LOGIC PROGRAMMING WITH FUNCTIONS AND PREDICATES: THE LANGUAGE BABEL[*]

JUAN JOSE MORENO-NAVARRO AND MARIO RODRIGUEZ-ARTALEJO[†]

▷ We investigate the experimental programming language BABEL, designed to achieve integration of functional programming (as embodied in HOPE, Standard ML, or MIRANDA) and logic programming (as embodied in PROLOG) in a simple, flexible, and mathematically well-founded way. The language relies on a constructor discipline, well suited to accommodate PROLOG terms and HOPE-like patterns. From the syntactical point of view, BABEL combines pure PROLOG with a first order functional notation. On the other side, the language uses narrowing as the basis of a lazy reduction semantics which embodies both rewriting and SLD resolution and supports computation with potentially infinite data structures. There is also a declarative semantics, based on Scott domains, which provides a notion of least Herbrand model for BABEL programs. We develop both semantics and prove the existence of least Herbrand models, as well as a soundness result for the reduction semantics w.r.t. the declarative one. We also sketch a completeness result for the reduction semantics and illustrate the features of the language through some programming examples. ◁

1. INTRODUCTION

Interest, other than purely academic, in declarative (that is, functional and logical) programming languages has greatly increased since VLSI technology opened the realistic possibility of building parallel machines capable of executing declarative

Address correspondence to Juan José Moreno-Navarro, Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software, Facultad de Informática, UPM, Campus de Montegancedo s/n, E-28660 Madrid, Spain. E-mail: `jjmoreno@fi.upm.es`.

Accepted 9 November 1990.

[*]This work has been partially supported by "Acción Integrada Hispano Alemana" No. 35B, granted by "Dirección General de Investigación Científica y Técnica" and "Deutscher Akademischer Austauschdienst", and by the PRONTIC project TIC 89/0104.

[†]Sección Departamental de Informática y Automática, Facultad de C.C. Matemáticas, UCM, Avenida Complutense s/n, E 28040 Madrid. E-mail: `W450@EMDUCM11.BITNET`.

programs efficiently. Current progress in the improvement of the implementation techniques on conventional machines has also helped to arouse interest. Moreover, there are several intrinsic reasons to favor declarative languages. Being closer to the abstraction level of specifications, they enable one to write more concise programs, thereby increasing software productivity. Having a very clean mathematical semantics, they enjoy referential transparency and support powerful formal methods, such as transformation rules, to assist in software design and maintenance. Finally, they are especially well suited for rapid prototyping and applications to such fields as symbolic computation, artificial intelligence, and knowledge based systems.

Functional programming has an older tradition than logic programming. Its mathematical foundations (λ-calculus, equational logic, rewriting) are well understood (see, Barendregt [2], Huet and Oppen [26]); powerful compilation and implementation techniques have already been developed (see Peyton-Jones [45]); and excellent programming environments exist, for LISP—usually not considered a functional language in the strict sense—and also for more modern functional languages. For a survey on functional programming, see Hudak [24].

Logic programming is most commonly understood in a restricted sense, namely, programming with Horn clauses and SLD resolution; see, Lloyd [37]. There is for practical purposes only one working programming language, PROLOG, that can be considered as a significant realization of the logic programming paradigm. Efficient implementation techniques for PROLOG have also been investigated; see Campbell [7], Warren [57].

Neither PROLOG nor the functional languages enjoy all the benefits of declarative programming. PROLOG lacks such programming facilities as evaluable functions, types, higher order programming, and lazy evaluation, while functional languages lack the computing power provided by logical variables, unification, and deductive inference. During the last years, many attempts have been made to design declarative programming languages integrating the functional and logical paradigms. DeGroot and Lindstrom [11] collects significant papers on this field, while Bellia and Levi [3] analyzes and classifies the main existing approaches to the integration.

The existing proposals for integrated logic plus functional programming languages differ in their degree of mathematical rigor and semantic clarity. Our view is that the integration should be founded on a clean declarative semantics, related to the operational semantics through soundness and completeness results that allow one to interpret programs as theories, computations as deductions, and the whole programming language as a logical one which embodies the logic of functions, relations, and equality.

A key role for a semantically and mathematically well-founded amalgamation of the functional and logical paradigms is played by equational logic and the related notion of rewriting (Huet and Oppen [26]), as well as by narrowing, a combination of unification (Robinson [50], Lassez et al. [34]) and rewriting that originally arose in the context of theorem proving (Lankford [32], Slagle [55]), and that has been used to solve some problems of E-unification and matching (Fay [19], Hullot [27], Siekman and Szabo [54], Gallier and Snyder [21], Nutt et al. [43]) and is embedded in the operational semantics of several logic programming languages with an equational flavor (Dershowitz and Josephson [13], Dershowitz [12], Dershowitz and Plaisted [14], Josephson and Dershowitz [28], Fribourg [20], Goguen and Meseguer [23], Subrahmanyam and You [56]). Some of these languages use conditional rewrite rules (Kaplan [29], Remy and Zhang [49], Kaplan and Jouannoud [30]), while some others are based on equational Horn clauses.

Usually, languages of this kind use narrowing as a basis for E-unification algorithms (Giovannetti and Moiso [22], Dincbas and van Henternryck [16]), which in turn support evaluable functions and equality. Some approaches exploit the fact that narrowing can be simulated by SLD resolution (van Emden and Yukawa [18], Bosco et al. [6]) to gain facility for the implementation of logic + functional programming languages which have a modified form of SLD resolution as their single computation mechanism (Levi et al. [35], Bosco and Giovannetti [5]).

Other possible approaches to the integration of the functional and logical paradigms consist in enriching functional languages with logic variables and unification (Darlington et al. [8], Darlington and Guo [9, 10]) or in building so-called functional logic languages, which keep a functional syntax, but use narrowing as operational semantics (Reddy [46–48]).

The language BABEL that we present in this paper can be best described as a first order functional logic language in Reddy's sense. It is based on a constructor discipline (O'Donnell [44]) and works with only two elementary types (constructed terms and boolean values). Predicates are identified with boolean functions; this provides two truth values and enables the use of propositional connectives. In particular, there is a boolean negation that is certainly not the logical one (because of the undefined boolean value) but comes closer to it than in PROLOG. The operational semantics of the language uses a lazy version of narrowing as the single computation mechanism. SLD resolution (and hence pure PROLOG computations) are simulated by lazy narrowing, which supports also computations with infinite data structures (built from constructors) and lazy evaluation as in lazy functional languages. BABEL is also equipped with a computable approximation to equality, since the true identity between possibly infinite data becomes uncomputable. The declarative semantics of the language uses interpretations based on Scott domains (Scott [52]) and allows to prove the existence of least Herbrand models, as well as soundness and completeness theorems for the operational semantics. This is related to similar results for Horn clause logic programs (van Emden and Kowalski [17], Apt and van Emden [1]) as well as for other lazy logic + functional languages (Levi et al. [35]).

The design of BABEL has been specially influenced by the view of the operational semantics in Reddy [47] and the view of the declarative semantics in Levi et al. [35]. The most distinctive feature of our approach is perhaps the decision to take narrowing as the single computation mechanism and to handle SLD resolution as a particular kind of narrowing. Other researchers have advocated the simulation of narrowing by means of SLD resolution, in order to capitalize on the extensive experience already available for PROLOG implementations (van Emden and Yukawa [18], Bosco et al. [6], Levi et al. [35]). We have chosen the opposite view for similar reasons, namely, in order to take advantage of the available experience in implementation techniques for functional languages. An extension of BABEL with higher order functions and polymorphic typing has been implemented using a graph narrowing abstract machine (Kuchen et al. [31]) which was designed as an extension of the sequential kernel of a purely functional, parallel (programmed) graph reduction machine (Loogen et al. [36]) by unification and backtracking mechanisms inspired by Warren's abstract machine for PROLOG (Warren [57]). A prototype emulator of the abstract BABEL machine has been programmed in OCCAM and runs on transputer systems.* Presently, the BABEL abstract machine supports

*Currently, the emulator is programmed in C and runs on SUN workstations.

only sequential and eager evaluation. Our final aim is a parallel machine which works with lazy evaluation. Much work remains to be done, but our hope is that purely applicative programs will run on the BABEL machine almost as efficiently as in the original graph reduction machine, though some overhead due to the different parameter passing mechanism (unification instead of matching) cannot be avoided.

The rest of the paper is organized in the following way: In Section 2 we introduce BABEL's syntax for terms, expressions, rules, and programs. In Section 3 we illustrate the expressive power of the language by means of some simple examples, chosen to allow for comparison with other declarative languages. In Section 4 we define lazy narrowing, specify the operational semantics, and apply it to some computations related to the previous examples. In Section 5 we present BABEL's declarative semantics (including the existence of least Herbrand models for BABEL programs), prove a soundness theorem for the operational semantics, and sketch a completeness theorem which has been proved in Moreno-Navarro [41]. Finally, Section 6 summarizes our conclusions and refers to future work planned for improving the design and implementation of the language.

2. BABEL'S SYNTAX

We start with five disjoint sets of symbols:

Data variables	$X, Y, Z \in \mathrm{DV}$
Boolean variables	$X, Y, Z \in \mathrm{BV}$
Constructors	$c, d, e \in \mathrm{CS}$
Function symbols	$f, g, h \in \mathrm{FS}$
Predicate symbols	$p, q, r \in \mathrm{PS}$

We assume that DV and BV are countably infinite and fixed. Notice that we use the same metavariables X, Y, Z to range over DV and BV. We shall denote $\mathrm{DV} \cup \mathrm{BV}$ as VS. The set $\Sigma = \mathrm{CS} \cup \mathrm{FS} \cup \mathrm{PS}$ is called the signature and may change according to the program we consider. We assume that signatures are finite. Signature symbols are assumed to have associated arities. In concrete examples, we shall use identifiers starting with an uppercase (lowercase) letter for variables (signature symbols).

Definition 2.1. Data Σ-terms ($t \in \mathrm{DTerm}_\Sigma$), boolean Σ-terms ($b \in \mathrm{BTerm}_\Sigma$), data Σ-expressions ($E \in \mathrm{DExp}_\Sigma$), and boolean Σ-expressions ($B, C \in \mathrm{BExp}_\Sigma$) are defined as follows:

$t ::= X$	% data variable
$\quad \mid c$	% constant, i.e. nullary constructor
$\quad \mid c(t_1, \ldots, t_n)$	% construction
$b ::= X$	% boolean variable
$\quad \mid \mathrm{true}$	% truth
$\quad \mid \mathrm{false}$	% falsity
$E ::= t$	% data term
$\quad \mid c(E_1, \ldots, E_n)$	% constructor application
$\quad \mid f(E_1, \ldots, E_n)$	% function application
$\quad \mid (C \rightarrow E)$	% guarded expression
$\quad \mid (C \rightarrow E_1 \Box E_2)$	% conditional expression

$B ::= b$	% boolean term
$\mid p(E_1, \ldots, E_n)$	% predicate application
$\mid E_1 = E_2$	% weak equality
$\mid \neg B$	% negation
$\mid (B, C)$	% conjunction
$\mid (B; C)$	% disjunction
$\mid (C \rightarrow B)$	% guarded boolean expression
$\mid (C \rightarrow B_1 \Box B_2)$	% conditional boolean expression

The sets of Σ-terms and Σ-expressions are defined as $\text{Term}_\Sigma = \text{DTerm}_\Sigma \cup \text{BTerm}_\Sigma$ and $\text{Exp}_\Sigma = \text{DExp}_\Sigma \cup \text{Bexp}_\Sigma$, respectively. In the sequel, we reserve R, L, M, N for expressions and rely on the context to determine whether they are boolean or not.

The distinction between data expressions and boolean expressions (and accordingly, between functions and predicates) is the only type discipline in BABEL.

Constructors are well known in functional programming languages. They represent free functions and correspond to PROLOG's functors.

Expressions of the form	$C \rightarrow M$	$C \rightarrow M_1 \Box M_2$
are intended to mean	*if* C *then* M *else* undefined	*if* C *then* M_1 *else* M_2

respectively. A weak equality $E_1 = E_2$ is intended to hold iff the values of E_1 and E_2 are finite, defined, and identical. Since the language allows for infinite values, this means that weak equality is only an approximation of identity. In particular, it is not reflexive.

Definition 2.2. A BABEL rule has one of the two following forms;

(F) $f(t_1, \ldots, t_n) := \{C \rightarrow\}E.$ % function rule

(P) $p(t_1, \ldots, t_n) := \{C \rightarrow\}B.$ % predicate rule

where curly braces indicate that the presence of "$C \rightarrow$" is optional.

We shall use the following terminology:

(1) $f(t_1, \ldots, t_n)$ or $p(t_1, \ldots, t_n)$ is the rule's left hand side (lhs).

(2) $C \rightarrow E$ or $C \rightarrow B$ is the rule's right hand side (rhs).

(3) C is the rule's guard.

(4) E or B is the rule's body.

Any BABEL rule must satisfy two restrictions:

Restriction 2.1. (Left linearity). No variable is allowed to have multiple occurrences in the lhs.

Restriction 2.2. (Free variables). Any variable that occurs in a rhs and does not occur in the corresponding lhs is called free. Such free variables are allowed in guards, but not in bodies.

Assume a rule

$$k(t_1, \ldots, t_n) := \{C \rightarrow\} M.$$

where k is either a function or a predicate symbol. Let $X_1, \ldots, X_r$ be the variables in the lhs, and $Y_1, \ldots, Y_s$ be the free variables. The intended logical meaning of the rule is then

$$\forall X_1 \cdots \forall X_r \{\forall Y_1 \cdots \forall Y_s (C \Rightarrow\} k(t_1, \ldots, t_n) \equiv M\{)\}$$

or, equivalently,

$$\forall X_1 \cdots \forall X_r \{(\exists Y_1 \cdots \exists Y_s C \Rightarrow\} k(t_1, \ldots, t_n) \equiv M\{)\},$$

where $\equiv$ stands for semantical identity. A precise definition will be given in Section 5.

Notice that any BABEL rule can be written as $L := R$, where L is the lhs and R is the rhs. We shall use this notation later.

We also adopt a convention to present some special rules in a nicer form.

Convention 2.1. (PROLOG-like rules). The following "sugarings" are allowed:

Raw form	Sugared form
$p(t_1, \ldots, t_n) :=$ true.	$p(t_1, \ldots, t_n)$.
$p(t_1, \ldots, t_n) := C \rightarrow$ true.	$p(t_1, \ldots, t_n) :- C$.
$p(t_1, \ldots, t_n) :=$ false.	$\neg p(t_1, \ldots, t_n)$.
$p(t_1, \ldots, t_n) := C \rightarrow$ false.	$\neg p(t_1, \ldots, t_n) :- C$.

Some notational conventions in BABEL's syntax have been chosen for compatibility with PROLOG. This is the reason that we use "," and ";" for conjunction and disjunction, respectively. The sign " := " between the lhs and rhs of a rule was chosen because of its similarity with the neck sign " :– " in PROLOG clauses. Moreover, this sign emphasizes that a rule's lhs gets, by definition, the value of the corresponding rhs.

An important design decision is related to guarded expressions. In BABEL, guards are not intended to behave as in concurrent logic programming languages (Shapiro [53]). Their main role in the language is to allow for rules with conditional rhs. It is technically important that the guard in such a conditional rhs may have free variables which do not occur in the lhs. As shown in Convention 2.1 above, this facility may be used to mimic PROLOG clauses, whose body may have variables that do not occur in the head.

Before we define programs, we still need some auxiliary notions on boolean expressions.

Definition 2.3. The finite set PC(B) of the prime components of a given boolean expression is computed recursively:

$$\begin{aligned}
\mathrm{PC}(\mathrm{true}) = \mathrm{PC}(\mathrm{false}) &= \varnothing, \\
\mathrm{PC}(\neg B) &= \mathrm{PC}(B), \\
\mathrm{PC}(B_1, B_2) = \mathrm{PC}(B_1; B_2) &= \mathrm{PC}(B_1) \cup \mathrm{PC}(B_2), \\
\mathrm{PC}(C \rightarrow B) &= \mathrm{PC}(C) \cup \mathrm{PC}(B), \\
\mathrm{PC}(C \rightarrow B_1 \square B_2) &= \mathrm{PC}(C) \cup \mathrm{PC}(B_1) \cup \mathrm{PC}(B_2), \\
\mathrm{PC}(B) &= \{B\} \quad \text{in any other case.}
\end{aligned}$$

Definition 2.4. A boolean expression B is propositionally unsatisfiable iff the truth value of B with respect to any evaluation

$$\mathbf{V} : \mathrm{PC}(B) \rightarrow \{\text{true}, \text{false}, \perp_b\}$$

is either false or the ***undefined boolean value*** $\perp_b$. The truth value w.r.t. $\mathbf{V}$ must be computed according to the truth tables for the propositional connectives, given in Section 5. Notice that propositional satisfiability is a decidable property.

We are now in a position to define BABEL programs.

Definition 2.5. A BABEL program is any recursively enumerable set Π of BABEL rules that satisfies a nonambiguity restriction.

Restriction 2.3. (Nonambiguity). Given any two rules in Π for the same symbol k:

$$k(t_1, \ldots, t_n) := \{B \rightarrow\} M.$$
$$k(s_1, \ldots, s_n) := \{C \rightarrow\} N.$$

at least one of the following conditions must hold:

(a) No superposition: $k(t_1, \ldots, t_n)$ and $k(s_1, \ldots, s_n)$ are not unifiable.

(b) Fusion of bodies: $k(t_1, \ldots, t_n)$ and $k(s_1, \ldots, s_n)$ have a m.g.u. θ such that $M\theta$ and $N\theta$ become identical.

(c) Incompatibility of guards: $k(t_1, \ldots, t_n)$ and $k(s_1, \ldots, s_n)$ have a m.g.u. θ such that $(B, C)\theta$ is propositionally unsatisfiable.

When necessary, we shall speak of Σ-rules and Σ-programs, to make explicit the signature they are built from.

Our conditions on programs should be compared with those considered for rewriting systems by Huet and Levy [25]. These authors work with unconditional rewrite rules and impose linearity and nonoverlapping conditions on the left hand sides. They prove that such hypotheses ensure confluence even in the absence of termination. Our more liberal nonambiguity requirements will also allow us to obtain a kind of confluence result in Section 5.

We also would like to mention that our condition on incompatibility of guards should be replaced in practice by a more flexible one which takes the semantics of weak equality into account. According to our present definition, the two guards $E_1 = E_2$ and $\neg(E_2 = E_1)$ are not regarded as incompatible (unless E_1 and E_2 are syntactically identical).

3. PROGRAMMING IN BABEL

In this section we present some examples of simple BABEL programs in order to illustrate the expressive power of the language and to allow comparison with the programming style in PROLOG and in functional languages. The semantics and behavior of these programs will become more clear in the two following sections.

We present the programs together with declarations of the different symbols in their signatures. We also allow ourselves to use the commonly accepted sugarings for the syntax of natural numbers and lists, and write some function and predicate symbols in infix form, to improve legibility.

Example 3.1. (Appending lists). This program merely shows that pure PROLOG corresponds to a subset of BABEL. Notice that weak equality must be used to ensure left linearity, by adding equalities between variables to bodies of clauses (which are guards of BABEL rules in the raw form of the syntax):

```
constructors
  []/0       % empty list
  [·|·]/2    % list constructor
predicates
  append/3
rules
  /* A1 */   append([], Ys, Zs) :- Zs = Ys.
  /* A2 */   append([X | Xs], Ys, [Z | Zs]) :- Z = X, append(Xs, Ys, Zs).
```

This appending program admits multiple use, as in PROLOG; i.e., the arguments of append have no directionality.

Of course, append can also be programmed as a function in BABEL. The point is that the version just given behaves like the PROLOG append predicate. For this purpose, guards (with free variables) are necessary.

Example 3.2. (Testing binary trees for equality of frontiers). This program illustrates the cooperation of functions and predicates:

```
constructors
  []/0, [·|·]/2   % as above
  tip/1           % constructor of leaves
  node/2          % constructor of compound nodes
functions
  frontier/1
predicates
  equal-frontier/2
  equal-list/2
  equal-atom/2
rules
  /* EF  */ equal-frontier (A, B) := equal-list(frontier(A), frontier(B)).
  /* EL1 */ equal-list ([], Ys) := Ys = [].
  /* EL2 */ equal-list ([X | Xs], [Y | Ys]) :=
                                   equal-atom (X, Y), equal-list (Xs, Ys).
  % Some rules for equal-atom should be added at this place
  /* FT1 */ frontier (tip(X)) := [X].
  /* FT2 */ frontier (node(tip(X), B)) := [X | frontier (B)].
  /* FT3 */ frontier (node(node(A, B), C)) := frontier(node(A, node(B, C))).
```

When used to evaluate ground (i.e. variable free) expressions such as

$$\text{equal-frontier}\Big(\text{node}\big(\text{node}(\text{tip}(1), \text{tip}(2)), \text{tip}(3)\big),\\ \text{node}\big(\text{node}(\text{tip}(1), \text{tip}(3)), \text{tip}(2)\big)\Big)$$

the program behaves essentially as a functional program; but it can also be used to solve such expressions as

$$\text{equal-frontier}\big(\text{node}(\text{tip}(X), A), \text{node}(B, \text{tip}(Y))\big)$$

yielding a boolean result and answers for the variables X, Y, A, B.

Notice that we might change the first rule to

/* EF′ */ equal-frontier(a, b) :– equal-list(frontier(A), frontier(B)).

Without sugaring, this amounts to

/* EF′ */ equal-frontier(A, B) :– equal-list(frontier(A), frontier(B)) $\to$ true.

Now, equal-frontier is more akin to a PROLOG predicate; it can succeed (by computing the result true) or fail, but it is unable to compute the result false, because of the guard in the rhs.

Example 3.3. (The Alpine Club puzzle). In Malachi et al. [38] we found the statement of the following puzzle, which was the subject of discussion of a few contributors to a PROLOG electronic mailing list:

Tony, Mike, and John belong to the Alpine Club. Every member of the Alpine Club is either a skier or a mountain climber or both. No mountain climber likes rain, and all skiers like snow. Mike dislikes whatever Tony likes and likes whatever Tony dislikes. Tony likes rain and snow. Is there a member of the Alpine Club who is a mountain climber, but not a skier?

The following BABEL solution to the puzzle illustrates the more liberal approach to negation allowed by BABEL than by PROLOG.

```
constructors
   tony/0, mike/0, john/0, rain/0, snow/0
predicates
   alpinist/1, climber/1, skier/1, likes/2
rules
   /* AC1  */ alpinist(tony).
   /* AC2  */ alpinist(mike).
   /* AC3  */ alpinist(john).
   /* SC   */ climber(X) :– alpinist(X), ¬skier(X).
   /* LK1  */ ¬likes(X, rain) :– climber(X).
   /* LK2  */ ¬skier(X) :– ¬likes(X, snow).
   /* LK3  */ ¬likes(mike, X) :– likes(tony, X).
   /* LK4  */ likes(mike, X) :– ¬likes(tony, X).
   /* LK5  */ likes(tony, rain).
   /* LK6  */ likes(tony, snow).
```

To solve the puzzle, BABEL must solve the boolean expression

G: $(\text{alpinist}(X), \text{climber}(X), \neg\text{skier}(X)) \to \text{true}$.

obtaining true as result, and binding X to an answer.

Some comments on the BABEL formalization of the puzzle's clues may help to understand the limitations of BABEL with respect to predicate logic. Firstly, notice that rules LK3, LK4 do not violate nonambiguity, since the guards are incompatible. Next, notice that the knowledge embodied in rules SC, LK1, and LK2 could be alternatively expressed as follows:

```
/* SC'  */ skier(X) :- alpinist(X), ¬climber(X).
/* LK1' */ ¬climber(X) :- likes(X, rain).
/* LK2' */ likes(X, snow) :- skier(X).
```

In fact, LK2′ corresponds more closely to the English statement of one of the clues. From the viewpoint of knowledge representation, one could argue that it would be more fair to add both rules LK2 and LK2′ to the program. But then, for the same reason, we should add the rule LK1′ as an alternative expression of the knowledge contained in LK1, and then LK1′, SC would violate nonambiguity. The chosen formalization can perhaps be justified on the following grounds: the lhs of rules SC and LK2 match the signs of the conjuncts in the goal expression G, while rule LK1 corresponds to the English statement of one of the clues.

Example 3.4. (Computing Hamming numbers). Dijkstra [15] attributes to Hamming the problem of building the infinite ascending sequence of all positive numbers greater than 1 containing no prime factors other than 2, 3, and 5. The following solution illustrates again the cooperation of functions and predicates, as well as the use of conditional expressions and lazy lists:

```
constructors
  []/0, [·|·]/2   % lists again
  0/0             % zero
  suc/1           % successor
functions
  hamming-seq/0
  merge-3/3, merge-2/2
  seq-prod/2
  +/2, */2
predicates
  nth-hamming/2
  nth-member/3
  </2
rules
  /* NH  */ nth-hamming (N, M) :- nth-member (N, hamming-seq, M).
  /* NM1 */ nth-member(1, [X | Xs], Y) :- Y = X.
  /* NM2 */ nth-member(suc(suc(N)), [X | Xs], Y) :-
                                            nth-member(suc(N), Xs, Y).
  /* HS  */ hamming-seq := merge-3(seq-prod(2, [1 | hamming-seq]),
                                   seq-prod(3, [1 | hamming-seq]),
                        seq-prod(5, [1 | hamming-seq])).
/* SP   */ seq-prod(X, [Y | Ys]) := [X*Y | seq-prod(X, Ys)].
/* M3   */ merge-3(Xs, Ys, Zs) := merge-2(Xs, merge-2(Ys, Zs)).
/* M2   */ merge-2([X | Xs], [Y | Ys]) :=
               X < Y → [X | merge-2(Xs, [Y | Ys])] □
               Y < X → [Y | merge-2([X | Xs], Ys)] □
           /* otherwise */ [X | merge-2(Xs, Ys)].
/* +1   */ X + 0 := X.
/* +2   */ X + suc(Y) := suc(X + Y).
/* *1   */ X*0 := 0.
/* *2   */ X*suc(Y) := (X*Y) + X.
/* <1   */ ¬X < 0.
/* <2   */ 0 < suc(Y).
/* <3   */ suc(X) < suc(Y) := X < Y.
```

At first sight, *n*th-hamming looks like a PROLOG predicate. It may be used to solve goals in several modes:

```
nth-hamming(5, M)     % true for M = 6
nth-hamming(N, 10)    % true for N = 8
nth-hamming(N, M)     % infinitely many answers
```

However, solving these goals involves functions and requires lazy evaluation, as we shall see in the next section.

4. BABEL'S OPERATIONAL SEMANTICS

In this section we develop BABEL's computation mechanism, which is based on a lazy version of narrowing and defines the operational semantics of the language. Our narrowing method is similar to the lazy narrowing strategy outlined by Reddy in [47]. It was designed aiming at soundness and completeness results with respect to the declarative semantics presented in the next section.

Let us first give some preliminary definitions. We follow the usual notation and terminology in the term rewriting literature; cf. Huet and Oppen [26].

Definition 4.1. Let BS be the set of BABEL predefined symbols for weak equality, negation, conjunction, disjunction, guards, and conditionals. Let TV be the set {true, false}.

Any Σ-expression M is viewed as a finite labeled tree, also denoted by M. The labeled tree is a partial function from the set $\mathbb{N}_+^*$ of finite sequences of positive integers to VS ∪ CS ∪ FS ∪ PS ∪ BS ∪ TV, whose finite domain $\mathbf{O}(M)$ satisfies:

(i) $\mathbf{O}(M)$ is not empty and prefix closed.

(ii) If $u \in \mathbf{O}(M)$ and $M[u]$ is a n-ary symbol, then $u.i \in \mathbf{O}(M)$ iff $1 \leq i \leq n$.

$\mathbf{O}(M)$ is the set of occurrences of M. The prefix ordering on $\mathbf{O}(M)$ is defined as

$u \leq v$ iff $\exists w\, u.w = v$,

$u < v$ iff $u \leq v$ and $u \neq v$.

For $u \in \mathbf{O}(M)$, we have:

$M[u]$:	Symbol at occurrence u
M/u:	Subexpression at occurrence u
$M[u \leftarrow N]$:	Result of replacing M/u by N in M

var(M) denotes the set of variables occurring in M. M is called ground iff var(M) = $\varnothing$.

The set of nonvariable occurrences of M is defined as

$$\mathbf{O}^+(M) = \{u \in \mathbf{O}(M) \mid M[u] \notin \mathrm{VS}\}.$$

Definition 4.2. A Σ-substitution is any mapping from variables to S-expressions, such that data and boolean variables are mapped to data and boolean expressions, respectively. Any substitution σ can be uniquely extended to a mapping from Exp_Σ to Exp_Σ, also denoted by σ. We denote the application of σ to M as $M\sigma$.

The domain of a substitution σ is defined as

$$\mathrm{dom}(\sigma) = \{X \in \mathrm{VS} \mid X\sigma \neq X\}.$$

σ is called finite iff dom(σ) is finite.

σ is called a d-substitution iff $X\sigma$ is a term for all $X \in \mathrm{dom}(\sigma)$.

σ is called a ground substitution iff $X\sigma$ has no variable occurrences, for all $X \in \mathrm{dom}(\sigma)$.

The restriction of a substitution σ to a set of variables $V \subseteq \mathrm{VS}$ is denoted as $\sigma \upharpoonright V$.

The composition of two substitutions σ, θ is denoted as $\sigma\theta$ and satisfies $M(\sigma\theta) = (M\sigma)\theta$.

In the sequel, we identify a finite substitution with an association list binding variables to expressions and denoted as $\sigma = [M_1/X_1, \ldots, M_r/X_r]$, where $\{X_1, \ldots, X_r\} = \mathrm{dom}(\sigma)$.

Next, we turn to unification. The notions of unifier and most general unifier (m.g.u.) are well known. To use narrowing as a computation mechanism for BABEL, we are interested in unifying expressions with left hand sides of rules. This gives rise to a particular kind of unification problems:

Definition 4.3. Let $k/n \in \mathrm{FS} \cup \mathrm{PS} \cup \mathrm{BS}$. For the purposes of this definition, we apply k in prefix form even if it is a predefined BABEL symbol. A linear unification problem asks for the unification of two expressions without shared variables:

$$k(M_1, \ldots, M_n), \qquad k(t_1, \ldots, t_n),$$

where $M \in \mathrm{Exp}_\Sigma$, $t \in \mathrm{DTerm}_\Sigma$, and $k(t_1, \ldots, t_2)$ is linear, i.e. has no multiple occurrences of variables.

Linear unification problems can be solved by any version of Robinson's unification algorithm [50]. However, in order to control the lazy behavior of narrowing, we must distinguish such cases where an attempted unification does not succeed because of a clash between a constructor c and a function, predicate, or predefined symbol k, since such a situation can be viewed as a demand for further evaluation of k. The following is a straightforward adaptation of well-known unification algorithms (e.g. Lassez et al. [34]) which serves as a precise formulation of this idea. Notice that unification can now succeed, fail, or suspend.

Definition 4.4. (Unification algorithm for linear unification problems).

Input: $k(M_1, \ldots, M_n)$, $k(t_1, \ldots, t_n)$ as in Definition 4.3.

Stage 1: Build $U_0 = \{M_1 \downarrow_1 t_1, \ldots, M_n \downarrow_n t_n\}$, $\sigma_0 = \varepsilon$.

Stage 2: Don't care nondeterministically rewrite the initial unification configuration (U_0, σ_0), using the rules which follow, until a nonreducible configuration (U, σ) is reached. If $U = \emptyset$, report "SUCCESS" and output σ as m.g.u. If $U = \{\text{FAILURE}\}$, report "FAILURE". Otherwise, consider the set I of all integers i with $1 \leq i \leq n$ and such that U has some member of the form $l(N_1, \ldots, N_u)\downarrow_i c(s_1, \ldots, s_v)$, where $l/u \in \text{FS} \cup \text{PS} \cup \text{BS}$, $c/v \in \text{CS} \cup \text{TV}$, and $u, v \geq 0$. Then, report "SUSPENDED" and output I as the set of indexes of demanded arguments.

The rules for reducing unification configurations are:

(UR1) $(\{c(N_1, \ldots, N_u)\downarrow_i c(s_1, \ldots, s_u)\} \cup U, \sigma) \rightarrow (\{N_1 \downarrow_i s_1, \ldots, N_u \downarrow_i s_u\} \cup U, \sigma)$, where $c/u \in \text{CS} \cup \text{TV}$, $u \geq 0$.

(UR2) $(\{X \downarrow_i s\} \cup U, \sigma) \rightarrow (U[s/X], \sigma[s/X])$, where the term s is not a variable.

(UR3) $(\{M \downarrow_i X\} \cup U, \sigma) \rightarrow (U[M/X], \sigma[M/X])$.

(UR4) $(\{c(N_1, \ldots, N_u)\downarrow_i d(s_1, \ldots, s_v)\} \cup U, \sigma) \rightarrow (\{\text{FAILURE}\}, \sigma)$, where $c/u, d/v \in \text{CS} \cup \text{TV}$, $c \neq d, u, v \geq 0$.

Notice that our unification algorithm has no occur check. It is not needed, because of linearity. Moreover, the algorithm has three possible outcomes. The SUSPENDED one acts as a mechanism for detecting that the pattern of a rule is demanding further evaluation of some argument. This information will help us to achieve laziness.

The behavior of the algorithm is explained by the following result, which can be proved by well-known techniques (Lassez et al. [34]).

Theorem 4.1. The above unification algorithm, when given any linear unification problem

Unify: $k(M_1, \ldots, M_n), k(t_1, \ldots, t_n)$

as input, always halts. Moreover, it reports SUCCESS *iff the two input expressions are unifiable. In this case, it outputs a m.g.u. σ that can be presented as the union of two substitutions with disjoint domains*:

$$\sigma = \sigma_{\text{out}} \uplus \sigma_{\text{in}},$$

where

$$\sigma_{\text{out}} = \sigma \restriction \text{var}(k(M_1, \ldots, M_n)),$$

$$\sigma_{\text{in}} = \sigma \restriction \text{var}(k(t_1, \ldots, t_n))$$

Moreover, σ_{out} is a d-substitution.

Notice that the part σ_{out} of a computed m.g.u. reflects the flow of data (constructed terms) from a rule to an expression, while the part σ_{in} represents the flow of information (expressions to be evaluated) from an expression to a rule. We show now three small examples to illustrate the possible outcomes of the unification algorithm.

Example 4.1. (Successful unification).

Input: $f(X, \text{cons}(g(X), \text{nil})), f(\text{suc}(X'), \text{cons}(Y', Z'))$.

Output: SUCCESS with

$$\sigma_{\text{in}} = [g(\text{suc}(X'))/Y', \text{nil}/Z'],$$
$$\sigma_{\text{out}} = [\text{suc}(X')/X].$$

Example 4.2. (Failed unification).

Input: $f(X, \text{cons}(X, \text{nil})), f(\text{suc}(X'), \text{cons}(0, Y'))$.

Output: FAILURE.

Example 4.3. (Suspended unification).

Input: $f(g(X), Y, \text{cons}(g(Y), \text{nil})), f(\text{suc}(X'), 0, \text{cons}(\text{suc}(Y'), Z'))$.

Output: SUSPENDED with $I = \{1, 3\}$.

The operational meaning of BABEL's predefined symbols can be specified by the same kind of rules as for the user defined functions and predicates. Let us now introduce these rules.

Definition 4.5. Let Σ be any BABEL signature, with set of constructors CS. The implicit Σ-rules for BABEL primitives are as follows:

Rules for guards and conditionals:

$$(\text{true} \rightarrow X) := X.$$
$$(\text{true} \rightarrow X \,\square\, Y) := X. \qquad (\text{false} \rightarrow X \,\square\, Y) := Y.$$

Rules for propositional connectives:

$$\neg\text{true} := \text{false}.$$
$$\neg\text{false} := \text{true}.$$
$$(\text{false}, Y) := \text{false}. \qquad (\text{false}; Y) := Y.$$
$$(\text{true}, Y) := Y. \qquad (\text{true}; Y) := \text{true}.$$
$$(X, \text{false}) := \text{false}. \qquad (X; \text{false}) := X.$$
$$(X, \text{true}) := X. \qquad (X; \text{true}) := \text{true}.$$

Rules for weak equality:

$$c = c := \text{true}. \qquad \text{for all} \quad c/0 \in \text{CS} \cup \text{TV}.$$
$$c(X_1, \ldots, X_n) = c(Y_1, \ldots, Y_n) := X_1 = Y_1, \ldots, X_n = Y_n.$$
$$\text{for all} \quad c/n \in \text{CS}, \quad n \geq 1.$$
$$c(X_1, \ldots, X_n) = d(Y_1, \ldots, Y_n) := \text{false}.$$
$$\text{for all} \quad c/n, d/m \in \text{CS}, \quad c \neq d, \quad n, m \geq 0.$$

Given a BABEL program Π of signature Σ, we denote as $\hat{\Pi}$ the result of expanding Π with all implicit Σ-rules.

Notice that for any program Π, $\hat{\Pi}$ is still a nonambiguous set of rules in BABEL's sense. By contrast, the implicit rules of our parallel conjunction and disjunction violate Huet and Levy's [25] nonambiguity.

We are now in a position to define rule applicability and lazy narrowing.

Definition 4.6. Let Π be a Σ-program and let N be a Σ-expression.

(a) A rule of $\hat{\Pi}$ applies to, fails for, or suspends for N iff some variant of the rule standing apart from N (i.e., sharing no variables with N) applies to, fails for, or suspends for N, respectively.

(b) A rule $L := R$ standing apart from N applies to, fails for, or suspends for N iff the unification of N, L yields a SUCCESS, FAILURE, or SUSPENDED outcome, respectively. If the unification succeeds with m.g.u. σ, we say that $L := R$ applies to N via σ. If the unification yields a suspended outcome with the set I of demanded argument indices, we say that $L := R$ is suspended at i for any integer $i \in I$.

Lazy narrowing works by narrowing expressions through application of rules. Laziness is achieved by trying to select outer redexes first and going inner only when demanded by the lhs patterns in suspended rules. The following definitions capture this idea.

Definition 4.7. Assume a Σ-program Π and a Σ-expression M. We say that $u \in \mathbf{O}^+(M)$ is a redex occurrence iff some rule in $\hat{\Pi}$ applies to M/u. We also say that a redex occurrence u of M is lazy iff u belongs to the set $\mathrm{LR}_\Pi[M]$ defined by recursion on M's structure as follows. In the following lines, k stands for a symbol belonging to $\mathrm{FS} \cup \mathrm{PS} \cup \mathrm{BS}$, where FS and PS are given by Σ, and BS is as in Definition 4.1. For the purposes of this definition, we apply k in prefix form even if it belongs to BS:

$$\mathrm{LR}_\Pi[M] = \varnothing \qquad \text{if } M \text{ is a term.}$$

$$\mathrm{LR}_\Pi[c(M_1, \ldots, M_n)] = \bigsqcup_{i=1}^{n} i.\mathrm{LR}_\Pi[M_i]$$

$$\text{if } c/n \in \mathrm{CS},\ n > 0, \text{ and some } M_i \text{ is not a term.}$$

$$\mathrm{LR}_\Pi[k(M_1, \ldots, M_n)] = \{\varepsilon \mid \text{some rule in } \hat{\Pi} \text{ applies to } k(M_1, \ldots, M_n)\} \cup \bigsqcup_{i \in I} i.\mathrm{LR}_\Pi[M_i],$$

where $I = \{i \mid 1 \le i \le n, \text{ some rule in } \hat{\Pi} \text{ is suspended at } i\}$.

Notice that $k(M_1, \ldots, M_n)$ is regarded as having no redex occurrences if all rules fail for it, since in such a situation the expression cannot be reduced to an expression with a constructor at the outermost occurrence, and only constructed terms are accepted as evaluated values.

Definition 4.8. Let Π, M be as above. If $u \in \mathbf{O}^{+}(M)$ is a redex occurrence for a (variant of *a*) rule $L := R$ in $\hat{\Pi}$ that applies to M/u via $\sigma = \sigma_{\text{out}} \uplus \sigma_{\text{in}}$ (remember Theorem 4.1), we say that M narrows in one step to the new expression $M[u \leftarrow R]\sigma$ and write

$$M\,{}^{\mathrm{N}} \overset{\Pi}{\to}{}_{\sigma_{\text{out}}} M[u \leftarrow R]\sigma.$$

If $\sigma_{\text{out}} = \varepsilon$, we write

$$M \overset{\Pi}{\to} M[u \leftarrow R]\sigma_{\text{in}}$$

and say that M rewrites in one step to $M[u \leftarrow R]\sigma_{\text{in}}$.

Notice that σ_{out} records the part of σ that has an effect on M. We also say that M narrows to N, or reduces to N via narrowing, and write

$$M\,{}^{\mathrm{N}} \overset{\Pi}{\to}{}^{*}_{\sigma_{\text{out}}} N$$

iff there is some narrowing sequence

$$M = M_0\,{}^{\mathrm{N}} \overset{\Pi}{\to}{}_{\sigma_{\text{out},1}} M_1 \cdots M_{i-1}\,{}^{\mathrm{N}} \overset{\Pi}{\to}{}_{\sigma_{\text{out},i}} M_i \cdots {}^{\mathrm{N}} \overset{\Pi}{\to}{}_{\sigma_{\text{out},l}} \mathrm{M}_l = \mathrm{N}$$

with $l \geq 0$ and $\sigma_{\text{out}} = \sigma_{\text{out},1} \cdots \sigma_{\text{out},i} \cdots \sigma_{\text{out},l} \upharpoonright \text{var}(M)$. Analogously, the notation

$$M \overset{\Pi}{\to} {}^{*}N$$

indicates the existence of a rewriting sequence.

Finally, we say that M reduces to N via lazy narrowing iff there is some narrowing sequence with the property that each narrowing step in it applies a rule at a lazy redex occurrence. To indicate this, we shall use the notation

$$M\,{}^{\mathrm{N}}\!-\!1 \overset{\Pi}{\to}{}^{*}_{\sigma_{\text{out}}} N.$$

In the case of lazy rewriting, we shall write

$$M - 1 \overset{\Pi}{\to} {}^{*}N.$$

In practice, the program will be known by context and Π will be omitted in the notation.

We propose to adopt lazy narrowing as the operational (reduction) semantics of BABEL. By inspecting the implicit rules, we can see that BABEL primitives display the behavior that should be expected of them under demand driven evaluation. Notice, in particular, that guarded expressions and conditional expressions demand the evaluation of their conditions first, while our parallel conjunction and disjunction demand the evaluation of both arguments.

As already said in the introduction, narrowing has been used to solve problems of E-unification (Fay [19], Hullot [27], Siekmann and Szabo [54], Giovannetti and Moiso [22], Dincbas and van Heternryck [16], Gallier and Snyder [21], Nutt et al. [43]) and is embedded in the operational semantics of several logic + functional programming languages (Dershowitz and Josephson [13], Dershowitz [12], Dershowitz and Plaisted

[14], Josephson and Dershowitz [28], Fribourg [20], Goguen and Meseguer [23], Levi et al. [35], Bosco and Giovannetti [5], Reddy [46–48], Subrahmanyam and You [56]). In BABEL, lazy narrowing can be used to solve a set of "equations" (in the sense of weak equality) by reducing a boolean expression

$$(M_1 = N_1, \ldots, M_r = N_r) \to \text{true}$$

to true via lazy narrowing. The resulting extended unification algorithm behaves similarly to the "unification with lazy surderivation" in Dincbas and van Henteryck [16].

BABEL, however, is intended for a more general use, which we explain now.

Definition 4.9. Let Σ be a signature with set of constructors CS. The shell $|N|$ of any given Σ-expression N is defined recursively:

$|\text{true}| = \text{true}$, $\quad |\text{false}| = \text{false}$,

$|N| = \perp_b$ $\quad$ for any other boolean expression,

$|c| = c$ $\quad$ for $c/0 \in \text{CS}$,

$|c(N_1, \ldots, N_m)| = c(|N_1|, \ldots, |N_m|)$ $\quad$ for $c/m \in \text{CS}$, $\quad m > 0$,

$|N| = \perp_d$ $\quad$ for any other data expression.

The declarative meaning of the "undefined" symbols $\perp_b$ and $\perp_d$ will be formally defined in Section 5.

Let Π be a Σ-program. For any narrowing derivation

$$M \;{\scriptstyle N}\overset{\Pi}{\to} {}^{*}_{\sigma_{\text{out}}} N$$

(which does not need to be lazy), we define the outcome as the pair $(|N|, \sigma_{\text{out}})$. We also say that $|N|$ is the result of σ_{out} is the answer. Notice that the result can be partially defined, if $|N|$ has occurrences of $\perp_b$ or $\perp_d$.

Two special kinds of outcomes are important: functional outcomes, where $|N|$ is a ground term and σ_{out} is ε, and PROLOG-like outcomes, where $|N|$ is true.

Notice that, for any outcome, σ_{out} is a d-substitution, because of Theorem 4.1 and Definition 4.8.

Now, we can imagine BABEL computations as lazy narrowing reductions that yield some outcome. This view includes functional and PROLOG-like computations as particular cases. Let us go back to the examples from Section 3 to provide concrete illustrations.

Example 4.4. (Appending lists; see Example 3.1). The append predicate is frequently used in PROLOG to nondeterministically split a given list into two factors. We show how this can be done in BABEL:

$$\begin{array}{lll}
 & \text{append}(X_s, Y_s, [a, b, c]) & \\
\text{N}-1 \overset{A2}{\to}_{\sigma_{\text{out},1}} & (a = X_1, \text{append}(X_{s1}, Y_s, [b, c])) \to \text{true} & \\
 & & \%\ \text{where } \sigma_{\text{out},1} = [[X_1 \mid X_{s1}]/X_s] \\
\text{N}-1 \overset{IR}{\to} {}^{*}_{\sigma_{\text{out},2}} & \text{append}(X_{s1}, Y_s, [b, c]) \to \text{true} & \%\ \text{where } \sigma_{\text{out},2} = [a/X_1] \\
\text{N}-1 \overset{A1}{\to}_{\sigma_{\text{out},3}} & ([b, c] = Y_s \to \text{true}) \to \text{true} & \%\ \text{where } \sigma_{\text{out},3} = [[\,]/X_{s1}] \\
\text{N}-1 \overset{IR}{\to} {}^{*}_{\sigma_{\text{out},4}} & \text{true} & \%\ \text{where } \sigma_{\text{out},4} = [[b, c]/Y_s]
\end{array}$$

The outcome is (true, σ_{out}) with $\sigma_{\text{out}} = [[a]/X_s, [b, c]/ Y_s]$. We have abbreviated some steps and used IR to indicate the application of implicit rules.

Notice that SLD resolution has been simulated by lazy narrowing acting on PROLOG-like rules (cf. Convention 2.1). The role of guards in PROLOG-like rules is essential to achieve the simulation.

Example 4.5. (Testing binary trees for equality of frontiers; see Example 3.2). When used to evaluate a ground expression of the form equal-frontier(t_1, t_2), the program behaves in a purely functional way. Assuming that the given trees t_1, t_2 have different frontiers, BABEL behaves as a lazy functional language and is able to detect inequality without having to evaluate the whole frontiers; e.g.

$$\begin{array}{ll}
& \text{equal-frontier(node(tree(tip(1), tip(2)), tip(3)),} \\
& \qquad\qquad \text{node(tree(tip(1), tip(3)), tip(2)))} \\
\overset{\text{EF}}{-1\rightarrow} & \text{equal-list(frontier(node(node(tip(1), tip(2)), tip(3))),} \\
& \qquad\quad \text{frontier(node(node(tip(1), tip(3)), tip(2))))} \\
-1\rightarrow^* & \text{equal-list([2 | frontier(tip(3))], [3 | frontier(tip(2))])} \\
-1\rightarrow^* & \text{false}
\end{array}$$

We have omitted some intermediate steps. The computation illustrates lazy rewriting as a particular case of lazy narrowing.

The same program can solve expressions including variables; e.g.

$$\begin{array}{ll}
& \text{equal-frontier(node(tip}(X), A), \text{node}(B, \text{tip}(Y))) \\
\overset{\text{EF}}{-1\rightarrow} & \text{equal-list(frontier(node(tip}(X), A)), \text{frontier(node}(B, \text{tip(1))))} \\
\overset{\text{FT2}}{-1\rightarrow} & \text{equal-list(}[X \mid \text{frontier}(A)], \text{frontier(node}(B, \text{tip}(Y)))) \\
\text{N} - 1 \underset{\sigma_{\text{out},1}}{\overset{\text{FT2}}{\longrightarrow}} & \text{equal-list(}[X \mid \text{frontier}(A)], [Z \mid \text{frontier(tip}(Y))]) \\
& \qquad\qquad\qquad \%\ \text{where } \sigma_{\text{out},1} = [\text{tip}(Z)/B] \\
\overset{\text{EL2}}{-1\rightarrow^*} & \text{equal-atom}(X, Z), \text{equal-list(frontier}(A), \text{frontier(tip}(Y)))
\end{array}$$

etc.; there are infinitely many possible outcomes, each one with a result true or false and an answer for X, Y, A, and B.

Example 4.6. (The Alpine Club puzzle; see example 3.3). A BABEL computation of the outcome (true, [mike/X]) to the Alpine Club puzzle is as follows:

$$\begin{array}{ll}
& (\text{alpinist}(X), \text{climber}(X), \neg\text{skier}(X)) \rightarrow \text{true} \\
\text{N} - 1 \underset{\sigma_{\text{out},1}}{\xrightarrow{\text{AC2, IR}}}{}^* & (\text{climber(mike)}, \neg\text{skier(mike)}) \rightarrow \text{true} \qquad \%\ \text{where } \sigma_{\text{out},1} = [\text{mike}/X] \\
\overset{\text{LK2}}{-1 \longrightarrow} & (\text{climber(mike)}, \neg(\neg\text{likes(mike, snow)} \rightarrow \text{false})) \rightarrow \text{true}
\end{array}$$

$$\overset{\text{LK3}}{-1\!\!\rightarrow} \quad (\text{climber(mike)}, \neg(\neg(\text{likes(tony, snow)} \rightarrow \text{false}) \rightarrow \text{false})) \rightarrow \text{true}$$

$$\overset{\text{LK6}}{-1\!\!\rightarrow} \quad (\text{climber(mike)}, \neg(\neg(\text{true} \rightarrow \text{false}) \rightarrow \text{false})) \rightarrow \text{true}$$

$$\overset{\text{IR}}{-1\!\!\rightarrow^*} \quad \text{climber(mike)} \rightarrow \text{true}$$

$$\overset{\text{SC}}{-1\!\!\rightarrow} \quad ((\text{alpinist(mike)}, \neg\text{skier(mike)}) \rightarrow \text{true}) \rightarrow \text{true}$$

$$-1\!\!\xrightarrow{\text{AC2, IR}}{}^{*} \quad (\neg\text{skier(mike)} \rightarrow \text{true}) \rightarrow \text{true}$$

$$-1\!\!\rightarrow^* \quad (\text{true} \rightarrow \text{true}) \rightarrow \text{true} \qquad \%\ \text{same reduction as before}$$

$$\overset{\text{IR}}{-1\!\!\rightarrow^*} \quad \text{true}$$

As in Example 4.4, this essentially mimics a PROLOG computation; but some (restricted) use of logical negation is possible.

Example 4.7. (Computing Hamming numbers; see Example 3.4). In the context of this program, the presence of potentially infinite data structures (streams) makes lazy evaluation essential. A BABEL computation of the fifth Hamming number would behave essentially as in a lazy functional language and could start as follows:

$$\textit{n}\text{th-hamming}(5, M)$$

$$\overset{\text{NH}}{-1\!\!\rightarrow} \quad \textit{n}\text{th-member}(5, \text{hamming-seq}, M) \rightarrow \text{true}$$

Now hamming-seq is demanded by the rules for *n*th-member; so other rules will be applied to partially evaluate it until its first member 2 is computed. At this moment, the state of the evaluation will be

$$\textit{n}\text{th-member}(5, [2 \mid \text{RHS}], M) \rightarrow \text{true}$$

(where RHS is an expression denoting the rest of the Hamming sequence), and the reduction will proceed lazily:

$$\overset{\text{NM2}}{-1\!\!\rightarrow} \quad (\textit{n}\text{th-member}(4, \text{RHS}, M) \rightarrow \text{true}) \rightarrow \text{true}$$

and so on. In few words: The Hamming sequence will be produced lazily, until its fifth member is found. BABEL's unification mechanism takes care of the detecting demanded arguments through SUSPENDED outcomes of the unification algorithm; remember Definition 4.4.

It is known that the computation of the first n Hamming numbers, using the algorithm embodied in the present program and lazy evaluation, takes $O(n)$ steps; see Bird and Wadler [4].

The program also admits modes of use going beyond functional programming; e.g.

$$\textit{n}\text{th-hamming}(N, 10)\ {\scriptstyle\text{N}} -1 \xrightarrow{\Pi}{}^{*}_{[8/N]}\ \text{true}$$

$$\textit{n}\text{th-hamming}(N, M)\ {\scriptstyle\text{N}} -1 \xrightarrow{\Pi}{}^{*}_{[6/N, 8/M]}\ \text{true}$$

Before closing this section, let us say that BABEL's lazy evaluation is a clear and simple way to specify a demand driven reduction mechanism; but it is not an "optimal computation rule". In fact, lazy narrowing (in the technical sense of Definition 4.8) allows narrowing sequences where some inner narrowing step does not contribute to any later steps. Such derivations are, strictly speaking, not so lazy. They can arise in cases where one has, at the same time, applicable rules and suspended rules. In such cases, the search space for lazy narrowing (i.e., the tree whose root is the initial expression and whose path correspond to all possible lazy narrowing sequences) usually includes duplicate solutions.

The following example illustrates this. Incidentally, it serves to show that both outermost and innermost narrowing can miss solutions that are computable via lazy narrowing. Even infinitely many solutions can be lost.

Example 4.8. Let Π be the following program:

constructors
$0/0, \mathrm{suc}/1$ % for natural numbers
$[\,]/0, [\cdot \mid \cdot]/2$ % for lists
functions
$f/2 \quad g, h/1$
rules
/* F_0 */ $f(0, X) := h(X)$.
/* F_1 */ $f(\mathrm{suc}(N), [Y \mid Y_s]) := [\mathrm{suc}(Y) \mid f(N, Y_s)]$.
/* G */ $g(N) := [N \mid g(\mathrm{suc}(N))]$.
/* H */ $h([Y \mid Y_s]) := [Y]$.

In Figure 1 we show a fragment of Π's search space for the initial expression $f(N, g(0))$, using lazy narrowing.

Of the two lazy narrowing sequences $\mathrm{NS}_1, \mathrm{NS}'_1$, only the first is "really lazy"; the second starts with a reduction that does not contribute to the second step and could have been delayed until the third one (obtaining, in fact, NS_1). There is an exact characterization of the "really lazy" derivations: They correspond to the outside-in derivations introduced by Huet and Levy [25] to investigate left linear, nonambiguous, uncondi-

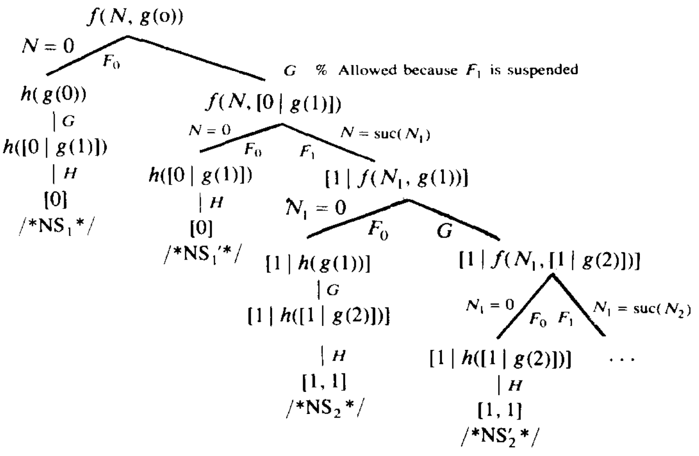

FIGURE 1.

tional rewriting systems. Briefly speaking, the definition is obtained by forcing the later use of the rule that demands evaluation of a suspended argument. Darlington and Guo [9] have adapted this notion to a kind of lazy narrowing based on constructors, and You [58–60] has used a very similar notion (outer narrowing derivation) to investigate E-unification and E-matching problems for a class of constructor based unconditional rewriting systems.

With innermost narrowing, the search space for the same program and initial expression would consist of a single infinite branch, corresponding to the nonterminating evaluation of $g(0)$. No solutions would be found. Finally, we show the search space using outermost narrowing, which retains only one solution:

$$\begin{tikzcd}
f(N, g(0)) \arrow[d, dash, "N=0"', "F_0"] \\
h(g(0)) \arrow[d, dash, "G"] \\
h([0 \mid g(1)]) \arrow[d, dash, "H"] \\
{[0]}
\end{tikzcd}$$

5. BABEL'S DECLARATIVE SEMANTICS

In this section we present a declarative semantics for BABEL, which is based on Scott domains. We introduce interpretations and define how to view a BABEL rule as a logical statement that can hold or not hold in a given interpretation. We pay special attention to Herbrand interpretations and prove that any BABEL program has a least Herbrand model. This result generalizes the well-known corresponding one for Horn clause programs (van Emden and Kowalski [17], Apt and van Emden [1]). A very similar approach has been used for the language K-LEAF (Levi et al. [35]). With the help of our declarative semantics, we are able to prove soundness, confluence, and completeness results.

We give first some preliminary definitions on Scott's domains; cf. Scott [52] and Mulmuley [42].

Definition 5.1. A partially ordered set, with order $\sqsubseteq$, is called a complete partial order (cpo) iff it has a least element $\perp$ (usually called "bottom") and any directed subset S of D has a least upper bound (lub) $\sqcup S$ in D.

Given a cpo D, an element z is called finite iff, for any directed subset S of D, $z \sqsubseteq \sqcup S$ implies that there exists an element $y \in S$ such that $z \sqsubseteq y$. An element x is called total iff the only upper bound of x in D is x itself.

D is algebraic iff for all $x \in D$, the set

$$S_x = \{ z \mid z \sqsubseteq x \text{ and } z \text{ is finite} \}$$

is directed and $x = \sqcup S_x$. D is ω-algebraic if, in addition, the set Fin_D of its finite elements is countable.

A subset S of D is consistent iff any finite subset S_0 of S has some upper bound in D. A cpo D is called consistently complete iff any consistent subset of D has a lub in D.

In the rest of this section, by domain we mean a consistently complete ω-algebraic cpo. For a different characterization of domains, see Scott [52].

Definition 5.2. Let D be a domain. An enumeration of Fin_D is any mapping e from $\mathbb{N}$ onto Fin_D that maps 0 to $\perp$; i.e.

$$e_0 = \perp \quad \text{and} \quad \{e_i \mid i \in \mathbb{N}\} = \mathrm{Fin}_D.$$

Assume some fixed one to one mapping from $\mathbb{N}$ onto the set of all finite subsets of $\mathbb{N}$ (cf. Rogers [51]). Let I_n be the finite set coded by $n \in \mathbb{N}$ under this bijection. Then we say that e is an effective presentation of D iff the two following predicates on natural numbers are decidable (i.e. recursive):

$$\mathrm{con}(n) \quad \Leftrightarrow_{\mathrm{def}} \quad S_n \text{ is consistent in } D,$$

$$\mathrm{lub}(n, j) \quad \Leftrightarrow_{\mathrm{def}} \quad \sqcup S_n = e_j \ \text{ in } D$$

where, for $n \in \mathbb{N}$, $S_n = \{e_i \mid i \in I_n\} \subseteq \mathrm{Fin}_D$.

A domain is called effectively presented iff it has some effective presentation.

Definition 5.3. Given two domains C and D, a mapping f from C to D is called monotonic iff $x \sqsubseteq_C y$ implies $f(x) \sqsubseteq_D f(y)$ for all $x, y \in C$.

A monotonic function f is called continuous iff, for any directed subset S of C, $f(\sqcup_C S) = \sqcup_D f(S)$.

If C and D are domains, we shall denote by $[C \to D]$ the set of continuous functions from C to D.

Definition 5.4. Let C, D be domains with effective presentations e and e'. A continuous mapping $f : C \to D$ is called computable iff its graph, defined as the predicate on $\mathbb{N}$ given by

$$G_f(i, j) \quad \Leftrightarrow_{\mathrm{def}} \quad e'_j \sqsubseteq f(e_i),$$

is recursively enumerable. An element $x \in D$ is called computable iff the set of natural numbers

$$G_x =_{\mathrm{def}} \{i \in \mathbb{N} \mid e'_i \sqsubseteq x\}$$

is recursively enumerable.

The following results are well known; cf. Scott [52] and Larsen and Winskel [33].

Theorem 5.1. Let C, D be domains. Then $[C \to D]$, *under the pointwise ordering, and* $C \times D$, *under the componentwise ordering, are again domains. They are effectively presentable if C and D are.*

Moreover, domains as objects with continuous mappings as morphisms constitute a cartesian closed category.

Theorem 5.2. Let D be a domain. Any continuous mapping $f : D \to D$ *has a least fixpoint* $x \in D$ *which satisfies* $f(x) = x$ *and* $x \sqsubseteq y$ *for any* $y \in D$ *such that* $f(y) \sqsubseteq y$. *In fact, this least fixpoint is the lub of the chain*

$$\perp \sqsubseteq f(\perp) \sqsubseteq \cdots \sqsubseteq f^n(\perp) \sqsubseteq \cdots.$$

Moreover, the mapping $\mathrm{fix} : [D \to D] \to D$ *which assigns to any* $f \in [D \to D]$ *its least fixpoint is itself continuous; i.e.,* $\mathrm{fix} \in [[D \to D] \to D]$.

If D is effectively presentable, then fix *is computable, and* fix(f) *is computable for any computable* $f \in [D \rightarrow D]$.

We now define two special domains related to a given BABEL signature.

Definition 5.5. Let Σ be a BABEL signature with set of constructors CS.

The Herbrand domain H_Σ has as elements all (finite or infinite) trees with nodes labeled by ranked symbols from CS $\cup \{\bot_d\}$, where $\bot_d$ is a new symbol of arity 0. Such trees can be viewed as functions from $\mathbb{N}_+^*$; remember Definition 4.1.

Elements s, $t \in H_\Sigma$ are called Σ-constructs. In particular, all ground data Σ-terms are Σ-constructs. By a partial data Σ-term we shall understand a finite Σ-construct.

The ordering $\sqsubseteq$ on H_Σ is defined as follows:

$s \sqsubseteq t$ iff t can be obtained from s by substituting some constructs for occurrences of $\bot_d$.

The boolean domain BOOL is the flat cpo given by the diagram

$$\begin{tikzcd}[column sep=small] \text{true} & & \text{false} \\ & \bot_b \arrow[ul, dash] \arrow[ur, dash] & \end{tikzcd}$$

where $\bot_b$ is a new symbol of arity 0, and true, false are BABEL's boolean constants. By partial Σ-term we understand the partial data Σ-terms together with true, false, and $\bot_b$.

Simply by checking the corresponding definitions, we obtain:

Theorem 5.3. H_Σ and BOOL *are domains. Moreover, H_Σ has the following properties*:

(i) *The finite elements coincide with the finite constructs, i.e., those having a finite number of nodes.*

(ii) *The total elements are the constructs without any occurrence of* $\bot_d$.

(iii) *H_Σ is effectively presentable.*

We are now in a position to define interpretations and models for BABEL programs.

Definition 5.6. Let Σ be a signature with CS, FS, PS. A Σ-interpretation is any algebra

$$I = \langle D, (c_I)_{c \in \mathrm{CS}}, (f_I)_{f \in \mathrm{FS}}, (p_I)_{p \in \mathrm{PS}} \rangle$$

where D is a domain and the following conditions hold:

$c_I \in D$ for $c/0 \in \mathrm{CS}$,

$c_I \in [D^n \rightarrow D]$ for $c/n \in \mathrm{CS}$, $n > 0$,

$f_I \in D$ for $f/0 \in \mathrm{FS}$,

$f_I \in [D^n \rightarrow D]$ for $f/n \in \mathrm{FS}$, $n > 0$,

$p_I \in$ BOOL for $p/0 \in \mathrm{PS}$,

$p_I \in [D^n \rightarrow \text{BOOL}]$ for $p/n \in \mathrm{PS}$, $n > 0$.

I is called a computable interpretation iff D is effectively presentable and c_I, f_I, p_I are computable mappings.

I is called a Herbrand interpretation iff D is H_Σ and the interpretation of constructors is free, i.e.,

$$c_I = c \quad \text{for} \quad c/0 \in \mathrm{CS},$$

$$c_I(t_1, \ldots, t_n) = \begin{array}{c} c \\ t_1 \; \cdots\cdots \; t_n \end{array} \quad \text{for} \quad c/n \in \mathrm{CS}, \quad n > 0, \text{ and } t_i \in H_\Sigma.$$

In any given interpretation, BABEL expressions denote domain elements.

Definition 5.7. Let I be a Σ-interpretation. An environment over I is any mapping ρ from $\mathrm{VS} = \mathrm{DV} \cup \mathrm{BV}$ to $D \cup \textsc{bool}$ such that $\rho(X) \in D$ for all $X \in \mathrm{DV}$ and $\rho(Y) \in \textsc{bool}$ for all $Y \in \mathrm{BV}$.

For any BABEL Σ-expression M, the valuation $[\![M]\!]_I(\rho)$ of M in I under ρ is defined by recursion on M's syntactical structure. It belongs to BOOL if M is a boolean expression; otherwise, it belongs to D:

$$[\![\mathrm{true}]\!]_I(\rho) = \mathrm{true},$$

$$[\![\mathrm{false}]\!]_I(\rho) = \mathrm{false},$$

$$[\![X]\!]_I(\rho) = \rho(X) \quad \text{for} \quad X \in \mathrm{VS},$$

$$[\![c]\!]_I(\rho) = c_I \quad \text{for} \quad c/0 \in \mathrm{CS},$$

$$[\![c(M_1, \ldots, M_n)]\!]_I(\rho) = c_I([\![M_1]\!]_I(\rho), \ldots, [\![M_n]\!]_I(\rho)) \quad \text{for} \quad c/n \in \mathrm{CS}, \quad n > 0,$$

$$[\![k]\!]_I(\rho) = k_I \quad \text{for} \quad k/0 \in \mathrm{FS} \cup \mathrm{PS},$$

$$[\![k(M_1, \ldots, M_n)]\!]_I(\rho) = k_I([\![M_1]\!]_I(\rho), \ldots, [\![M_n]\!]_I(\rho)) \quad \text{for} \quad k/n \in \mathrm{FS} \cup \mathrm{PS}, \quad n > 0,$$

$$[\![E_1 = E_2]\!]_I(\rho) = \mathrm{eq}([\![E_1]\!]_I(\rho), [\![E_2]\!]_I(\rho)),$$

$$[\![\neg B]\!]_I(\rho) = \mathrm{not}([\![B]\!]_I(\rho)),$$

$$[\![(B_1, B_2)]\!]_I(\rho) = \mathrm{and}([\![B_1]\!]_I(\rho), [\![B_2]\!]_I(\rho)),$$

$$[\![(B_1; B_2)]\!]_I(\rho) = \mathrm{or}([\![B_1]\!]_I(\rho), [\![B_2]\!]_I(\rho)),$$

$$[\![(B \to M)]\!]_I(\rho) = \mathrm{if\_then}\ ([\![B]\!]_I(\rho), [\![M]\!]_I(\rho)),$$

$$[\![(B \to M_1 \Box M_2)]\!]_I(\rho) = \mathrm{if\_then\_else}\ ([\![B]\!]_I(\rho), [\![M_1]\!]_I(\rho), [\![M_2]\!]_I(\rho)),$$

where the semantic functions eq, not, and, or, if_then, and if_then_else are continuous and computable and are defined as follows:

$$\mathrm{eq}(d_1, d_2) = \begin{cases} \mathrm{true} & \text{if } d_1 = d_2 \text{ is a finite and total element in } D, \\ \mathrm{false} & \text{if } \{d_1, d_2\} \text{ is inconsistent in } D, \\ \bot_b & \text{otherwise} \end{cases}$$

(the fact that D is consistently complete is needed to ensure the continuity of eq),

b	not(b)
true	false
false	true
$\bot_b$	$\bot_b$

and(b_1, b_2)

$b_2 \backslash b_1$	true	false	$\bot_b$
true	true	false	$\bot_b$
false	false	false	false
$\bot_b$	$\bot_b$	false	$\bot_b$

or(b_1, b_2)

$b_2 \backslash b_1$	true	false	$\bot_b$
true	true	true	true
false	true	false	$\bot_b$
$\bot_b$	true	$\bot_b$	$\bot_b$

$$\text{if\_then}(b, m) = \begin{cases} m & \text{if } b = \text{true}, \\ \bot & \text{otherwise}, \end{cases}$$

$$\text{if\_then\_else}(b, m_1, m_2) = \begin{cases} m_1 & \text{if } b = \text{true}, \\ m_2 & \text{if } b = \text{false}, \\ \bot & \text{otherwise}, \end{cases}$$

where m, m_1, m_2 may belong to D or BOOL, and $\bot$ stands for $\bot_d$ or $\bot_b$.

Knowing how to evaluate expressions, any BABEL rule can be interpreted as the statement that the lhs value is always at least as well defined as the rhs value:

Definition 5.8. Let Π, I be a Σ-program and a Σ-interpretation. Remember $\hat{\Pi}$ from Definition 4.5.

I is a model of Π (in symbols: $I \models \Pi$) iff I is a model of every rule in $\hat{\Pi}$.

I is a model of a rule $L := R$ (in symbols: $I \models L := R$) iff $[\![L]\!]_I(\rho) \sqsupseteq [\![R]\!]_I(\rho)$ for all environments ρ over I. Here, $\sqsubseteq$ stands for the partial ordering over BOOL if R is boolean; otherwise, it stands for the ordering over D.

We are now going to prove the existence of a least Herbrand model for any given BABEL program. This generalizes the well-known result of van Emden and Kowalski [17] for Horn clause programs, and is related to a similar result that holds for the logic + functional language K-LEAF (Levi et al. [35]). Intuitively, the least Herbrand model of Π is the "most economic model": It has no data objects but the constructs, and it defines functions and predicates only as much as demanded by the program's rules (including the implicit ones). Similar intuitions lay behind initial models for algebraic specifications; cf. Goguen and Meseguer [23].

As in Apt and van Emden [1], we are going to obtain minimal models as least fixpoints of continuous interpretation transformers. First, we show that Σ-interpretations are the members of a domain.

Theorem 5.4. For any BABEL *signature* Σ, *the set* H-INT$_\Sigma$ *of all the Herbrand* Σ-*interpretations, equipped with the partial ordering*

$I \sqsubseteq J$ *iff* $k_I \sqsubseteq k_J$ *(in the corresponding domain) for all* $k \in \mathrm{FS} \cup \mathrm{PS}$,

is an effectively presentable domain.

PROOF. Remember that BABEL signatures are finite. According to Definition 5.6, any Herbrand Σ-interpretation can be identified with the finite tuple

$$\langle (f_I)_{f \in \mathrm{FS}}, (p_I)_{p \in \mathrm{PS}} \rangle,$$

where FS, PS are given by Σ. Therefore, H-INT$_\Sigma$ can be viewed as the cartesian product

$$\prod_{f/n \in \mathrm{FS}} [D^n \to D] \times \prod_{p/n \in \mathrm{PS}} [D^n \to \mathrm{BOOL}],$$

where D is H_Σ, and where $[D^n \to D]$ and $[D^n \to \mathrm{BOOL}]$ are understood as D and BOOL, respectively, whenever n is 0. By Theorem 5.3, H_Σ is an effectively presentable domain. We can then conclude that H-INT$_\Sigma$ is an effectively presentable domain, by Theorem 5.1. □

Next, we define interpretation transformers:

Definition 5.9. Let Π be a BABEL program of signature Σ.

A ground infinitary Σ-substitution is any mapping σ from VS to $H_\Sigma \cup \mathrm{BOOL}$ mapping boolean variables to BOOL and data variables to H_Σ. Ground infinitary substitutions generalize ground substitutions (cf. Definition 4.2) and have unique extensions to mappings from Exp_Σ to $H_\Sigma \cup \mathrm{BOOL}$. For any $M \in \mathrm{Exp}_\Sigma$, $M\sigma$ is called a ground instance of M. Notice that the evaluation of M_σ in a given Herbrand interpretation I, denoted by $[\![M\sigma]\!]_I$, can be defined as $[\![M]\!]_I(\sigma)$, since σ is an environment over I.

The interpretation transformer associated to Π is the mapping

$$\mathcal{T}_\Pi : \text{H-INT}_\Sigma \to \text{H-INT}_\Sigma$$

defined as follows: For any $I \in \text{H-INT}_\Sigma$, $\mathcal{T}_\Pi(I)$ is the Herbrand interpretation J such that, for any $k/n \in \mathrm{FS} \cup \mathrm{PS}$ and $t_1, \ldots, t_n \in H_\Sigma$,

$$k_J(t_1, \ldots, t_n) = \max T,$$

where $T = \{[\![R']\!]_I \mid k(t_1, \ldots, t_n) := R'$ is a ground instance of some rule in $\Pi\}$

Theorem 5.5. For any Σ-*program* Π, $\mathcal{T}_\Pi$ *is well defined, continuous, and computable. Moreover, for any* $I \in \text{H-INT}_\Sigma$, *one has* $I \models \Pi$ *iff* $\mathcal{T}_\Pi(I) \sqsubseteq I$.

PROOF. To prove that $\mathcal{T}_\Pi$ is well defined, we must show that the set T from Definition 5.8 has a maximum element. Let $[\![R'_1]\!]_I$, $[\![R'_2]\!]_I$ be any two members of T. We claim that they must be identical, unless one of the two is bottom. In fact, by construction of T, there are two ground instances of rules in Π:

$$k(t_1, \ldots, t_n) := \{B'_1 \to\} M'_1,$$

$$k(t_1, \ldots, t_n) := \{B'_2 \to\} M'_2,$$

where $\{B'_i\} \to M'_i$ is R'_i $(i = 1, 2)$. Since rules are left linear, this means that the left hand sides of the two rules are unifiable (we could not conclude this without left linearity, since the constructs t_i may be infinite). Because of the nonambiguity restriction for BABEL programs (remember Definition 2.5), M_1' and M_2' must be identical, unless (B_1', B_2') is propositionally unsatisfiable. If M_1' and M_2' are identical, then $[\![R_1']\!]_I = [\![R_2']\!]_I$ unless one or both are bottom. Otherwise, $[\![B_1']\!]_I$ and $[\![B_2']\!]_I$ cannot be both true, and at least one of the two values $[\![R_1']\!]_I$, $[\![R_2']\!]_I$ must be bottom.

It can be checked that each k_J is continuous. This essentially reduces to the continuity of the functions in I and the semantic functions.

We now argue that $\mathscr{T}_\Pi$ is continuous. Monotonicity is obvious from the definition. Let $\{I_\alpha\}_{\alpha\in A}$ be a directed set of Herbrand interpretations. Let $I = \sqcup_{\alpha\in A} I_\alpha$. Put $J = \mathscr{T}_\Pi(I)$, $J_\alpha = \mathscr{T}_\Pi(I_\alpha)$. We must show that $J = \sqcup_{\alpha\in A} J_\alpha$. But $\sqcup_{\alpha\in A} J_\alpha \sqsubseteq J$ follows from monotonicity. To show $J \sqsubseteq \sqcup_{\alpha\in A} J_\alpha$, consider any $k/n \in \mathrm{FS} \cup \mathrm{PS}$ and $t_1, \ldots, t_n \in H_\Sigma$. Let $\mathscr{R}'$ be the set of all R' such that $k(t_1, \ldots, t_n) := R'$ is a ground instance of some rule in Π. Then

$$\begin{aligned}
k_J(t_1, \ldots, t_n) &= \max\{[\![R']\!]_I \mid R' \in \mathscr{R}'\}\\
&= \max\Big\{ \bigsqcup_{\alpha\in A} [\![R']\!]_{I\alpha} \Big| R' \in \mathscr{R}' \Big\}\\
&\qquad\qquad \Big(\text{by } I = \bigsqcup_{\alpha\in A} I_\alpha \text{ and continuity of evaluation}\Big)\\
&= \bigsqcup_{\alpha\in A} \max\{[\![R']\!]_{I\alpha} \mid R' \in \mathscr{R}'\}\\
&\qquad \Big(\text{since this is an upper bound of } \bigsqcup_{\alpha\in A} [\![R']\!]_{I\alpha} \text{ for any fixed } R' \in \mathscr{R}'\Big)\\
&= \bigsqcup_{\alpha\in A} k_{J_\alpha}(t_1, \ldots, t_n).
\end{aligned}$$

Let us now show that $\mathscr{T}_\Pi$ is computable. By the proof of Theorem 5.4 and known results from domain theory (see e.g. Larsen and Winskel [33]) we can assume that any finite Herbrand Σ-interpretation I is presented as a finite tuple $\langle (k_I)_{k\in \mathrm{FS}\cup\mathrm{PS}}\rangle$, where each k_I is presented in turn as a partial Σ-term s (if k's arity is 0) or as a finite set of pairs $(\bar{s}, s)$, where s is a partial Σ-term, and $\bar{s}$ is a tuple of finite Σ-constructs. This set of pairs is called the graph of k_I and determines k_I's effect as a mapping, by requiring

$$k_I(t) = \sqcup \{s \mid (\bar{s}, s) \in \mathrm{graph}(k_I),\ \bar{s} \sqsubseteq \bar{t}\}$$

to hold for all tuples $\bar{t}$ of Σ-constructs.

By appealing to Church's thesis and Definition 5.4, we may restrict ourselves to give an informal proof showing that

$$\{(I, J) \mid I, J \in \text{H-INT}_\Sigma \text{ are finite and such that } J \sqsubseteq \mathscr{T}_\Pi(I)\}$$

is a r.e. set. Indeed, for finite I, $J \in \text{H-INT}_\Sigma$ we have:

$J \sqsubseteq \mathscr{T}_\Pi(I)$

iff $k_J \sqsubseteq k_I$ for all $k \in \mathrm{FS} \cup \mathrm{PS}$

iff for all $k \in \mathrm{FS} \cup \mathrm{PS}$ and for all $(\bar{s}, s) \in \mathrm{graph}(k_J)$ $s \sqsubseteq k_{\mathscr{T}_\Pi(I)}(\bar{s})$ (here $\bar{s}$ can be omitted if k is nullary)

iff for all $k \in \mathrm{FS} \cup \mathrm{PS}$ and for all $(\bar{s}, s) \in \mathrm{graph}(k_J)$ there is some finite ground instance $k(\bar{s}) := R'$ of a rule in Π such that $s \sqsubseteq [\![R']\!]_I$.

In view of Definition 5.9, we have to justify the restriction to finite ground instances in the last step above. Since $\bar{s}$ is finite and local variables in BABEL rules occur only in the guards, the ground instance R' under consideration cannot have any occurrences of infinite constructs outside the guard. On the other side, the denotation of a guard in I depends continuously on the valuation of its variables. If some ground instance of the guard denotes true, the same happens already for some finite ground instance. Hence, if $s \sqsubseteq [\![R']\!]_I$ for some ground instance R', then this is also the case for some finite ground instance.

We note that we have obtained a definition of a r.e. set, because FS $\cup$ PS is finite, each k_J has a finite graph, Π is r.e., and each $[\![R']\!]_I$ is finite and effectively computable from R' and I.

Finally, assume that $\mathcal{T}_\Pi(I) = J$. The condition $J \sqsubseteq I$ means that $k_I(t_1, \ldots, t_n) \sqsupseteq k_J(t_1, \ldots, t_n)$ must hold for all $k/n \in \mathrm{FS} \cup \mathrm{PS}$ and all $t_1, \ldots, t_n \in H_\Sigma$. By the definition of $\mathcal{T}_\Pi$, this amounts to saying that $[\![L']\!]_I \sqsupseteq [\![R']\!]_I$ must hold for any ground instance of any rule in Π. This happens exactly when I is a model of all rules in Π, because building all ground instances is the same as considering all possible environments. Since all implicit rules hold in all Herbrand models, we may conclude that $\mathcal{T}_I(I) \sqsubseteq I$ iff $I \models \Pi$. □

We are now in a position to prove the main result of this section:

Theorem 5.6. Any BABEL *program* Π *of signature* Σ *has a least Herbrand model* I_Π. *Moreover,* I_Π *is a computable element of the domain* H-INT$_\Sigma$.

PROOF. Let I_Π be the least fixpoint of $\mathcal{T}_\Pi$. By Theorems 5.2 and 5.5, I_Π is a computable element of H-INT$_\Sigma$ and can be described as

$$I_\Pi = \bigsqcup_{j \in \mathbb{N}} I_\Pi^j,$$

where

$$I_\Pi^0 = \bot \text{ (bottom of H-INT}_\Sigma) \quad \text{and} \quad I_\Pi^{j+1} = \mathcal{T}_\Pi\left(I_\Pi^j\right).$$

Theorem 5.5 also guarantees that I_Π is a model of Π. Let I be any other Herbrand model of Π. By Theorem 5.5 again, we know that $\mathcal{T}_\Pi(I) \sqsubseteq I$. Since $\mathcal{T}_\Pi$ is monotonic, it follows by induction on j that $I_\Pi^j \sqsubseteq I$ for all $j \in \mathbb{N}$. We conclude that $I_\Pi \sqsubseteq I$ because I_Π's characterization as a lub. Hence, I_Π is the least Herbrand model. □

We can prove that BABEL's reduction semantics always computes logically sound outcomes.

Theorem 5.7. (Soundness of the reduction semantics). Let Π *be a* Σ*-program. Any (not necessarily lazy) narrowing sequence*

$$M \overset{\Pi}{\underset{\mathrm{N}}{\longrightarrow}}{}^{*}_{\sigma_{\mathrm{out}}} N$$

computes a sound outcome in the sense that

$$[\![M\sigma_{\mathrm{out}}]\!]_I(\rho) \sqsupseteq [\![\, | N | \,]\!]_I$$

holds for all models $I \models \Pi$ *and all environments* ρ *over* I.

PROOF. Assume that the length of the reduction is l. Consider any model $I \models \Pi$ and any environment ρ over I. It is easy to check that $[\![N]\!]_I(\rho) \sqsupseteq [\![\,|N|\,]\!]_I$. Hence, we may replace $|N|$ by N in our thesis. We use induction on l. The case $l = 0$ is trivial. For $l > 0$, we may assume

$$M \;{}_{\mathrm{N}}\!\xrightarrow[\sigma_{\mathrm{out},1}]{\Pi} M_1 \;{}_{\mathrm{N}}\!\xrightarrow[\sigma_{\mathrm{out,rest}}]{\Pi}{}^{*} N,$$

where, for some $u \in \mathbf{O}^+(M)$ and some variant $L := R$ of a rule in Π, M/u and L are unifiable with m.g.u. $\sigma_1 = \sigma_{\mathrm{out},1} \uplus \sigma_{\mathrm{in},1}$, and M_1 is $M[u \leftarrow R]\sigma_1$. Now we can reason as follows, knowing that $\sigma_{\mathrm{out}} = \sigma_{\mathrm{out},1}\sigma_{\mathrm{out,rest}} \upharpoonright \mathrm{var}(M)$:

$$[\![M\sigma_{out}]\!]_I(\rho) = [\![M\sigma_{\mathrm{out},1}\sigma_{\mathrm{out,rest}}]\!]_I(\rho)$$
$$= [\![M\sigma_{\mathrm{out},1}]\!]_I(\mu)$$

[where μ is such that $\mu(X) = [\![X\sigma_{\mathrm{out,rest}}]\!]_I(\rho)$ for all $X \in \mathrm{VS}$]

$$\sqsupseteq [\![M[u \leftarrow R]\sigma_1]\!]_I(\mu)$$

(because $I \models \Pi$, $L := R$ is a rule in Π, and the context of $M\sigma_{\mathrm{out},1}$ at occurrence u behaves monotonically in any interpretation)

$$= [\![M_1]\!]_I\mu$$
$$= [\![M_1\sigma_{\mathrm{out,rest}}]\!]_I(\rho) \qquad \text{(by construction of } \mu)$$
$$\sqsupseteq [\![N]\!]_I(\rho) \qquad \text{(by induction hypothesis)}.$$

This completes the proof. □

This result has a reciprocal one. It says that any logically sound outcome is subsumed by some other outcome that can be computed via lazy narrowing.

Theorem 5.8. (Completeness of the reduction semantics). Let Π *be a* Σ*-program. Assume a* Σ*-expression* M*, a partial* Σ*-term* s*, and a finite d-substitution* θ *with* $\mathrm{dom}(\theta) \subseteq \mathrm{var}(M)$ *such that* $[\![M\theta]\!]_I(\rho) \sqsupseteq [\![s]\!]_I$ *holds for any* $I \models \Pi$ *and any environment* ρ *over* I*. Then there exists a lazy narrowing sequence*

$$M \;{}_{\mathrm{N}}\!-1 \xrightarrow[\sigma_{\mathrm{out}}]{\Pi}{}^{*} N$$

and a finite d-substitution λ*, such that* $\theta = \sigma_{out}\lambda \upharpoonright \mathrm{var}(M)$ *and* $|N\lambda| \sqsupseteq s$.

SKETCH OF PROOF. Remember that the least Herbrand model I_Π of Π is the lub of a chain of approximations I^j_Π (Theorem 5.6). By the hypothesis and the continuity of evaluation, there is some $j \in \mathbb{N}$ such that the inequality $[\![M\theta]\!]_I(\rho) \sqsupseteq [\![s]\!]_I$ holds for $I = I^j_\Pi$. Using induction on j, it is possible to prove that $M\theta \text{-l}\rightarrow^* N'$ for some expression N' such that $|N'| \sqsupseteq s$. Notice that N' is obtained by lazy rewriting, binding no variables in $M\theta$. The lazy narrowing derivation $M \;{}_{\mathrm{N}}\!-1 \rightarrow {}^*N$ needed to prove the theorem is then produced by a lifting construction, similar to those known for SLD resolution (Lloyd [37]) and classical narrowing (Hullot [27]). A full proof of this completeness result can be found in Moreno-Navarro [41]. □

To finish, we present a kind of confluence result that can be derived from the correctness theorem.

Corollary 5.1. ("Confluence" of rewriting). Let Π *be a* Σ*-program. Assume ground* Σ*-expressions* M, N_1, N_2 *such that*

$$M \overset{\Pi}{\rightarrow} {}^* N_1 \quad \text{and} \quad M \overset{\Pi}{\rightarrow} {}^* N_2.$$

Then N_1 *and* N_2 *cannot have different constructors at any common occurrence; i.e.,* $\{ |N_1|, |N_2| \}$ *is consistent in* H_Σ *(or in* BOOL, *if* M *is boolean).*

PROOF. By applying Theorem 5.7 to the least Herbrand model of Π, we can conclude that $[M]_{I_\Pi}$ is a common upper bound of $|N_1|$ and $|N_2|$ $\square$

6. CONCLUSIONS

We believe we have provided a semantic framework which allows us to amalgamate functional and logic programming in a conceptually simple and semantically coherent way. We have chosen to design a functional logic language with an essentially functional syntax, a lazy form of constructor based conditional narrowing as operational semantics, and a declarative semantics that is based on Scott domains and provides least Herbrand models. Our approach reduces SLD resolution to narrowing, by viewing definite clauses as a particular kind of conditional rewrite rules, and extends to functional programs the declarative semantics typically used for Horn clause logic programs.

In contrast with other approaches that have advocated the reduction of narrowing to SLD resolution in order to exploit the extensive experience on efficient PROLOG implementations (van Emden and Yukawa [18], Bosco et al. [6], Levi et al. [35]), our aim has been to capitalize on the available experience in efficient implementation techniques for functional languages. The sequential kernel of a parallel (programmed) graph reduction machine (Loogen et al. [36]) has been extended with unification and backtracking mechanisms inspired by Warren's abstract machine for PROLOG (Warren [57]), yielding an abstract machine BAM for BABEL (Kuchen et al. [31]). A prototype emulator of BAM has been programmed in C and runs on SUN workstations. Early experiences with the prototype let us hope that purely applicative programs will run in the BABEL machine almost as efficiently as in the original graph reduction machine, though some overhead due to the different parameter passing mechanisms (unification instead of matching) cannot be avoided.

We are presently working on improving the design and implementation of BABEL along several lines. The BAM implementation actually supports a higher order extension of BABEL with a polymorphic type system (Milner [40], Martins-Damas [39]), where narrowing is kept as the evaluation mechanism, but with the restriction that higher order logic variables are not allowed; that is, higher order variables are never affected by narrowing, but used only for rewriting.

REFERENCES

1. Apt, K. R. and van Emden, M. H., Contributions to the Theory of Logic Programming, *J. Assoc. Comput. Mach.* 29:841-862 (1982).
2. Barendregt, H. P., *The Lambda Calculus: Its Syntax and Semantics*, North Holland, 1981.
3. Bellia, M. and Levi, G., The Relation between Logic and Functional Languages: A Survey, *J. Logic Programming* 3:217-236 (1986).

4. Bird, R. and Wadler, P., *Introduction to Functional Programming*, Prentice-Hall, 1988.
5. Bosco, P. G. and Giovannetti, E., IDEAL: An Ideal Deductive Applicative Language, in: *Proceedings of the IEEE International Symposium on Logic Programming 1986*, IEEE Computer Soc. Press, pp. 89-94.
6. Bosco, P. G., Giovannetti, E., and Moiso, C., Narrowing vs. SLD-Resolution, *Theoret. Comput. Sci.* 59:3-23 (1988).
7. Campbell, J. (ed.), *Implementations of PROLOG*, Ellis Horwood, 1984.
8. Darlington, J., Field, A. J., and Pull, H., The Unification of Functional and Logic Languages, in: [11], pp. 37-70.
9. Darlington, J. and Guo, Y., Narrowing and Unification in Functional Programming—an Evaluation Mechanism for Absolute Set Abstraction, Technical Report, Working Draft, Imperial College, Nov. 1988.
10. Darlington, J. and Guo, Y., The Unification of Functional and Logic Languages, towards Constraint Functional Programming, Technical Report, Imperial College, Working Draft, Sept. 1989.
11. DeGroot, D. and Lindstrom, G. (eds.), *Logic Programming: Functions, Relations and Equations*, Prentice-Hall, 1986.
12. Dershowitz, N., Computing with Rewrite Systems, *Inform. and Control* 65:122-157 (1985).
13. Dershowitz, N. and Josephson, A., Logic Programming by Completion, in: *Proceedings of the 2nd International Conference on Logic Programming*, Uppsala, Sweden, July 1984, pp. 313-320.
14. Dershowitz, N. and Plaisted, D. A., Logic Programming cum Applicative Programming, in: *Proceedings of the IEEE International Symposium on Logic Programming*, Boston, July 1985, IEEE Computer Soc. Press, pp. 54-66.
15. Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall, 1976.
16. Dincbas, M. and van Henternryck, P., Extended Unification Algorithms for the Integration of Functional Programming into Logic Programming, *J. Logic Programming* 4:197-227 (1987).
17. van Emden, M. H. and Kowalski, R., Semantics of Predicate Logic as a Programming Language, *J. Assoc. Comput. Mach.* 23:733-742 (Oct. 1976).
18. van Emden, M. H. and Yukawa, K., Logic Programming with Equations, *J. Logic Programming* 4:256-288 (1987).
19. Fay, M., First-Order Unification in an Equational Theory, in: *Proceedings of the 4th Workshop on Automated Deduction*, Austin, Tex., Feb. 1979, pp. 161-167.
20. Fribourg, L., SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting, in: *Proceedings of the IEEE International Symposium on Logic Programming*, Boston, July 1985, IEEE Computer Soc. Press, pp. 172-184.
21. Gallier, J. H. and Snyder, W., Complete Set of Transformations for General *E*-unification, *Theoret. Comput. Sci.* 67:203-260 (1989).
22. Giovannetti, E. and Moiso, C., A completeness result for *E*-unification algorithms based on conditional narrowing, in: *Proceedings Workshop on Foundations of Logic and Functional Programming*, Trento, Italy, Dec. 1986, Lecture Notes in Comput. Sci. 306, Springer-Verlag, 1986, pp. 157-167.
23. Goguen, J. A. and Meseguer, J., EQLOG: Equality, Types and Generic Modules for Logic Programming, in: [11], p. 295-363.
24. Hudak, P., Conception, Evolution and Application of Functional Programming Languages, *ACM Comput. Surveys* 21(3):359-411 (1989).
25. Huet, G. and Levy, J. J., Computations in Nonambiguous Linear Rewriting Systems, Technical Report 359, INRIA Le Chesnay, France, 1979.
26. Huet, G. and Oppen, D. C., Equations and Rewrite Rules: A Survey, in: R. V. Book (ed.), *Formal Language Theory: Perspectives and Open Problems*, Academic, 1980, pp. 349-405.
27. Hullot, J. M., Canonical Forms and Unification, in: *Proceedings of the 5th Conference on*

Automated Deduction, July 1980, Lecture Notes in Comput. Sci. 87, Springer-Verlag, 1980, pp. 318–334.

28. Josephson, A. and Dershowitz, N., An Implementation of Narrowing: The RITE Way, in: *Proceedings Conference on Logic Programming 1986*, IEEE Computer Soc. Press, 1986, pp. 187–197.
29. Kaplan, S., Conditional Rewrite Rules, *Theoret. Comput. Sci.* 33:175–193 (1984).
30. Kaplan, S. and Jouannoud, J. P. (eds.), *Conditional Term Rewriting Systems, Proceedings*, Lecture Notes in Comput. Sci. 308, Springer-Verlag, 1988.
31. Kuchen, H., Loogen, R., Moreno-Navarro, J. J., and Rodriguez Artalejo, M., Graph-Based Implementation of a Functional Logic Language, in: *Proceedings of the European Symposium on Programming (ESOP) 1990*, Copenhagen, Lecture Notes in Comput. Sci. 432, Springer-Verlag, 1990, pp. 271–290.
32. Lankford, D. S., Canonical Inference, Tech. Report ATP-32, Dept. of Mathematics and Computer Science, Univ. of Texas at Austin.
33. Larsen, K. G. and Winskel, G., Using Information Systems to Solve Recursive Equations Effectively, in: *Lecture Notes in Comput. Sci.* 173, Springer-Verlag, 1984, pp. 109–129.
34. Lassez, J. L., Maher, M. J., and Marriott, K., Unification Revisited, in: M. Boscarol, L. Carducci Aiello, and G. Levi (eds.), *Foundations of Logic and Functional Programming, Workshop*, Trento, Italy, Dec. 1986, Lecture Notes in Comput. Sci. 306, Springer-Verlag, 1987, pp. 67–113.
35. Levi, G., Bosco, P. G., Giovannetti, E., Moiso, C., and Palamidesi, C., A Complete Semantic Characterization of K-LEAF, a Logic Language with Partial Functions, in: *Proceedings 4th Symposium on Logic Programming*, San Francisco, 1987, pp. 1–27.
36. Loogen, R., Kuchen, H., Indermark, K., and Damm, W., Distributed Implementation of Programmed Graph Reduction, in: *Proceedings Conference on Parallel Architectures and Languages Europe (PARLE) 1989,* Lecture Notes in Comput. Sci. 365, Springer-Verlag, 1989.
37. Lloyd, J. W., *Foundations of Logic Programming*, 2nd ed., Springer-Verlag, 1987.
38. Malachi, Y., Manna, Z. and Waldinger, R., TABLOG: A New Approach to Logic Programming, in: [11], pp. 365–394.
39. Martins-Damas, L. M., Type Assignment in Programming Languages, Ph.D. Thesis, Univ. of Edinburgh, 1985.
40. Milner, R., A Theory of Type Polymorphism in Programming, *J. Comput. System Sci.* 17(3):348–375 (1978).
41. Moreno-Navarro, J. J., BABEL: Diseño, Semántica e Implementación de un Lenguaje que Integra la Programación Functional y Lógica (in Spanish), Ph.D. Thesis, Facultad de Informática de Madrid, July 1989.
42. Mulmuley, K., Full Abstraction and Semantic Equivalence, ACM Doctoral Dissertation Award, 1986, MIT Press, 1987.
43. Nutt, W., Réty, P., and Smolka, G., Basic Narrowing Revisited, *J. Symbolic Comput.* 7:295–317 (1988).
44. O'Donnell, M. J., *Equational Logic as a Programming Language*, MIT Press, 1985.
45. Peyton-Jones, S., *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.
46. Reddy, U. S., Transformation of Logic Programs into Functional Programs, in: *Proceedings of the IEEE International Symposium on Logic Programming*, IEEE Computer Soc. Press, 1984, pp.187–197.
47. Reddy, U. S., Narrowing as the Operational Semantics of Functional Languages, in: *Proceedings of the IEEE International Symposium on Logic Programming,* IEEE Computer Soc. Press, July 1985, pp. 138–151.
48. Reddy, U. S., Functional Logic Languages, Part I, in: J. H. Fasel and R. M. Keller (eds.), *Proceedings of a Workshop on Graph Reduction*, Lecture Notes in Comput. Sci. 279, Springer-Verlag, 1987, pp. 401–425.

49. Remy, J. L. and Zhang, H., Contextual Rewriting, in: *Proceedings of the Conference on Rewriting Techniques and Applications*, Dijon, 1985.
50. Robinson, J. A., A Machine Oriented Logic Based on the Resolution Principle, *J. Assoc. Comput. Mach.* 12:23-41 (1965).
51. Rogers, H., *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, 1967.
52. Scott, D. S., Domains for Denotational Semantics, in: *Proceedings ICALP'82*, Lectures Notes in Comput. Sci. 140, Springer-Verlag, 1982, pp. 577-613.
53. Shapiro, E., The Family of Concurrent Logic Programming Languages, *ACM Comput. Surveys* 21:413-510 (1989).
54. Siekmann, J. and Szabo, P., Universal Unification and a Classification of Equational Theories, in: *Proceedings 6th Conference on Automated Deduction*, New York, June 1982, Lecture Notes in Comput. Sci. 13, Springer-Verlag, 1982, pp. 369-389.
55. Slagle, J. R., Automated Theorem Proving with Theories with Simplifiers, Commutativity and Associativity, *J. Assoc. Comput. Mach.* 21:622-642 (1974).
56. Subrahmanyam, P. A. and You, J. H., FUNLOG: A Computational Model Integrating Logic Programming and Functional Programming, in: [11], pp. 157-198.
57. Warren, D. H. D., An Abstract PROLOG Instruction Set, Technical Note 309, SRI International, Menlo Park, Calif., Oct. 1983.
58. You, J. H., Outer Narrowing for Equational Theories Based on Constructors, in: T. Lepistö and A. Salomaa (eds.), *Proceedings ICALP'88*, Lecture Notes in Comput. Sci. 317, Springer-Verlag, 1988, pp. 727-741.
59. You, J. H., Solving Equations in an Equational Language, in: J. Grabowski, P. Lescanne, and W. Wechler (eds.), *Proceedings Algebraic and Logic Programming 1988*, Lecture Notes in Comput. Sci. 343, Springer-Verlag, 1989, pp. 245-254.
60. You, J., Enumerating Outer Narrowing Derivations for Constructor-Based Term Rewriting Systems, *J. Symbolic Comput.* 7:319-341 (1989).