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
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 (\(\lambda\)-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 [43]); 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], Sickman 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], Subrahmanym 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 Hentenryck [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** $X, Y, Z \in DV$
- **Boolean variables** $X, Y, Z \in BV$
- **Constructors** $c, d, e \in CS$
- **Function symbols** $f, g, h \in FS$
- **Predicate symbols** $p, q, r \in PS$

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 $DV \cup BV$ as VS. The set $\Sigma = CS \cup FS \cup PS$ 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 $\Sigma$-terms ($t \in D\text{Term}_\Sigma$), boolean $\Sigma$-terms ($b \in B\text{Term}_\Sigma$), data $\Sigma$-expressions ($E \in D\text{Exp}_\Sigma$), and boolean $\Sigma$-expressions ($B, C \in B\text{Exp}_\Sigma$) are defined as follows:

$t:: = X$ \hfill \% data variable
\[ | c \] \hfill \% constant, i.e. nullary constructor
\[ | c(t_1, \ldots, t_n) \] \hfill \% construction

$b:: = X$ \hfill \% boolean variable
\[ | \text{true} \] \hfill \% truth
\[ | \text{false} \] \hfill \% falsity

$E:: = t$ \hfill \% data term
\[ | c(E_1, \ldots, E_n) \] \hfill \% constructor application
\[ | f(E_1, \ldots, E_n) \] \hfill \% function application
\[ | (C \rightarrow E) \] \hfill \% guarded expression
\[ | (C \rightarrow E_1 \mathbin{\square} E_2) \] \hfill \% conditional expression
The sets of \( \Sigma \)-terms and \( \Sigma \)-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 \textsc{babel}.

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

Expressions of the form \( C \cdot M \), \( C \cdot M_1 \cdot \square M_2 \)
are intended to mean
\[
\begin{align*}
\text{if } C \text{ then } M \\
\text{else undefined}
\end{align*}
\]
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 \textsc{babel} rule has one of the two following forms:

\[
\begin{align*}
(P) & \quad f(t_1, \ldots , t_n) := \{ C \to \} E. \quad \% \text{function rule} \\
(P) & \quad p(t_1, \ldots , t_n) := \{ C \to \} B. \quad \% \text{predicate rule}
\end{align*}
\]
where curly braces indicate that the presence of \( "C\!\to\!" \) is optional.

We shall use the following terminology:

\[
\begin{align*}
(1) & \quad f(t_1, \ldots , t_n) \text{ or } p(t_1, \ldots , t_n) \text{ is the rule's left hand side (lhs).} \\
(2) & \quad C \to E \text{ or } C \to B \text{ is the rule's right hand side (rhs).} \\
(3) & \quad C \text{ is the rule's guard.} \\
(4) & \quad E \text{ or } B \text{ is the rule's body.}
\end{align*}
\]

Any \textsc{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
\[ \forall X_1 \cdots \forall X_r \forall Y_1 \cdots \forall Y_s (C := k(t_1, \ldots, t_n) \equiv M\}) \]
or, equivalently,
\[ \forall X_1 \cdots \forall X_r \{ (\exists Y_1 \cdots \exists Y_s C := k(t_1, \ldots, t_n) \equiv M\}) \]
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:

<table>
<thead>
<tr>
<th>Raw form</th>
<th>Sugared form</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p(t_1, \ldots, t_n) := )</td>
<td>( p(t_1, \ldots, t_n) ).</td>
</tr>
<tr>
<td>( p(t_1, \ldots, t_n) := C \rightarrow ) true.</td>
<td>( p(t_1, \ldots, t_n) := C ).</td>
</tr>
<tr>
<td>( p(t_1, \ldots, t_n) := ) false.</td>
<td>( \neg p(t_1, \ldots, t_n) ).</td>
</tr>
<tr>
<td>( p(t_1, \ldots, t_n) := C \rightarrow ) false.</td>
<td>( \neg p(t_1, \ldots, t_n) ).</td>
</tr>
</tbody>
</table>

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:

\[
PC(\text{true}) = PC(\text{false}) = \emptyset, \\
PC(\neg B) = PC(B), \\
PC(B_1, B_2) = PC(B_1 \land B_2) = PC(B_1) \cup PC(B_2), \\
PC(C \rightarrow B) = PC(C) \cup PC(B), \\
PC(C \rightarrow B_1 \sqcap B_2) = PC(C) \cup PC(B_1) \cup PC(B_2), \\
PC(B) = \{ B \} \quad \text{in any other case.}
\]
Definition 2.4. A boolean expression \( B \) is propositionally unsatisfiable iff the truth value of \( B \) with respect to any evaluation

\[
V : PC(B) \rightarrow \{ \text{true}, \text{false}, \bot \}
\]

is either false or the **undefined boolean value** \( \bot \). The truth value w.r.t. \( V \) 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 \textsc{babel} programs.

Definition 2.5. A \textsc{babel} program is any recursively enumerable set \( \Pi \) of \textsc{babel} rules that satisfies a nonambiguity restriction.

Restriction 2.3. (Nonambiguity). Given any two rules in \( \Pi \) 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. \( \theta \) 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. \( \theta \) such that \( (B, C)\theta \) is propositionally unsatisfiable.

When necessary, we shall speak of \( \Sigma \)-rules and \( \Sigma \)-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 \textsc{babel} programs in order to illustrate the expressive power of the language and to allow comparison with the programming style in \textsc{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**
- \([0]/0\)  % empty list
- \([\cdot | \cdot ]/2\) % list constructor

**predicates**
- append/3

**rules**
- /* A1 */ append([], Y, Z) :- Z = Y.
- /* 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**
- \([0]/0, [\cdot | \cdot ]/2\) % 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 ([I, Ys]) := Ys = [I.
- /* 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

\[
\text{equal-frontier}\left(\text{node}\left(\text{node}(\text{tip}(1), \text{tip}(2)), \text{tip}(3))\right), \text{node}(\text{node}(\text{tip}(1), \text{tip}(3)), \text{tip}(2))\right)
\]

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

\[
\text{equal-frontier}\left(\text{node}(\text{tip}(X), A), \text{node}(B, \text{tip}(Y))\right)
\]

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

Notice that we might change the first rule to

\[/* \text{EF' */ equal-frontier}(a, b) :- \text{equal-list}(\text{frontier}(A), \text{frontier}(B)).\]

Without sugaring, this amounts to

\[/* \text{EF' */ equal-frontier}(A, B) :- \text{equal-list}(\text{frontier}(A), \text{frontier}(B)) \to \text{true}.\]

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**

\[\text{tony/0}, \text{mike/0}, \text{john/0}, \text{rain/0}, \text{snow/0}\]

**predicates**

\[\text{alpinist/1}, \text{climber/1}, \text{skier/1}, \text{likes/2}\]

**rules**

\[/* \text{AC1 */ alpinist}(tony).\]
\[/* \text{AC2 */ alpinist}(mike).\]
\[/* \text{AC3 */ alpinist}(john).\]
\[/* \text{SC */ climber}(X) :- alpinist(X), \neg \text{skier}(X).\]
\[/* \text{LK1 */ \neg \text{likes}(X, \text{rain}) :- \text{climber}(X).\]
\[/* \text{LK2 */ \neg \text{skier}(X) :- \neg \text{likes}(X, \text{snow}).\]
\[/* \text{LK3 */ \neg \text{likes}(\text{mike}, X) :- \text{likes}(\text{tony}, X).\]
\[/* \text{LK4 */ \text{likes}(\text{mike}, X) :- \neg \text{likes}(\text{tony}, X).\]
\[/* \text{LK5 */ \text{likes}(\text{tony}, \text{rain}).\]
\[/* \text{LK6 */ \text{likes}(\text{tony}, \text{snow}).\]
To solve the puzzle, **BABEL** must solve the boolean expression

\[ G: (\text{alpinist}(X), \text{climber}(X), \neg \text{skier}(X)) \rightarrow \text{true}. \]

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:

\[
/* \text{SC}’ */ \text{skier}(X) :- \text{alpinist}(X), \neg \text{climber}(X).
/* \text{LK1}’ */ \neg \text{climber}(X) :- \text{likes}(X, \text{rain}).
/* \text{LK2}’ */ \text{likes}(X, \text{snow}) :- \text{skier}(X).
\]

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**

\[
[\ ]/0, [\cdot \cdot \cdot ]/2 \quad \% \text{ lists again}
0/0 \quad \% \text{ zero}
suc/1 \quad \% \text{ successor}
\]

**functions**

\[
\text{hamming-seq}/0
\text{merge-3}/3, \text{merge-2}/2
\text{seq-prod}/2
\text{+}/2, \ast/2
\]

**predicates**

\[
\text{nth-hamming}/2
\text{nth-member}/3
\text{<}/2
\]

**rules**

\[
/* \text{NH} */ \text{nth-hamming}(N, M) :- \text{nth-member}(N, \text{hamming-seq}, M).
/* \text{NM1} */ \text{nth-member}(1, [X | X_s], Y) :- Y = X.
/* \text{NM2} */ \text{nth-member}(\text{suc}(\text{suc}(N)), [X | X_s], Y) :-
\text{nth-member}(\text{suc}(N), X_s, Y).
/* \text{HS} */ \text{hamming-seq} := \text{merge-3(}\text{seq-prod}(2, [1 | \text{hamming-seq}]),
\text{seq-prod}(3, [1 | \text{hamming-seq}]),
\]

seq-prod(5, [1 | hamming-seq])).

/* SP */ seq-prod(X, [Y | Y]) := [X * Y | seq-prod(X, Y)].

/* M3 */ merge-3(X_s, Y_s, Z_s) := merge-2(X_s, merge-2(Y_s, Z_s)).

/* M2 */ merge-2([X | X_s], [Y | Y_s]) :=

\[ X < Y \rightarrow [X | \text{merge-2}(X_s, [Y | Y_s])] \]
\[ Y < X \rightarrow [Y | \text{merge-2}([X | X_s], Y_s)] \]

/* otherwise */ [X | merge-2(X_s, Y_s)].

/* +1 */ X + 0 := X.

/* +2 */ X + suc(Y) := suc(X + Y).

/* *1 */ X * 0 := 0.

/* *2 */ X * suc(Y) := (X * Y) + X.

/* < 1 */ \( \neg X < 0 \).

/* < 2 */ 0 < suc(Y).

/* < 3 */ suc(X) < suc(Y) := X < Y.

At first sight, \( \text{nth-hamming} \) looks like a PROLOG predicate. It may be used to solve goals in several modes:

\( \text{nth-hamming}(5, M) \) % true for \( M = 6 \)
\( \text{nth-hamming}(N, 10) \) % true for \( N = 8 \)
\( \text{nth-hamming}(N, M) \) % 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 \( \Sigma \)-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 \( VS \cup CS \cup FS \cup PS \cup BS \cup TV \), whose finite domain \( O(M) \) satisfies:

(i) \( O(M) \) is not empty and prefix closed.

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

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

\[ u \leq v \iff \exists \ w \ u . w = v , \]
\[ u < v \iff u \leq v \text{ and } u \neq v . \]
For \( u \in O(M) \), we have:

\begin{align*}
M[u] & : \text{Symbol at occurrence } u \\
M/u & : \text{Subexpression at occurrence } u \\
M[u \leftarrow N] & : \text{Result of replacing } M/u \text{ by } N \text{ in } M
\end{align*}

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

The set of nonvariable occurrences of \( M \) is defined as

\( O^+(M) = \{ u \in O(M) | M[u] \notin VS \} \).

**Definition 4.2.** A \( \Sigma \)-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 \( \sigma \) can be uniquely extended to a mapping from \( \text{Exp}_\Sigma \) to \( \text{Exp}_\Sigma \), also denoted by \( \sigma \). We denote the application of \( \sigma \) to \( M \) as \( M\sigma \).

The domain of a substitution \( \sigma \) is defined as

\( \text{dom}(\sigma) = \{ X \in VS | X \sigma \neq X \} \).

\( \sigma \) is called finite iff \( \text{dom}(\sigma) \) is finite.

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

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

The restriction of a substitution \( \sigma \) to a set of variables \( V \subseteq VS \) is denoted as \( \sigma \uparrow V \).

The composition of two substitutions \( \sigma, \theta \) 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 \( \nu = [M_1/X_1, \ldots, M_r/X_r] \), where \( \{ X_1, \ldots, X_r \} = \text{dom}(\sigma) \).

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 \textsc{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 \( k/n \in \text{FS} \cup \text{PS} \cup \text{BS} \). For the purposes of this definition, we apply \( k \) in prefix form even if it is a predefined \textsc{babel} symbol. A linear unification problem asks for the unification of two expressions without shared variables:

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

where \( M \in \text{Exp}_\Sigma, t \in \text{DTerm}_\Sigma \), 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 \( U_0 = \{M_1 \downarrow t_1, \ldots, M_n \downarrow t_n\}, \sigma_0 = \varepsilon \).

**Stage 2:** Don't care nondeterministically rewrite the initial unification configuration \( (U_0, \sigma_0) \), using the rules which follow, until a nonreducible configuration \( (U, \sigma) \) is reached. If \( U = \emptyset \), report ``SUCCESS'' and output \( \sigma \) 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 \( l(N_1, \ldots, N_n) \downarrow_i c(s_1, \ldots, s_v) \), where \( l/u \in FS \cup PS \cup BS \), \( c/v \in CS \cup TV \), 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) \( \{c(N_1, \ldots, N_n) \downarrow_i c(s_1, \ldots, s_u)\} \cup U, \sigma \rightarrow \{N_1 \downarrow s_1, \ldots, N_n \downarrow s_u\} \cup U, \sigma \), where \( c/u \in CS \cup TV \), \( u \geq 0 \).

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

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

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

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

\[ \text{Unify: } k(M_1, \ldots, M_n), k(t_1, \ldots, t_n) \]

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

\[ \sigma = \sigma_{\text{out}} \cup \sigma_{\text{in}}, \]

where

\[ \sigma_{\text{out}} = \sigma \upharpoonright \text{var}(k(M_1, \ldots, M_n)), \]

\[ \sigma_{\text{in}} = \sigma \upharpoonright \text{var}(k(t_1, \ldots, t_n)) \]

Moreover, \( \sigma_{\text{out}} \) is a d-substitution.
Notice that the part $\sigma_{\text{out}}$ of a computed m.g.u. reflects the flow of data (constructed terms) from a rule to an expression, while the part $\sigma_{\text{in}}$ 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

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

$$\sigma_{\text{out}} = \left[ \text{suc}(X')/X \right].$$

**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 $\Sigma$ be any BABEL signature, with set of constructors $CS$. The implicit $\Sigma$-rules for BABEL primitives are as follows:

**Rules for guards and conditionals:**

$$(\text{true} \rightarrow X) := X.$$

$$(\text{true} \rightarrow X \sqcap Y) := X.$$

$$(\text{false} \rightarrow X \sqcap Y) := Y.$$

**Rules for propositional connectives:**

$$\neg \text{true} := \text{false}.$$  

$$\neg \text{false} := \text{true}.$$  

$$(\text{false}, Y) := \text{false}.$$  

$$(\text{false}; Y) := Y.$$  

$$(\text{true}, Y) := \text{true}.$$  

$$(\text{true}; Y) := \text{false}.$$  

$$(X, \text{false}) := \text{false}.$$  

$$(X; \text{false}) := X.$$  

$$(X, \text{true}) := X.$$  

$$(X; \text{true}) := \text{true}.$$  

**Rules for weak equality:**

$$c = c := \text{true}.$$  

for all $c/n \in CS \cup TV$.

$$c(X_1, \ldots, X_n) = c(Y_1, \ldots, Y_n) := X_1 = Y_1, \ldots, X_n = Y_n.$$  

for all $c/n \in CS$, $n \geq 1$.

$$c(X_1, \ldots, X_n) = d(Y_1, \ldots, Y_n) := \text{false}.$$  

for all $c/n, d/m \in CS$, $c \neq d$, $n, m \geq 0$. 


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

Notice that for any program $\Pi$, $\hat{\Pi}$ 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 $\Pi$ be a $\Sigma$-program and let $N$ be a $\Sigma$-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. $\sigma$, we say that $L := R$ applies to $N$ via $\sigma$. 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 $\Sigma$-program $\Pi$ and a $\Sigma$-expression $M$. We say that $u \in O^+(M)$ 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 $LR_{\Pi}[M]$ defined by recursion on $M$'s structure as follows. In the following lines, $k$ stands for a symbol belonging to $FS \cup PS \cup BS$, where $FS$ and $PS$ are given by $\Sigma$, 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 $BS$.

$$LR_{\Pi}[M] = \emptyset \quad \text{if } M \text{ is a term.}$$

$$LR_{\Pi}\left[c(M_1, \ldots, M_n)\right] = \bigcup_{i=1}^{n} i.LR_{\Pi}[M_i] \quad \text{if } c/n \in CS, \ n > 0, \text{ and some } M_i \text{ is not a term.}$$

$$LR_{\Pi}\left[k(M_1, \ldots, M_n)\right] = \{\epsilon | \text{ some rule in } \hat{\Pi} \text{ applies to } k(M_1, \ldots, M_n)\} \cup \bigcup_{i \in I} i.LR_{\Pi}[M_i],$$

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

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

$$M \Downarrow \Pi \sigma_{\text{out}} M[u \leftarrow R] \sigma.$$ 

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

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

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

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

$$M \Downarrow \Pi \sigma_{\text{out}} N$$

iff there is some narrowing sequence

$$M = M_0 \Downarrow \Pi \sigma_{\text{out}, 1} M_1 \cdots M_{l-1} \Downarrow \Pi \sigma_{\text{out}, i} M_i \cdots N \Downarrow \Pi \sigma_{\text{out}, l} M_l = N$$

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

$$M \Downarrow \star$$

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

$$M \Downarrow \Pi \star M \Downarrow \Pi \star N.$$ 

In the case of lazy rewriting, we shall write

$$M \Downarrow \Pi \star N.$$ 

In practice, the program will be known by context and $\Pi$ 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 Hentenryck [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

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

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 $\Sigma$ be a signature with set of constructors $CS$. The shell $|N|$ of any given $\Sigma$-expression $N$ is defined recursively:

|true| = true,  
|false| = false,  
|c| = $c_0$ for any other boolean expression,  
|c_{N_1, \ldots, N_m}| = $c_{|N_1|, \ldots, |N_m|}$ for $c \in CS$,  
|N| = $\bot_d$ for any other data expression.

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

Let $\Pi$ be a $\Sigma$-program. For any narrowing derivation

$$MN \xrightarrow{\Pi} N$$

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

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

Notice that, for any outcome, $o_{out}$ 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:

```prolog
append(X_s, Y_s, [a, b, c])
N \xrightarrow{A_2} o_{out,1} (a = X_1, append(X_{s1}, Y_s, [b, c])) \rightarrow \text{true} \quad \% \text{where } o_{out,1} = [[X_1 \mid X_{s1}]/X_s]
N \xrightarrow{IR} \text{true} \quad \% \text{where } o_{out,2} = [a/X_1]
N \xrightarrow{A_1} (b, c) \rightarrow \text{true} \quad \% \text{where } o_{out,3} = [[1]/X_{s1}]
N \xrightarrow{IR} \text{true} \quad \% \text{where } o_{out,4} = [[b, c]/Y_s]
```
The outcome is (true, \( \sigma_{\text{out}} \)) with \( \sigma_{\text{out}} = [\{a\}/X, [b, c]/Y] \). 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, \textit{BABEL} behaves as a lazy functional language and is able to detect inequality without having to evaluate the whole frontiers; e.g.

\[
\text{equal-frontier}(\text{node}(\text{tree}(\text{tip}(1), \text{tip}(2)), \text{tip}(3)), \\
\text{node}(\text{tree}(\text{tip}(1), \text{tip}(3)), \text{tip}(2)))
\]

\[
\begin{align*}
\texttt{EF} & \quad -1 \rightarrow \quad \text{equal-list}(\text{frontier}(\text{node}(\text{node}(\text{tip}(1), \text{tip}(2)), \text{tip}(3))), \text{frontier}(\text{node}(\text{node}(\text{tip}(1), \text{tip}(3)), \text{tip}(2)))) \\
\texttt{FT2} & \quad -1 \rightarrow \quad \text{equal-list}(\{2 \mid \text{frontier}(\text{tip}(3))\}, \{3 \mid \text{frontier}(\text{tip}(2))\}) \\
\texttt{N} & \quad -1 \rightarrow \quad \text{false}
\end{align*}
\]

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.

\[
\text{equal-frontier}(\text{node}(\text{tree}(X), A), \text{node}(B, \text{tip}(Y)))
\]

\[
\begin{align*}
\texttt{EF} & \quad -1 \rightarrow \quad \text{equal-list}(\text{frontier}(\text{node}(\text{tree}(X), A)), \text{frontier}(\text{node}(B, \text{tip}(1)))) \\
\texttt{FT2} & \quad -1 \rightarrow \quad \text{equal-list}(\{X \mid \text{frontier}(A)\}, \{Z \mid \text{frontier}(\text{tip}(Y))\}) \\
\texttt{N} & \quad -1 \rightarrow \quad \sigma_{\text{out}, 1} \quad \text{equal-list}(\{X \mid \text{frontier}(A)\}, \{Z \mid \text{frontier}(\text{tip}(Y))\}) \\
\texttt{EL2} & \quad -1 \rightarrow \quad \text{equal-atom}(X, Z), \text{equal-list}(\text{frontier}(A), \text{frontier}(\text{tip}(Y)))
\end{align*}
\]

\%

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

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

\[
\begin{align*}
\text{alpinist}(X), \text{climber}(X), \lnot \text{skier}(X) & \rightarrow \text{true} \\
\text{climber}(\text{mike}), \lnot \text{skier}(\text{mike}) & \rightarrow \text{true} \quad \%
\end{align*}
\]

\[
\begin{align*}
\texttt{AC2, IR} & \quad -1 \rightarrow \quad \sigma_{\text{out}, 1} \quad \text{alpinist}(X), \text{climber}(X), \lnot \text{skier}(X) \rightarrow \text{true} \\
\texttt{LK2} & \quad -1 \rightarrow \quad (\text{climber}(\text{mike}), \lnot \text{skier}(\text{mike})) \rightarrow \text{true} \quad \%
\end{align*}
\]

\[
\text{climber}(\text{mike}), \lnot (\text{likes}(\text{mike}, \text{snow}) \rightarrow \text{false}) \rightarrow \text{true}
\]

\%

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:

\[
\text{nth-hamming}(5, M) \\
\text{\texttt{nth-member}(5, \text{hamming-seq}, M) \rightarrow true}
\]

Now hamming-seq is demanded by the rules for nth-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

\[
\text{nth-member}(5, [2 | \text{RHS}], M) \rightarrow true
\]

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

\[
\text{nth-member}(4, \text{RHS}, M) \rightarrow true
\]

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.

\[
\text{nth-hamming}(N, 10)^N - 1 \rightarrow_{[9/N]} \text{true}
\]

\[
\text{nth-hamming}(N, M)^N - 1 \rightarrow_{[6/N, 8/M]} \text{true}
\]

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:

```plaintext
constructors
doctor/0, sus/1  % for natural numbers
[ ]/0, [· · ]/2  % for lists
functions
    f/2    g, h/1
rules
/* F_0 */ f(0, X) := h(X).
/* F_1 */ f(suc(N), [Y Y]) := [suc(Y) | f(N, Y)].
/* G */ g(N) := [N | g(suc(N))].
/* H */ h([Y Y]) := [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 \( NS_1, NS'_1 \), 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-
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:

\[
\begin{align*}
\text{f}(N, g(0)) \\
N = 0 & | F_0 \\
h(g(0)) & | G \\
h(0 | g(1)) & | H \\
[0]
\end{align*}
\]

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 Emde [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 \( \bot \) (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

\[
S_x = \{ z | z \sqsubseteq x \text{ and } z \text{ is finite} \}
\]

is directed and \( x = \sqcup S_x \). \( D \) is \( \omega \)-algebraic if, in addition, the set \( \text{Fin}_D \) 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 \( \omega \)-algebraic cpo. For a different characterization of domains, see Scott [52].
Definition 5.2. Let $D$ be a domain. An enumeration of $\text{Fin}_D$ is any mapping $e$ from $\mathbb{N}$ onto $\text{Fin}_D$ that maps 0 to $\perp$; i.e.,

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

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):

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

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

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

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 $x \leq y$ implies $f(x) \leq f(y)$ for all $x, y \in C$.

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

If $C$ and $D$ are domains, we shall denote by $[C \rightarrow D]$ 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 $f : C \rightarrow D$ is called computable iff its graph, defined as the predicate on $\mathbb{N}$ given by

$$G_f(i, j) \quad \text{def} \quad e'_j \subseteq f(e_i),$$

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

$$G_x = \{i \in \mathbb{N} | e'_i \subseteq x\}$$

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 $[C \rightarrow D]$, 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 $f : D \rightarrow D$ has a least fixpoint $x \in D$ which satisfies $f(x) = x$ and $x \subseteq y$ for any $y \in D$ such that $f(y) \subseteq y$.

In fact, this least fixpoint is the lub of the chain

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

Moreover, the mapping $\text{fix} : [D \rightarrow D] \rightarrow D$ which assigns to any $f \in [D \rightarrow D]$ its least fixpoint is itself continuous; i.e., $\text{fix} \in [[D \rightarrow D] \rightarrow D]$. 
If $D$ is effectively presentable, then $\text{fix}$ is computable, and $\text{fix}(f)$ is computable for any computable $f \in [D \to D]$.

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

**Definition 5.5.** Let $\Sigma$ be a BABEL signature with set of constructors $CS$.

The Herbrand domain $H_\Sigma$ 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 $\Sigma$-constructs. In particular, all ground data $\Sigma$-terms are $\Sigma$-constructs. By a partial data $\Sigma$-term we shall understand a finite $\Sigma$-construct.

The ordering $\sqsubseteq$ on $H_\Sigma$ is defined as follows:

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

The boolean domain $\text{BOOL}$ is the flat cpo given by the diagram

\[
\begin{array}{c}
\text{true} \\
\bot_b \\
\text{false}
\end{array}
\]

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

Simply by checking the corresponding definitions, we obtain:

**Theorem 5.3.** $H_\Sigma$ and $\text{BOOL}$ are domains. Moreover, $H_\Sigma$ 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) $H_\Sigma$ is effectively presentable.

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

**Definition 5.6.** Let $\Sigma$ be a signature with $CS$, $FS$, $PS$. A $\Sigma$-interpretation is any algebra

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

where $D$ is a domain and the following conditions hold:

$c_I \in D$ for $c / 0 \in CS$,

$c_I \in [D^n \to D]$ for $c / n \in CS$, $n > 0$,

$f_I \in D$ for $f / 0 \in FS$,

$f_I \in [D^n \to D]$ for $f / n \in FS$, $n > 0$,

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

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


I is called a computable interpretation iff \( D \) is effectively presentable and \( c_f, f_t, p_t \) are computable mappings.

I is called a Herbrand interpretation iff \( D \) is \( H_\Sigma \) and the interpretation of constructors is free, i.e.,

\[
c_f = c \quad \text{for } c/0 \in CS,
\]

\[
c(t_1, \ldots, t_n) = \underbrace{t_1 \cdots t_n}^c \quad \text{for } c/n \in CS, \ n > 0, \text{ and } t_i \in H_\Sigma.
\]

In any given interpretation, BABEL expressions denote domain elements.

**Definition 5.7.** Let \( I \) be a \( \Sigma \)-interpretation. An environment over \( I \) is any mapping \( \rho \) from \( VS = DV \cup BV \) to \( D \cup BOOL \) such that \( \rho(X) \in D \) for all \( X \in DV \) and \( \rho(Y) \in BOOL \) for all \( Y \in BV \).

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

\[
[M]_I(\rho) = \begin{cases} 
\text{true}, & \text{if } M = \text{true}; \\
\text{false}, & \text{if } M = \text{false}; \\
\rho(X), & \text{if } X \in VS; \\
c_f, & \text{if } c/0 \in CS; \\
c([M_1, \ldots, M_n])_I(\rho) = c([M_1]_I(\rho), \ldots, [M_n]_I(\rho)), & \text{for } c/n \in CS, \ n \geq 0; \\
[k]_I(\rho) = k_f, & \text{if } k/0 \in FS \cup PS; \\
[k(M_1, \ldots, M_n)]_I(\rho) = k_f([M_1]_I(\rho), \ldots, [M_n]_I(\rho)), & \text{for } k/n \in FS \cup PS, \ n \geq 0; \\
[E_1 = E_2]_I(\rho) = \text{eq}([E_1]_I(\rho), [E_2]_I(\rho)); \\
\neg B]_I(\rho) = \neg ([B]_I(\rho)); \\
[B_1 \land B_2]_I(\rho) = \text{and}([B_1]_I(\rho), [B_2]_I(\rho)); \\
[B_1 \lor B_2]_I(\rho) = \text{or}([B_1]_I(\rho), [B_2]_I(\rho)); \\
[(B \rightarrow M)]_I(\rho) = \text{if_then}([B]_I(\rho), [M]_I(\rho)); \\
[(B \rightarrow M_1 \land M_2)]_I(\rho) = \text{if_then_else}([B]_I(\rho), [M_1]_I(\rho), [M_2]_I(\rho)),
\end{cases}
\]

where the semantic functions \( \text{eq}, \neg, \land, \lor, \text{if_then}, \text{and if_then_else} \) are continuous and computable and are defined as follows:

\[
\text{eq}(d_1, d_2) = \begin{cases} 
\text{true}, & \text{if } d_1 = d_2 \text{ is a finite and total element in } D; \\
\text{false}, & \text{if } \{d_1, d_2\} \text{ is inconsistent in } D; \\
\perp, & \text{otherwise}
\end{cases}
\]
(the fact that $D$ is consistently complete is needed to ensure the continuity of eq),

<table>
<thead>
<tr>
<th>$b$</th>
<th>not($b$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>⊥_b</td>
<td>⊥_b</td>
</tr>
</tbody>
</table>

and($b_1$, $b_2$)

<table>
<thead>
<tr>
<th>$b_1$</th>
<th>$b_2$</th>
<th>true</th>
<th>false</th>
<th>⊥_b</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>⊥_b</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>⊥_b</td>
</tr>
<tr>
<td>⊥_b</td>
<td>⊥_b</td>
<td>false</td>
<td>false</td>
<td>⊥_b</td>
</tr>
</tbody>
</table>

or($b_1$, $b_2$)

<table>
<thead>
<tr>
<th>$b_1$</th>
<th>$b_2$</th>
<th>true</th>
<th>false</th>
<th>⊥_b</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>⊥_b</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>⊥_b</td>
<td>⊥_b</td>
<td>⊥_b</td>
</tr>
</tbody>
</table>

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

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 $\text{BOOL}$, 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 $\Pi, I$ be a $\Sigma$-program and a $\Sigma$-interpretation. Remember $\tilde{\Pi}$ from Definition 4.5.

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

$I$ is a model of a rule $L := R$ (in symbols: $I \models L := R$) iff $\llbracket L \rrbracket_\rho \subseteq \llbracket R \rrbracket_\rho$ for all environments $\rho$ over $I$. Here, $\subseteq$ stands for the partial ordering over $\text{BOOL}$ 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 Emde 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 $\Pi$ 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 Emde [1], we are going to obtain minimal models as least fixpoints of continuous interpretation transformers. First, we show that $\Sigma$-interpretations are the members of a domain.
Theorem 5.4. For any BABEL signature $\Sigma$, the set $H\text{-INT}_\Sigma$ of all the Herbrand $\Sigma$-interpretations, equipped with the partial ordering

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

is an effectively presentable domain.

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

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

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

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

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

Next, we define interpretation transformers:

Definition 5.9. Let $\Pi$ be a BABEL program of signature $\Sigma$.

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

The interpretation transformer associated to $\Pi$ is the mapping

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

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

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

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

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

PROOF. To prove that $\mathcal{F}_\Pi$ 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 $\Pi$:

$$k(t_1, \ldots, t_n) := \{ B'_1 \to \} M'_1,$$
$$k(t_1, \ldots, t_n) := \{ B'_2 \to \} M'_2,$$

where $\{ R'_i \} \to M'_i$ 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 nonam
biguity 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 $[R_{1}'], J = [R_{2}'], J$ unless one or both are bottom.
Otherwise, $[B_{1}'], J$ and $[B_{2}'], J$ cannot be both true, and at least one of the two values $[R_{1}'], J, [R_{2}'], J$ 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 $\mathcal{F}_{\Pi}$ is continuous. Monotonicity is obvious from the definition.
Let $\{I_{a}\}_{a \in A}$ be a directed set of Herbrand interpretations. Let $I = \bigcup_{a \in A} I_{a}$.
Put $J = \mathcal{F}_{\Pi}(I)$, $J_{a} = \mathcal{F}_{\Pi}(I_{a})$. We must show that $J = \bigcup_{a \in A} J_{a}$. But $\bigcup_{a \in A} J_{a} \subseteq J$ follows
from monotonicity. To show $J \subseteq \bigcup_{a \in A} J_{a}$, consider any $k/n \in FS \cup PS$ and $t_{1}, \ldots, t_{n}$
eq H_{\Sigma}$. Let $\mathcal{R}'$ be the set of all $R'$ such that $k(t_{1}, \ldots, t_{n}) := R'$ is a ground instance of
some rule in $\Pi$. Then

$$k_{J}(t_{1}, \ldots, t_{n}) = \max \{ [R']_{J} | R' \in \mathcal{R}' \}$$

(by $I = \bigcup_{a \in A} I_{a}$ and continuity of evaluation)

$$= \bigcup_{a \in A} \max \{ [R']_{I_{a}} | R' \in \mathcal{R}' \}$$

(since this is an upper bound of $\bigcup_{a \in A} [R']_{I_{a}}$ for any fixed $R' \in \mathcal{R}'$)

$$= \bigcup_{a \in A} k_{J_{a}}(t_{1}, \ldots, t_{n}).$$

Let us now show that $\mathcal{F}_{\Pi}$ 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 $\Sigma$-interpretation $I$ is presented as a finite tuple $((k_{J})_{k \in FS \cup PS})$, where
each $k_{J}$ is presented in turn as a partial $\Sigma$-term $s$ (if $k$'s arity is 0) or as a finite set of
pairs $(\tilde{s}, s)$, where $s$ is a partial $\Sigma$-term, and $\tilde{s}$ is a tuple of finite $\Sigma$-constructs. This set
of pairs is called the graph of $k_{J}$ and determines $k_{J}$'s effect as a mapping, by requiring

$$k_{J}(t) = \bigcup \{ s | (\tilde{s}, s) \in \text{graph}(k_{J}), \tilde{s} \subseteq I \}$$

to hold for all tuples $I$ of $\Sigma$-constructs.

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

$$\{(I, J) \mid I, J \in H-INT_\Sigma \text{ are finite and such that } J \subseteq \mathcal{F}_{\Pi}(I) \}$$

is a r.e. set. Indeed, for finite $I$, $J \in H-INT_\Sigma$ we have:

$$J \subseteq \mathcal{F}_{\Pi}(I)$$

iff $k_{J} \subseteq k_{I}$ for all $k \in FS \cup PS$

iff for all $k \in FS \cup PS$ and for all $(\tilde{s}, s) \in \text{graph}(k_{J})$ $s \subseteq k_{J}(I)(\tilde{s})$ (here $\tilde{s}$ can
be omitted if $k$ is nullary)

iff for all $k \in FS \cup PS$ and for all $(\tilde{s}, s) \in \text{graph}(k_{J})$ there is some finite ground
instance $k(\tilde{s}) := R'$ of a rule in $\Pi$ such that $s \subseteq [R']_{I}$. 
In view of Definition 5.9, we have to justify the restriction to finite ground instances in the last step above. Since \( \hat{s} \) is finite and local variables in \text{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 \( s \subseteq [R']_I \) 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, \( \Pi \) is r.e., and each \([R']_I\) is finite and effectively computable from \( R' \) and \( I \).

Finally, assume that \( \mathcal{F}_\Pi(I) = I \). The condition \( J \subseteq I \) means that \( k_f(t_1, \ldots, t_n) \subseteq k_f(t'_1, \ldots, t'_n) \) must hold for all \( k/n \in FS \cup PS \) and all \( t_1, \ldots, t_n \in H_\Sigma \). By the definition of \( \mathcal{F}_\Pi \), this amounts to saying that \([L']_I \subseteq [R']_I \) must hold for any ground instance of any rule in \( \Pi \). This happens exactly when \( I \) is a model of all rules in \( \Pi \), 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 \( \mathcal{F}_I(I) \subseteq I \) iff \( I \models \Pi \).

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

**Theorem 5.6.** Any \text{BABEL} program \( \Pi \) of signature \( \Sigma \) has a least Herbrand model \( I_\Pi \). Moreover, \( I_\Pi \) is a computable element of the domain \( H_{INT_\Sigma} \).

**Proof.** Let \( I_\Pi \) be the least fixpoint of \( \mathcal{F}_\Pi \). By Theorems 5.2 and 5.5, \( I_\Pi \) is a computable element of \( H_{INT_\Sigma} \) and can be described as

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

where

\[
I_\Pi^0 = \bot \quad \text{(bottom of} \ H_{INT_\Sigma}) \quad \text{and} \quad I_\Pi^{j+1} = \mathcal{F}_\Pi(I_\Pi^j).
\]

Theorem 5.5 also guarantees that \( I_\Pi \) is a model of \( \Pi \). Let \( I \) be any other Herbrand model of \( \Pi \). By Theorem 5.5 again, we know that \( \mathcal{F}_\Pi(I) \subseteq I \). Since \( \mathcal{F}_\Pi \) is monotonic, it follows by induction on \( j \) that \( I_\Pi^j \subseteq I \) for all \( j \in \mathbb{N} \). We conclude that \( I_\Pi \subseteq I \) because \( I_\Pi \)'s characterization as a lub. Hence, \( I_\Pi \) is the least Herbrand model.

We can prove that \text{BABEL}'s reduction semantics always computes logically sound outcomes.

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

\[
M^N \rightarrow^*_{\sigma_{\text{out}}} N
\]

computes a sound outcome in the sense that

\[
[M_{\sigma_{\text{out}}}]_I(\rho) \supseteq [N]_I
\]

holds for all models \( I \models \Pi \) and all environments \( \rho \) over \( I \).
PROOF. Assume that the length of the reduction is $l$. Consider any model $I \models \Pi$ and any environment $\rho$ over $I$. It is easy to check that $[N](\rho) \supseteq [N]$. 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

$$M \xrightarrow{\Pi} \sigma_{out,1} M \xrightarrow{\Pi} \ast_{\sigma_{out,rest}} N,$$

where, for some $u \in O^+(M)$ and some variant $L := R$ of a rule in $\Pi$, $M \circ u$ and $L$ are unifiable with m.g.u. $\sigma_i = \sigma_{out,1} \cup \sigma_{in,1}$, and $M_i$ is $M(u \leftarrow R)\sigma_i$. Now we can reason as follows, knowing that $u \sigma_{in,1} = a\sigma_{out,1}$

$$[M_{\sigma_{out}}](\rho) = [M_{\sigma_{out,1}}{\sigma_{out,rest}}](\rho)$$

$$= [M_{\sigma_{out,1}}](\mu)$$

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

$$\supseteq [M(u \leftarrow R)\sigma_i](\mu)$$

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

$$= [M_i](\mu)$$

$$= [M_i\sigma_{out,rest}](\rho) \quad \text{(by construction of $\mu$)}$$

$$\supseteq [N](\rho) \quad \text{(by induction hypothesis)}.$$}

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 $\Pi$ be a $\Sigma$-program. Assume a $\Sigma$-expression $M$, a partial $\Sigma$-term $s$, and a finite $d$-substitution $\theta$ with $\text{dom}(\theta) \subseteq \text{var}(M)$ such that $[M\theta](\rho) \supseteq [s]$, holds for any $I \models \Pi$ and any environment $\rho$ over $I$. Then there exists a lazy narrowing sequence

$$M \xrightarrow{\Pi} \ast_{\sigma_{out}} N$$

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

**Sketch of Proof.** Remember that the least Herbrand model $I_\Pi$ of $\Pi$ is the lub of a chain of approximations $I^j_\Pi$ (Theorem 5.6). By the hypothesis and the continuity of evaluation, there is some $j \in \mathbb{N}$ such that the inequality $[M\theta](\rho) \supseteq [s]$ holds for $I - I^j_\Pi$. Using induction on $j$, it is possible to prove that $M\theta \xrightarrow{l-\ast} N'$ for some expression $N'$ such that $|N'| \supseteq s$. Notice that $N'$ is obtained by lazy rewriting, binding no variables in $M\theta$. The lazy narrowing derivation $M \xrightarrow{l-\ast} N$ 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 \( \Pi \) be a \( \Sigma \)-program. Assume ground \( \Sigma \)-expressions \( M, N_1, N_2 \) such that
\[
M \xrightarrow{\Pi} *N_1 \quad \text{and} \quad M \xrightarrow{\Pi} *N_2.
\]
Then \( N_1 \) and \( N_2 \) cannot have different constructors at any common occurrence; i.e., \(|N_1|, |N_2|\) is consistent in \( H_\Sigma \) (or in \( \text{BOOL} \), if \( M \) is boolean).

**Proof.** By applying Theorem 5.7 to the least Herbrand model of \( \Pi \), we can conclude that \([M]_{\Pi} \) is a common upper bound of \(|N_1|\) and \(|N_2|\).

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

27. Hullot, J. M., Canonical Forms and Unification, in: *Proceedings of the 5th Conference on


