langle D_a, \langle A, '\mathbf{nil}_a\rangle_p\rangle_p\rangle_p\}, \phi))_s.\end{aligned}$$

The subexpression **DeqA**?$(D_a, '\mathbf{d})$ simplifies to **AeqA**?$(D, '\mathbf{d})$. Because only the left argument is a logical variable, this is rewritten to **AeqA**?$('\mathbf{d}, D)$, which further simplifies to **is'd**?(D). The greater expression therefore simplifies to:

$$(term(A).atom(D).\mathbf{if(if(is'd?}(D), \mathbf{TRUE, FALSE)} \{\langle A, \langle D_a, \langle A, '\mathbf{nil}_a\rangle_p\rangle_p\rangle_p\}, \phi))_s.$$

This splits into two subsets, one which assumes that $B \neq 'd$, which quickly simplifies to $\varnothing$, and the following, which assumes the opposite:

$$(term(A).\mathbf{if(if(is'd?}('d), \mathbf{TRUE, FALSE)} \{\langle A, \langle '\mathbf{d}_a, \langle A, '\mathbf{nil}_a\rangle_p\rangle_p\rangle_p\}, \phi))_s,$$

which simplifies to:

$$_s(term(A).\{\langle A, \langle '\mathbf{d}_a, \langle A, '\mathbf{nil}_a\rangle_p\rangle_p\rangle_p\})_s.$$

Thus, the set contains all lists of the form

$$\langle A, \langle '\mathbf{d}_a, \langle A, '\mathbf{nil}_a\rangle_p\rangle_p\rangle_p,$$

where A is any arbitrary element of **hu**.

4.2.5 Discussion

The technique just described is analogous to narrowing in constructor-based term-rewriting systems (Reddy, 1985), where, to enable reduction obstructed by a logical variable, one instantiates the variable in all possible ways which would permit further reduction. The computational explosion usually associated with narrowing is avoided, however, because we need only narrow with respect to a handful of semantic primitives, into which all PowerFuL programs are compiled.

As discussed earlier, this technique is analogous to resolution's computation of a Horn logic program's most general computed answer substitutions. The handling of inequality resembles Lee Naish's inequality predicate for Prolog (Naish, 1985). His inequality predicate also fails when two terms are identical, succeeds when two terms cannot be unified, and delays until further instantiation has occurred when two terms are unifiable but not identical. If all other subgoals have succeeded, but the variables in an inequality constraint are still insufficiently bound to ensure satisfaction of the constraint, Naish's Prolog gives an error message. In such a situation, we prefer to report the inequality constraint as part of the solution, i.e. as a kind of negative binding (Khabaza discusses negative unification in (Khabaza 1984). Constraint logic programming (Jaffar and Lassez, 1987) provides a precedent for permitting constraints in answers, and one might choose to expand PowerFuL's capabilities along those lines.

In PowerFuL, logical variables must be instantiated to simplify a whole range of primitives, not just equality. Therefore, unification cannot be a single monolithic operation, as it is in Prolog, but has to be broken into its component parts. That is, the binding of one logical variable to another is handled separately from the narrowing of a variable's range into subdomains. This follows the approach taken by Robinson in LogLisp (Berkling *et al.*, 1982). Traces of logic programs in PowerFuL may therefore seem longer than their Prolog equivalents, but execution is not necessarily slower, because in actual execution of Prolog, unification must still be performed one step at a time.

4.2.6 Correctness observations

Soundness

Given any *parameterised* derivation $t \to^* t_t$, for every instantiation σ which replaces each logical variable with an element (from **hu**, **atoms** or **bools**, as is appropriate) that satisfies the logical variable's constraints, $t_t\sigma[\perp]$ approximates t.

This is true because of the meaning of a parameterised expression (in terms of '+'), and the fact that all steps in a parameterised derivation replace expressions by equals.

Completeness
Any element of a set which can be computed by a non-optimised derivation can be computed as an instantiation of a parameterised derivation.

This is true because when dividing a parameterised expression into cases (for the purpose of simplifying a primitive), every possible instantiation of logical variables which satisfies the constraints is a possible instantiation of one of the subcases. No possible instantiation is ever lost. Furthermore, any computation which can be performed after replacement of the enumeration variable by a term can be performed on the parameterised body.

4.2.7 Summary

A sound and complete operational semantics was developed via direct implementation of the denotational semantics. Then, we demonstrated that computation with logical variables can avoid the need to explicitly enumerate the Herbrand universe. This optimisation required only minor modifications to the denotational definition's direct implementation. Only when the set abstraction generator is the Herbrand universe is the enumeration parameter treated as a logical variable. Logical variables are instantiated only to the extent needed to perform primitive simplifications. Wherever simplification of a primitive operation depends on specific information about the element represented by a logical variable, the set is divided into subsets, each making a different assumption about the element being represented. Within each subset, the primitive now has enough information to simplify.

The syntax and denotational semantics of PowerFuL made no reference to logical variables. The logical variable is merely an *operational* concept to improve the execution efficiency when hu is used as a generator. More complicated sets are therefore permitted as generators (e.g. such as sets of functions, sets of sets, etc.), and in these cases, the default mechanism (generate, instantiate, and continue) is used.

5 Conclusions

Throughout the past decade, there has been substantial interest in combining functional and logic programming. This section evaluates the language PowerFuL by comparing it to previous work, and presents areas of further work.

5.1 Related work

Many of the early proposals sought to weave together the *operational semantics* of Lisp-Like and Prolog-like languages (Robinson and Sibert, 1982; Lindstrom, 1985; Reddy, 1985; Smolka and Panangadan, 1985; Darlington *et al.*, 1986). Often there was little regard as to whether the resulting language contained an easily-described declarative subset. Since our drive to combine functional and logic programming was motivated by the observation that both were declarative paradigms, we consider the development of a pure and declarative combined paradigm to be a key aspect of the problem.

Much of the recent work has dealt with *equational programming* (Goguen and

Meseguer, 1984; Dershowitz and Plaisted, 1985; Jayaraman and Silbermann, 1986; You and Subrahmanyam, 1986; Darlington and Guo, 1989). In this approach, functions over a domain of first-order terms are defined and implemented by a set of equations which provide a confluent set of left-to-right rewrite rules. These rewrite rules may be used to reduce an expression to a normal form and, through the technique of *narrowing* (Goguen and Meseguer, 1984; Dershowitz and Plaisted, 1985) to solve for variables in a goal equation. This approach recalls from functional programming the ability to define and use functions, and from logic programming, the ability to solve for values which satisfy constraints. Nevertheless, approaches based on syntactic term-rewriting fail to address the idea of functions as first-class objects.

Incorporation of higher-order functions into equational programming requires *higher-order unification* (Robinson, 1986), which is not generally computable. Though it is always possible to *simulate* higher-order functions in a first-order language (Warren, 1982; Smolka and Panangaden, 1985; Goguen, 1988), there is no guarantee of *referential transparency*; reliance upon such simulation reduces the usefulness of the formal semantics in reasoning about programs making use of this feature. One would prefer the language's semantics to speak of such objects (and their properties) *explicitly*.

Equality is not decidable over the functions implementable in typical functional languages. Therefore, an implementable higher-order functional language will either relinquish referential transparency through the use of an efficient but unreliable equality test (Abelson and Sussman, 1985) or let the language refuse to compare higher-order objects for equality (Milner, 1984). Pure higher-order functional programming must take the latter choice. Even this approach is not as readily available in logic programming, where equality (or rather a generalisation of equality, i.e. unification) is the basis for propagation of procedure arguments. For these reasons, we chose functional programming as the foundation.

Using the primitives of domain theory, we defined and implemented a lazy higher-order functional programming language. We prefer domain theory to purely syntactic formal theories because it combines the computational aspects of function application (β-reduction of λ-expressions) with the mathematical definition of function as a mapping from one domain into another, understood intuitively as a set of input/output pairs. One advantage of this approach is the ability to reason about recursively-defined objects, e.g. functions and infinite lists, via fixed point induction.

After enriching the semantic domain with the lower powerdomain (whose elements represent subsets of the enriched domain), we added a relative set abstraction construct (based on Zermelo–Frankel set notation (Hudak, 1989)) to denote powerdomain elements. This feature not only maintains the language's simplicity, regularity and orthogonality, but with it one can define the Herbrand universe of a first-order logic program, and model Horn logic relations as subsets of the domain of lists, whose elements are drawn from the Herbrand universe. When construction of the Herbrand universe is built into the implementation, computation of set abstractions generated by the Herbrand universe may be optimised, treating the enumeration parameters as logical variables, rather than using the more general method of 'generate and instantiate' (the default technique used with other set abstraction generators). These optimisations provide the efficiency of unification-based evaluation ((Lloyd, 1987), and together with the simplification rules for (weak) equality, implements a kind of lazy unification. In our language, the logical variable is *not* an explicit construct in the declarative reading of a program, but results instead from an implementation optimisation.

The optimised-evaluation procedure provides a kind of *lazy narrowing* (Reddy, 1985). This degree of narrowing is more efficient than narrowing in general term-rewriting systems (Goguen and Meseguer, 1984; Dershowitz and Plaisted, 1985), because our narrowing is performed in the semantic domain, rather than in the syntactic program domain. Narrowing only need be performed with respect to a handful of predefined semantic primitives, rather than with respect to an arbitrary set of user-defined functors.

The use of set abstraction in functional programming for logic programming capability was pioneered by Darlington (Darlington *et al.* 1986; Darlington and Guo, 1989) and Robinson (Robinson and Sibert, 1982). Darlington's *absolute set abstraction* differs from our relative set abstraction in that the first-order universe of terms was the *only* permitted set abstraction generator. This decision had two effects: (i) In Darlington's language, set abstraction was a top-level construct only; there was no natural way to construct new sets from simpler sets, or to treat sets as first-class objects; (ii) The enumeration of set abstraction parameters by the use of the first-order universe of terms is understood and unstated; hence the syntactic differences between relative and first-order absolute set abstraction. A computable generalisation to higher-order absolute set abstraction has not been discovered. Robinson's set abstraction in LogLisp (Robinson and Sibert, 1982) was not limited to the top-level; however, his language did not support lazy evaluation.

The syntax of relative set abstraction was proposed by Darlington and implemented in Turner's Miranda (Turner, 1985). If PowerFuL's optimisations were omitted, and if a binary set union's two subsets are evaluated sequentially rather than in parallel, then computation of our set abstraction would be similar to Turner's, except that Turner uses `cons` instead of set union (and the result may be used as may any other list). Though the user might *intend* that the resulting *list comprehension* (Peyton Jones, 1987) represent a set, the declarative semantics show no loyalty to this understanding. Turner's syntactic 'set union' is neither commutative, associative, nor idempotent. PowerFuL maintains these properties by limiting use of set abstraction to those operations which may be modelled by the lower powerdomain. In PowerFuL, set union is lazy. For completeness, computation requires a degree of parallel evaluation, though not so much as the full parallel-outermost rule. In practice, this is likely to be implemented via backtracking. Sequential reduction suffices outside of set abstraction, thus preserving the efficiency of purely functional computation.

The reader may feel disappointed that PowerFuL does not implement higher-order unification. As stated earlier, higher-order equality (and therefore its generalisation, higher-order unification) is undecidable, and therefore has no place in a functional language which maintains referential transparency. Pure first-order Horn logic programming lacks higher-order unification as well (by default, since it supports no higher-order objects at all). We consider it reasonable to omit a feature that is absent from both base paradigms.

5.2 Further work

There are several areas of additional research:

(1) Many modern functional languages are enriched with polymorphic type systems. In adding a type system to PowerFuL, one would have to consider its effect on the optimisations for computing with logical variables, since many constructors would be available. Our feeling is that it might make the interpreter more complex, but ought not to hurt efficiency. A very interesting theoretical task would be the generalisation of the denotational framework to deal with polymorphism. Another interesting theoretical task would be investigation of analogs to *negation by failure* in Horn Logic (Lloyd, 1987). This would probably require use of the Egli–Milner powerdomain (Main, 1987) in place of the lower powerdomain.

(2) Practical programming would inevitably lead to the need for features such as negation-as-failure (Clocksin and Mellish, 1981), which has been found to be useful in languages like Prolog. (Prolog, in fact, does not implement negation-as-failure correctly, due to its omission of the groundness check for negated goals). While our own interest is that the purely declarative subset of the language be as powerful and reliable as possible, these extensions are legitimate topics for investigation.

(3) To make PowerFuL a practical programming language, we also must consider the incorporation of arithmetic. The addition should cause less violence to PowerFuL's basic paradigm as compared with arithmetic in Prolog (Prolog is sometimes unable to solve for logical variables when important system predicates are used (Clocksin and Mellish, 1981), since in PowerFuL, inability to compute with logical variables over some types (e.g. functions) does not prevent the use of those types as first-class objects. We would also like to adapt the results from research in *constraint logic programming* (Jaffar and Lassez, 1987; Hickey, 1989).

(4) This work strives to unify the declarative aspects of functional and logic programming. It would be interesting to consider how to unify the efficient implementation strategies developed for these kinds of languages (Warren, 1983; Peyton Jones, 1987). Another issue is that the computation of infinite sets requires closer interaction between user and interpreter. This places additional demands on the language environment, so new programming environments may also need to be developed.

(5) Some aspects of the operational semantics, in particular the assumed completeness of safe computation rules (see section 4.1), and the correctness of the optimisations for logic programming (see section 4.2) have been given only intuitive justification. Provision of rigorous proofs of these ideas is one of our priorities.

Acknowledgements

In its course of this research, we have received insightful and constructive comments from many people interested in our work. Among them have been David Schmidt, David Plaisted, John Darlington, Dean Brock, Jan Prins, Don Stanat, Yi-ke Guo, and Boum Belkhouche. Special thanks go to Sylvia Takahashi whose current work on the implementation of PowerFuL has resulted in the discovery and correction of errors and omissions in the denotational description.

References

Abelson, H. and Sussman, G. 1985. *Structure and Interpretation of Computer Programs.* MIT Press.

Ashcroft, E. A. and Wadge, W. W. 1982. Prescription for semantics. *Trans. On Programming Languages and Systems*, **4** (2): April, 283–294.

Bellia, M. and Levi, G. 1986. The relation between logic and functional languages: a survey. *J. Logic Prog.* **3**: 217–236.

Berkling, K., Robinson, J. A. and Siebert, E. E. G. 1982. *A Proposal for a Fifth Generation Logic and Functional Programming System, Based on Highly Parallel Reduction Machine Architecture.* Syracuse Univ., November.

Clocksin, W. F. and Mellish, C. S. 1981. *Programming in Prolog.* Springer-Verlag.

Darlington, J., Field, A. J. and H. Pull. 1986. Unification of functional and logic languages. In DeGroot, D. and Lindstrom, G. eds., *Logic Programming, Relations, Functions and Equations*, Prentice-Hall, 37–70.

Darlington, J. and Guo, Y. 1989. Narrowing and unification in functional programming – an evaluation mechanism for absolute set abstraction. In *3rd International Conf. Rewriting Techniques and Applications*, Chapel Hill, NC: 92–108.

DeGroot, D. and Lindstrom, G. 1986. *Logic Programming: Functions, Equations, and Relations.* Prentice-Hall.

Dershowitz, N. and Plaisted, D. A. 1985. Applicative programming *cum* logic programming. In *Symposium on Logic Programming*, Boston, MA; 54–66.

Goguen, J. 1988. *Higher Order Functions Considered Unnecessary for Higher Order Programming.* SRI Project No. 1243, SRI International.

Goguen, J. A. and Meseguer, J. 1984. Equality, types, modules, and (why not?) generics for logic programming. *J. Logic Prog.* **2**: 179–210.

Gunter, C. A. and Scott, D. S. 1990. Semantic domains. In van Leeuwen, J. ed., *Handbook of Theoretical Computer Science*, Elsevier: 634–652.

Hickey, T. 1989. CLP and structure abstraction. In *16th ACM POPL*, Austin, TX: 124–133.

Hudak, P., Wadler, P., *et al.* 1988. *Report on the Functional Programming Language Haskell.* Yale Research Report DCS/RR-666, December.

Hudak, P. 1989. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, **21** (3), September: 359–411.

Hughes, J. 1990. Why functional programming matters. In Turner, D. A., ed., *Research Topics in Functional Programming*, Addison-Wesley.

Jaffar, J. and Lassez, J.-L. 1987. Constraint logic programming. In *14th ACM POPL*, Munich: 111–119.

Jayaraman, B. and Silbermann, F. S. K. 1986. Equations, sets, and reduction semantics for functional and logic programming. In *1986 ACM Conf. on LISP and Functional Programming*, Boston, MA: 320–331.

Khabaza, T. 1984. Negation as failure and parallelism. In *International Symposium on Logic Programming*, Atlantic City, NJ: 70–75.

Lindstrom, G. 1985. Functional programming and the logical variable. In *12th ACM Symposium on Principles of Programming Languages*, New Orleans, LA: 266–280.

Lloyd, J. 1987. *Foundations of Logic Programming* (*2nd edition*), Springer-Verlag.

McCarthy, J., *et al.* 1965. *LISP 1.5 Programmer's Manual*, MIT Press.

Main, M. G. 1987. A powerdomain primer. *Bull. Euro. Assoc. for Theoretical Computer Sci.*, **33**; October.

Manna, Z. 1974. *Mathematical Theory of Computation*, McGraw-Hill.

Milner, R. 1984. A proposal for standard ML. In *ACM Symposium on LISP and Functional Programming*, Austin, TX: 184–197.

Naish, L. 1985. *Negation and Control in Prolog.* Doctoral Dissertation, University of Melbourne.

Pleban, U. F. 1984. Compiler prototyping using formal semantics. In *Proc. ACM SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices*, **19** (6), June.

Peyton Jones, S. L. 1987. *The Implementation of Functional Programming Languages*, Prentice-Hall.

Reddy, U. S. 1985. Narrowing as the operational semantics of functional languages. In *1985 Symposium on Logic Programming*, Boston, MA: 138–151.

Robinson, J. A. and Sibert, E. 1982. LOGLISP: An alternative to PROLOG. *Machine Intelligence*, **10**: 299–314.

Robinson, J. A. 1984. *New Generation Knowledge Processing: Syracuse University Parallel Expression Reduction.* First Annual Progress Report, December.

Robinson, J. A. 1986. The future of logic programming. In *Symposium on Logic in Computer Science*, Ireland.

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

Scott, D. S. 1982. *Domains for denotational semantics.* Volume 140 of *Lecture Notes in Computer Science*, Springer.

Silbermann, F. S. K. and Jayaraman, B. 1989. Set abstraction in functional and logic programming. In *Fourth International Conf. on Functional Programming and Computer Architecture*, London, UK.

Smolka, G. and Panangadan, P. 1985. *A Higher-order Language with Unification and Multiple Results*, Tech. Report TR 85-685, Cornell University, May.

Stoy, J. E. 1977. *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*, MIT Press.

Turner, D. A. 1985. Miranda: A non-strict functional language with polymorphic types. In *Conf. on Functional Prog. Langs. and Comp. Arch.*, Nancy, France: 1–16.

van Emden, M. H. and Kowalski, R. A. 1976. The semantics of predicate logic as a programming language. *J. ACM*, 23 (4), 733–743.

Vuillemin, J. 1974. Correct and optimal implementations of recursion in a simple programming language. *J. Computer and System Sci.*, **9**: 332–354.

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

Warren, D. H. D. 1982. Higher-order extensions of Prolog: are they needed? *Machine Intelligence*, 10: 441–454.

Warren, D. H. D. 1983. *An Abstract Instruction Set for Prolog*, Tech. Note 309, SRI International.

You, J-H. and Subrahmanyam, P. A. 1986. Equational logic programming: an extension to equational programming. In *13th ACM Symposium on Principles of Programming Languages*, St. Petersburg, FL: 209–218.