Fresh: A Higher-Order Language with Unification and Multiple Results

Gert Smolka
Prakash Panangaden

TR 85-685
May 1985

Department of Computer Science
Cornell University
Ithaca, New York 14853

Fresh: A Higher-Order Language with Unification and Multiple Results

Gert Smolka and Prakash Panangaden

Cornell University

May 1985

Abstract

This paper presents Fresh, a language that integrates logic programming features into higher-order functional programming. The language incorporates unification, multiple results and a collection construct. Many examples illustrate that these extensions of functional programming are useful. We define an operational semantics along the lines of Plotkin's structural approach. The semantics is of intrinsic interest since it covers backtracking and the collection construct. To illustrate the conceptual similarities and differences between functional and logic programming, we begin with a purely functional core language and add first unification and then backtracking. With each addition we discuss the enhanced eloquence of the language and the concomitant modifications to the semantics.

1 Introduction

This paper presents Fresh, a language that integrates logic programming features into higher-order functional programming. This language represents an attempt to explore the connections between functional and logic programming, the two major applicative programming styles that have developed over the last twenty years. Our presentation of Fresh will focus on two themes: Fresh as a practical programming language and Fresh as a laboratory for exploring the semantic ideas that are needed to describe the combination of reduction, unification and backtracking.

The starting point in our discussion is FM, a higher-order functional language tailored for the extensions to come. FM departs from other functional languages by employing a failure-based evaluation strategy and by incorporating unrestricted pattern matching. In FM, pattern matching serves as the sole computational mechanism for binding variables and for decomposing and comparing objects. The first step from FM to Fresh is the generalization of matching to unification. Variables can now be bound to nonground terms, and unification may refine such incomplete values by binding the variables contained in them. These delayed bindings make it possible to build objects incrementally by imposing constraints as information becomes available. The second extension of FM is the incorporation of multiple results and backtracking. The list of all results of an expression can be computed with a collection construct. The combination of backtracking and collection provides an attractive alternative to recursion. These features give Fresh expressive data base capabilities.

We define an operational semantics for Fresh following Plotkin's structural approach [26]. The semantics is of intrinsic interest because it covers backtracking and the collection construct. The meaning of each construct is defined by a set of proof rules, which justify instances of the reduction relation. A proof for a reduction is a tree whose structure follows closely the structure of the expression being reduced. The result of an expression is computed by constructing the proof tree that justifies the reduction to the result. The semantics is compositional, a property frequently cited for preferring a denotational definition.

Fresh subsumes Horn logic with equality under a depth-first search strategy. Thus a first-order subset of Fresh could be characterized in model theoretic terms. However, full Fresh does not have model-theoretic semantics, since it contains negation as failure [3, 20], collection and higher-order functions. We feel that these features are crucial for a practical programming language. The major difficulties with the logic programming approach are negation and the incorporation of control. So far, logic programming languages do not include a *general form* of negation, and we are not aware of any research in progress that may change this situation. Furthermore, the integration of control constructs necessarily leads to a language that cannot be described in model theoretic terms. Eqlog [9], a recent logic language based on Horn logic with equality, has a clean model theoretic semantics, but includes neither negation nor control constructs. Prolog [3] is no more a logic programming language than Fresh is. In fact, Fresh is an attempt to overcome Prolog's ad hoc features like the cut or the call predicate and to integrate Prolog's useful computational innovations smoothly into a functional framework.

The presentation begins with an informal description of FM, the functional core language of Fresh. To make the comparison between functional programming and Fresh more perspicuous, we then give a formal account of matching and FM's semantics. This also provides a familiar and simple base to introduce our style of semantics. The next two sections discuss the incorporation of unification and backtracking into FM. Many examples illustrate how Fresh relates to Prolog and why the integration of these features is useful. These informal discussions are followed by two technical sections, which give a rigorous account of unification and Fresh's semantics. Finally, we discuss related research and future directions.

2 A Functional Language Based on Matching

In this section we shall discuss a functional language, called FM, based on matching. This language will serve as the starting point in our discussion of the relation between functional and logic programming. The point in presenting FM is to establish a familiar base that can be naturally extended to subsume logic programming. FM contains nearly all constructs of the full language, Fresh. Fresh is obtained by generalizing matching to unification and adding constructs that introduce and eliminate multiple results.

Modern functional languages, like KRC [33], HOPE [2] and Standard ML [22], offer matching constructs as a means to decompose data objects. However, these languages employ matching in a rather restricted form; for instance, patterns cannot contain the same variable twice. Consequently, this stripped down form of matching does not provide for the comparison of data objects. Since decomposition and comparison often interact, these restrictions deprive the programmer of a significant part of matching's expressive power. There are two reasons for these restrictions. First, since general matching is an operation that can fail, a smooth integration into a language whose evaluation strategy does not account for failure is impossible. Second, general matching requires equality on all objects, a property that is not naturally satisfied by higher-order objects.

Since we are aiming at a language that incorporates unification, which is a generalisation of matching, FM should overcome these problems. Equality for all objects is obtained by hiding functions behind names, called designators. Abstractions now become reducible expressions. They reduce to a new designator that is bound to the function defined by the abstraction. Although this change is for the most part invisible, it provides the unusual feature that function names can be compared with other objects. The failure problem is addressed by incorporating failure as a general concept. Consequently, the reduction of an expression either *succeeds* with a result or *fails* without a result. This binary character of reduction renders traditional boolean expressions obsolete; the conditional now tests for success or failure. It turns out that the proper integration of matching and failure not only prepares the ground for the addition of unification and multiple results, but also increases the eloquence and compactness of the functional language. Matching now becomes the basic computational construct that binds variables and compares and decomposes data objects. Patterns in FM can contain the same variable more than once and can also contain variables that are bound in an outer context.

Matching and unification are operations that solve equations between terms. Matching is a restricted form of unification and solves equations where one side is a ground term. Consequently, all data objects in FM are ground terms, while patterns are terms that can contain variables.

2.1 Terms and Matching

Terms are tree-structured objects built from atoms, designators and variables. *Atoms* are primitive objects that include identifiers starting with a lower case letter (e.g., maria, apple, x or y), numbers (e.g., 612), or special symbols (e.g., (), %, + or −). *Designators* serve as function names and are written as bold-face identifiers or symbols, for instance, **append** or **+**. Functional objects are hidden behind designators since matching and unification are first-order operations, which require equality for terms. *Variables* are identifiers starting with a capital letter, for instance, X, Y or Person. A *term* is either an atom, a designator, a variable, or a *pair* (s, t) that consists of two terms s and t. A term is *ground* if it contains no variables.

Matching and unification solve equations between terms with respect to variables and syntactic equality. Matching is a restricted form of unification that solves equations where one side is a ground term. For instance, the equation

(X, (Y, (red, X))) ≡ ((apple, green), (fun, (red, (apple, green))))

has the solution X ≡ (apple, green) and Y ≡ fun. The equation

(X, (Y, (red, X))) ≡ ((apple, green), (fun, (red, (apple, yellow))))

has no solution. Thus an attempt to solve it will *fail*. The left-hand side of a matching equation is called the *pattern* and the right-hand side is called the *argument* of the equation. We say that the pattern is *matched* against the argument. If the match *succeeds* it produces a set of *variable bindings*, which is the unique solution of the equation. If we replace all variables in the pattern by their *values*, that is, by the ground terms the variables are bound to in the solution, we obtain a term that is equal to the argument.

To obtain a palatable syntax for terms, the comma is treated as a right-associative infix operator. Thus (1, 2, 3, 4) is a syntactical variant of (1, (2, (3, 4))). Furthermore, redundant parentheses are possible; for instance, (((1))) is a syntactical variant of the atom 1. These syntactical refinements do not affect matching or other semantic operations, but are just a notational convenience that is invisible at the semantic level.

We call the terms defined above *binary terms* to distinguish them from other terms. Binary terms are simpler than Prolog's first-order terms. However, first-order terms can be expressed as binary terms and their usual syntax can be recovered by a simple abbreviation rule. A *labeled term* is a term (%, *s*, *t*) where % is the atom % and *s* is either an atom or a variable. Syntactically, such a term can be written as *s*(*t*). Thus

plus(times(X, Y), F(Z, 6))

is a syntactical variant of the term

(%, plus, (%, times, X, Y), %, F, Z, 6) ,

which, when deprived of all syntactical sugar, becomes

(%, (plus, ((%, (times, (X, Y))) , (%, (F, (Z, 6)))))) .

An input processor for FM would translate the first variant into the third variant, which is the unique semantic form. An output processor would retranslate into the refined form. Semantically, labeled terms are just a subset of all terms and do not feature any special properties. The reader may check that matching first-order terms in binary representation is equivalent to matching first-order terms directly, provided that every function symbol is always used with the same number of arguments. This holds as well for unification. One advantage of the binary representation is that we can now solve equations like

F(A) ≡ plus(4, 7).

where the variable F matches the function symbol and A matches the argument tuple of the coded first-order term plus(4, 7); the solution is F ≡ plus and A ≡ (4, 7).

Another important subclass of terms are lists. A *list* is either the atom () (the *empty list*) or a term (*s*, *l*) where *l* is a list. For instance, (1, 2, 3, ()) is a list with three elements.

Although binary terms are similar to Lisp's S-expressions, we shall now see that their integration into FM is quite different.

2.2 Expressions and Reduction

Figure 1 defines FM's abstract syntax. Note that, for instance, **A** denotes the set of all atoms and that a, b, c are metavariables that range over atoms. Whenever the reader encounters in the following the symbols a, b and c it is understood that they denote an atom. The same holds for the metavariables for designators, variables, terms and expressions. Note that every term is an expression.

The execution of expressions is called *reduction*. The reduction of an expression either *succeeds* or *fails*; if it succeeds it produces a *result*, which is a ground term. A successful reduction also produces a function environment, which

Atoms: $a, b, c \in \mathbf{A}$

Designators: $d \in \mathbf{D}$

Variables: $x, y, z \in \mathbf{V}$

Terms: $s, t, u, v \in \mathbf{T}$

$$s ::= a \mid d \mid x \mid (s, t)$$

Expressions: $e, f, g, h \in \mathbf{E}$

$e ::=$	a	*atom*
	d	*designator*
	x	*variable*
	(e, f)	*pair*
	$s \equiv e \rightarrow f; g$	*conditional let-clause*
	$s \blacktriangleright e$	*abstraction*
	$f[e]$	*application*

Figure 1 FM's abstract syntax.

binds the designators in the result. Besides the two regular outcomes of reduction, the reduction of certain expressions may not terminate. In contrast to other functional languages, run-time errors cannot occur in FM. Situations that do not allow a successful continuation of reduction produce a failure. For instance, the application of something that is not a function to an argument will simply fail. Of course, a type discipline for FM would statically enforce that such conditions cannot occur.

An expression is *canonical* if it reduces to itself. The canonical expressions of FM are exactly the ground terms. Atoms and designators reduce to themselves. A pair (e, f) reduces to (u, v) if e reduces to u and f reduces to v.

Reduction takes place in the presence of two environments. The *term environment* binds variables to ground terms and is augmented by solving matching equations. The *function environment* binds designators to functions and is augmented by the reduction of abstractions, which creates new functions. Section 4 gives statically verifiable conditions that ensure that all variables and designators are bound when they are reduced.

A variable reduces to its *value*, that is, the ground term to which it is bound in the term environment. Thus a term always reduces to a ground term, which is obtained by replacing all variables by their values.

As mentioned before, a pair (e, f) reduces to (u, v) if e reduces to u and f reduces to v. If one of the components fails the reduction of the entire pair fails.

A *conditional let-clause* has form $s \equiv e \rightarrow f; g$ (read "if s matches e then f else g") where s is the *pattern*, $s \equiv e$ is the *condition part*, f is the *then-part* and g is the *else-part.* Note that the metavariables indicate that e, f and g can be arbitrary expressions and s can be any term. The reduction of a conditional let-clause begins with the reduction of e. If e *succeeds* with the result u, the solution of the equation $s \equiv u$ under the current term environment is attempted. If the equation is solvable its solution yields an augmented term environment, under which the then-part is reduced. If the then-part fails, the conditional let-clause fails; otherwise its result is the result of the then-part. The else-part is only reduced if the reduction of the condition part fails; that is, either e fails or the match between the pattern and the result of e fails. If the else-part fails the conditional let-clause fails; otherwise, the result of the else-part is the result of the conditional let-clause.

When we discuss functions we will define an expression fail[] that always fails. This expression is helpful for defining other useful extensions. A simple let-clause can now be defined as

$$s \equiv e \rightarrow f \qquad \text{is} \qquad s \equiv e \rightarrow f;\ \mathrm{fail[]}\ .$$

With that we can express Lisp's car and cdr operation, which yield the head and the tail of a list:

$$(\mathrm{H, T}) \equiv \mathrm{L} \rightarrow \mathrm{H} \qquad \textit{car operation}$$

$$(\mathrm{H, T}) \equiv \mathrm{L} \rightarrow \mathrm{T} \qquad \textit{cdr operation}$$

The variable L is supposed to be bound to a list. If L is bound to the empty list the expression fails. This is in contrast to Lisp, where the application of car or cdr to the empty list results in a run-time error. Note that, in FM, a cons operation (cons $e\ l$) just becomes the pair (e, l).

Another useful syntactic extension is the *wildcard symbol* _ that stands for a variable that occurs only once. With that a *conditional* can be defined as

$$e \rightarrow f;\ g \qquad \text{is} \qquad \_ \equiv e \rightarrow f;\ g\ .$$

A match with the wildcard symbol always succeeds. The presence of failure makes the conditional independent of distinguished boolean values. If the condition succeeds the then-part determines the result; if the condition fails the else-part is reduced. The boolean connectives are now easily defined. Conjunction becomes pairing, for instance, $(e, f) \rightarrow g;\ h$. Conditional disjunction (also called alternation) becomes

$$e\ ;f \qquad \text{is} \qquad x \equiv e \rightarrow x;\ f$$

where x is a new variable. Negation can be defined as

$$\neg e \qquad \text{is} \qquad e \rightarrow \mathrm{fail[];\ true}$$

where true is the atom true. Equality could be defined as

$$e\ =\ f \qquad \text{is} \qquad x \equiv e \rightarrow (x \equiv f \rightarrow \mathrm{true})$$

where x is a new variable. Note that the inner pattern is a variable that is bound by the outer pattern.

Let us study reduction in more detail. For now we are not concerned with functions, so reduction operates on configurations $\rho\{e\}$ where ρ is the term environment and e is the expression to be reduced. We start with the configuration

$$\{\, \mathsf{X} \equiv 1 \;\rightarrow\; \mathsf{L} \equiv (1, 1, 2, ()) \;\rightarrow\; (\mathsf{X}, \mathsf{X}, \mathsf{R}) \equiv \mathsf{L} \;\rightarrow\; \mathsf{R} \,\}$$

whose term environment is empty. The conditional arrow $\rightarrow$ binds right-associative; thus the expression above is parsed as

$$\mathsf{X} \equiv 1 \;\rightarrow\; (\, \mathsf{L} \equiv (1, 1, 2, ()) \;\rightarrow\; (\, (\mathsf{X}, \mathsf{X}, \mathsf{R}) \equiv \mathsf{L} \;\rightarrow\; \mathsf{R}\,)\,)\,.$$

The innermost let-clause checks whether the first two elements of the list L are equal to X and, if so, returns the rest of the list. Thus the entire expression reduces to the list (2, ()).

Formally, the expression is reduced by first reducing the condition part $\mathsf{X} \equiv 1$ of the outer let-clause. Since 1 reduces to 1, the equation yields a new variable binding. Thus reduction continues with the configuration

$$\mathsf{X} \equiv 1 \;\{\, \mathsf{L} \equiv (1, 1, 2, ()) \;\rightarrow\; (\mathsf{X}, \mathsf{X}, \mathsf{R}) \equiv \mathsf{L} \;\rightarrow\; \mathsf{R} \,\}\,.$$

By proceeding as before ((1, 1, 2, ()) is a ground term and reduces to itself), we obtain the configuration

$$\mathsf{X} \equiv 1 \quad \mathsf{L} \equiv (1, 1, 2, ()) \;\{\, (\mathsf{X}, \mathsf{X}, \mathsf{R}) \equiv \mathsf{L} \;\rightarrow\; \mathsf{R} \,\}\,.$$

This time the right-hand side of the condition part is the variable L, which reduces to its value (1, 1, 2, ()). Thus the next step is to solve the *matching problem*

$$\mathsf{X} \equiv 1 \quad \mathsf{L} \equiv (1, 1, 2, ()) \;\{\, (\mathsf{X}, \mathsf{X}, \mathsf{R}) \equiv (1, 1, 2, ()) \,\}\,,$$

which consists of the environment and an equation. First, the equation is split into three equations, yielding

$$\mathsf{X} \equiv 1 \quad \mathsf{L} \equiv (1, 1, 2, ()) \;\{\, \mathsf{X} \equiv 1 \quad \mathsf{X} \equiv 1 \quad \mathsf{R} \equiv (2, ()) \,\}\,.$$

Now the left-hand side of the first equation $\mathsf{X} \equiv 1$ is replaced by its value under the environment, yielding

$$\mathsf{X} \equiv 1 \quad \mathsf{L} \equiv (1, 1, 2, ()) \;\{\, 1 \equiv 1 \quad \mathsf{X} \equiv 1 \quad \mathsf{R} \equiv (2, ()) \,\}\,.$$

Since $1 \equiv 1$ is a tautology it can be discarded. Analogously, the second equation is discarded, yielding

$$X \equiv 1 \quad L \equiv (1, 1, 2, ()) \quad \{ R \equiv (2, ()) \} .$$

Since the left-hand side of the remaining equation is a variable that is not bound in the environment, the equation defines a new binding that is included into the environment, yielding the augmented environment

$$X \equiv 1 \quad L \equiv (1, 1, 2, ()) \quad R \equiv (2, ())$$

as the solution of the matching problem. Thus the reduction of the innermost let-clause continues with the then-part, that is, with the configuration

$$X \equiv 1 \quad L \equiv (1, 1, 2, ()) \quad R \equiv (2, ()) \; \{ R \} ,$$

which reduces to the value of R. Thus the entire expression reduces to (2, ()).

We complete the informal account of FM's semantic with the description of abstraction and application, which create and apply functions. As in other functional languages, functions have always exactly one argument and their global variables are bound statically, that is, when the function is created rather than when the function is applied. One argument is sufficient since pairs provide implicitly for multiple arguments.

An *abstraction* has form $s \blacktriangleright e$ where the *pattern* s can be any term and the *body* e can be any expression. As a special case we have the form $x \blacktriangleright e$ that is similar to $\lambda x.e$ in the lambda calculus. In contrast to the lambda calculus, however, abstractions are not canonical in FM. Furthermore, variables in the pattern of an abstraction can be bound in the outer context, as, for instance, in X▸(X▸body). An abstraction reduces to a new designator, which is then bound to the closure of the abstraction under the current term environment. The closure represents the created function and consists of the abstraction together with all bindings that exist so far for variables that occur in the abstraction. For instance, the closure of the abstraction (X,Y) ▸Y will contain the binding $X \equiv 1$ if the current term environment binds X to 1 and does not bind Y. This closure is equivalent to a closure that contains the abstraction (1,Y) ▸Y and no bindings.

An *application* has form $f\,[e]$ where f and e are expressions. It is reduced by first reducing the pair (f, e). If the pair does not reduce to a pair (d, u) where d is a designator, the application fails. Otherwise, let the designator be bound to the closure $(\rho, s \blacktriangleright g)$. Then the matching equation $s \equiv u$ between the abstraction pattern and the reduced argument is solved under the closure environment ρ. If the equation is unsolvable the application fails. Otherwise, the abstraction body g is reduced under the augmented closure environment and yields the result of the application. As an example, consider the reduction of the application $((X,Y) \blacktriangleright Y)[L]$ under the environment $X \equiv 1$ and $L \equiv (1, 2, ())$. Then the pair $((X,Y \blacktriangleright Y), L)$ reduces to $(d, (1, 2, ()))$ where d is a new designator that is bound to a closure with the environment $X \equiv 1$ and the

$s(e)$	is $(\%, s, e)$	*labeled term* $(s \in \mathbf{A} \cup \mathbf{V})$
_	is a new variable	*wildcard*
fail[]	is $(1 \blacktriangleright 1)[2]$	*expression that always fails*
$e \to f; g$	is $\_ \equiv e \to f; g$	*conditional*
$\neg e$	is $e \to$ fail[]; true	*negation*
$e \to f$	is $e \to f$; fail[]	*if-clause*
$s \equiv e \to f$	is $s \equiv e \to f$; fail[]	*let-clause*
$f \leftarrow e$	is $e \to f$; fail[]	*where-clause*
$f \leftarrow s \equiv e$	is $s \equiv e \to f$; fail[]	*where-clause with matching*
$e\, ; f$	is $x \equiv e \to x; f$	*alternation* $(x$ *is new*$)$
$s_1 \blacktriangleright e_1;;$ $s_2 \blacktriangleright e_2;;$ $\vdots$ $s_n \blacktriangleright e_n$	is $x \blacktriangleright (s_1 \equiv x \to e_1\, ;$ $(s_2 \equiv x \to e_2\, ;$ $\vdots$ $(s_n \equiv x \to e_n\, ;$ fail[]) ...))	*clausal definition* $(x$ *is new*$)$

Operator Precedence: $;; \;\|\; \blacktriangleright \;\|\; \to \;\; ; \;\; \leftarrow \;\|\; , \;\|\; \neg \;\|\; \equiv \;\|\; f[e] \;\; s(e)$

right-associative: $;; \;\; \blacktriangleright \;\; \to \;\; ; \;\; \neg \;\; \equiv$

left-associative: $f[e] \;\; \leftarrow$

Figure 2 Syntactic extensions.

abstraction $(X,Y) \blacktriangleright Y$. Since the matching problem $X \equiv 1 \{ (X,Y) \equiv (1, 2, ()) \}$ reduces to the environment $X \equiv 1$ and $Y \equiv (2, ())$, the application $((X,Y) \blacktriangleright Y)[L]$ reduces to $(2, ())$.

The expression that always fails can be defined as

fail[] is $(1 \blacktriangleright 1)[2]$.

Note that without functions failure is not possible since the else-part of the conditional let-clause catches a possible failure of its condition part.

Figure 2 summerizes the syntactic extensions used in this paper. Since FM is a sublanguage of Fresh, all syntax and semantic discussed so far is valid for Fresh as well. As it will become clear in the following examples, clausal definition (see figure 2) is the intended style for function definitions. The conditional let-clause rather than a regular conditional was included as the base construct so that the clausal definition construct can be defined. The reader may convince himself that conditional, abstraction and application are not strong enough to express the clausal definition construct.

As we know from the lambda calculus, abstraction and application alone suffice for the definition of recursive functions. However, to have a convenient facility for the definition of recursive functions, FM supports top-level function declarations. A *function declaration* has the form

d of $s \blacktriangleright e$. (read "d of s is e")

where d is a designator and $s \blacktriangleright e$ is an abstraction. A *program* is a sequence of function definitions followed by an expression that has to be reduced under this function environment.

2.3 Examples

We begin with some examples of functions acting on lists. Recall that a list is either the atom () (the empty list) or a pair whose right component is a list. The types that appear in the function declarations are comments and have no semantic significance. List concatenation can be defined as

```
append: list(T) × list(T) → list(T)  of
        (), L ▸ L;;
    (H,T), L ▸ H, append[T,L].
```

The comma binds tighter than the abstraction operator (see figure 2); thus the second clause of append is parsed as ((H,T), L) ▸ (H, append[T,L]) . Technically, append has like any other function just one argument. The type indicates that the argument must be a pair of two lists. A membership function is

```
member: T × list(T) → {true}  of
    X, (X, T) ▸ true;;
    X, (H, T) ▸ member[X, T].
```

This function takes a term and a list and returns true if the term is in the list and fails otherwise. Note how the test for membership is essentially being carried out by matching with a pattern containing two occurrences of X; the first clause of member can only succeed if the first element of the list is equal to the "first argument" of member. A typical higher-order function is

```
map: (S → T) → list(S) → list(T)  of
    F ▸ (    () ▸ ();;
          H, T ▸ F[H], map[F][T] ).
```

which maps a list by applying a function to each element. Note that map fails if its argument function fails for one element of the list. Thus map can also be used to test that all elements of a list have a certain property. The function

```
islist: term → {true}  of
        () ▸ true;;
      H,T ▸ islist[T].
```

succeeds with the atom true if its argument is a list. The function

```
succeeds: term → {true}  of
        X ▸ true.
```

succeeds for every argument with true. The function

```
comp: (R → S) × (S → T)  →  (R → T)  of
        F, G ▸ X ▸ F[G[X]].
```

builds the composition of two functions. Thus the expression comp[succeeds, map[islist]] reduces to a function that succeeds with the atom true if and only if its argument is a list of lists. The next function

```
powerlist: list(T) → list(list(T)) of
        () ▸ (), ();;
        H,T ▸ append[ P, map[ L▸H,L ] [P] ]  ←  P ≡ powerlist[T].
```

constructs the list of all sublists of a list. We shall see a declarative version of this function when we introduce backtracking. The next function takes a function and a list and returns the result of applying the function to the first member of the list on which the function succeeds.

```
. get: (S → T) → list(S) → T  of
        F ▸  H, T ▸ F[H]; get[F][T].
```

If s is a term the expression get[s▸s] reduces to a function that, when applied to a list, yields the first element that matches the pattern s. For instance, get[(X,2) ▸ (X,2)] yields a function that retrieves the first element that is a pair whose right component is the atom 2. The function

```
getvalue:  list( identifier × T) × identifier → T of
        L, I ▸ get[ I,V▸V] [L].
```

gets a value from an association list. Note that the pattern of the inner abstraction contains the variable I, which is bound by the pattern of the outer abstraction. The last example is a function for the symbolic differentiation of arithmetic expressions:

```
d: unknown → arithmeticExpression → arithmeticExpression  of
        U ▸ (   X + Y  ▸ d[U][X] + d[U][Y] ;;
                X − Y  ▸ d[U][X] − d[U][Y] ;;
                X*Y    ▸ d[U][X] *Y + X*d[U][Y] ;;
                X / Y  ▸ ( d[U][X] *Y − X*d[U][Y] ) / Y*Y ;;
                  U    ▸ 1 ;;
                  C    ▸ 0 ).
```

Note that the pattern of the second last clause is the variable U, which is bound by the outer pattern to the unknown with respect to which the derivative is computed. The fancy syntax is obtained by declaring the atoms +, −, *, / as infix operators with the appropriate precedence. Then, for instance, X + Y is a syntactical variant of the labeled term + (X, Y), which in turn is an abbreviation for (%, (+ , (X, Y))).

As the examples above illustrate, the existence of designators makes no difference to the programming style. Designators were introduced since matching and unification are first-order operations based on syntactical equality. By hiding functions behind designators and treating designators like atoms, matching and unification can cover all objects. Consequently, FM offers the unusual feature that function names can be compared with any term. The suppression of this extra-capability at the computational level would lead to numerous dynamic error conditions. Moreover, the formulation of the algebraic properties of matching and unification would become rather awkward. On the other hand, a type discipline could easily ensure that designators are not compared with other objects.

The next two sections present a formal account of matching and FM's semantics. This prepares the ground for the formal semantics of Fresh and will illustrate the semantic differences between the functional and unification-based language. The reader may skip these formal sections and first read the informal description of the full language, which appears in sections 5 and 6.

3 Matching

This section gives a formal account of substitutions, equations and matching. Substitutions serve as term environments in the formal semantics of FM and Fresh. Equations arise during the reduction of conditional let-clauses and applications. Section 7 will continue this section with the discussion of unification.

Recall the definition of terms given in figure 1:

Atoms: $a, b, c \in \mathbf{A}$

Designators: $d \in \mathbf{D}$

Variables: $x, y, z \in \mathbf{V}$

Terms: $s, t, u, v \in \mathbf{T}$ $\qquad$ $s ::= a \mid d \mid x \mid (s, t)$

$\mathbf{V}(s)$ denotes the set of all variables that occur in the term s.

A *substitution* is a function θ: $\mathbf{T} \to \mathbf{T}$ such that $\theta a = a$ for all atoms a, $\theta d = d$ for all designators d, and $\theta(s, t) = (\theta s, \theta t)$ for all pairs (s, t), where θs denotes the application of θ to the term s. We write $\psi\theta$ for the composition of two substitutions θ and ψ. As always, application and composition are compatible, that is, $(\theta\psi)s = \theta(\psi s)$ holds for all substitutions θ and ψ and all terms s. It is easy to verify that two substitutions are equal if they are equal for all variables. The identity on $\mathbf{T}$ is called *empty substitution* and is denoted by ε.

An *equation* is an expression $s \equiv t$ whose *left-hand side s* and *right-hand side t* are terms. An *equation system* is a bag of equations. We use bags rather than sets to avoid that an implementation has to check for duplicates. A substitution θ is a *solution* for an equation system E if $\theta s = \theta t$ for all equations $s \equiv t$ in E.

The *domain* of a substitution θ is $\mathbf{D}(\theta) := \{ x \in \mathbf{V} \mid \theta x \neq x \}$. A substitution is *finite* if its domain is finite. A finite substitution θ can be *represented* by the finite equation system $\{ x \equiv \theta x \mid x \in \mathbf{D}(\theta) \}$. In the following we will freely switch between viewing a substitution as an equation system and viewing it as a mapping from terms to terms. Furthermore, we will only consider finite substitutions; so in the following, "substitution" always stands for finite substitution. A *ground substitution* is a substitution which maps all variables of its domain to ground terms. Note that a ground substitution is a solved equation system, that is, the solution of θ viewed as an equation system is θ.

A *matching problem* is a pair $\rho\{M\}$ where ρ is a ground substitution and M is a finite equation system whose right-hand sides are ground terms. A *solution* of a matching problem $\rho\{M\}$ is a substitution θ that solves M and ρ (that is, ρ viewed as an equation system) and satisfies $\mathbf{D}(\theta) = \mathbf{D}(\rho) \cup \mathbf{V}(M)$, where $\mathbf{V}(M)$ is the set of all variables that occur in the equations of M. It is easy to show that a matching problem has at most one solution and that a solution is a ground substitution. If θ solves both M and ρ then θ solves the instantiated equation system ρM. Note that the converse does not hold.

Matching is the process of computing the solution of a matching problem. An indeterministic algorithm for solving matching problems is best specified by *matching rules*, which reduce a matching problem $\rho\{M\}$ by growing the *solved part* ρ

and shrinking the *unsolved part* M. Note that ρ already solves ρ. The *failure symbol* $\Box$ is the normal form for unsolvable matching problems.

(Taut)	$\rho\{s \equiv s,\ M\} \ {}_{\mathrm{m}}\!\!\rightarrow\ \rho\{M\}$		iff s is an atom or a designator
(Subst)	$\rho\{x \equiv u,\ M\} \ {}_{\mathrm{m}}\!\!\rightarrow\ \rho\{\rho x \equiv u,\ M\}$		iff $x \in \mathrm{D}(\rho)$
(Bind)	$\rho\{x \equiv u,\ M\} \ {}_{\mathrm{m}}\!\!\rightarrow\ \{x \equiv u\}\rho\{M\}$		iff $x \notin \mathrm{D}(\rho)$
(Split)	$\rho\{(s, t) \equiv (u, v),\ M\} \ {}_{\mathrm{m}}\!\!\rightarrow\ \rho\{s \equiv u, t \equiv v,\ M\}$		
(Fail)	$\rho\{M\} \ {}_{\mathrm{m}}\!\!\rightarrow\ \Box$		iff none of the rules above applies and $M \neq \emptyset$

The expression $\rho\{s \equiv u,\ M\}$ denotes the matching problem $\rho\{M'\}$ where M' is obtained from M by adding the equation $s \equiv u$. Furthermore, $\{x \equiv u\}\rho$ is the composition of the substitutions $\{x \equiv u\}$ and ρ. The following proposition is easy to prove and states that the matching rules are sound.

Proposition 3.1. Let $\rho\{M\} \ {}_{\mathrm{m}}\!\!\rightarrow \rho'\{M'\}$. Then a substitution θ solves $\rho\{M\}$ if and only if θ solves $\rho'\{M'\}$.

The symbol ${}_{\mathrm{m}}\!\!\rightarrow^*$ denotes the reflexive and transitive closure of ${}_{\mathrm{m}}\!\!\rightarrow$ and $\rho\{M\} \ {}_{\mathrm{m}}\!\!\rightarrow^* \rho'$ is an abbreviation for $\rho\{M\} \ {}_{\mathrm{m}}\!\!\rightarrow^* \rho'\{\emptyset\}$. For the the next theorem we need some notations and a theorem from Huet [14]. Let $\rightarrow$ be a binary relation and let $\rightarrow^*$ be its reflexive and transitive closure. Then $\rightarrow$ is *locally confluent* if for all x, y, z such that $x \rightarrow y$ and $x \rightarrow z$ there exists a u such that $y \rightarrow^* u$ and $z \rightarrow^* u$. Furthermore, $\rightarrow$ is *confluent* if for all x, y, z such that $x \rightarrow^* y$ and $x \rightarrow^* z$ there exists a u such that $y \rightarrow^* u$ and $z \rightarrow^* u$. Finally, $\rightarrow$ is *noetherian* if there are no infinite sequences $x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \cdots$. Huet proves that a noetherian relation is confluent if it is locally confluent.

Theorem 3.2. The reduction ${}_{\mathrm{m}}\!\!\rightarrow$ is confluent and noetherian. Furthermore, $\rho\{M\} \ {}_{\mathrm{m}}\!\!\rightarrow^* \rho'$ if an only if ρ' is the solution of $\rho\{M\}$; and $\rho\{M\} \ {}_{\mathrm{m}}\!\!\rightarrow^* \Box$ if and only if $\rho\{M\}$ has no solution.

Proof. The matching reduction is noetherian since each rule decreases either the number of variable occurrences or the sum of the depths of the equations in the unsolved part of the matching problem. Since the reduction is noetherian it is confluent if it is locally confluent. The local confluence of ${}_{\mathrm{m}}\!\!\rightarrow$ is easy to verify. The remaining two claims follow by induction from proposition 3.1. $\Box$

Theorem 3.2 tells us that the matching rules in fact specify an algorithm for solving matching problems. Given a matching problem, such an algorithm just applies matching rules as long as they are applicable. Since ${}_{m}\!\!\rightarrow$ is noetherian such an algorithm will always terminate. And since ${}_{m}\!\!\rightarrow$ is confluent it will always compute the same solution.

4 Reduction Tree Semantics of FM

In this section we shall formalize the informal description of FM and obtain a compact and easy to understand specification of FM's semantics. Recall that the reduction of an expression takes place in the presence of a term environment that binds variables and a function environment that binds designators. Thus reduction starts with a *configuration* $\delta\,\rho\,\{e\}$ where δ is the function environment, ρ is the term environment and e is the expression to be executed. As we already know, the reduction of such a configuration either fails or produces a result together with a function environment that binds the occurring designators. Thus we can write the proposition

$$\delta\,\rho\,\{e\} \;\rightarrow\; \square$$

to express that the reduction of $\delta\,\rho\,\{e\}$ fails; analogously, the proposition

$$\delta\,\rho\,\{e\} \;\rightarrow\; \delta'\{u\}$$

says that the reduction of $\delta\,\rho\,\{e\}$ succeeds with the result u and the function environment δ'. The final function environment δ' is necessary since u can contain designators that are not bound by the initial function environment δ, because the reduction of abstractions creates new designators and functions.

The propositions $\delta\,\rho\,\{e\} \rightarrow \square$ and $\delta\,\rho\,\{e\} \rightarrow \delta'\{u\}$ are called *reductions*. To specify the semantics of FM now means to say when a reduction is *valid*, that is, to say when the configuration on the right is indeed the result of executing the configuration on the left. This will be accomplished by proof rules, which allow us to *prove* that a certain reduction is valid. For instance, the proof rule

$$\frac{\delta\,\rho\,\{a\} \;\rightarrow\; \delta\{a\}}{}$$

specifies the reduction of atoms by saying that a reduction $\delta\, \rho \{ a \} \rightarrow \delta \{ a \}$ is always valid. The proof rule

$$\frac{\delta\, \rho \{ (e, f) \} \;\rightarrow\; \square}{\begin{array}{l} \delta\, \rho \{ e \} \;\rightarrow\; \square \\ \delta\, \rho \{ f \} \;\rightarrow\; F \end{array}}$$

says that a pair reduces to failure if its left component reduces to failure and its right component has a valid reduction. There will be further proof rules for pairs that handle the other cases. The pair rule suggests that proofs for the validity of reductions will have a tree structure that follows the structure of the expression being reduced.

The proof rules are called *reduction rules* and the proofs are called *reduction trees*. The *proof theoretic reading* of the rules is complemented by an *operational reading*. Under the operational reading the rules specify an abstract interpreter that starts with a configuration $\delta\, \rho \{ e \}$ and computes a result F such that $\delta\, \rho \{ \mathrm{e} \} \rightarrow F$ is a valid reduction. Operationally, a reduction tree is a valid computation. To understand the rules, the declarative or proof theoretic reading is superior since it is less detailed than the operational reading. The operational details are obvious once the declarative meaning of the rules is understood.

Reduction tree semantics is compositional since the tree for a particular reduction is obtained by combining the trees for its components. Lack of compositionality has been the standard reason for preferring a denotational description over an operational description.

We begin with the formalization of environments, configurations and reductions. A *term environment* is a substitution. A *closure* $(\rho, s \blacktriangleright e)$ consists of a term environment ρ and an abstraction $s \blacktriangleright e$. A *function environment* δ is a mapping from a finite set of designators into the set of closures, where $\mathbf{D}(\delta)$ denotes the *domain* of δ. An *initial configuration* $\delta\, \rho \{ e \}$ consists of a function environment δ, a term environment ρ, and an expression e. A *final configuration* is either the *failure configuration* $\square$ or a pair $\delta \{ u \}$ where δ is a function environment and and u is a term. A *reduction* has the form $I \rightarrow F$ where I is an initial configuration and F is a final configuration.

A reduction rule consists of a *conclusion* (the reduction above the line), none, one or several *premises* (the reductions beneath the line) and possibly an *application condition* (at the right-hand side of the rule). The conclusion is valid if all premises are valid and the application condition is satisfied.

Atoms, Designators and Variables

$$\frac{\delta\,\rho\{a\} \;\rightarrow\; \delta\{a\}}{} \qquad \frac{\delta\,\rho\{d\} \;\rightarrow\; \delta\{d\}}{} \qquad \frac{\delta\,\rho\{x\} \;\rightarrow\; \delta\{\rho x\}}{}$$

The rules express the fact that atoms and designators reduce to themselves while variables are replaced by their values. These rules justify leaves of reduction trees.

Pairs

$$\frac{\delta\,\rho\{(e,f)\} \;\rightarrow\; \delta'\cup\delta''\{(u,v)\}}{\begin{array}{l}\delta\,\rho\{e\} \;\rightarrow\; \delta'\{u\}\\ \delta\,\rho\{f\} \;\rightarrow\; \delta''\{v\}\end{array}} \qquad \text{iff } \mathbf{D}(\delta')\cap\mathbf{D}(\delta'') \subset \mathbf{D}(\delta)$$

$$\frac{\delta\,\rho\{(e,f)\} \;\rightarrow\; \square}{\begin{array}{l}\delta\,\rho\{e\} \;\rightarrow\; \square\\ \delta\,\rho\{f\} \;\rightarrow\; F\end{array}} \qquad \frac{\delta\,\rho\{(e,f)\} \;\rightarrow\; \square}{\begin{array}{l}\delta\,\rho\{e\} \;\rightarrow\; F\\ \delta\,\rho\{f\} \;\rightarrow\; \square\end{array}}$$

The condition $\mathbf{D}(\delta')\cap\mathbf{D}(\delta'') \subset \mathbf{D}(\delta)$ enforces that newly introduced designators are distinct. Note that an existing designator or variable binding is never changed. Thus the condition is sufficient to ensure that the union $\delta'\cup\delta''$ is well-defined. The failure rules require that both components have valid reductions. Operationally, this means that the reduction of a pair terminates if and only if the reduction of both components terminates.

Conditional Let-Clause

$$\frac{\delta\,\rho\,\{\,s \equiv e \rightarrow f;\, g\,\} \;\rightarrow\; F}{\begin{array}{l}\delta\,\rho\,\{\,e\,\} \;\rightarrow\; \delta'\{\,u\,\}\\ \delta'\,\rho'\,\{\,f\,\} \;\rightarrow\; F\end{array}} \qquad \text{iff} \quad \rho\,\{\,s \equiv u\,\}\; {}_{\mathrm{m}}\!\!\rightarrow^{*}\; \rho'$$

$$\frac{\delta\,\rho\,\{\,s \equiv e \rightarrow f;\, g\,\} \;\rightarrow\; F}{\begin{array}{l}\delta\,\rho\,\{\,e\,\} \;\rightarrow\; \Box\\ \delta\,\rho\,\{\,g\,\} \;\rightarrow\; F\end{array}}$$

$$\frac{\delta\,\rho\,\{\,s \equiv e \rightarrow f;\, g\,\} \;\rightarrow\; F}{\begin{array}{l}\delta\,\rho\,\{\,e\,\} \;\rightarrow\; \delta'\{\,u\,\}\\ \delta\,\rho\,\{\,g\,\} \;\rightarrow\; F\end{array}} \qquad \text{iff} \quad \rho\,\{\,s \equiv u\,\}\; {}_{\mathrm{m}}\!\!\rightarrow^{*}\; \Box$$

Abstraction

$$\frac{\delta\,\rho\,\{\,s \blacktriangleright e\,\} \;\rightarrow\; \delta[d \leftarrow (\rho', s \blacktriangleright e)]\,\{\,d\,\}}{} \qquad \text{where } d \notin \mathbf{D}(\delta) \text{ and } \rho' = \rho|_{\mathbf{V}(s \blacktriangleright e)}$$

An abstraction reduces to a new designator, which is bound to the closure of the abstraction under the initial term environment. The closure environment ρ' is the restriction of the initial environment to the variables occurring in the abstraction. $\delta[d \leftarrow (\rho', s \blacktriangleright e)]$ is obtained from δ by adding the new binding $d \leftarrow (\rho', s \blacktriangleright e)$.

Application

$$\frac{\delta\,\rho\,\{\,f\,[e]\,\} \;\rightarrow\; F}{\begin{array}{l}\delta\,\rho\,\{\,(f, e)\,\} \;\rightarrow\; \delta'\{\,(d, u)\,\}\\ \delta'\,\rho''\,\{\,g\,\} \;\rightarrow\; F\end{array}} \qquad \begin{array}{l}\text{iff} \quad \rho'\,\{\,s \equiv u\,\}\; {}_{\mathrm{m}}\!\!\rightarrow^{*}\; \rho''\\ \text{where } (\rho',\, s \blacktriangleright g) = \delta'(d)\end{array}$$

$$\frac{\delta\,\rho\,\{\,f\,[e]\,\} \;\rightarrow\; \square}{\delta\,\rho\,\{\,(f,e)\,\} \;\rightarrow\; \square}$$

$$\frac{\delta\,\rho\,\{\,f\,[e]\,\} \;\rightarrow\; \square}{\delta\,\rho\,\{\,(f,e)\,\} \;\rightarrow\; \delta'\,\{\,(v,u)\,\}} \qquad \text{iff } v \text{ is not a designator}$$

$$\frac{\delta\,\rho\,\{\,f\,[e]\,\} \;\rightarrow\; \square}{\delta\,\rho\,\{\,(f,e)\,\} \;\rightarrow\; \delta'\,\{\,(d,u)\,\}} \qquad \text{iff } \rho'\,\{\,s \equiv u\,\}\ {}_{m}\!\rightarrow^{*}\ \square \quad \text{where } (\rho',\ s\blacktriangleright g) = \delta'(d)$$

The third rule covers the case where the first component of an application does not reduce to a function. Other functional languages handle this situation with a run-time error. In FM such an application simply fails. A type discipline for FM would statically enforce conditions that prevent such situations from occurring. Below we shall give statically verifiable conditions that ensure that all designators and variables are bound when they are reduced. For that reason there is no rule that handles unbound designators.

We now complete the semantic account of FM by defining reduction trees. An *instance of a rule* is a tuple $(R, R_1, \ldots, R_n)$ where $n \geq 0$ and R and $R_1, \ldots, R_n$ are instances (under the same interpretation) of the conclusion and the premises (in top-down order) of the rule such that the conditions at the right-hand side of the rule are satisfied. Note that the conclusion R and the premises $R_1, \ldots, R_n$ are reductions.

A *reduction tree for a reduction* R is an expression $(R, T_1, \ldots, T_n)$ such that $T_1, \ldots, T_n$ are reduction trees for $R_1, \ldots, R_n$ where $n \geq 0$ and $(R, R_1, \ldots, R_n)$ is an instance of a rule. A *reduction tree for an initial configuration* I is a reduction tree for some reduction $I \rightarrow F$.

The next theorem formulates the fact that the reduction rules are *deterministic*, that is, a reduction has at most one reduction tree up to designator renaming.

Theorem 4.1. All reduction trees for an initial configuration are equal up to consistent designator renaming.

Proof. By induction on the depth of reduction trees. □

Besides being deterministic, the reduction rules have the additional property that all designators and variables are bound when they are reduced, provided the initial configuration is "closed" under its environments. We begin the formalization of this property by defining free variables of expressions. Variables are bound by patterns of abstractions and conditional let-clauses. The set $\mathbf{FV}(e)$ of variables that are free in the expression e is defined inductively:

$$\mathbf{FV}(a) = \varnothing \qquad \mathbf{FV}(d) = \varnothing$$

$$\mathbf{FV}(x) = \{x\} \qquad \mathbf{FV}((e, f)) = \mathbf{FV}(e) \cup \mathbf{FV}(f)$$

$$\mathbf{FV}(s \equiv e \to f; g) = \mathbf{FV}(e) \cup (\mathbf{FV}(f) - \mathbf{V}(s)) \cup \mathbf{FV}(g)$$

$$\mathbf{FV}(s \blacktriangleright e) = \mathbf{FV}(e) - \mathbf{V}(s) \qquad \mathbf{FV}(f[e]) = \mathbf{FV}(f) \cup \mathbf{FV}(e)$$

An expression is *closed* if it has no free variables. Note that this notion of closed expressions is different from the lambda calculus. In the lambda calculus the meaning or reduction of a closed expression is completely independent of the context in which it occurs. This property does not hold for FM since we allowed patterns to contain variables that are bound in an outer context. For instance, although the abstraction $\mathsf{X} \blacktriangleright \mathsf{X}$ is closed, its meaning depends on the environment as the expression $(\mathsf{X} \blacktriangleright \mathsf{X} \blacktriangleright \mathsf{X})[4]$ illustrates.

A term environment ρ is *closed under a function environment* δ if for all $x \in \mathbf{D}(\rho)$ all designators occurring in ρx are bound by δ, that is, are in $\mathbf{D}(\delta)$. A function environment δ is *consistent* if for all $d \in \mathbf{D}(\delta)$ where $\delta(d) = (\rho, s \blacktriangleright e)$ the closure environment ρ is a ground substitution closed under δ, the abstraction body e is closed under δ, and $\mathbf{FV}(s \blacktriangleright e) \subset \mathbf{D}(\rho) \subset \mathbf{V}(s \blacktriangleright e)$.

An initial configuration $\delta\,\rho\,\{e\}$ is *consistent* if

- δ is consistent and ρ is a ground substitution closed under δ.
- e is *closed under* δ, that is, e contains only designators bound by δ.
- e is *closed under* ρ, that is, all free variables of e are bound by ρ.

A final configuration $\delta\{u\}$ is *consistent* if δ is consistent and u is a ground term closed under δ. A reduction $\delta\,\rho\,\{e\} \to \delta'\,\{\,u\,\}$ is *consistent* if $\delta\,\rho\,\{e\}$ and $\delta'\,\{\,u\,\}$ are consistent and the final function environment is an extension of the initial function environment, that is, $\delta = \delta'|_{\mathbf{D}(\delta)}$. Furthermore, a failure reduction $\delta\,\rho\,\{e\} \rightarrow \square$ is consistent if its initial configuration is consistent. The following theorem is easily proven by structural induction on reduction trees.

Theorem 4.2. Every reduction in every reduction tree for a consistent initial configuration is consistent.

We say that the reduction of a consistent initial configuration I

- *fails* if $I \rightarrow \square$ has a reduction tree.
- *succeeds with result* u *and function environment* δ' if the reduction $I \rightarrow \delta'\{\,u\,\}$ has a reduction tree.
- *does not terminate* if there exists no final configuration F such that the reduction $I \rightarrow F$ has a reduction tree.

We know by theorem 4.1 that every consistent initial configuration has exactly one of these properties. Nonexistence of a reduction tree is equivalent to a nonterminating computation since the rules are *complete* for consistent configurations, that is, an interpreter cannot reach a configuration to which no rule is applicable. The rules are incomplete for inconsistent configurations since there are no rules that handle unbound designators or variables.

The given semantics is essentially proof theoretic rather than denotational or purely operational. As in denotational semantics, all expressions that appear in the initial configurations of a reduction tree are contained in the static program. This is in contrast to the lambda calculus where the β-rule instantiates the bodies of abstractions. Moreover, the rules are "effective" and yield an abstract interpreter. The tree structure suggests several possibilities for parallelism. The most important one is the parallel execution of the components of a pair. The only communication overhead involved is that the component reductions must introduce distinct designators.

5 Generalizing Matching to Unification

Functional languages are based on a rigid pattern of information flow. Variables become bound when they are introduced and their values cannot be updated or refined. Functions cannot change their argument and convey all output through their result. In this section we generalize FM by extending the rôle of variables. Unbound variables are now treated as first-class objects and can be part of canonical objects. As in FM, variable bindings are created by solving equations between patterns and canonical objects. Since canonical objects can now contain variables, matching is generalized to unification, which solves equations containing variables on both sides. Bindings created for variables in the canonical side are called *delayed bindings*. Since such variables can occur in the values of other variables, a delayed binding may *refine* or *narrow* already existing bindings. The application of a function now not only produces a result, but can also refine the environment by producing delayed bindings. The value of a variable can now be built incrementally by beginning with a skeleton and binding contained variables later as information becomes available.

This model of variables underlies Prolog, the first unification-based language. Such variables are often called "logical variables" since they appeared with logic programming. This paper tries to illustrate that in a functional framework there are virtually no applications of "logical variables" that exhibit significant advantages over a purely functional formulation and that are compatible with the logic programming paradigm. However, if we abandon the logic programming paradigm and use free variables as first-class objects, then in fact many interesting applications become possible.

"Logical variables" and unification do not come free. Delayed variable bindings are side effects and complicate reasoning about programs. We shall discuss *transparency*, a property that, if satisfied, allows a declarative understanding of delayed variable bindings. Roughly, the result of a transparent expression remains unchanged if the produced delayed bindings are supplied a priori. Nontransparency arises because free variables become first-class objects and one can test for properties that are metalinguistic in functional languages.

FU is a conservative extension of FM; the result of a consistent FM-program remains unchanged under FU's semantics. FU has the same syntax as FM. However, in FU a conditional suffices as base construct since delayed bindings make the definition of a conditional let-clause possible. Thus the bare syntax of FU is:

$$e ::= a \mid d \mid x \mid (e, f) \mid s \blacktriangleright f \mid f[e] \mid e \rightarrow f; g\,.$$

The only construct that produces variable bindings is application. All bindings are obtained as solutions of equations between patterns and reduced arguments. The main semantic innovations are:

- Unbound variables reduce to themselves. Thus FU's canonical objects are terms that can contain variables. In particular, functions can be applied to nonground terms and can produce nonground results.

- Equations between patterns and arguments are solved by unification. Thus an application can bind variables contained in the argument. These *delayed bindings* are propagated to the context of an application. Thus a successful reduction not only produces a result but also refines the term environment. Refinement means that new bindings are added, which can apply to variables in the values of existing bindings.

The new features are best illustrated with the example of *open unification*, a function defined by

unif : *term* × *term* → *term* of
  X, X ▸ X.

In FM, unif is equality, that is, unif $[e, f]$ succeeds with ground term u if and only if both e and f reduce to u. In FU, the results of e and f can be nonground terms u and v, and unif $[e, f]$ succeeds with a most general instance of u and v together with the bindings that form a most general solution of $u \equiv v$. Let us consider the reduction of unif[(1,X), Y] under the environment Y ≡ (Z,U) in detail. The argument reduces to the nonground term ((1, X), (Z, U)). To avoid variable clashes, the reduced argument is applied to a variant of the closure where all variables are renamed to new ones. In general, this renaming is also necessary if the argument and the closure do not share variables, since variables of the closure may become part of the result or the refined environment and are then exported to the outer context. In our example, let the renamed closure be (V,V ▸ V). Then the next step is to solve the equation

(V, V) ≡ ((1,X), (Z,U)) .

A most general solution is $\{V \equiv (1,U),\ X \equiv U,\ Z \equiv 1\}$. Obviously, infinitely many other solutions can be obtained by binding U to some term. A second most general solution is $\{V \equiv (1,X),\ U \equiv X,\ Z \equiv 1\}$. In general, a solvable equation system has always a most general solution and there are only finitely many most general solutions. As in the example, most general solutions differ only in the direction they bind variables to variables. Unification is a nondeterministic algorithm that computes one of the most general solutions. This indeterminacy has no significant impact on the language; results are unique up to designator and variable renaming.

To continue the example, we have now to reduce the body V of the renamed closure under an environment that binds V to (1,U). Thus the result of the application is (1, U). But the reduction also refines the term environment by including the delayed bindings produced for variables in the reduced argument. Formally, we have the valid reduction (the function environments are not shown)

$$Y \equiv (Z,U)\ \{\mathbf{unif}[(1,X), Y]\} \ \rightarrow\ Y \equiv (1,U)\ \ X \equiv U\ \ Z \equiv 1\ \{(1,U)\}.$$

Note that the reduction *refines* or *narrows* the value of Y. It is easy to verify that the application has the same result if we supply the delayed bindings for X and Z a priori, that is, the reduction

$$Y \equiv (1,U)\ \ X \equiv U\ \ Z \equiv 1\ \{\mathbf{unif}[(1,X), Y]\} \ \rightarrow\ Y \equiv (1,U)\ \ X \equiv U\ \ Z \equiv 1\ \{(1,U)\}.$$

is valid as well. In general, an expression is called *transparent*, if it reduces to the same result if the refined environment or any further refinement of it is supplied a priori .

A conditional $e \rightarrow f\,;\, g$ is reduced by first reducing e. If e succeeds the then-part f is reduced under the augmented environments produced by the reduction of e. Otherwise, the else-part is reduced under the initial environments. The expression

$$\mathbf{unif}[s, e] \ \rightarrow\ f\,;\, g$$

is a generalization of FM's conditional let-clause. In the following we will abbreviate $\mathbf{unif}\,[e, f]$ to $e \equiv f$ and use all syntactic extensions defined for FM. If we interpret all conditional let-clauses in a consistent FM-program as above, the resulting FU program is equivalent and produces the same result.

We complete the informal account of FU's semantics by describing the reduction of pairs. As in FM, a pair is reduced by reducing the components independently. If one or both components fail, the pair fails. Otherwise, each component yields a result and a refined term environment. Since the components may have produced delayed bindings for the same variables, a simple union of the two refined environments will not work. Since environments are (solved) equation systems, we build their union and take the solution of this equation system as the refined term environment of the entire pair. If the union of the component environments is unsolvable the reduction of the pair fails. Here are some examples for valid reductions (the function environments are not shown):

$\{(X \equiv 1,\ X \equiv 2)\} \ \rightarrow\ \square$

$\{(X \equiv 1,\ Y \equiv 2)\} \ \rightarrow\ X \equiv 1\ \ Y \equiv 2\ \{(1, 2)\}$

$\{(X \equiv (1, Y),\ X \equiv (Z, 4))\} \ \rightarrow\ X \equiv (1,2)\ \ Y \equiv 4\ \ Z \equiv 1\ \{((1, 4),\ (1, 4))\}$

Let us consider the last example in detail. The first component $X \equiv (1, Y)$ produces the result $(1, Y)$ and the refined environment $X \equiv (1, Y)$. The second component $X \equiv (Z, 4)$ produces the result $(Z, 4)$ and the refined environment $X \equiv (Z, 4)$. Thus the union of the component environments is $\{X \equiv (1, Y),\ X \equiv (Z, 4)\}$ and has the solution $\{X \equiv (1,2), Y \equiv 4, Z \equiv 1\}$. The new bindings $Y \equiv 4$ and $Z \equiv 1$ are applied to the results of the components and yield $((1, 4),\ (1, 4))$ as the result of the pair.

Pairs offer a clear possibility for parallelism. Sequentialization can be accomplished with an if-clause $e \rightarrow f$.

Once a closure is created, it is separated from the context and is not affected by new variable bindings. For instance, the expression

$X \equiv 1,\ Y \equiv (X \blacktriangleright \mathrm{true})\ \rightarrow\ Y[2]$

succeeds with the atom true and the bindings $X \equiv 1$ and $Y \equiv d$ where d is a new designator bound to the closure $X \blacktriangleright \mathrm{true}$. The closure contains no binding for X since such a binding does not exist in the context where the closure is created. Thus the above expression is nontransparent with respect to X; the expression $(1 \equiv 1, Y \equiv (1 \blacktriangleright \mathrm{true})) \rightarrow Y[2]$ fails.

Free variables are first-class objects in FU. The following function succeeds with the atom true if and only if its argument is a free variable.

var of $X \blacktriangleright\ \neg\neg((\neg\neg X \equiv 1),\ (\neg\neg X \equiv 2))$.

Recall that negation as failure $\neg e$ is defined as $e \rightarrow$ fail[]; true. Since a failure reduction does not produce bindings, a double negation $\neg\neg e$ succeeds if and only if e succeeds, but discards all bindings that the reduction of e may have produced. A second, quite different possibility to define a variable test is:

```
var of X ▸ (Y▸true) [ (X▸true)[1], (X▸true)[2] ]
```

This works since the variables in a closure become renamed when the closure is applied. We can go further and define equality for free variables:

```
eqvar of X, Y ▸ var[X], var[Y] → ¬(X ≡ 1, Y ≡ 2).
```

For instance, eqvar[X, Y] reduces to true if X and Y are free. On the other hand, X ≡ Y → eqvar[X, Y] will fail since before the test X gets bound to Y or vice versa. However, the pair (X ≡ Y, eqvar[X,Y]) will succeed since the components of a pair are reduced independently.

The existence of a variable test allows for nontransparent first-order expressions, for instance, (var[X], X ≡ 1). Furthermore, FU is necessarily nontransparent with respect to variables that become bound to higher-order objects. For instance, the expression X ≡ (Y▸Y) reduces to result d and binding X ≡ d where d is a new designator, but d ≡ (Y▸Y) fails since an abstraction always reduces to a new designator.

5.1 The Dictionary Example

This classical example for the use of incomplete data structures appeared first in Warren [36]. The task is to construct a dictionary of pairs (key, value) that is organized as a binary search tree with respect to a linear order on the set of keys. The basic idea is to represent empty subdictionaries as free variables, so that new entries can be inserted by binding an empty subdictionary rather than by constructing a new dictionary. The type of our dictionary is

$$dict(K,V) = ?(\{\text{emptydict}\} + K \times V \times dict(K,V) \times dict(K,V))$$

where K and V are type variables representing the key and the value type. A dictionary is now either a free variable (defined by the ?-operator), the atom emptydict or a tuple consisting of a key, a value, and a left and right subdictionary. As long as the dictionary is not complete all empty subdictionaries are represented as free variables. After all entries are made a dictionary can be *closed* by binding all empty subdictionaries to the atom emptydict. The following predicate defines membership for a closed dictionary:

```
contains: (K × K → {true}) → dict(K,V) × K × ?V → {true} of
    LT ▸ ( (K, V, L, R),    K,  V    ▸ true ;;
           (K, V, L, R), NK, NV   ▸ contains [LT] [ ( LT[NK, K] → L ; R ), NK, NV ] ).
```

The first argument LT supplies the order according to which the dictionary is organized.

The **contains** predicate can retrieve values from a closed dictionary. The application **contains** [lt] [D, K,V] , where lt is the order, D is bound to a closed dictionary, K is bound to a key and V is a free variable, succeeds if and only if D contains an entry for the key K. In this case the variable V gets bound to the corresponding value. However, the most interesting thing about **contains** is that it can be used to insert a new entry into an open dictionary (that is, all empty subdictionaries are variables). When applied to an open dictionary, **contains** tries to refine (that is, instantiate) the dictionary such that it contains the new entry. This insertion operation fails if the dictionary already contains an entry for the key in question, but with a different value. Thus **contains** can built a dictionary beginning with a variable. The next predicate refines a dictionary such that it is closed:

```
closed: dict(K,V) → {true} of
    emptydict      ▸ true ;;
    (K, V, L, R)   ▸ closed [L], closed [R] → true.
```

We can now built a dictionary, close it and retrieve information from it. For instance, the expression

```
contains [lt] [D, b, 1] → contains [lt] [D, a, 2] → contains [lt] [D, c, 3] → closed [D]
   → contains [lt] [D, a, V] → V
```

yields the result 2 and binds D to

```
(b, 1,  (a, 2, emptydict, emptydict),  (c, 3, emptydict, emptydict) ) .
```

This expression is transparent; if the binding for D is supplied a priori, the expression reduces to the same result.

What do we do if retrieval is necessary before the dictionary is complete? If we check with **contains** for a key that is not in an open dictionary, a new entry is made, and that is probably not what we want. If we assume that no value in the dictionary is a variable, the following function implements retrieval on open dictionaries:

```
get: (K × K → {true}) → dict(K,V) × K → V of
      LT ▸ D, K ▸ contains [LT] [D, K, V] → ¬var[V] → V.
```

This function cannot be expressed in a logic programming language since it tests for a metalogical property. Furthermore, get and **contains** are higher-order since they take the key order as argument. So far, higher-order logic programming languages have not been developped.

5.2 Polymorphic Type Inference

Since polymorphic type inference [7] employs unification as a basic operation, it should yield a prime example for FU's expressive power. This is in fact the case, but only if we employ some nontransparent features. Let us consider a simple functional language with pairs, abstractions, applications and conditionals

expn ::= *identifier* | (*expn*, *expn*) | abs(*identifier*, *expn*) |
app(*expn*, *expn*) | cond(*expn*, *expn*, *expn*)

and the type structure

type ::= *identifier* | *variable* | prod(*type*, *type*) | fun(*type*, *type*)

where *variable* stands for FU-variables. The nonground term fun(T, T) is an example for a polymorphic type. The function

```
typeof: assignment → base × expression → type of
      A ▸ ( B,    abs(I,E)    ▸ fun(S, typeof[A][((I,S), B), E]) ;;
            B,    app(F,E)    ▸ T ← typeof[A][B, F] ≡ fun(typeof[A][B, E], T);;
            B,  cond(C,E,F)   ▸ typeof[A][B, E] ≡ typeof[A][B, F] ← bool ≡ typeof[A][B, C];;
            B,     (E,F)      ▸ prod( typeof[A][B, E], typeof[A][B, F] );;
            B,       I        ▸ getvalue[B, I] ; A[I] ).
```

infers the type of an expression under a type assignment and a base. The assignment assigns types to defined functions, that is, it is a function from identifiers to types. The base assigns types to identifiers that are introduced by abstractions and is represented as a list of pairs of identifiers and types. The last clause of **typeof** accesses the base and the assignment. The base is given priority since its identifiers correspond to inner scopes. The function getvalue is defined in section 2. If the base B contains no binding for the identifier I getvalue fails and the assignment is tried. Since the assignment is a function it can be accessed by application. If the assignment contains no binding for the identifier the entire type inference fails.

In the following examples for typeof we assume the following binding for the type assignment:

A = (id ▸ fun(T, T);; eq ▸ fun(prod(T, T), bool))

The reader may think of id as the identity function and of eq as the equality function. The application

typeof [A] [(), app(id, id)]

yields as expected the result fun(X, X) for the self-application app(id, id). Note that X is a new variable. It will now become clear why we separated the type bindings into the assignment and the base. Since the assignment is a function the variables in its abstraction become renamed upon application. Hence, every access of the assignment provides a type with new type variables. This allows the inference of the correct type for the self-application.

Each clause of typeof corresponds directly to a type rule. This succinctness is possible since FU can solve equations that contain function applications. An examples that occurs in typeof is fun(typeof[A][B, E], T) ≡ typeof[A, B][F]; this equation is solved for T and for all variables that occur in the base B. Most of the computational work is done by unification; new bindings for type variables are propagated as side effects. For instance, the application

typeof [A] [((x, int), (y, Y), (z, list), (v, V), ()),
        cond(appl(eq, (x, y)), z, v)]

yields the result list as the type of the expression and the variable bindings Y ≡ int and V ≡ list for the types of the identifiers y and v in the base. Note that typeof is transparent with respect to "type variables" in the base. The application

typeof [A] [((x, int), (y, real), (z, list), (v, V), ()),
        cond(appl(eq, (x, y)), z, v)]

fails since the expression is not well-typed under the given base.

To build a complete type checker we must be able to decide whether a type is an instance of another type. Since types are terms it suffices if we can decide this question for terms. We say that a term *s* *subsumes* a term *t* if there exists a substitution θ such that $s = \theta t$. A subsumption test can be defined as follows:

```
subsumes: term × term → {true} of
   S, T ▸ ¬ ¬ ( L ≡ vars[T] → S ≡ T → map[ V▸var[V] ] [L] ).

vars: term → list(term) of
     X  ▸ var[X]    → X, () ;
          X ≡ S, T  → append [ vars [S], vars [T] ] ;
                      ().
```

The function vars computes a list that contains all variables that occur in its argument. If S subsumes T than S and T can be unified without binding a variable of T to anything but a variable. Thus the application of map in subsumes succeeds if no variable in T was bound to a nonvariable. The double negation makes sure that subsumes does not bind variables contained in its arguments. It is clear that subsumes is not a transparent function. Since it tests for a metalogical property it cannot be written in a logic programming language. To formulate polymorphic type checking in a logic programming language, we were forced to represent type variables as ground terms and were thus deprived of the advantages of the built-in unification. It is questionable whether a formulation in, say, Horn logic would have any advantages over a functional formulation in FM.

To continue the type checking example, let us assume that programs consist of a sequence of function definitions followed by an expression:

program ::= fundef(*identifier*, *type*, abs(*identifier*, *expn*)), *program* | *expn*.

The type checker takes a type assignment and a program and succeeds with the most general type of the result if the program is well-typed. The type assignment specifies the built-in functions and their types.

```
welltyped: assignment × program → type of
   A, P ▸ welltyped1 [ fullassignment [A, P], P].
```

```
fullassignment: assignment × program → assignment of
    A, (fundef(F, T, E), P)   ▸ ¬A[F]  →  fullassignment [ (F▸T;; X▸A[X]) , P] ;;
    A, E                      ▸ A.

welltyped1: assignment × program → type of
    A, (fundef(F, T, E), P)   ▸ subsumes [ T, typeof [A] [(), E] ]  →  welltyped1[ A, P] ;;
    A, E                      ▸ typeof [A] [(), E].
```

In the first pass through the program the checker collects all defined functions and their types and makes sure that their names do not conflict with each other or the built-in functions. The second pass checks that all defined functions satisfy their declared types and finally computes the most general type of the expression to be executed. Note that the type checker handles recursive function definitions and that the order of the function definitions is irrelevant.

The goal of this section was to illustrate that unification can be smoothly integrated into a functional language. Two applications were presented that illustrate the use of unification in such a framework and show the advantages over purely functional formulations. Both applications are incompatible with the logic programming paradigm, since they depend crucially on the metalogical test whether a variable is free. There seem to be virtually no applications of practical relevance that employ "logical variables" in an entirely transparent fashion and still show significant advantages over a purely functional formulation.

6 Adding Multiple Results

Fresh is obtained from FU by adding three constructs for the introduction and elimination of multiple results. Operationally, multiple results lead to a reduction strategy with chronological backtracking. Multiple results and matching yield a language with expressive data base capabilities. Furthermore, the combination of backtracking and collection can often replace recursion and lead to intuitive formulations with a declarative reading. Finally, the incorporation of unification and multiple results yields a language that subsumes Horn logic with equality.

Multiple results are introduced by *disjunctions*. The results of a disjunction $e \mid f$ are the results of e followed by the results of f. Like Prolog, Fresh employs a left-to-right depth-first search strategy. For instance, the expression (1| 2) | (1| 3) has the *result sequence* 1, 2, 1, 3. Note that the result 1 occurs twice. The combination of recursion and backtracking allows expressions that have an infinite result sequence. For instance, an application inf[()] where

inf of () ▸ 1 | inf[()].

has the infinite result sequence 1, 1, 1, ... A useful example for the combination of recursion and backtracking is a list enumerator

el : *list(T)* → T of
  H, T ▸ H | el[T].

with the declarative reading "an element of a list is either the head or an element of the tail". The application el[1, 2, 3, ()] yields the result sequence 1, 2, 3, □ where the failure result □ stems from the empty list ().

Multiple results extend naturally to pairs and applications. Here are some examples:

((1| 2), (3| 2))	yields	(1, 3), (1, 2), (2, 3), (2, 2)
(X ≡ (1| 2), X ≡ (2| 3))	yields	□, □, (2, 2), □
((1 ▸ (first | second)) | (X ▸ X)) [1| 2| 3]	yields	first, second, □, □, 1, 2, 3

The extension of the conditional to multiple results is somewhat more subtle. We introduce a new conditional $e \Rightarrow f\,;\, g$ called *soft conditional*. It is reduced by first reducing e. If e has no success result the reduction proceeds with g and the result sequence of g is the result sequence of the entire conditional. If the condition part e has a success result, the reduction proceeds with f under the first environment created by e. The result sequence of the entire conditional is then obtained by concatenating the result sequences of the then-part f under the refined environments produced by the condition part e. In other words, execution can backtrack from the then-part to the condition part. Here are examples:

X ≡ (1| 2| 3) ⇒ X; 5	yields	1, 2, 3
X ≡ (1| 2| 3) ⇒ ((X ≡ (2|3)), 4) ; 5	yields	□, □, (2, 4), □, □, (3, 4)
X ≡ 1 ⇒ X ≡ 2; 5	yields	□
((X ≡ 1), (1 ≡ 2)) ⇒ X ≡ 1; (4| 5| X)	yields	4, 5, X

The last example shows that bindings produced by the condition part (here $X \equiv 1$) are not propagated to the else-part.

Confinement (written $!e$) discards all but the first success result of an expression. Here are examples:

!(5, X)	yields	(5, X)
!(1| 2| 3)	yields	1
!(X ≡ (1| 2| 2) ⇒ X ≡ 2 ; 5)	yields	2
!(0 ≡ 1)	yields	□

The last example shows that confinement fails if the expression it is applied to has no success result.

We can now define a *hard conditional*

$$e \rightarrow f ; g \qquad \text{is} \qquad !e \Rightarrow f ; g$$

that does not allow backtracking from the then-part to the condition part. In a program without disjunctions the soft and the hard conditional are equivalent since then there are no multiple results that can be cut away.

Collection (written $\{e\}$) is the second construct for the elimination of multiple results. The expression $\{e\}$ reduces to the *list* of all success results of the expression e. If e fails, that is has only failure results, the collection of e reduces to the empty list. Thus collection always has exactly one success result. Here are some examples:

{1}	yields	(1, ())
{1| 2| 3}	yields	(1, 2, 3, ())
{ X ≡ (1| 2| 3) | 5 }	yields	(1, 2, 3, 5, ())
{0 ≡ 1}	yields	()
{**el** [1, 2, 3, ()] }	yields	(1, 2, 3, ())

Delayed bindings created during the reduction of a collection are not propagated to the outer context. Furthermore, variables in the elements of the result list are renamed to new ones such that no two elements share variables. For instance, the expression

$$X \equiv (1, Y) \rightarrow \{ X \equiv (Z, (2| 3| U| U)) \}$$

yields the result ((1, 2), (1, 3), (1, A), (1, B), ()) and the delayed binding X ≡ (1, Y) where A and B are new variables. During the reduction of the collection the bindings Y ≡ 2, Y ≡ 3, Y ≡ U, Y ≡ U are created successively. Since these bindings are independent, a propagation of a binding for Y outside of the collection construct is not sensible. This semantics for the collection construct are also supported by an evolving type discipline for Fresh.

A collection of an expression with an infinite result sequence will not terminate. This problem could be resolved if the language employed a lazy reduction strategy for collections. Then confinement could be defined by the collection construct as (F,S) ≡ $\{e\} \Rightarrow$ F.

Collection constructs also exist in Prolog [4] and KRC [33]. However, these languages integrate collection somewhat artificially since syntax and semantics inside a collection are different from the rest of the language. In Fresh collection can be applied to any expression.

Fresh's bare syntax is

$$e ::= a \mid d \mid x \mid (e, f) \mid s \blacktriangleright f \mid f[e] \mid e \Rightarrow f; g \mid e|f \mid !e \mid \{e\}.$$

The connection to FM and FU is made by the syntactic extension

$$e \rightarrow f; g \qquad \text{is} \qquad !e \Rightarrow f; g$$

which defines the *hard conditional*. The results of FM or FU programs remain unchanged under Fresh's semantics. Disjunction provides for a second form of the clausal definition construct:

$$\begin{array}{lll}
s_1 \blacktriangleright e_1 \; \| & \text{is} & \mathrm{X} \blacktriangleright (\;\; (s_1 \blacktriangleright e_1) \mid \\
s_2 \blacktriangleright e_2 \; \| & & \qquad\;\; (s_2 \blacktriangleright e_2) \mid \\
\quad\vdots & & \qquad\quad\;\; \vdots \\
s_n \blacktriangleright e_n & & \qquad\;\; (s_n \blacktriangleright e_n) \;\;)[\mathrm{X}].
\end{array}$$

If the patterns s_i are pairwise nonunifiable (that is, have a common instance) this construct is equivalent to the clausal definition construct based on the conditional. Otherwise, the disjunction-based construct allows that more than one clause contributes to the result sequence.

6.1 Examples

Recall the definition of list enumeration

```
el : list(T) → T of
      H, T ▸ H | el[T].
```

which reads "an element of a list is either its head or an element of its tail". The collection { (el[l_1], el[l_2]) } yields the cartesian product of the lists l_1 and l_2. Furthermore, the sublist of all elements of a list l that satisfy a predicate p is obtained by { X ⇐ p[X ≡ el[l]] } where the *where-clause* $f \Leftarrow e$ is a syntactical variant of $f \Rightarrow e$; fail[].

In section 2 we defined in FM a function that yields the power list of a list. We can now define a declarative version of this function:

```
sublist: list(T) → list(T) of
          () ▸ ();;
       H, T ▸ S | H,S ⇐ S ≡ sublist[T].
```

This may be read: "The sublist of the empty list is the empty list; and a sublist of a nonempty list is either a sublist of the tail or the head together with a sublist of the tail". The power list of a list l is now simply computed by { sublist [l] }.

The next example illustrates Fresh's data base capabilities. The task is to implement a grade book that contains entries of the form (student, test, grade). A straightforward solution is to represent the grade book as a list of such tuples. In the following we assume that the variable B is bound to such a grade book. The query "all students who got an A in test CS211" is realized by the expression

```
{ S ⇐ (S, cs211, a) ≡ el [B] }.
```

The query "all students who ever flunked a test" becomes

```
{ S ⇐ (S, T, f) ≡ el [B] }.
```

A list of all students who ever flunked a test together with the list of all tests a student flunked is computed by

```
{ (S, {T ⇐ (S, T, f) ≡ el [B] }) ⇐ (S, T1, f) ≡ el [B] }.
```

The query "all students who flunked data bases and got an A in theory" becomes (the where symbol binds left-associative)

$$\{ S \Leftarrow (S, \text{database}, f) \equiv \text{el}\,[B] \Leftarrow (S, \text{theory}, a) \equiv \text{el}\,[B] \} .$$

Finally, the grade book obtained from B by deleting all entries for the test CS211 is computed by

$$\{ E \Leftarrow \neg T \equiv \text{cs211} \Leftarrow E \equiv (S, T, G) \equiv \text{el}\,[B] \} .$$

6.2 Comparison with Prolog

A Prolog program is a set of predicates. Each predicate is defined by a sequence of Horn clauses. Two typical examples are list concatenation and reversal:

```
append( nil, L, L ) ← true.
append( cons(H, T), L, cons(H, TL) ) ← append(T, L, TL) .

reverse(nil, nil) ← true.
reverse( cons(H, T), L) ← reverse(T, R), append( R, cons(H, nil), L) .
```

The empty list is the atom nil and a list with head h and tail t is the term cons(h, t). Horn clauses and predicates can be expressed in Fresh as well:

```
append of
                nil, L, L          ▸ true ||
    cons(H, T), L, cons(H, TL)     ▸ append [T, L, TL] .

reverse of
                nil, nil     ▸ true ||
    cons(H, T), L            ▸ reverse [T, R]  ⇒  append [R, cons(H, nil), L] .
```

Predicates become functions that either fail or yield the atom true as result. Prolog's conjunction is sequential and translates therefore into an if-clause. The two Fresh functions have precisely the same semantics as the corresponding Prolog predicates. In other words, Horn logic with depth-first search is a subset of Fresh. Prolog's cut is easily replaced by confinement and conditional, and Prolog's negation as failure is negation in Fresh.

Horn clauses are not the intended programming style for Fresh. In fact, Fresh is an attempt to offer Prolog's capabilities in a framework that avoids Prolog's drawbacks:

- In Fresh, all constructs can be nested. This avoids the need for auxiliary variables and leads to a compact notation.

- Functions and nesting make information flow explicit. In Prolog, information flow can only be accomplished by side effects (delayed bindings) and is thus difficult to analyze.

- In Fresh, full unification and delayed bindings are the exception. If we consider an open unification $e \equiv f$ where one side reduces to a ground term as matching, only the examples in section 5 require full unification and employ side effects. In Prolog, every predicate application requires full unification and conveys output through side effects.

- In Fresh, backtracking is introduced explicitly by disjunction and is localized by confinement and collection. A sufficient condition that implies that a function has at most one result can be easily tested at compile-time. In Prolog, backtracking is the default. Since most applications do not involve backtracking, intellectual effort and numerous cuts are needed to suppress backtracking.

- Higher-order functions and functional abstraction are a structured alternative to Prolog's obscure assert and call constructs. While higher-order functions are easily covered by a static type discipline, this is not the case with Prolog's assert and call constructs.

- Fresh's pair construct offers a clear possibility for parallelism. Since Prolog's conjunction is defined sequentially, the exploitation of and-parallelism requires complicated compile-time analysis. In Fresh, sequentialization is naturally expressed by nesting and conditionals.

7 Unification

In Fresh, all variable bindings are obtained by solving equations between patterns and reduced arguments. In contrast to FM, both sides of an equation can contain variables. This complicates the situation since such equations can have infinitely many solutions. For instance, the equation

(a, X, U, (b, c)) ≡ (a, (Y, c), V, X)

is certainly solved by {X ≡ (b, c), Y ≡ b, U ≡ V}. However, {X ≡ (b, c), Y ≡ b, V ≡ U} is a solution as well; and infinitely many further solutions can be obtained by binding the variables V or U to arbitrary terms. Fortunately, solvable unification problems have most general solutions, from which all other solutions can be obtained. In our example, the first two solutions are most general. We will prove that most general solutions are unique up to reversal of bindings that bind variables to variables. Thus the task of a unification algorithm is to decide whether an equation system is solvable and, if so, to compute a most general solution.

This section presents the basic notions and results from unification theory in a framework that is tailored for the semantic account of Fresh. Idempotent substitutions will play the rôle of ground substitutions and be most general solutions of unification problems and term environments in the reduction tree semantics. A nondeterministic unification algorithm is obtained by extending the matching rules. The key results are theorem 7.5 (most general solutions are unique up to binding reversal), theorem 7.9 (the unification rules are noetherian) and theorem 7.12 (the unification rules are confluent up to binding reversal and yield a most general solution).

7.1 Subsumption, Idempotence and Equivalence of Substitutions

A substitution θ *subsumes* a substitution ψ, written $\theta \leq \psi$, if $\theta = \theta\psi$. A substitution θ is *idempotent* if it subsumes itself, that is, $\theta = \theta\theta$. It is easy to show that $\theta \leq \psi$ is a preorder (that is, reflexive and transitive) on the set of idempotent substitutions. Thus, $\theta \sim \psi \Leftrightarrow \theta \leq \psi \wedge \psi \leq \theta$ defines an equivalence relation $\sim$ on idempotent substitutions. We say that two substitutions θ and ψ are *equivalent* if they are idempotent and subsume each other, that is, $\theta \leq \psi$ and $\psi \leq \theta$. The letter ρ will always range over idempotent substitutions.

Recall that we consider only finite substitutions, and that a finite substitution θ can be identified with its representation, that is, the equation system $\{ x \equiv \theta x \mid x \in \mathbf{D}(\theta) \}$. The following is easy to prove:

Proposition 7.1. θ subsumes ψ if and only if θ solves ψ.

The *variables introduced by a substitution* θ, written $\mathbf{I}(\theta)$, are the variables contained in the right-hand sides of its representation, that is, $\mathbf{I}(\theta) = \bigcup \{ \mathbf{V}(\theta \mathbf{x}) \mid x \in \mathbf{D}(\theta) \}$. Obviously, a substitution is idempotent if and only if its domain and its introduced variables are disjoint. $\mathbf{V}(\theta) := \mathbf{D}(\theta) \cup \mathbf{I}(\theta)$ is the set of all variables occurring in the finite representation of θ. Two substitutions θ and ψ are *independent* if $\mathbf{V}(\theta)$ and $\mathbf{V}(\psi)$ are disjoint. If θ and ψ are independent, then $\theta \cup \psi$ denotes the substitution whose representing equation system is the union of the representing equation systems of θ and ψ.

Proposition 7.2. If θ and ψ are independent then $\theta\psi = \psi\theta = \theta \cup \psi$.

Our next goal is to show that two substitutions are equivalent if and only if they are equal up to binding reversal. The concept of a direct variant makes precise what we mean by binding reversal. A substitution ψ is a *direct variant* of a substitution θ if there exist pairwise distinct variables $x_1, \ldots, x_n, y_1, \ldots, y_n$ such that $\theta x_i = y_i$ for all i and $\psi = \{y_1 \equiv x_1, \ldots, y_n \equiv x_n\}\theta$. For example, the substitution $\{X \equiv Y, Z \equiv Y\}$ is a direct variant of $\{Y \equiv Z, X \equiv Z\}$ since $\{X \equiv Y, Z \equiv Y\} = \{Z \equiv Y\} \{Y \equiv Z, X \equiv Z\}$. The following proposition is easy to prove:

Proposition 7.3. Direct variants of idempotent substitutions are idempotent.

The following lemma is needed for the proof of the next theorem.

Lemma 7.4. Let θ and ψ be equivalent, but distinct substitutions. Then there exist distinct variables x and y such that $\theta x = y$ and $\psi y = x$.

Proof. Let $|s|$ denote the depth of a term s. First we observe that $|\theta s|=|\psi s|\geq|s|$ since $|\theta s|=|\theta\psi s|\geq|\psi s|=|\psi\theta s|\geq|\theta s|\geq|s|$. Now let s be a term such that $\theta s\neq\psi s$. We prove the claim by induction on $|\theta s|$.

$|\theta s|=1$. Since substitutions cannot disagree on atoms or designators s must be a variable. Since $\theta s\neq\psi s$ and $\psi s=\psi\theta s$ we have that $y:=\theta s$ is a variable. Since $\psi s\neq\theta s$ and $\theta s=\theta\psi s$ we have that $x:=\psi s$ is a variable. Thus $\theta x=y$ and $\psi y=x$ and x and y are distinct.

$|\theta s|>1$. Then $\theta s=(u_1, u_2)=(\theta u_1, \theta u_2)$ and $\psi s=(v_1, v_2)=(\psi v_1, \psi v_2)$ since θ and ψ are idempotent. Since $\theta s\neq\psi s$ we have $\theta u_1\neq\psi v_1$ without loss of generality. Now the claim follows from the induction hypothesis. □

Theorem 7.5. Let θ and ψ be idempotent substitutions. Then θ and ψ are equivalent if and only if ψ is a direct variant of θ.

Proof. *Right to left.* Let $\psi=\{y_1\equiv x_1,\ldots,y_n\equiv x_n\}\theta$ and $x_1,\ldots,x_n, y_1,\ldots,y_n$ be pairwise distinct variables such that $\theta x_i=y_i$ for all i. Then ψ subsumes θ since θ subsumes θ. On the other hand, θ subsumes ψ (that is, $\theta=\theta\{y_1\equiv x_1,\ldots,y_n\equiv x_n\}\theta$) since $\theta\{y_1\equiv x_1,\ldots,y_n\equiv x_n\}=\theta$. Thus θ and ψ are equivalent.

Left to right. We prove that ψ is a direct variant of θ by induction on $n:=|\{x\in\mathbf{V}|\,\theta x\neq\psi x\}|$, that is, the number of variables θ and ψ disagree on.

$n=0$. Then $\theta=\psi$.

$n>0$. Then θ and ψ are distinct. Thus by lemma 7.4 there are distinct variables x and y such that $\theta x=y$ and $\psi y=x$. Then $\theta':=\{y\equiv x\}\theta$ is a direct variant of θ. Furthermore, we know by proposition 7.3 that θ' is idempotent. Thus we have by the right to left part of the proof that $\theta\sim\theta'$. Since we assumed that $\theta\sim\psi$ we have $\psi\sim\theta'$. Now the induction hypothesis applies to ψ and θ' and yields that ψ is a direct variant of θ'. Thus there exist pairwise distinct variables $x_1,\ldots,x_n, y_1,\ldots,y_n$ such that $\theta' x_i=y_i$ for all i and $\psi=\{y_1\equiv x_1,\ldots,y_n\equiv x_n\}\theta'$. Since $\theta' x=x$ we have that x is distinct from all x_i; since $\theta' x=x$ and $\psi x=x$ (ψ is idempotent and $\psi y=x$) we have that x is distinct from all y_i. Since $\theta' y=x$, y and x are distinct and θ' is idempotent we have that y is distinct from all y_i; finally, since $\theta' y=x$ and x is different from all y_i we have that y is different from all x_i. Thus $\psi=\{y_1\equiv x_1,\ldots, y_n\equiv x_n\}\theta'=\{y_1\equiv x_1,\ldots, y_n\equiv x_n\}\{y\equiv x\}\theta=\{y_1\equiv x_1,\ldots,y_n\equiv x_n, y\equiv x\}\theta$ since $\{y_1\equiv x_1,\ldots,y_n\equiv x_n\}$ and $\{y\equiv x\}$ are idempotent. Thus ψ is a direct variant of θ. □

7.2 The Unification Rules

The next lemma justifies a method for the stepwise construction of idempotent substitutions, which is employed by the unification rules.

Lemma 7.6. The composition $\{x \equiv \rho s\}\rho$ is an idempotent substitution if $x \notin \mathbf{D}(\rho) \cup \mathbf{V}(\rho s)$. Furthermore, $\{x \equiv \rho s\}\rho$ subsumes ρ.

Proof. The second claim is obvious since ρ subsumes ρ. Let y be a variable. To prove that $\{x \equiv \rho s\}\rho$ is idempotent it suffices to show that $\{x \equiv \rho s\}\rho\{x \equiv \rho s\}\rho y = \{x \equiv \rho s\}\rho y$. This equality holds since $\mathbf{D}(\{x \equiv \rho s\}\rho) = \mathbf{D}(\rho) \cup \{x\}$ and $\mathbf{D}(\rho) \cup \{x\}$ and $\mathbf{V}(\{x \equiv \rho s\}\rho y)$ are disjoint. □

A *unification problem* is a pair $\rho\{E\}$ where ρ is an idempotent substitution and E is a finite equation system. A *solution* for a unification problem $\rho\{E\}$ is a substitution θ that solves ρ and E. *Unification* is the process of computing the solution of a unification problem. The unification rules reduce a unification problem $\rho\{E\}$ by growing the *solved part* ρ and shrinking the *unsolved part* E. Note that ρ already solves ρ. There are seven unification rules:

(Taut)	$\rho\{s \equiv s,\ E\} \ {}_{\mathrm{u}}\!\!\rightarrow\ \rho\{E\}$		iff $s \in \mathbf{A} \cup \mathbf{D} \cup \mathbf{V}$
(LSubst)	$\rho\{x \equiv u,\ E\} \ {}_{\mathrm{u}}\!\!\rightarrow\ \rho\{\rho x \equiv u,\ E\}$		iff $x \in \mathbf{D}(\rho)$
(LBind)	$\rho\{x \equiv u,\ E\} \ {}_{\mathrm{u}}\!\!\rightarrow\ \{x \equiv \rho u\}\rho\{E\}$		iff $x \notin \mathbf{D}(\rho) \cup \mathbf{V}(\rho u)$
(RSubst)	$\rho\{s \equiv x,\ E\} \ {}_{\mathrm{u}}\!\!\rightarrow\ \rho\{s \equiv \rho x,\ E\}$		iff $x \in \mathbf{D}(\rho)$
(RBind)	$\rho\{s \equiv x,\ E\} \ {}_{\mathrm{u}}\!\!\rightarrow\ \{x \equiv \rho s\}\rho\{E\}$		iff $x \notin \mathbf{D}(\rho) \cup \mathbf{V}(\rho s)$
(Split)	$\rho\{(s, t) \equiv (u, v),\ E\} \ {}_{\mathrm{u}}\!\!\rightarrow\ \rho\{s \equiv u, t \equiv v,\ E\}$		
(Fail)	$\rho\{s \equiv u,\ E\} \ {}_{\mathrm{u}}\!\!\rightarrow\ \square$		iff none of the rules above applies

The unification rules extend the matching rules. Since variables can now occur in the right-hand sides as well, there are symmetric substitution and binding rules for right-hand sides. The significant difference between unification and matching is in the formulation of the binding rules. The augmented environment $\{x \equiv \rho u\}\rho$ is obtained by left-composition with $\{x \equiv \rho u\}$. Since the right-hand sides of the representation of ρ can now contain variables, they may become instantiated under the new binding. A new binding $x \equiv \rho u$ can only be made if it is not cyclic, that is, $x \notin \mathbf{V}(\rho u)$. This is the so-called *occurrence check*. For instance, the equation $X \equiv (X, X)$ has no solution.

Most Prolog systems ignore the occurrence check since it requires a considerable computational overhead. These systems are not only logically inconsistent, but their unification algorithm will also not terminate for certain cyclic equation systems. Colmerauer [5, 6] has developed a consistent theory of cyclic unification and infinite regular trees. However, to guarantee termination, he incorporates a condition that is as expensive as the occurrence check. Van Emden and Lloyd [20] show that Colmerauer's cyclic unification is sound for Horn logic with an augmented equality theory. For Fresh, the occurrence check does not cause problems, since the big majority of unifications can be implemented by matching. In typed Fresh, which is currently under development, compile-time analysis will automatically decide for which equations matching is sufficient. Furthermore, typed Fresh will exploit conditions (like those formulated by Plaisted [25]) that imply that occurrence checks are redundant although both sides contain variables.

In the following ${}_{\mathrm{u}}{\to}^{*}$ denotes the reflexive and transitive closure of ${}_{\mathrm{u}}{\to}$ on the set of unification problems. Furthermore, $\rho\{E\}\ {}_{\mathrm{u}}{\to}^{*}\ \rho'$ is an abbreviation for $\rho\{E\}\ {}_{\mathrm{u}}{\to}^{*}\ \rho'\{\varnothing\}$. The next proposition says that unification generalizes matching.

Proposition 7.7. Let $\rho\{M\}$ be a matching problem. Then $\rho\{M\}\ {}_{\mathrm{m}}{\to}^{*}\ \rho'$ if and only if $\rho\{M\}\ {}_{\mathrm{u}}{\to}^{*}\ \rho'$.

Proof. If the right-hand sides of equations are ground terms, the right substitution and binding rules do not apply. Furthermore, the occurrence check in the left binding rule is always satisfied. Thus the applicable unification rules are equivalent to the matching rules. □

Lemma 7.8. Let $\rho\{E\}\ {}_{\mathrm{u}}{\to}^{*}\ \rho'\{E'\}$. Then ρ' is an idempotent substitution and subsumes ρ. Furthermore, $\mathbf{V}(\rho') \subseteq \mathbf{V}(\rho) \cup \mathbf{V}(E)$.

Proof. The first claim follows from lemma 7.6 by induction on the length of the reduction sequence. The second claim is obvious since none of the rules introduces new variables. □

The following theorem states that the unification rules do not allow infinite reduction sequences.

Theorem 7.9. The reduction $_{\mathrm{u}}\!\rightarrow$ is noetherian.

Proof. Suppose there is an infinite reduction sequence starting from $\rho\{E\}$. Since $\rho\{E\}$ contains only finitely many variables, new variables are not introduced, and the binding rules increase the number of variables in the domain of the idempotent substitution ρ, we can assume without loss of generality that the infinite reduction sequence does not employ the binding rules. The remaining rules do not increase the number of occurrences of variables contained in the domain of the idempotent substitution ρ. Since both substitution rules decrease the number of those occurrences at least by one, we can now assume that the infinite reduction sequence employs only the tautology and splitting rule which, is impossible. $\square$

The next lemma states that a solvable unification problem cannot be reduced to the failure configuration $\square$.

Lemma 7.10. If $\rho\{E\}$ has a solution and E is nonempty, then there exist ρ' and E' such that $\rho\{E\}$ $_{\mathrm{u}}\!\rightarrow$ $\rho'\{E'\}$.

Proof. Let θ be a solution for $\rho\{E\}$ and E be nonempty. Suppose none of the rules is applicable. Since there is a solution, E does not contain equations of the form $a \equiv b$, $a \equiv (s, t)$, or $(s, t) \equiv a$ where a is either an atom or a designator. Furthermore, since the tautology and splitting rule are not applicable, all equations in E must have form $x \equiv (s, t)$ or $(s, t) \equiv x$. Without loss of generality let $x \equiv (s, t)$ be in E. Since the substitution rule is not applicable we have $x \notin \mathbf{D}(\rho)$. Since the binding rule is not applicable, we have $x \in \mathbf{V}(\rho s)$ without loss of generality. Thus we have $\theta x = \theta(s, t) = (\theta s, \theta t)$. Consequently $|\theta x| > |\theta s| = |\theta\rho s|$ since θ subsumes ρ. Since $x \in \mathbf{V}(\rho s)$ we have $|\theta\rho s| \geq |\theta x|$, which is a contradiction. $\square$

The next theorem states that the unification rules are sound, that is, the set of solutions is invariant under reduction with a unification rule.

Theorem 7.11. Let $\rho\{E\}$ $_{\mathrm{u}}\!\rightarrow$ $\rho'\{E'\}$. Then θ is a solution of $\rho\{E\}$ if and only if θ is a solution of $\rho'\{E'\}$.

Proof. We have to prove the claim for each except the failure rule. For the tautology and the splitting rule the claim is obvious. Since the left and the right version of the substitution and binding rules are symmetric, it suffices to prove the claim for the left substitution and the left binding rule.

Left substitution rule, left to right. Let θ be a solution of $\rho\{x \equiv s, E\}$. It suffices to prove that $\theta\rho x = \theta s$. This is the case since θ subsumes ρ and θ solves $x \equiv s$.

Left substitution rule, right to left. Let θ be a solution of $\rho\{\rho x \equiv s, E\}$. It suffices to prove that $\theta x = \theta s$. This is the case since θ subsumes ρ and θ solves $\rho x \equiv s$.

Left binding rule, left to right. Let θ solve $\rho\{x \equiv s, E\}$. It suffices to prove that θ subsumes $\{x \equiv \rho s\}\rho$. Since θ subsumes ρ this is the case if θ subsumes $\theta\{x \equiv \rho s\}$. Let y be a variable. If y is different from x, $\theta y = \theta\{x \equiv \rho s\}y$ obviously holds. Otherwise we have $\theta x = \theta s = \theta\rho s = \theta\{x \equiv \rho s\}x$ since θ solves $x \equiv s$ and subsumes ρ.

Left binding rule, right to left. Let θ solve $\{x \equiv \rho s\}\rho\ \{E\}$ and $x \notin \mathbf{D}(\rho)$. We have to show that θ subsumes ρ and solves $x \equiv s$. We have $\theta = \theta\{x \equiv \rho s\}\rho = \theta\{x \equiv \rho s\}\rho\rho = \theta\rho$ since θ subsumes $\{x \equiv \rho s\}\rho$ and ρ is idempotent. Thus θ subsumes ρ. Furthermore, $\theta x = \theta\{x \equiv \rho s\}\rho x = \theta\{x \equiv \rho s\}x = \theta\rho s = \theta s$ since θ subsumes $\{x \equiv \rho s\}\rho$, $x \notin \mathbf{D}(\rho)$, and θ subsumes ρ. Thus θ solves $x \equiv s$. $\square$

Let Θ be a set of substitutions. Then a substitution $\theta \in \Theta$ is *most general for* Θ if θ is subsumed by every substitution in Θ. It is easy to see that most general substitutions are idempotent and all most general substitutions for Θ are equivalent. The following theorem tells us that if a unification problem can be solved, then the unification rules will solve it and produce a most general solution.

Theorem 7.12. A unification problem $\rho\{E\}$ has a solution if and only if there is a ρ' such that $\rho\{E\}\ {}_{\mathrm{u}}{\to}^{*}\ \rho'$; and $\rho\{E\}\ {}_{\mathrm{u}}{\to}^{*}\ \square$ if and only if $\rho\{E\}$ has no solution. Furthermore, if $\rho\{E\}\ {}_{\mathrm{u}}{\to}^{*}\ \rho'$ then ρ' is a most general solution of $\rho\{E\}$ and satisfies $\mathbf{V}(\rho') \subset \mathbf{V}(\rho) \cup \mathbf{V}(\mathrm{E})$.

Proof. The first and second claim follow by induction on the length of reduction sequences by lemma 7.10 (no dead ends) and theorems 7.9 (no infinite reduction sequences) and 7.11 (solutions are invariant under reductions). Note that a solved problem $\rho\{\emptyset\}$ is solved by ρ. Furthermore, recall that θ is a solution of $\rho\{\emptyset\}$ if and only if θ subsumes ρ. Thus theorem 7.11 yields the third claim by induction on the length of reduction sequences. The last claim is part of lemma 7.8. $\square$

The last step in the reduction of pairs is the unification of the component term environments. Term environment will be idempotent substitutions and can thus be regarded as solved equation systems. We call two substitutions *compatible* if they are subsumed by a common substitution.

Corollary 7.13. Let θ and ψ be substitutions. Then there exists a substitution that subsumes θ and ψ if and only if there exists ρ such that $\varepsilon\{\theta \cup \psi\}\ {}_{u}{\to}^{*}\ \rho$; if ρ exists it is a most general substitution that subsumes θ and ψ.

Proof. From proposition 7.1 we know that θ subsumes ψ if and only if θ solves ψ's equation system. Thus the claim follows immediately from theorem 7.12. □

The reader may consult Siekmann [31] for an overview of the current state of first-order unification theory. Unification was first studied by Herbrand [11], who gave the first unification algorithm. But unification only became of real importance with the advent of automatic theorem provers, and unification algorithms were independently rediscovered by Robinson [28], Guard et al. [10] and Knuth and Bendix [16]. Fast unification algorithms exploit a structure sharing representation for terms and were devised by Huet [13], Martelli and Montanari [21], and Patterson and Wegman [24]. Patterson and Wegman's algorithm is linear-time. There is some work on higher-order unification by Huet [12, 13]. Goldfarb [8] shows that second-order unification is undecidable by reducing Hilbert's tenth problem to it. Thus programming languages must use first-order unification.

8 Reduction Tree Semantics of Fresh

In FM, a success reduction has the form

$$\delta\,\rho\{e\} \to \delta'\{u\}$$

where δ and δ' are function environments, ρ is a term environment, e is the expression to be reduced, and u is the result. An interpreter would begin with the initial configuration $\delta\,\rho\{e\}$ and compute the final configuration $\delta'\{u\}$. The final function environment is an extension of the initial one and contains the functions that were created during the reduction process. The term environment is a ground substitution, that is, it binds variables to ground terms.

In Fresh, free variables are first-class objects. The term environment can bind variables to nonground terms and the reduction process can produce delayed bindings for variables that occur in the values of other variables. Delayed bindings are propagated to the outer context. For instance, delayed bindings produced for the condition part of a conditional are available during the reduction of the then-part. To propagate delayed bindings, reductions for Fresh employ a final term environment ρ',

$$\delta\, \rho\, \{ e \} \rightarrow \delta'\, \rho'\, \{ u \}$$

which is obtained from the initial one by including delayed bindings. Thus the final term environment will subsume the initial one, that is, $\rho' = \rho'\rho$. Subsumption captures what we described as refinement or narrowing of the term environment. The fact that the final term environment subsumes the initial one implies $\rho'x = \rho'\rho x$ for all variables x, that is, the final value $\rho'x$ is an instance of the initial value ρx. This means that bindings cannot be arbitrarily changed but only be refined. Once a variable is bound to a ground term (always the case in FM), its value is final and cannot be changed anymore.

Since reduced arguments in Fresh can contain variables, the variables in a closure must be renamed to new ones upon application to avoid variable clashes. New variables can be exported to the context of an application since they may be included in the result or the refined term environment. Thus being a "new variable" is a property that cannot be decided in the local context of an application but must be defined globally. To solve this problem, we will employ reductions of the form

$$\delta\, \rho\, \{ e \} \rightarrow \delta'\, \rho'\, \{ u \}\, V$$

where V is the set of all variables that are newly introduced in the reduction tree justifying the reduction. The rules that combine reductions will come with conditions that prevent variable clashes.

With these extensions, FU's reduction rules can be easily obtained from the rules for FM. However, to obtain the rules for Fresh, we also have to account for multiple results. Intuitively, when building a reduction tree, we have at every disjunction the choice of proceeding with either the left or the right side. We will make the reduction process deterministic by equipping initial configurations with a choice string that contains the necessary decisions a priori. Thus, the final form of reductions for Fresh is

$$\delta\,\rho\,\{\,e\,\}\,\xi \;\rightarrow\; \delta'\,\rho'\,\{\,u\,\}\,V$$

where the choice string ξ is a string over the characters L (left) and R (right). The semantics of disjunctions is then captured by the rules

$$\frac{\delta\,\rho\,\{\,e\,|\,f\,\}\,\mathsf{L}\xi \;\rightarrow\; C}{\delta\,\rho\,\{\,e\,\}\,\xi \;\rightarrow\; C} \qquad \frac{\delta\,\rho\,\{\,e\,|\,f\,\}\,\mathsf{R}\xi \;\rightarrow\; C}{\delta\,\rho\,\{\,f\,\}\,\xi \;\rightarrow\; C} \quad .$$

We now begin the formalization of Fresh's semantics. A *choice string* is a string over the characters L and R. The empty choice string is denoted by ε and $\xi\xi'$ denotes the concatenation of two choice strings ξ and ξ'. The relation $\xi < \xi'$ (read "ξ is immediately before ξ'") is defined by

$$\xi < \xi' \;:\Leftrightarrow\; \exists\,\zeta.\;\; \xi = \zeta\mathsf{L}\mathsf{R}\cdots\mathsf{R} \;\wedge\; \xi' = \zeta\mathsf{R}\mathsf{L}\cdots\mathsf{L}$$

where ζ, $\mathsf{R}\cdots\mathsf{R}$ and $\mathsf{L}\cdots\mathsf{L}$ can be empty strings. This relation is used to formalize chronological backtracking. A sequence $\xi_1, \ldots, \xi_n$ of choice strings is a *chain* if $\xi_1 = \mathsf{L}\cdots\mathsf{L}$ and $\xi_1 < \cdots < \xi_n$. Informally, a chain records in chronological order all choice strings beginning with the first possible one up to ξ_n. A chain $\xi_1, \ldots, \xi_n$ is *maximal* if $\xi_n = \mathsf{R}\cdots\mathsf{R}$. A maximal chain contains all possible choice strings for an expression in chronological order.

As for FM, we first define a general form of reductions and proof then that every reduction in a reduction tree for a consistent initial configuration is consistent. *Term environments* are substitutions and *function environments* are finite maps from designators to closures. The *domain* of a function environment δ is denoted by $\mathbf{D}(\delta)$. *Closures* consist of a term environment and an abstraction. An *initial configuration* has form $\delta\;\rho\;\{\;e\;\}\;\xi$ where δ is a function environment, ρ is a term environment, e is the expression to be reduced, and ξ is a choice string. A *final configuration* is either the *failure configuration* $\Box$ or has the form $\delta\;\rho\;\{\;u\;\}\;V$ where δ is a function environment, ρ is a term environment, u is a term (the result), and V is a set of variables. A *reduction* has the form $I \rightarrow F$ where I is an initial and F is a final configuration.

Atoms and Designators

$$\frac{\delta\,\rho\,\{\,a\,\}\,\varepsilon \;\rightarrow\; \delta\,\rho\,\{\,a\,\}\varnothing}{} \qquad\qquad \frac{\delta\,\rho\,\{\,d\,\}\,\varepsilon \;\rightarrow\; \delta\,\rho\,\{\,d\,\}\varnothing}{}$$

Variables

$$\frac{\delta\,\rho\,\{\,x\,\}\,\varepsilon \;\rightarrow\; \delta\,\rho\,\{\,\rho x\,\}\varnothing}{}$$

Pairs

$$\frac{\delta\,\rho\,\{\,(e,f)\,\}\,\xi_1\xi_2 \;\rightarrow\; (\delta_1\cup\delta_2)\,\rho_3\,\{\,\rho_3(u,v)\,\}\,V_1\cup V_2}{\begin{array}{l}\delta\,\rho\,\{\,e\,\}\,\xi_1 \;\rightarrow\; \delta_1\,\rho_1\,\{\,u\,\}V_1\\ \delta\,\rho\,\{\,f\,\}\,\xi_2 \;\rightarrow\; \delta_2\,\rho_2\{\,v\,\}\,V_2\end{array}}$$

iff $\rho_1\{\,\rho_2\}\ {}_{\mathrm{u}}\!\!\rightarrow^*\ \rho_3$
and $\mathbf{D}(\delta_1)\cap\mathbf{D}(\delta_2)\subset\mathbf{D}(\delta)$
and V_1, V_2 and $\mathbf{V}((e,f))$
are pairwise disjoint

$$\frac{\delta\,\rho\,\{\,(e,f)\,\}\,\xi_1\xi_2 \;\rightarrow\; \square}{\begin{array}{l}\delta\,\rho\,\{\,e\,\}\,\xi_1 \;\rightarrow\; \delta_1\,\rho_1\,\{\,u\,\}V_1\\ \delta\,\rho\,\{\,f\,\}\,\xi_2 \;\rightarrow\; \delta_2\,\rho_2\{\,v\,\}\,V_2\end{array}}$$

iff $\rho_1\{\,\rho_2\}\ {}_{\mathrm{u}}\!\!\rightarrow^*\ \square$
and V_1, V_2 and $\mathbf{V}((e,f))$ are
pairwise disjoint

$$\frac{\delta\,\rho\,\{\,(e,f)\,\}\,\xi_1\xi_2 \;\rightarrow\; \square}{\begin{array}{l}\delta\,\rho\,\{\,e\,\}\,\xi_1 \;\rightarrow\; \square\\ \delta\,\rho\,\{\,f\,\}\,\xi_2 \;\rightarrow\; F\end{array}} \qquad\qquad \frac{\delta\,\rho\,\{\,(e,f)\,\}\,\xi_1\xi_2 \;\rightarrow\; \square}{\begin{array}{l}\delta\,\rho\,\{\,e\,\}\,\xi_1 \;\rightarrow\; F\\ \delta\,\rho\,\{\,f\,\}\,\xi_2 \;\rightarrow\; \square\end{array}}$$

As in FM, the components of a pair are reduced independently. Then the produced function and term environments are combined. The additional failure rule handles the case where the unification of the term environments fails. The expression $(X \equiv 1,\ X \equiv 4)$ is an example for this case. The condition $\rho_1\,\{\rho_2\,\}\ {}_{\mathrm{u}}\!\!\rightarrow^*\ \square$ is equivalent to $\rho_2\{\rho_1\,\}\ {}_{\mathrm{u}}\!\!\rightarrow^*\ \square$ and $\varepsilon\,\{\,\rho_1\cup\rho_2\,\}\ {}_{\mathrm{u}}\!\!\rightarrow^*\ \square$ (ε is the empty substitution). The unified term environment must be applied to the result of both components since they can contain variables that are bound by new bindings in the unified term environment. The expression (X ≡ (1, Y), X ≡ (Z, 4)) is an example that was discussed in section 5. The conditions for designators and variables ensure that there are no clashes with new designators and variables.

Soft Conditional

$$\frac{\delta\,\rho\,\{\,e \Rightarrow f\,;\,g\,\}\,\xi_1\xi_2 \;\rightarrow\; \delta_2\,\rho_2\,\{\,v\,\}\,V_1 \cup V_2}{\begin{array}{l}\delta\,\rho\,\{\,e\,\}\,\xi_1 \;\rightarrow\; \delta_1\,\rho_1\,\{\,u\,\}\,V_1\\ \delta_1\,\rho_1\,\{\,f\,\}\,\xi_2 \;\rightarrow\; \delta_2\,\rho_2\,\{\,v\,\}\,V_2\end{array}} \qquad \text{iff } V_1, V_2 \text{ and } \mathbf{V}((e,f)) \text{ are pairwise disjoint}$$

$$\frac{\delta\,\rho\,\{\,e \Rightarrow f\,;\,g\,\}\,\xi_1\xi_2 \;\rightarrow\; \square}{\begin{array}{l}\delta\,\rho\,\{\,e\,\}\,\xi_1 \;\rightarrow\; \delta_1\,\rho_1\,\{\,u\,\}\,V_1\\ \delta_1\,\rho_1\,\{\,f\,\}\,\xi_2 \;\rightarrow\; \square\end{array}} \qquad \text{iff } V_1 \text{ and } \mathbf{V}(f) \text{ are disjoint}$$

$$\frac{\delta\,\rho\,\{\,e \Rightarrow f\,;\,g\,\}\,\xi \;\rightarrow\; F}{\begin{array}{l}\delta\,\rho\,\{\,e\,\} \;\rightarrow^{a}\; \square\\ \delta\,\rho\,\{\,f\,\}\,\xi \;\rightarrow\; F\end{array}}$$

The expression $\delta\,\rho\,\{\,e\,\} \rightarrow^{a} \square$ is an abbreviation for a nonempty sequence

$$\delta\,\rho\,\{\,e\,\}\,\xi_1 \rightarrow \square\;, \ldots,\; \delta\,\rho\,\{\,e\,\}\,\xi_n \rightarrow \square$$

of reductions such that $\xi_1, \ldots, \xi_n$ is a maximal chain. Thus the third rule says that the else-part is only reduced if the condition part has no success result.

Abstraction

$$\frac{\delta\,\rho\,\{\,s \blacktriangleright e\,\}\,\varepsilon \;\rightarrow\; \delta[d \leftarrow (\rho', s \blacktriangleright e)]\,\rho\,\{\,d\,\}\,\varnothing}{} \qquad \text{iff } d \notin \mathbf{D}(\delta) \text{ and } \rho' = \rho|_{\mathbf{V}(s \blacktriangleright e)}$$

Once an abstraction is reduced to a closure, the closure is not affected by further variable bindings. For instance, the expression X ≡ (1,Y) → V ≡ (Z ▸X) → Y ≡ 5 → (V, X) reduces to $(d, (1, 5))$ where d is bound to a closure with term environment X ≡ (1,Y) and abstraction Z▸X. Thus side-effects do not apply to closures.

Technically, this is easily achieved since closures are kept in a separate environment and new bindings only affect the term environment. This is one reason for having designators.

Application

$$\begin{array}{l}
\delta\,\rho\,\{\,f[e]\,\}\,\xi_1\xi_2 \;\rightarrow\; \delta_2\,\rho_4\,\{\,v\,\}\,V_1 \cup V_2 \cup V_3 \\
\hline
\delta\,\rho\,\{\,(f,e)\,\}\,\xi_1 \;\rightarrow\; \delta_1\,\rho_1\,\{\,(d,u)\,\}\,V_1 \\
\delta_1\,\rho_3\,\{\,g\,\}\,\xi_2 \;\rightarrow\; \delta_2\,\rho_4\,\{\,v\,\}\,V_3
\end{array}
\qquad
\begin{array}{l}
\text{iff } \rho_1\rho_2\,\{\,s \equiv u\,\}\ {}_{\mathrm{u}}{\rightarrow^*}\ \rho_3 \\
\text{and } (\rho_2,\ s{\blacktriangleright}g) \in \mathbf{VAR}(\delta_1(d)) \\
\text{and } V_2 = \mathbf{V}(\rho_2) \cup \mathbf{V}(s{\blacktriangleright}g)
\end{array}$$

and V_1, V_2, V_3 are pairwise disjoint
and V_2 and $\mathbf{V}(\rho_1) \cup \mathbf{V}(u)$ are disjoint

$$\begin{array}{l}
\delta\,\rho\,\{\,f[e]\,\}\,\xi_1\xi_2 \;\rightarrow\; \square \\
\hline
\delta\,\rho\,\{\,(f,e)\,\}\,\xi_1 \;\rightarrow\; \delta_1\,\rho_1\,\{\,(d,u)\,\}\,V_1 \\
\delta_1\,\rho_3\,\{\,g\,\}\,\xi_2 \;\rightarrow\; \square
\end{array}
\qquad
\begin{array}{l}
\text{iff } \rho_1\rho_2\,\{\,s \equiv u\,\}\ {}_{\mathrm{u}}{\rightarrow^*}\ \rho_3 \\
\text{and } (\rho_2,\ s{\blacktriangleright}g) \in \mathbf{VAR}(\delta_1(d))
\end{array}$$

and $\mathbf{V}(\rho_2) \cup \mathbf{V}(s{\blacktriangleright}g)$ and $\mathbf{V}(\rho_1) \cup \mathbf{V}(u)$ are disjoint

$$\begin{array}{l}
\delta\,\rho\,\{\,f[e]\,\}\,\xi \;\rightarrow\; \square \\
\hline
\delta\,\rho\,\{\,(f,e)\,\}\,\xi \;\rightarrow\; \delta_1\,\rho_1\,\{\,(d,u)\,\}\,V_1
\end{array}
\qquad
\begin{array}{l}
\text{iff } \rho_1\rho_2\,\{\,s \equiv u\,\}\ {}_{\mathrm{u}}{\rightarrow^*}\ \square \\
\text{and } (\rho_2,\ s{\blacktriangleright}g) \in \mathbf{VAR}(\delta_1(d))
\end{array}$$

and $\mathbf{V}(\rho_2) \cup \mathbf{V}(s{\blacktriangleright}g)$ and $\mathbf{V}(\rho_1) \cup \mathbf{V}(u)$ are disjoint

$$\begin{array}{l}
\delta\,\rho\,\{\,f[e]\,\}\,\xi \;\rightarrow\; \square \\
\hline
\delta\,\rho\,\{\,(f,e)\,\}\,\xi \;\rightarrow\; \delta_1\,\rho_1\,\{\,(v,u)\,\}\,V_1
\end{array}
\qquad
\text{iff } v \text{ is not a designator}$$

$$\begin{array}{l}
\delta\,\rho\,\{\,f[e]\,\}\,\xi \;\rightarrow\; \square \\
\hline
\delta\,\rho\,\{\,(f,e)\,\}\,\xi \;\rightarrow\; \square
\end{array}$$

$\mathbf{VAR}(\delta_1(d))$ is the set of all *variants* of the closure $\delta_1(d)$. A variant of $\delta_1(d)$ is obtained by renaming the variables in $\delta_1(d)$ consistently.

Disjunction

$$\frac{\delta\,\rho\,\{\,e\,|\,f\,\}\,\mathsf{L}\xi \;\rightarrow\; F}{\delta\,\rho\,\{\,e\,\}\,\xi \;\rightarrow\; F} \qquad \frac{\delta\,\rho\,\{\,e\,|\,f\,\}\,\mathsf{R}\xi \;\rightarrow\; F}{\delta\,\rho\,\{f\}\,\xi \;\rightarrow\; F}$$

Confinement

$$\frac{\delta\,\rho\,\{\,!e\,\}\,\varepsilon \;\rightarrow\; F}{\delta\,\rho\,\{\,e\,\} \;\rightarrow^{1}\; F} \qquad \frac{\delta\,\rho\,\{\,!e\,\}\,\varepsilon \;\rightarrow\; \Box}{\delta\,\rho\,\{\,e\,\} \;\rightarrow^{a}\; \Box}$$

The abbreviation $\delta\,\rho\,\{\,e\,\} \rightarrow^{a} \Box$ was explained with the conditional rules. The expression $\delta\,\rho\,\{\,e\,\} \rightarrow^{1} F$ abbreviates a nonempty sequence

$$\delta\,\rho\,\{\,e\,\}\,\xi_1 \rightarrow \Box \;,\ldots,\; \delta\,\rho\,\{\,e\,\}\,\xi_{n-1} \rightarrow \Box\;,\; \delta\,\rho\,\{\,e\,\}\,\xi_n \rightarrow F$$

of reductions such that $\xi_1, \ldots, \xi_n$ is a chain. Thus F is the first success result under chronological backtracking.

Collection

$$\frac{\delta\,\rho\,\{\,\{e\}\,\}\,\varepsilon \;\rightarrow\; (\delta_1 \cup \cdots \cup \delta_n)\,\rho\,\{\,(u'_1,\ldots,u'_n)\,\}\,\mathbf{V}((u'_1,\ldots,u'_n))}{\delta\,\rho\,\{\,e\,|\,()\,\} \;\rightarrow^{a}\; \delta_1\,\{u_1\},\ldots,\delta_n\,\{u_n\}}$$

$$\begin{aligned} \text{iff}\quad \forall\, i,j. \qquad & i \neq j \;\Rightarrow\; \mathbf{D}(\delta_i) \cap \mathbf{D}(\delta_j) \subset \mathbf{D}(\delta) \\ & \wedge\;\; i \neq j \;\Rightarrow\; \mathbf{V}(u'_i) \cap \mathbf{V}(u'_j) = \varnothing \\ & \wedge\;\; u'_i \in \mathbf{VAR}(u_i) \;\;\wedge\;\; \mathbf{V}(u'_i) \cap \mathbf{V}(\rho) = \varnothing \end{aligned}$$

The expression $\delta\,\rho\,\{f\} \rightarrow^{a} \delta_1\,\{u_1\},\ldots,\delta_n\,\{u_n\}$ abbreviates a sequence of reductions for $\delta\,\rho\,\{f\}$ such that their choice strings form a maximal chain and δ_i and u_i are the function environments and results of the success results. $\mathbf{VAR}(u)$ is the set of all variants of u where a *variant* of u is obtained by renaming the variables in u consistently. All elements of the result list are renamed such that they do not share variables with each other or the final term environment. A collection has no side effects since the final configuration inherits the initial term environment.

Reduction trees are defined as for FM. The next proposition says that the existence of a reduction tree does not depend on the choice of designators and variables.

Proposition 8.1. Let R be a reduction. Then R has a reduction tree if and only if every variant of R (under consistent designator and variable renaming) has a reduction tree.

Since initial configurations carry a choice string that determines which side of a disjunction is reduced, Fresh's reduction rules are deterministic. Thus we have:

Theorem 8.2. All reduction trees for an initial configuration are equal up to consistent designator and variable renaming.

We now define consistency of function environments, configurations and reductions. A term environment ρ is *closed* under a function environment δ if for all $x \in \mathbf{D}(\rho)$ all designators in ρx are in $\mathbf{D}(\delta)$. A function environment δ is *consistent* if for all $d \in \mathbf{D}(\delta)$ such that $\delta(d) = (\rho,\ s \blacktriangleright e)$ the closure environment ρ and the abstraction body e are closed under δ and $\mathbf{D}(\rho) \subset \mathbf{V}(s \blacktriangleright e)$. An initial configuration $\delta\, \rho \{ e \} \xi$ is *consistent* if its function environment is consistent, its term environment and its expression are closed under its function environment, and its term environment is idempotent. A final configuration $\delta\, \rho \{ u \}\, V$ is *consistent* if its function environment is consistent, its term environment and its result are closed under its function environment, and its term environment is idempotent. A reduction $\delta\, \rho \{ e \} \xi \rightarrow \delta'\, \rho'$ $\{ u \}\, V$ is *consistent* if its initial and final configuration are consistent, the final function environment is an extension of the initial function environment (that is, $\delta = \delta'|_{\mathbf{D}(\delta)}$), the final term environment subsumes the initial term environment (that is, $\rho' = \rho'\rho$), and V and $\mathbf{V}(\rho) \cup \mathbf{V}(e)$ are disjoint. A failure reduction $I \rightarrow \square$ is consistent if its initial configuration is consistent.

Theorem 8.3. Every reduction in every reduction tree for a consistent initial configuration is consistent.

8 Conclusion and Related Work

Fresh is an attempt to reformulate Prolog's computational innovations in the framework of higher-order functional programming. The paper shows that a smooth integration without syntactic or semantic overhead is possible. Since Fresh contains Horn logic with equality, it can be used like a logic programming language. On the other hand, Fresh offers all the advantages of functional programming. Higher-order functions and predicates replace Prolog's obscure assert and call constructs. Information flow can be expressed explicitly by functions and nesting of expressions rather than by delayed bindings and auxiliary variables.

. I am currently working on a polymorphic type discipline for Fresh. The type system will distinguish between ground and nonground types. Ground types are types whose elements are ground terms, while elements of nonground terms can contain variables. Typed Fresh will provide for many compile-time optimizations. For instance, the distinction between ground and nonground types makes it possible to determine at compile-time which instances of unification can be implemented by matching. There has been very little work on type systems for unification-based languages. Mycroft and O'Keefe [23] have adapted ML's type system to a subset of Prolog, but their type discipline does not distinguish between ground and nonground types.

The integration of lazy reduction into Fresh seems promising. This would allow to apply collection to expressions with infinite result sequences and would thus interface backtracking more satisfying. Subrahmanyam and You [32] discuss the combination of unification and lazy reduction. Another interesting research topic is the integration of Concurrent Prolog's [29, 30] communication and synchronization mechanisms. Since these mechanisms depend mainly on logical variables and unification, an adaption to Fresh seems straightforward.

Van Emden and Kowalski [34] and Apt and Van Emden [1] provide several precise characterizations of the semantics of Horn clauses. However, these semantics do not apply to Prolog since they do not cover backtracking, the cut or collection. Jones and Mycroft [15] define an operational and a denotational semantics for a subset of Prolog that includes the cut but excludes the collection, assert and call constructs.

There are many recent papers on the integration of functional and logic programming. One direction was initiated by Kornfeld [17] and is based on Horn logic with equality. In this framework first-order functions can be defined by equality axioms. Goguen and Meseguer's Eqlog [9] extends this approach by employing a many-sorted logic. Eqlog comes with a complete and simple model theoretic semantics and thus is truly a logic programming language. However, this is only possible since Eqlog does not include critical features like negation, higher-order objects, collection or control constructs. Work at Utah emphasizes the incorporation of unification into functional programming. Lindstrom [19] shows how to incorporate logical variables into FEL, a functional language featuring lazy evaluation, and gives an implementation technique suitable for reduction architectures. Reddy [27] discusses a first-order functional language, which generalizes reduction to narrowing or resolution. He gives a denotational semantics, examines the extension to lazy narrowing, and discusses a parallel implementation. Reddy's language does not include multiple results.

Acknowledgements

The paper benefited from discussions with Alan Demers and James Hook.

References

1. Apt, K. R. and M. H. Van Emden. Contributions to the Theory of Logic Programming. *Journal of the ACM 29.3* (1982), 841-862.

2. Burstall, R. M., D. B. MacQueen and D. T. Sanella. HOPE: an Experimental Applicative Language. *Lisp Conference*, ACM, 1980, 136-143.

3. Clark, K. L. Negation as Failure. In *Logic and Data Bases*, H. Gallaire and J. Minker, Eds., Plenum Press, 1978, 293-322.

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

5. Colmerauer, A. Prolog and Infinite Trees. In K. L. Clark and S.-A. Tärnlund, Eds., *Logic Programming*, Academic Press, 1982.

6. Colmerauer, A, H. Kanoui, M. Van Caneghem. Prolog, Theoretical Principles and Current Trends. *Technology and Science of Informatics*, 2, 4 (1983), 255-292.

7. Damas, L. and R. Milner. Principal Type-Schemes for Functional Programs. *Proc. Principles of Programming Languages*, ACM, 1982, 207-212.

8. Goldfarb, D. The Undecidability of the Second Order Unification Problem. *Journal of Theoretical Computer Science*, 13 (1981), 225-230.

9. Goguen, J. A. and J. Meseguer. Eqlog: Equality, Types, and Generic Modules for Logic Programming. In D. DeGroot, and G. Lindstrom, Eds., *Functional and Logic Programming*, Prentice Hall, 1985.

10. Guard, J. R., F. C. Oglesby, J. H. Benneth and L. G. Settle. Semi-Automated Mathematics. *Journal of the ACM*, 18, 1 (1969).

11. Herbrand, J. Recherches sur la Theorie de la Demonstration. In J. Van Heijenoort, Ed., *From Frege to Gödel: a Source Book in Mathematical Logic, 1879-1931*, Harvard University Press, 1967.

12. Huet, G. Unification in Typed Lambda Calculus. *Proc. Symp. on λ-Calculus and Computer Science*, Springer LNCS 37, 1975, 192-212.

13. Huet, G. Resolution d'Equations dans les Languages d'Ordre 1, 2, ... , ω. These d'Etat, Specialite Mathematiques, Universite Paris VII, 1976.

14. Huet, G. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems. *Journal of the ACM*, 27, 4 (1980), 797-821.

15. Jones, N. D. and A. Mycroft. Stepwise Development of Operational and Denotational Semantics for Prolog. *First Int. Symp. on Logic Programming*, IEEE, 1984, 281-288.

16. Knuth, D. and P. Bendix. Simple Word Problems in Universal Algebras. In J. Leech, Ed., *Computational Problems in Abstract Algebras*, Pergamon Press, 1970, 263-297.

17. Kornfeld, W. A. Equality for Prolog. *Proc. 7th Int. Joint Conf. on Artificial Intelligence*, 1983, 514-519.

18. Kowalski, R. A. Algorithm = Logic + Control. *Communications of the ACM*, 22, 7 (1979), 424-436.

19. Lindstrom, G. Functional Programming and the Logical Variable. *Proc. Principles of Programming Languages*, ACM, 1985, 266-280.

20. Lloyd, J. W. Foundations of Logic Programming. Springer Verlag, 1984.

21. Martelli, A. and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems* 4, 2 (1982), 258-282.

22. Milner, R. A Proposal for Standard ML. *Proc. Symp. on Lisp and Functional Programming*, 1984, 184-197.

23. Mycroft, A. and R. A. O'Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence 15* (1984).

24. Paterson, M. S. and M. N. Wegman. Linear Unifcation. *Journal of Computer and System Science*, 16 (1978), 158-167.

25. Plaisted, D. A. The Occur-Check Problem in Prolog. *First Int. Symp. on Logic Programming*, IEEE, 1984, 272-280.

26. Plotkin, G.D. A Structural Approach to Operational Semantics. DAIMI FN-19, Comp. Sc. Dept., Aarhus University, Denmark, 1981.

27. Reddy, U. Narrowing as the Operational Semantics of Functional Languages. *Second Int. Symp. on Logic Programming*, IEEE, 1985.

28. Robinson, J. A. A Machine-Oriented Logic based on the Resolution Principle. *Journal of the ACM*, 12, 1 (1965), 23-41.

29. Shapiro, E. Y. and A. Takeuchi. Object-Oriented Programming in Concurrent Prolog. *New Generation Computing*, 1,1 (1983), 25-48.

30. Shapiro, E. Systems Programming in Concurrent Prolog. *Proc. Principles of Programming Languages*, ACM, 1984, 93-105.

31. Siekmann, J. H. Universal Unification. *Proc 7th Int. Conf. on Automated Deduction*, Springer LNCS 170, 1984, 1-42.

32. Subrahmanyam, P. A. and J.-H. You. Pattern Driven Lazy Reduction: a Unifying Evaluation Mechanism for Functional an Logic Programs. *Proc. Principles of Programming Languages*, ACM, 1984, 228-234.

33. Turner, D. A. Recursion Equations as a Programming Language. In J. Darlington, P. Henderson and D. A. Turner, Eds., *Functional Programming and its Apllications*, Cambridge University Press, 1982.

34. Van Emden, M. H. and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23, 4 (1976), 733-742.

35. Van Emden, M. H. and J. W. Lloyd. A Logical Reconstruction of Prolog II. *Second Int. Logic Programming Conference*, 1984, 35-40.

36. Warren, D. Logic Programming and Compiler Writing. *Software—Practice and Experience*, 10 (1980), 97-125.