Logic Programming

Proceedings of the Fourth International Conference

edited by Jean-Louis Lassez

PUBLISHER'S NOTE
This format is intended to reduce the cost of publishing certain works in book form and to shorten the gap between editorial preparation and final publication. Detailed editing and composition have been avoided by photographing the text of this book from authors' prepared copy.

Printed and bound in Australia.

Library of Congress Cataloging-in-Publication Data

Logic programming.

(MIT Press series in logic programming)
Papers for the Fourth International Conference on Logic Programming at the University of Melbourne, Australia, in May 1987.
Includes index.
1. Logic programming—Congresses. I. Lassez, Jean-Louis. II. International Conference on Logic Programming (4th; 1987: University of Melbourne)
QA76.6.L589 1987 005.13'1 87-2937
ISBN 0-262-12125-5

AN EFFICIENT LOGIC PROGRAMMING LANGUAGE AND ITS APPLICATION TO MUSIC[1]

Kemal Ebcioğlu
IBM, Thomas J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598
(914) 789-7303

Abstract

This paper is about BSL, an efficient logic programming language intended for implementing expert systems based on the generate-and-test method. From the execution viewpoint, BSL is an Algol-class nondeterministic language with a single assignment restriction; but there is a simple mapping that translates a BSL program to a first order formula, so that each terminating execution of a BSL program without free variables amounts to a proof of the corresponding first order sentence. We provided a de Bakker style formal semantics for BSL, and we obtained a soundness result which relates BSL and first order logic. BSL has been used for implementing a music expert system for harmonizing four-part chorales; this application is briefly described in the paper, and a musical output example is given.

Introduction

In this paper, we will report on BSL, a new logic programming language intended for implementing expert systems; and we then will describe the application of BSL to a music expert system for harmonizing four-part chorales in the style of J.S. Bach.

BSL was born out of our music application. At the outset of our music project, we decided to use first order logic to represent musical knowledge. First order logic was felt to be well-suited to the application, because it allowed us to make precise, concrete assertions about properties of a piece of music, and because it was more formal and tractable than some other A.I. paradigms, such as unrestricted production systems [Forgy and McDermott 77]. We initially found over a hundred assertions in first order predicate calculus which later formed the seed of the knowledge base. These assertions were not in clausal form, and made free use of existential and universal quantifiers, e.g., like the assertions one would use to extend English in a formal topic such as [Rogers 67]. However, the Prolog interpreter then available to us on the VAX 11 architecture did not have a natural way of coding quantifiers, moreover, it did not offer the most efficient way for utilizing the native resources of a traditional CPU. On the other hand, our music application was well-suited to the native data types and operations of a traditional architecture, and was also known to be extremely computation intensive (we did have a fair idea about the potential problems of the application because we had previously written a smaller scale 16th century strict counterpoint program using a similar heuristic search method [Ebcioğlu 81]). We were thus led to

[1] This research was supported by NSF grant no. DCR-8316665, and the major portion of it was done in the department of Computer Science, S.U.N.Y. at Buffalo, under the direction of my advisor Prof. John Myhill.

look for a different logic programming language for implementing our project. Our requirements were: 1- the language had to have a natural way of coding universal and existential quantifiers directly; 2- the language had to utilize the native resources of a traditional architecture efficiently, in a manner competitive with deterministic Algol-class languages, so that we could use it to produce very high quality music in a reasonable time; 3- the language had to have a natural way of specifying preferred solutions as well as just correct ones (the musical importance of this will be explained in the sequel); 4- the language had to have a streamlined design in order to increase its chances of being theoretically tractable, moreover, we felt that striving to use a streamlined design was a better way to approach a large project. While we were going back and forth between the logical assertions and ways of "executing" them, a logic programming language called BSL was designed, which appears to satisfy each of the above-mentioned requirements.

The formal basis for BSL

From the execution point of view, BSL is an Algol-class nondeterministic language where variables cannot be assigned more than once except in controlled contexts. It has a Lisp-like syntax and is compiled into efficient backtracking programs in C, via a compiler written in Lisp. However, BSL differs from existing nondeterministic languages [e.g., Floyd 67, Smith and Enea 73, Cohen 79] and relates to first order logic in a special way that we will summarize in this introductory section. Our plan is to proceed as follows: We will first describe a programming language called L^*, which is a tractable subset of BSL. We then will describe a first order language L, and a mapping that translates programs of L^* to formulas of L. We then will describe a fixed structure M involving integers, arrays, records and operations on such objects, which will represent the models we are interested in. The operational semantics of L^* will then be described via a ternary relation Ψ, such that $\Psi(F, \sigma_0, \sigma)$ means L^* program F, when started in initial state σ_0, terminates in state σ, where a state is a mapping from variable names to the universe of M. We will finally cite a soundness result: *if* an L^* program terminates in a state σ, *then* the corresponding first order formula of L is true in σ (where the truth of a formula in a state σ is evaluated in the interpretation M, after replacing any free variables x in the formula by $\sigma(x)$). It will thus be seen that for the case of L^* programs without free variables, each terminating execution of an L^* program amounts to a proof of the corresponding first order sentence.

We will begin by describing a programming language L^*, which is a tractable subset of BSL: The basic syntactic building blocks of L^* are *constants*, that consist of integers such as -2, 0, 3, and record tags, which are identifiers such as ssn, salary; and *variables*, which are identifiers such as x, p, n, or emp (for convenience, we assume that certain *reserved words*, such as "array" or "integer," cannot be used as identifiers, and that variables are distinct from record tags). A record tag intuitively serves to name a particular field of a record object, like the salary field of an employee record. An L^* *term* can be a variable or a constant, and if t_1, t_2 are terms, then $(f\ t_1\ t_2)$ is also a term, where f is one of the function symbols +,-,*,/,sub, or dot (sub and dot are intuitively intended for subscripting an array, and extracting a field of a record, respectively). Examples of L^* terms are 0, (* 2 (dot emp salary)), or (+ x 1) (also abbreviated as (1+ x)). A L^* *lvalue* is a term that can appear on the left hand side of an assignment, and is either a standalone variable x, or a term of the form $(f_1\ (f_2 \ldots (f_k\ x \ldots) \ldots) \ldots)$, where each f_i is sub or dot. Lvalues are exemplified by x, (dot emp salary), or (sub p n) (which can also be abbreviated as (salary emp) or (p n), where

clear from context). The programs of L^* are called formulas, because of their similarity to formulas of first order logic. Assuming l is an lvalue, and t_1, t_2 are terms, an L^* *atomic formula* is defined to be either an *assignment* of the form $(:= l\ t_1)$, or a *test* of the form $(relop\ t_1\ t_2)$, where *relop* is one of the predicate symbols == (equal), != (not equal), <, >=, <=, or >. An L^* atomic formula is an L^* *formula*. Assuming F_1 and F_2 are L^* formulas, then so are the following: (and F_1 F_2), (or F_1 F_2),[2] $(A\ x\ t_1\ (< x\ t_2)\ (1+ x)\ F_1)$, $(E\ x\ t_1\ (< x\ t_2)\ (1+ x)\ F_1)$, and (E ((x typ)) F_1), where x is a variable, t_1 , t_2 are terms where x does not occur, and *typ* is an L^* type. The L^* types are similar to the type declarations of an Algol-class language, and allow inductively defined integer, array and record declarations. Examples of L^* types are: integer, (array (3) integer), and (record (ssn integer) (salary integer)). A more detailed definition of types will be given later.[3]

We will now define a first order language [Shoenfield 67] called L, that will provide a means to translate L^* programs into first order predicate calculus. The variables of L are those of L^*, and the constants (or 0-ary function symbols) of L are the elements of the universe $|M|$ which we will soon describe in detail, and which includes the integers, record tags, and types of L^*, plus other individuals such as array and record objects. The function symbols of L consist of the L^* binary function symbols +,-,*,/,sub, dot, and the unary function symbol, type (which is intended to return the type of an object). The predicate symbols of L consist of the binary =,≠, < ,≥,≤, and >.

We inductively define a translation $\lambda u[u']$ from terms ∪ predicate symbols ∪ function symbols ∪ formulas of L^* to terms ∪ predicate symbols ∪ function symbols ∪ formulas of L as follows: The translation of a constant or variable or function symbol is itself. The translation of the relational predicate symbols ==, !=, <, >=, <=, >, are =,≠,<, ≥,≤,>, respectively. The assignment symbol := of L^* is also translated into = in L (i.e. both == and := are mapped to = in the logical counterpart). The translation of an L^* term or atomic formula $(f\ t_1\ t_2)$ is $f'(t'_1, t'_2)$; but the standard infix abbreviations may be used in L, moreover, dot(u, v) and sub(u, v) may be abbreviated as $u.v$ and $u[v]$, respectively. The translations of L^*'s lvalues are also called lvalues in L. The translation of (and F_1 F_2) and (or F_1 F_2) are $[F'_1 \& F'_2]$, and $[F'_1 \vee F'_2]$, respectively. The translation of $(A\ x\ t_1\ (< x\ t_2)\ (1+ x)\ F_1)$ and $(E\ x\ t_1\ (< x\ t_2)\ (1+ x)\ F_1)$ are $(\forall x \mid t'_1 \le x < t'_2)[F'_1]$, and $(\exists x \mid t'_1 \le x < t'_2)[F'_1]$, respectively.[4] The translation of (E ((x typ)) F_1) is $(\exists x \mid \text{type}(x) = \text{"typ"})[F'_1]$ (which is also abbreviated as $(\exists x{:}typ)[F'_1]$).

The only models we are interested in for the first order translations of L^* formulas are those that involve integers, arrays, records and operations on such objects. Fol-

---

[2] If P is one of {and, or}, then the construct $(P\ F_1 \ldots F_{k-2}\ F_{k-1}\ F_k)$, $k > 2$, can be used as an abbreviation for $(P\ F_1 \ldots (P\ F_{k-1}\ (P\ F_{k-1}\ F_k))\ldots)$.

[3] In [Ebcioğlu 87], we described a more general version of L^*, which allowed the condition $(< x\ t_2)$ and the increment $(1+ x)$ within the quantifiers to be more complex, as in a (possibly non-terminating) while loop. The reason we are only describing a restricted form of the quantifiers here, is because the more general form tends to lengthen the formal exposition. The full BSL language does allow the more general form, however.

[4] The abbreviations $(\exists x \mid R)\ F$ and $(\forall x \mid R)\ F$ stand for $(\exists x)[R \& F]$, and $(\forall x)[R \Rightarrow F]$, respectively; and $x \le y < z$ of course stands for $[x \le y \& y < z]$. We will also assume the following precedence (from highest to lowest) for binary logical connectives in this paper for avoiding brackets: &, ∨, ⇒, ⇔.

lowing the approach of [de Bakker 79], we obtain our results with a fixed interpretation involving computer data structures; However, an axiomatization can of course be produced for a corresponding "theory of integers, arrays, and records," so that our results concerning the fixed interpretation also hold in all models of such a theory. We describe here a fixed structure M that is intended to represent data types and operations available to an Algol class language such as BSL. The universe $|M|$ of the structure consists of the record tags of L^*, the special individual $\perp$ which is used for patching undefined values of functions, objects (which include the integers, arrays and records), and types (which are constants that are the types of the objects). We will inductively define the objects and their types together. An object of type "integer" can either be an integer, or U (called the *unassigned* constant). If $x_0, \ldots, x_{n-1}$ $(n > 0)$ are objects of the same type *typ*, then $(x_0 \ \ldots \ x_{n-1})$ is an object (called an array), and its type is (array (n) *typ*). If $x_1, \ldots, x_n$ $(n > 0)$ are objects which have types $typ_1, \ldots, typ_n$, and $y_1, \ldots, y_n$ are distinct record tags, then $(y_1\ x_1\ \ldots\ y_n\ x_n)$ is an object (called a record), and its type is (record $(y_1\ typ_1)\ \ldots\ (y_n\ typ_n)$). There are no further objects, and no further types. Examples of objects are -3, U, which have type "integer"; "(1 2 U)", which has type "(array (3) integer)"; and "(ssn 999123456 salary 25000)", which has type "(record (ssn integer) (salary integer))". Objects of type integer are called scalar objects, the others are called aggregate objects. The binary operations +,-,* (multiplication), / (integer division) on the integers are given their traditional meaning in the structure M, but they yield the constant $\perp$ when not all of their operands are integers or when their result would be undefined. Examples: $3*2=6$, $1+U=\perp$, $1/0=\perp$. $\mathrm{sub}(x, y)$ extracts the y'th element of x when y is a nonnegative integer, and x is an array object which has a y'th element (the elements of an array are numbered 0,1,2,... in BSL); otherwise $\mathrm{sub}(x,y)$ yields $\perp$. Example: sub("(1 2 U)",2)=U. The function $\mathrm{dot}(x, y)$, when x is a record $(y_1\ x_1\ \ldots\ y_n\ x_n)$, and there exists i, $1 \le i \le n$, such that y is the record tag y_i, yields the object x_i. Otherwise, $\mathrm{dot}(x,y)$ yields $\perp$. Example: dot("(ssn 999123456 salary 25000)",salary)=25000. The function $\mathrm{type}(x)$ yields the type of x if x is an object, and $\perp$ otherwise. Example: type("(U U)") = "(array (2) integer)". The binary predicates $=$ and $\neq$ have their conventional meaning. The binary predicates $<$, $\le$, $>$, $\ge$ are defined to be the pairs of integers for which these predicates would conventionally be true. Examples: $1<2$ is true, $1<U$ is false, $1 \ge U$ is false.

To describe the semantics of an L^* program, we first need a few definitions: We define a state to be a mapping from variables to elements of the universe $|M|$. We let $\sigma, \sigma_0, \sigma_1, \ldots$ and $\tau, \tau_0, \tau_1, \ldots$ range over states. We say that a formula F of L is true in a state σ, or that σ satisfies F, iff F is true in the interpretation M after replacing any free variables x in F by $\sigma(x)$. We define the function $V(\sigma,t)$ to yield the value of the L term t in the interpretation M, after replacing any variables x in t by $\sigma(x)$. Let y be the leftmost variable that occurs in an lvalue l of L. We define the substitution of term t for lvalue l in state σ, denoted by $\sigma[t/l]$, to be that state τ which is identical to σ, except that the subpart (or subobject) of $\sigma(y)$ selected by l in σ, has been replaced by $V(\sigma,t)$ in $\tau(y)$. Both l and t must evaluate to scalar objects in σ, or l must be a variable, otherwise the substitution is defined to be the identity transformation on states. Example: let $\sigma(a)$="(1 2 U)", $\tau = \sigma[5/a[2]]$; then $\tau(a)$="(1 2 5)", and $\tau(x) = \sigma(x)$ for all $x \neq a$.

We will describe the operational semantics of an L^* program via a ternary predicate Ψ, such that $\Psi(F, \sigma_0, \sigma)$ means L^* program F terminates in final state σ when started

in initial state σ_0. The inductive style of this formal semantics of L^* was inspired from regular dynamic logic [Harel 79]. So here is how an L^* program is executed (informal explanations are given in parentheses):

If l is an L^* lvalue and t is an L^* term,

$\Psi((:= l\ t), \sigma_0, \sigma) \leftrightarrow [V(\sigma_0, l')$ is U & $V(\sigma_0, l')$ is an integer & $\sigma = \sigma_0[l'/l']]$.

(Thus, an assignment is performed in the conventional manner, but if an attempt is made to assign to an lvalue whose current value is not U, or to use a non-integer right hand side, the program does not terminate.)
If *relop* is a relational predicate symbol of L^*, and t_1 and t_2 are L^* terms,

$\Psi((relop\ t_1\ t_2), \sigma_0, \sigma) \leftrightarrow$
$[V(\sigma_0, t'_1), V(\sigma_0, t'_2)$ are integers & $V(\sigma_0, t'_1)\ relop'\ V(\sigma_0, t'_2)$ is true in M & $\sigma = \sigma_0]$.

(Thus, a test is performed through ordinary comparison; but if any term of a test evaluates to a non-integer value, or if the test fails, the program does not terminate. Otherwise, the test acts as a no-op.)
If F_1, F_2 are L^* formulas,

$\Psi((\text{and } F_1\ F_2), \sigma_0, \sigma) \leftrightarrow (\exists\sigma_1)[\Psi(F_1, \sigma_0, \sigma_1)\ \&\ \Psi(F_2, \sigma_1, \sigma)]$.

("and" acts like a semicolon; (and F_1 F_2) is executed by first executing F_1, then F_2.)

$\Psi((\text{or } F_1\ F_2), \sigma_0, \sigma) \leftrightarrow [\Psi(F_1,\sigma_0,\sigma) \vee \Psi(F_2,\sigma_0,\sigma)]$.

((or F_1 F_2) is executed by executing one of F_1 or F_2.)
If F is (E ((*x typ*)) F_1) where x is a variable, *typ* is a type, and F_1 is an L^* formula,

$\Psi(F, \sigma_0, \sigma) \leftrightarrow$
$(\exists s, t, \tau_0, \tau_1)$
$[s = V(\sigma_0, x)\ \&$
t is an object of type *typ* all of whose scalar subparts are U &
$\tau_0 = \sigma_0[t/x]\ \&$
$\Psi(F_1, \tau_0, \tau_1)\ \&$
$\sigma = \tau_1[s/x]]$.

((E ((*x typ*)) F_1) is executed by saving x, setting x to an object of type *typ* all of whose scalar subparts are unassigned, executing F_1, and finally restoring x. This construct is similar to a begin-end block with a local variable.)
If F is (A x t_1 ($< x\ t_2$) ($1+ x$) F_1) where x is a variable, t_1, t_2 are L^* terms that do not contain occurrences of x, and F_1 is an L^* formula:

$\Psi(F, \sigma_0, \sigma) \leftrightarrow$
$(\exists k \geq 0)(\exists \tau_0, \ldots \tau_k)(\exists s)$
$[s = V(\sigma_0, x)$ &
$V(\sigma_0, t'_1)$, $V(\sigma_0, t'_2)$ are integers &
$\tau_0 = \sigma_0[t'_1/x]$ &
$(\forall i \mid 0 \leq i < k)(\exists \tau)[x < t'_2$ is true in τ_i & $\Psi(F_1, \tau_i, \tau)$ & $\tau_{i+1} = \tau[x + 1/x]]$ &
$x < t'_2$ is false in τ_k &
$\sigma = \tau_k[s/x]]$.

((A x t_1 (< x t_2) (1+ x) F_1) is executed by saving x, setting x to t_1, while x is less than t_2 repetitively executing F_1 and incrementing x, and restoring the old value of x when x is finally not less than t_2. This construct is similar to a "for" loop with a local index variable.)
If F is (E x t_1 (< x t_2) (1+ x) F_1), where x, t_1, t_2, and F_1 are defined as in the case for (A x ...),

$\Psi(F, \sigma_0, \sigma) \leftrightarrow$
$(\exists k \geq 0)(\exists \tau_0, \ldots, \tau_k)(\exists s)$
$[s = V(\sigma_0, x)$ &
$V(\sigma_0, t'_1)$, $V(\sigma_0, t'_2)$ are integers &
$\tau_0 = \sigma_0[t'_1/x]$ &
$(\forall i \mid 0 \leq i < k)[x < t'_2$ is true in τ_i & $\tau_{i+1} = \tau_i[x + 1/x]]$ &
$x < t'_2$ is true in τ_k &
$(\exists \tau)[\Psi(F_1, \tau_k, \tau)$ & $\sigma = \tau[s/x]]]$.

((E x t_1 (< x t_2) (1+ x) F_1) is executed by saving x, setting x to t_1, repetitively checking that x is less than t_2 and incrementing x an arbitrary number of times (possibly zero times), checking that x is less than t_2 for the last time, executing F_1, and finally restoring the old value of x. If x is not less than t_2 at any point along the way, execution does not terminate.)

We say that σ is an extension of σ_0 iff σ is identical to σ_0, except perhaps for some variables x such that $\sigma(x)$ and $\sigma_0(x)$ are objects of the same type, and $\sigma(x)$ and $\sigma_0(x)$ are identical except that there exist one or more scalar subparts of $\sigma(x)$ which are integers, while the corresponding scalar subparts of $\sigma_0(x)$ are "U". As an example, consider two states σ_0 and σ_1, and a variable a, such that σ_0(a) = "(−1 U U)" and σ_1(a) = "(−1 7 10)" and $\sigma_0(y) = \sigma_1(y)$ for all $y \neq$ a. Then σ_1 is an extension of σ_0. We say that σ extensibly satisfies F, or F is extensibly true in σ, iff σ satisfies F, and for any extension τ of σ, τ also satisfies F.

The following proposition precisely defines the relationship between the semantics of a formula of L^* as a computer program and the semantics of the corresponding formula of L under the interpretation M.

Proposition: (soundness of L^* formula-programs) Let Ψ and $\lambda u[u']$ be defined as above. Let σ_0 be any state, and F be a formula of L^*. Then for all states σ, if $\Psi(F, \sigma_0, \sigma)$, then σ is an extension of σ_0, and σ extensibly satisfies F'.

Proof: By induction on the complexity of F. See [Ebcioğlu 87]. □

The following table of examples should clarify the intuition behind the relationship between an L^* program and its first order translation, which is true at its termination states. Notice that when the L^* program does not have free variables, as in the last example, the truth of the corresponding first order sentence is independent of the value of any variable in the termination state; thus, each successful execution of a closed L^* program is equivalent to constructively proving that the corresponding first order sentence is true in the interpretation M, or in all models of a suitably axiomatized "theory of integers, arrays and records" corresponding to M.

L^* program	initial assumptions	first order translation	can program terminate?
(== x 0)	x=0	x=0	yes
(== x 0)	x=1	x=0	no
(== x 0)	x=U	x=0	no
(:= x 0)	x=U	x=0	yes
(:= x 0)	x=1	x=0	no
(:= x (1+ x))	none	x=x+1	no
(:= x x)	none	x=x	no
(or (:= x 0) (:= x 1))	x=U	x=0 ∨ x=1	always yes
(or (== x 0) (== x 1))	x=0	x=0 ∨ x=1	yes (via (== x 0))
(and (:= x 0) (:= x 1))	none	x=0 & x=1	no

assumptions: emp is initially an array of 100 employee records. Some employee's salary is > 35000. ans is initially U.
program: select an employee record, test salary field, then assign ssn field to ans.
(E i 0 (< i 100) (1+ i)
 (and (> (salary (emp i)) 35000) (:= ans (ssn (emp i)))))
Program terminates if i was incremented the right number of times for selecting a suitable employee.

first order translation:

(∃i | 0≤i<100)
 [emp[i].salary>35000 & ans=emp[i].ssn].

assumptions: none
program: create an array whose elements are 0 or 1
(E ((a (array (10) integer)))
 (A i 0 (< i 10) (1+ i)
  (E ((d integer))
   (and (or (:= d 0) (:= d 1)) (:= (a i) d)))))
All executions terminate.

first order translation:

(∃a:(array (10) integer))
 (∀i | 0≤i<10)
  [(∃d:integer)[[d=0 ∨ d=1] & a[i]=d]].

Although the direct translation from a multiple assignment Algol-class language to first order logic has previously been studied by [Hehner 84] from a program correctness viewpoint, our approach of equating the execution of a program of a single assignment nondeterministic Algol-class language to a proof, appears to be original.[5]

Some further language features

The L^* language discussed above is only a subset of BSL. The full BSL language has a few more, but not many more features; we tried to keep BSL small. These are mainly user defined predicates and functions, global variables, enumeration type de-

[5] But [Voda 86] has independently found a logic programming language called nondeterministic Pascal, which has language constructs similar to BSL. Like us, Voda has chosen a fixed interpretation, his being a simple one involving Lisp S-expressions and operations on these, which he has axiomatized via a "theory of pairs." But Voda's language proves assertions directly through Gentzen-style inference rather than through the state transformation paradigm as in BSL; in particular, the choice between assignment and equality test is delayed to run time in Voda's language, which should probably have a negative impact on that language's efficiency.

finitions, and macro and constant definitions. We will now give an *informal* overview of these features by going through a small BSL program for proving that Semele has an ancestor (similar to a problem in [Kowalski 79]):

```
(dc P__SIZE 8)
(dt mythological (HARMONIA APHRODITE ARES HERA SEMELE DIONYSUS CADMUS ZEUS))

(dx p (array (P__SIZE) (record (p__child mythological) (p__parent mythological))) ;parent relation
(HARMONIA  APHRODITE  ARES      HERA    SEMELE    HARMONIA
DIONYSUS   SEMELE     HARMONIA  ARES    ARES      ZEUS
SEMELE     CADMUS     DIONYSUS  ZEUS))

(dp parent ((OUT x mythological) (OUT y mythological)) ;y is a parent of x if ...
 (E i 0 (< i P__SIZE) (1+ i) (and (== x (p__child (p i))) (:= y (p__parent (p i))))))

(dp ancestor ((OUT x mythological) (OUT y mythological)) ;y is an ancestor of x if ...
 (E ((z mythological)) (and (parent x z) (or (:= y z) (ancestor z y)))))

(E ((u mythological)) (ancestor SEMELE u)) ;main formula
```

The dc (define constant) statement at the beginning of the program above allows the programmer to use symbolic abbreviations instead of integers. The dt (define enumeration type) statement is also a constant definition facility that associates the symbolic enumeration constants with consecutive integers (or user specified values if given); and the type name is subsequently taken to be an abbreviation for "integer." (It is possible to read and write enumeration constants in symbolic form, however.) The dx (define global variable) statement specifies the defined variable's name, type, and optional initial values. The dp (define predicate) statements are essentially procedure declarations that each consist of a formal parameter list and a procedure body. A BSL program always ends with a "main formula."

A BSL program is executed nondeterministically, by first setting all scalar subparts of the global variables to U or to the specified initial values if given, and then executing the main formula. The main formula is executed like an L^* formula, as described in the previous section; except that predicate calls occurring in the context of an atomic formula are executed by dynamically replacing the call by the body of the predicate definition, after substituting the arguments of the call for the corresponding formal parameters in the body. In case a variable y occurring in an argument of the call would become enclosed in a quantified construct (A y ...), (E y ...) or (E ((y ...)) ...) in the predicate body, the offending y's are renamed by a fresh name throughout the quantified construct, before the substitution takes place.[6]

Now suppose we are given any BSL formula F, along with definitions for the predicates directly or indirectly called from F. Assume that L and $\lambda u[u']$ have been extended to include these additional predicate symbols, so that BSL formulas with calls to the given predicates can be translated to first order assertions, and so that a BSL predicate definition for a predicate p with formal parameters $x_1, \ldots, x_n$ is translated to

---

[6] The subscript expressions in the arguments must evaluate to an integer at the time of a predicate or function call, otherwise execution does not terminate. This restriction, plus single assignment, allows this call by name technique [Naur 63] to be implemented through call by reference (if a subscript is already defined at call time, then it cannot change during predicate execution because of the single assignment rule). There is also a call by value technique which is obtained by omitting the "OUT" indication from the desired predicate parameters.

an axiom of the form: [for all $x_1, \ldots, x_n$ of the specified types [the assertion corresponding to the predicate body $\Rightarrow p(x_1, \ldots, x_n)$]]. Suppose we adopt, for our truth evaluations, an arbitrary extension of the interpretation M, obtained by adding arbitrary relations of appropriate arity to M for each predicate called directly or indirectly from F. Then, we claim that F has the following property: for any state σ, *if* F terminates in σ, *then* [the conjunction of the axioms for the predicate definitions $\Rightarrow F'$] is extensibly true in σ.[7] It follows that a BSL program including global variables, predicate definitions and a main formula, also has a corresponding assertion which is true when it terminates. This assertion is [There exist global variables of the specified types, equal to their initial values if given, such that [the conjunction of the axioms of the predicate definitions $\Rightarrow$ the assertion corresponding to the main formula]]. This happens to be a sentence, since free (undeclared) variables are not allowed at the complete BSL program level. Thus, each terminating execution of the example program above amounts to showing that the following first order sentence is true in any extension of M that contains relations for "ancestor" and "parent":

```
(∃p)[p="((p__child HARMONIA p__parent APHRODITE) (p__child ARES p__parent HERA) ...)" &
      [[(∀x,y:mythological)
            [(∃i | 0≤i<P__SIZE)[x=p[i].p__child & y=p[i].p__parent]
            ⇒ parent(x,y)] &
       (∀x,y:mythological)
            [(∃z:mythological)[parent(x,z) & [y=z ∨ ancestor (z,y)]]
            ⇒ ancestor(x,y)]]
      ⇒ (∃u:mythological)[ancestor(SEMELE,u)]]].
```

(Note that we will be using the abbreviations made available by the constant, enumeration type, and macro definitions of a BSL program in its logical counterpart as well.)

Other language features of BSL include user-defined functions. For example, the unary function "add1" could be defined as (df add1 ((OUT x integer) (OUT y integer)) (:= y (1+ x))), which would be, just like a predicate, associated with an axiom: (∀x,y:integer)[y=x+1 ⇒ add1(x)=y]. The extra last parameter of a function definition is called the return variable. A function call is evaluated (assuming call by name) by substituting the arguments of the call for the parameters in the function body after suitable renaming, executing the body, and finally collecting the value of the return variable as the result of the call. Functions bodies must normally not assign to the global variables or the arguments, and must not produce choice points when compiled, which means that they must return at most a single result for any given set of arguments (there may still be "or"s in a function body, however). BSL also has macro definitions that allow access to the full procedural capabilities of Lisp. A limited form of the "not" connective is defined as a macro, which is expanded by moving the "not"s in front of the tests via de Morgan like transformations, and then eliminating the "not"s by changing (not (== x 0)) to (!= x 0), etc.. There are also "if-then-else" and "case" constructs whose nondeterministic semantics are defined through and, or, not; but which are compiled in the conventional efficient way. A predicate with a side-effect free body (P ...) can be negated by defining an equivalent 0-1 valued function f, which returns 1 if (P ...) succeeds, or returns 0 if (not (P ...))

[7] But for brevity we will sometimes say "F' is true (false) in σ" for "[relevant axioms $\Rightarrow F'$] is true (false) in σ," where this is clear from context.

succeeds; and by adopting the convention that (f ...) occurring in an atomic formula context is an abbreviation for (!= (f ...) 0).

The implementation of BSL on a real computer

A BSL program whose main formula is of the form (E ((x typ)) F) is implemented on a real, deterministic computer via a modified backtracking method, which in principle attempts to simulate all possible executions of the main formula, and prints out the value of x just before the end of every execution that turns out to be successful. Whenever a choice has to be made between executing F_1 and executing F_2 in the context (or F_1 F_2), the current state is pushed down to enable restarting by executing F_2, and F_1 is executed. Whenever a choice has to be made between executing F_1 and setting n to (1+ n) in the context (E n t_1 (< n t_2) (1+ n) F_1), the current state is pushed down to enable restarting by setting n to (1+ n), and F_1 is executed. Whenever a test ($relop$ t_1 t_2) is found to be false, or if (< n t_2) is found to be false in the context (E n t_1 (< n t_2) (1+ n) F_1), and each time after the top level (E ((x typ)) ...) is successfully executed and x is printed, the state that existed at the most recent choice point is popped from the stack, and simulation restarts at that choice point. Explicit assignment to a scalar variable or scalar subpart of a variable when its old value is not U, or the use of such a variable while its current value is still U, are considered errors and cause the simulation to be aborted; but run-time checks for detecting such error conditions may be omitted for efficiency reasons. Simulation begins with an empty choice-point stack and ends when an attempt is made to pop something from an empty stack.

A modification is made to this basic backtracking technique for the case of side effect free formulas F_1 in the context (or F_1 F_2), or (E n ... F_1). (A side effect is taken to be an assignment or a predicate call.) After a formula F_1 in such a context is successfully executed, the most recent choice point on the stack is discarded (which would be the choice point for restarting at F_2, or F_1 with a different value of n, assuming the modification is uniformly applied). This convention, similar to the "cut" operation of Prolog, serves to prevent duplicate solutions for x from being printed out (or redundant failures from occurring) when in a particular state F'_1 and F'_2 do not express mutually exclusive conditions, or when F'_1 is true for more than one n in its quantifier range.[8] To see why this modification is reasonable, consider the less obvious example (or (== x 0) (:= y 1)). If the program successfully went through (== x 0) and then failed, then there is no point in backtracking to (:= y 1) and trying again by going through (:= y 1), since after going through (:= y 1), the program would either attempt to re-assign to y and cause an error, or would fail in exactly the same way as the first failure, because the first failure could not have depended on y (which should have been U at the time (== x 0) was successfully executed).

Since backtracking is a notoriously inefficient search algorithm, we had to be careful about its implementation in BSL, in the hope of making the language usable on substantial problems. The present implementation of BSL omits the run-time checks about single assignment, and uses an aggressive method of saving and restoring vari-

[8] But in Prolog, the "cut" in the first of a pair of clauses analogous to a BSL (or F_1 F_2) must be manually supplied, and is not built-in. [Debray and Warren 86] propose a method for supplying the "cut" through a compiler when this would not change the original semantics of the program, e.g. when the clauses are already mutually exclusive.

ables that relies on the assumption that the program is correct, in the sense that in any execution of the program, no scalar variable or scalar part of an aggregate variable will be explicitly re-assigned when it already has an integer value, or used while it still has the unassigned value. In a choice point, the following facts apply to a typical variable:[9] If the variable is already assigned, then we need not save it, because it will not be re-assigned during the continuation of execution (because the program follows the single assignment rule), and because its storage space will not be deallocated while it is still needed (by a special storage allocation scheme - see [Ebcioğlu 87], this scheme reduces to static allocation (zero creation and destruction overhead) for the L^* subset of BSL). On the other hand, if the variable is not currently assigned, then no program path starting at the current point can use the old value of that variable (because the program follows the no-use-before-set rule), so we still do not have to save it and restore it, even though the variable may contain a garbage value assigned during a failed path when a backtracking return is made to this choice point. This technique is as unsafe as omitting subscript range checks in Fortran, but appears to provide the highest performance. The second optimization relies on using whenever possible the ordinary compare-and-branch compilation techique for Boolean expressions of Algol-class languages [Aho, Sethi, and Ullman 86], extended in an obvious way to bounded quantifiers, via iteration. For side effect free subformulas F_1 in the context (or F_1 F_2) and (E n ... F_1), the BSL compiler produces efficient compare and branch and looping statements,[10] instead of implementing the equivalent but inefficient semantics of first pushing down a choice point and then discarding it when F_1 is successful. This technique is also extended to the case where F_1 contains side effects, by insisting on emitting compare and branches using the ordinary Boolean compilation technique as long as possible, until a side effect is actually encountered within F_1, in which case the required pushdown operation(s) are emitted, the backtracking return point label is established, and the restore operations following that label (if any), are emitted. The motivation here is to increase the chance that, e.g. within (or F_1 F_2), F_1 will fail and branch directly to F_2 before a choice point needs to be pushed. While this compilation technique is straightforward for the case where F_1 is entirely side effect free, generation of efficient code for "(or ...)" and "(E n ...)" containing side effects at arbitrary points is more involved, and a compilation algorithm for the general case is given in [Ebcioğlu 87].[11] A further optimization used in

---

[9] The only variables to which these facts do not apply at a choice point are precisely the variables which are declared within the scope of a universal quantifier (A ...), and which are at the same time lexically known at the choice point (e.g. have been declared in quantifiers enclosing the choice point). Such variables are pushed down and restored by the present implementation, but they are usually limited to one or two quantifier indices in practice.

[10] As a concrete example, for (or (E i 0 (< i n) (1+ i) (and (>= (a i) 0) (<= (a i) (1+ i)))) F_1), where the (E ...) is clearly side-effect free, the compiler generates efficient compare and branch and looping instructions to execute the following: "if not ($\exists$i | 0≤i<n)[a[i]≥0 & a[i]≤i+1] then goto the code for F_2, else goto the continuation of (or (E ...) F_2)."

[11] Note that the use of the code generation technique for Boolean expressions with bounded quantifiers is a compiler optimization applicable in limited cases, and has nothing to do with BSL's language level semantics, where programs are executed for side effects and not for returning truth values, as is the case in Boolean expressions. SETL [Schwartz 73] is an example of a deterministic, multiple assignment language that extends Boolean expressions with bounded quantifiers at the language semantics level. Also of relevance is the subset of first order logic called "logic of cuttable formulas," described in [Gergely and Szots 84], where formulas are compiled into programs that consist only of Boolean expressions extended with bounded quantifiers, and where the execution of a Boolean expression returning "true" is shown to be equivalent to a proof of the corresponding first order formula. The programs of the cuttable formulas language are essentially a subset of the compiled backtracking programs of BSL.

BSL is a very low overhead, compiled intelligent backtracking technique, which can be triggered by a compiler option. However, we will not be able discuss this technique in the scope of the present paper.

Another language feature that relates to the implementation of BSL on a deterministic computer is the heuristics feature. The backtracking algorithm of BSL is suitable for applications where all "correct" solutions of a problem must be found, as in the molecular genetics problem described in [Stefik 78]; however, for the music application, the list of all "correct" solutions is either impractically long or very boring. It is a known fact that absolute rules such as treatise rules expressed in first order predicate calculus are not sufficient for producing beautiful music. Composers use much additional knowledge for choosing among the "correct" extensions of a partial composition at each stage of the compositional process. While our limited powers of introspection prevent us from replicating the thought process of such choices in an algorithm, we conjecture that a good algorithmic approximation can be obtained by using a large number of prioritized heuristics, or recommendations, based on style-specific musical knowledge (cf. the work of [Lenat 76] for heuristics on the equally intangible topic of "interesting" mathematical conjectures). The order of enumeration of the termination states of a BSL formula F can be controlled by enclosing it in the construct (H F (x_1 x_2 ...) F_k ... F_0), which is itself a BSL formula. x_1 , x_2,... are a list of lvalues that must include those that are assigned within F, and F_k ,..., F_0 are *heuristics*, which must be side effect free BSL formulas. After each terminating execution of F, a *worth* is assigned to the current termination state by summing the weights of the heuristics whose corresponding assertions are true in that state, and the state (as represented by the current values of the lvalues x_1 , x_2,...) is saved in a list. Heuristics are weighted by decreasing powers of 2: the weight of F_i , $0 \le i \le k$ is defined to be 2^i. If and when the ways to execute F are exhausted, (H F ...) terminates with the best termination state of F, i.e. the assignment to x_1, x_2,... with the highest worth (with ties being resolved randomly), and then, if backtracking occurs to this (H F ...), with the next best termination state of F, etc.. More formally, (H F ...) attains the termination states of F in an order σ_0, ... σ_n, which satisfies:

$(\forall i,j \mid 0 \le i,j \le n)$
  $[(\exists m \mid 0 \le m \le k)$
    $[(\forall l \mid k \ge l > m)[F'_l$ has the same truth value in both σ_i and $\sigma_j]$ & F'_m is true in σ_i & F'_m is false in $\sigma_j]$
    $\Rightarrow i < j]$.

The assertion F' corresponding to F is of course true in all the termination states σ_i , $i = 0, \ldots, n$. As for the nondeterministic semantics of (H F ...), we adopt the convention that it is executed merely by executing F, and its logical translation is taken to be F' ; i.e., heuristics are ignored in the nondeterministic semantics of the program. Notice that when an (H ...) construct occurs as an appropriate subformula of a program, as shown in the expert system example in the next section; it will have the effect of guiding the backtracking search toward desirable paths. Heuristics are used for determining the best continuation of a partial chorale in our music expert system.

We ran a number of programs to see how BSL's performance compares with Prolog and Lisp, using the language implementations available to us on the IBM 3090 under CMS, namely the VM/Prolog interpreter, the VM/Lisp compiler, and a C compiler derived from the PL.8 optimizing compiler [Warren et al. 86] (the BSL compiler itself is written in VM/Lisp and generates C code). All available optimizations such as iteration (do) constructs, unchecked fixed arithmetic, eq instead of equal, unchecked car/cdr operations, and noninterruptible code for VM/Lisp, and static clauses for

VM/Prolog, were used.[12] The table below lists the results of the comparisons, along with the logical translations of the BSL programs used in the benchmarks. The Lisp and Prolog versions of the "queens" benchmark are also given, in order to provide concrete examples of what we are comparing. These programs are all naive search algorithms derived directly from a logical specification (without any refinement). Faster algorithms are certainly known for these problems, for example, in the queens problem, keeping a record of the taken diagonals will achieve an obvious speedup. But the benchmarks should still give an idea about the raw search capability of the different language implementations, which is a very important capability for the design of complex and computation-intensive expert systems, where one usually has to opt for the simplest specifications anyway, and where hand-optimization of individual parts of the system is usually impractical. The same naive algorithms are used in all three languages, but the solution, when it is of an array type in BSL, is represented as a list of integers in the Lisp and Prolog programs, which only needs to be accessed sequentially, in order not to aggravate the differences due to array vs. list representations. The times given are the IBM 3090-200 virtual cpu time in seconds to exhaust the search space, without printing results.

program	BSL time	VM/Lisp time	Lisp/BSL ratio	VM/Prolog time	Prolog/BSL ratio
debruijn	2.38	10.84	4.55	78.5	33.0
triangle	7.86	14.60	1.86	192.3	24.5
permute	8.26	19.64	2.38	172.1	20.8
queens	2.95	9.54	3.23	87.1	29.5
dslalpha	2.75	12.37	4.50	19.5	7.09

debruijn: enumerate all de Bruijn sequences [Ralston 82], circular strings of length M**N composed of digits 0,...,M-1, where every N digit long substring is distinct. An array d of SIZE=M**N+N-1 elements that begins with N M-1's (and hence ends with N-1 M-1's) is used to represent the circular string. Here M=2 and N=5. Note: in the following logical translations, the assignments have been left intact, so that the original BSL programs can be recovered directly.

(∃d:(array (SIZE) integer))
 (∀n | 0≤n<SIZE)
  (∃j | 0≤j<M)[d[n]:=j & [n<N ⇒ d[n]=M-1] & (∀k | n-1≥k≥N-1)(∃i | 0≤i<N)[d[n-i]≠d[k-i]].

triangle: enumerate all triples of integers x,y,z, 0<x<y<z<400, such that x**2+y**2=z**2 (Pythagorean numbers).

(∃x,y,z:integer)(∃i | 1≤i<398)(∃j | i+1≤j<399)(∃k | j+1≤k<400)[i*i+j*j=k*k & x:=i & y:=j & z:=k].

Note: PL_8 does not move up (i*i+j*j) from the innermost quantifier, because the "inner loop" is re-entered in the middle after a backtracking return.

permute: enumerate all permutations of the digits 0,1,...,8

(∃p:(array (9) integer))(∀n | 0≤n<9)(∃j | 0≤j<9)[(∀k | n-1≥k≥0)[j≠p[k]] & p[n]:=j].

queens: find all solutions to the 11-queens problem. The rows and columns are numbered as 0,1,...,10, and the array elements p[0],...,p[10] represent the column no. of the queen on row 0,...,10, respectively. The Lisp and Prolog versions are also given.

(∃p:(array (11) integer))
 (∀n | 0≤n<11)(∃j | 0≤j<11)[(∀k | n-1≥k≥0)[j≠p[k] & j-p[k]≠n-k & p[k]-j≠n-k] & p[n]:=j].

[12] Without the equal->eq, fixed arithmetic, unchecked operation and noninterruptible code optimizations, VM/Lisp slows down by a factor of 9.8-16.4 (5.7 in dslalpha); and without the static clause optimization VM/Prolog is slowed down by a factor of 1.37-1.86 (1.07 on triangle); on these particular programs.

```
(compile '(queens1 (lambda (n s)
    (cond ((not (qslessp n 11)) (use s))
          (t   (do ((j 0 (qsinc1 j))) ((not (qslessp j 11)))
                 (cond ((do((k (qsdec1 n) (qsdec1 k)) (x s (qcdr x)))
                           ((or(null x)
                               (eq (qcar x) j)
                               (eq (qsdifference j (qcar x)) (qsdifference n k))
                               (eq (qsdifference (qcar x) j) (qsdifference n k)))
                            (null x)))
                        (queens1 (qsinc1 n) (cons j s)))))))))
(compile '(queens (lambda nil (queens1 0 nil))))
(compile '(use (lambda (x) nil)))
```

```
range(*i,*j,*x) <- lt(*i,*j) & range1(*i,*j,*x).
range1(*i,*j,*i).
range1(*i,*j,*x) <- sum(*i,1,*ip1) & lt(*ip1,*j) & range1(*ip1,*j,*x).
queen1(11,*x,*x) <- /().
queen1(*n,*x,*z) <- range(0,11,*j) & check(*x,*j,1) & sum(*n,1,*np1) & queen1(*np1,*j,*x,*z).
check(nil,*,*).
check(*pk.*rest,*j,*nminusk) <- ne(*j,*pk) & ¬diff(*j,*pk,*nminusk) & ¬diff(*pk,*j,*nminusk) &
sum(*nminusk,1,*newnmk) & check(*rest,*j,*newnmk).
queens() <- queen1(0,nil,*x) & fail().
```

dslalpha: enumerate the names of the suppliers who supply all parts. Executed 100,000 times. Taken from a DSL ALPHA query for the suppliers-parts database in [Date 77]. VM/Prolog is doing well here apparently because of clause indexing.

```
(∃s,p,sp)
 [s="((s__sno S1 s__sname SMITH s__status 20 s__city LONDON) ...)" &
 p="((p__pno P1 p__pname NUT p__color RED p__weight 12 p__city LONDON) ...)" &
 sp="((sp__sno S1 sp__pno P1 sp__qty 300) ...)" &
 (∃ans:snametype)
  (∃n | 0≤n<S__SIZE)
    [(∀i | 0≤i<P__SIZE)(∃j | 0≤j<SP__SIZE)[sp[j].sp__sno=s[n].s__sno & sp[j].sp__pno=p[i].p__pno]
     & ans:=s[n].s__sname]].
```

There are many factors that contribute to the efficiency of BSL on a traditional architecture vs. Lisp and Prolog. In BSL, the choice between assignment and equality test is made at compile time, unlike Prolog's unification algorithm, which often requires that the choice be deferred to run time (but the practical gain from this inefficiency in Prolog is relational programming, which is admittedly very flexible and useful for certain applications). BSL directly uses the native data types of a traditional architecture, such as integers, unlike Lisp and Prolog, which often have to manipulate machine words containing both data and tags. A considerable amount of push down and restore operations are eliminated in BSL because of the optimizations for side-effect free subformulas; and when there has to be backtracking, BSL saves and restores very little state because of the backtracking optimization described above (By contrast, some Prolog implementations may push down a choice point right in the beginning of the execution of a clause via a "try" instruction, or may have to use backtrackable assignment [Fagin and Dobry 85, Turk 86].) Bounded quantifiers are implemented in BSL via inline code (often simple loops), rather than via clauses as in Prolog, and this increases the chances of applying traditional compiler transformations to BSL code. Some of these optimizations can be of course be implemented in Prolog and Lisp;[13] so it is difficult to reach a conclusion about the inherent speeds of the

[13] E.g., mode declarations or automatic mode inference in Prolog can allow the choice between assignment and equality test to be made at compile time, and, e.g., [Kranz et al. 86] describe an optimizing compiler for Lisp that could approach C/Fortran performance in consless code. Considerably more performance can also be achieved by compiling Prolog: [Kurokawa et al. 86] report a Prolog compiler

languages. But nevertheless, BSL appears to achieve good utilization of the resources of a traditional architecture.

One parallel architecture most suited to BSL seems to be the emerging Very Long Instruction Word architecture [Ellis 86, Nicolau 85]. The newer VLIW machine compilation techniques for general sequential programs are not in the least daunted by the sequential backtracking semantics of BSL; in fact, the extraction of parallelism is simplified because of BSL's single assignment nature. The VLIW architecture can achieve only a modest speedup for BSL's backtracking execution, but a tightly coupled VLIW machine can incur much less communication delays than some proposed parallel architectures for Prolog (e.g. communication via packets [Onai et al. 85]), which is an advantage for applications where the inherent parallelism is not very high.

Designing expert systems in BSL

The problem with BSL is that it does not support multiple assignment as in impure Lisp or C, consequently, conventional algorithms of Algol-class languages are difficult to translate directly into BSL.[14] BSL also does not support list processing: it is limited to problems that allow an abstraction using Pascal-PLI-C style data types. Thus, BSL is not a replacement for Lisp or Prolog; one would choose Lisp or Prolog rather than BSL, for easily implementing an application such as a compiler, where list processing seems inevitable. Nevertheless, BSL does have an important application area, where neither its lack of multiple assignment nor its lack of list processing is a drawback; we found out this fact through our own application. BSL can be used for writing the rules of a generate-and-test expert system in a declarative style, thinking first in logic. In this declarative style, one virtually never feels a need to set, e.g., x:=x+1, since that would be tantamount to asserting x=x+1. Moreover, while implementing such an expert system, BSL can achieve very good utilization of the hardware resources of a traditional supercomputer, thus allowing the concepts of logic programming to be applied to large, computation intensive problems. We feel that BSL can be used for any computation intensive generate-and-test application where a natural abstraction with Pascal style data types is feasible: the molecular genetics problem described in [Stefik 78] is one such example. There is of course an occasional need for conventional programming in any project, which can always be done by calling C procedures from BSL. We did so for interactive graphics in our music expert system. The formal analog of an expert system based on the generate-and-test method can be implemented in BSL via a very long formula of the following form:

for the IBM 3090 which has a performance of 1.42 megalips on "append" (This compiler is probably 4.6-6.2 times faster than VM/Prolog with static clauses, according to our estimate based on the benchmarks given in that paper). On the other hand, BSL is not doing its best with an intervening C compiler, e.g., the traditional code motion optimization is occasionally inhibited because of the non-structured style of the C code generated by BSL.

[14] For example, for summing the elements of an array a in BSL, one method would be to use an additional array dimension to represent the values of the sum variable at successive times, and to code the program as follows: [s[0]:=0 & (∀i | 0≤i<n)[s[i+1]:=a[i]+s[i]] & ans:=s[n]]; where the array s consumes unnecessary space. Actually, since for each i in the backtracking execution of this code, $0 \le i < n$, s[i] and s[i+1] are never simultaneously live; it does not look difficult to design a compiler transformation to "coalesce" the whole array s into a single scalar variable [as in Chaitin et al. 81], but we have not attempted such a transformation in the present compiler.

```
(E ((s (array (N) op1)))
   (E ((inp op2))
      (and "initialize inp"
         (A n 0 (< n N) (1+ n)
            (H
               (and
                  (or (and conditions1 actions1)      ;
                      ...                             ; generate section
                      (and conditionsk actionsk))     ;
                  constraint1                         ;
                  ...                                 ; test section
                  constraintm)                        ;
               ((s n))
               heuristic1                             ;
               ...                                    ; recommendations section
               heuristicn)))))                        ;
```

In the generate-and-test paradigm of BSL, the computation proceeds by "generate-and-test steps," where each step consists of selecting and assigning an acceptable value to the n'th element of the solution array "s" depending on the elements 0,...,n-1, and on the given input data structure "inp". The condition-action pairs given here are the formal analogs of *production rules* [Davis and King 76], as they are used in a generate-and-test application. The conditions are side effect free subformulas that perform certain tests about elements 0,...,n-1 of the solution array and the program input, and the actions are subformulas that involve assignments to element n of the solution array. Thus a condition-action pair has the informal meaning "IF the conditions are true about the partial solution, THEN a new element as described by the actions can be added to the partial solution." The *constraints* are side effect free subformulas whose logical translations assert absolute rules about the elements 0,...,n of the solution array and the program input. They have the procedural effect of rejecting certain assignments to element n (this effect is also called *early pruning* [Hayes-Roth, Waterman and Lenat 83]). The *heuristics* are side effect free formulas whose logical translations assert desirable properties of the elements 0,...,n of the solution array and the program input. They have the procedural effect of having certain assignments to element n tried before others are, and they thus bias the search so that the solution first found is hopefully a good one. The attributes of each element of the solution array will often be subdivided into several groups which will be selected in sequence, in which case the "generate section" shown in the formula above will have some more structure.

An application of BSL: The CHORAL system

The limitation of the generate-and-test paradigm described above is that it allows only one view of the solution object, as defined by the attributes of the solution array. In complex expert systems, a need often arises for representing knowledge from *multiple viewpoints* (e.g., as in [Erman et al. 80, Sussman and Steele 80]). In fact, our music expert system, whose name is CHORAL, and whose purpose is to harmonize chorale melodies, maintains several viewpoints of its solution object, the chorale harmonization:[15] The *chord skeleton* view observes the chorale as a sequence of rhythmless chords and fermatas. The *fill-in* view observes the chorale as four inter-

[15] A chorale is a short musical piece that is sung by a choir consisting of men's and women's voices. There are four parts in a chorale (soprano, alto, tenor, bass) which are sung together; the soprano part is the main melody. Harmonization is the process of composing the alto, tenor and bass parts

acting automata that change states in lockstep, generating the actual notes of the chorale in the form of suspensions, passing tones and other ornamentations, depending on the underlying chord skeleton. The *time-slice* view observes the chorale as a sequence vertical time-slices each of which has a duration of an eighth note, and imposes harmonic constraints. The *melodic string* view observes the sequence of notes of the individual voices from a purely melodic point of view. The *Schenkerian analysis* view observes the chorale as the sequence of steps of two bottom up parsers that attempt to assign a hierarchical voice leading structure to the chorale, according to our hierarchical voice leading theory inspired from [Schenker 79]. Each viewpoint of the chorale is conceptually defined through a different set of primitive functions and predicates, which is then implemented through a solution array. However, the different solution arrays of the viewpoints are constructed in parallel, as if a dedicated generate-and-test process just like the program above were in control of each viewpoint ([Ebcioğlu 87] gives further details). The knowledge base of the system is generally very complex and computation hungry; some factors contributing to the computational and conceptual complexity being the production rules and constraints about the bold clashes of multiple simultaneous passing notes, neighbor notes, and suspensions, (without which the harmonizations would reduce to uninteresting student exercises), the long list of production rules for style-specific modulations and idiomatic cadences, and the difficult constraints on maintaining melodic interest in the inner voices.

The CHORAL system and BSL were originally implemented on VAX 11/750 and 780 computers under Franz Lisp and UNIX.[16] We have presently ported BSL and the CHORAL system to the 3081 and 3090 computers at the IBM Thomas J. Watson Research Center, they now run under VM/Lisp and CMS. The program takes an alphanumeric encoding of the chorale melody as input, and outputs the harmonization in conventional notation. The system presently incorporates over 350 production rules, constraints, and heuristics, which were found through empirical studies of the Bach chorales, personal intuitions, and traditional music treatises. It takes typically under half an hour of 3081 CPU time to harmonize a chorale, although some harmonizations require more time. At the end of the paper, we give a recent output example produced by the system, an harmonization of chorale no. 68 [Terry 64]. It has to be transposed to be singable. Many more output examples of similar quality, and the list of rules of the system can be found in [Ebcioğlu 87]. The program's harmonizations are very encouraging, but their resemblance to Bach is limited to patterns such as idiomatic cadences; they certainly lack the austere overall texture of Bach. Nevertheless, the CHORAL system seems to provide evidence that BSL can be useful for at least one substantial application, and we hope that our work with BSL will be of use to researchers looking for ways of applying the concepts of first order logic to large-scale problems.

References:

Aho, A.V., Sethi, R. and Ullman, J.V. "Compilers: Principles, Techniques and Tools" Addison-Wesley, 1986.

when the soprano is given. J.S. Bach has produced many beautiful chorale harmonizations [Terry 64], which, despite their seeming simplicity, constitute a very challenging style to imitate successfully.

[16] UNIX is a trademark of AT&T Bell Laboratories.

Chaitin, G.J., Auslander, M.A., Chandra, A.K., Cocke, J., Hopkins, M.E., and Markstein, P.W. "Register Allocation via Coloring" Computer Languages 6 (1981), pp. 47-57.

Cohen, J. "Non-deterministic Algorithms" Computing Surveys Vol. 11, No. 2, June 1979.

Date, C.J. "Introduction to Database Systems" Addison-Wesley, 1977.

Davis, R. and King, J. "An Overview of Production Systems" Elcock and Michie (eds.), Machine Intelligence 8, Wiley, 1976.

de Bakker, J. "Mathematical Theory of Program Correctness" North Holland, 1979.

Debray, S. and Warren D.S. "Detection and Optimization of Functional Computations in Prolog" Proc. third ICLP, 1986.

Ebcioğlu, K. "Computer Counterpoint" Proceedings of the 1980 International Computer Music Conference, Computer Music Association, San Francisco, California, 1981.

Ebcioğlu, K. "Report on the CHORAL project: An Expert System for Harmonizing Four-part Chorales" research report RC12628, IBM Thomas J. Watson Research Center, Yorktown Heights, 1987. (This is a revised version of the author's Ph.D. dissertation, "An Expert System for Harmonization of Chorales in the Style of J.S. Bach," technical report TR 86-09, Dept. of Computer Science, S.U.N.Y. at Buffalo, March 1986.)

Ellis, J.R. "Bulldog: A Compiler for VLIW Architectures" MIT Press, 1986.

Erman, L.D., Hayes-Roth, F., Lesser, V.R., and Reddy, D.R. "The Hearsay-II Speech Understanding System: Integrating Knowledge to Resolve Uncertainty" Computing Surveys, Vol 12, No 2, June 1980.

Fagin, B. and Dobry, T. "The Berkeley PLM Instruction Set: An Instruction Set for Prolog" Report no. UCB/CSD 86/257, Computer Science Division (EECS), University of California at Berkeley, September 1985.

Floyd, R. "Nondeterministic Algorithms" Journal of the ACM, Vol. 14, no. 4, October 1967.

Forgy, C. and McDermott, J. "OPS: A Domain Independent Production System Language" Proc. fifth IJCAI, 1977.

Gergely, T. and Szots, M. "Cuttable Formulas for Logic Programming" 1984 International Symposium on Logic Programming, February 1984.

Harel, D. "First Order Dynamic Logic" Lecture notes in Computer Science, Goos and Hartmanis (eds.), Springer-Verlag 1979.

Hayes-Roth, F., Waterman, D. and Lenat, D.B. (eds.) "Building Expert Systems" Addison-Wesley, 1983.

Hehner, E.C.R. "Predicative Programming Part I" Communications of the Association for Computing Machinery, February 1984.

Kowalski, R. "Logic for Problem Solving" North Holland 1979.

Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., Adams, N., "ORBIT: An Optimizing Compiler for Scheme" Proc. Sigplan '86 Symposium on Compiler Construction, June 1986.

Kurokawa, T., Tamura, N., Asakawa, Y., and Komatsu, H. "A Very Fast Prolog Compiler on Multiple Architectures" Proc. FJCC 1986.

Lenat, D.B. "A.M.: An Artificial Intelligence Approach to Discovery in Mathematics and Heuristic Search" Report no. STAN-CS-76-570, Stanford University, 1976.

Naur, P. (ed) "Revised Report on the Algorithmic Language ALGOL 60" Communications of the ACM, 6,1, 1-17, 1963.

Nicolau, A. "Percolation Scheduling: A Parallel Compilation Technique" TR 85-678, Dept. of Computer Science, Cornell University, May 1985.

Onai, R., Shimizu, H., Masuda, K., Matsumoto, A., Aso, M., "Architecture and Evaluation of a Reduction-based Parallel Inference Machine: PIM-R" Proc. 4th Conference on Logic Programming, Tokyo, 1985.

Ralston, A. "de Bruijn Sequences: A Paradigm of the Interaction of Discrete Mathematics and Computer Science" Mathematics Magazine, Vol. 55, 1982, pp. 131-143.

Rogers, H. "Theory of Recursive Functions and Effective Computability" McGraw-Hill, 1967.

Schenker, H. "Free Composition (*Der freie Satz*)" Translated and edited by Ernst Oster. Longman 1979.

Schwartz, J.T. "On Programming: An Interim Report on the SETL Project" Courant Institute of Mathematical Sciences, New York University, 1973.

Shoenfield, J. "Mathematical Logic" Addison-Wesley, 1967.

Smith, D.C. and Enea, H.J. "Backtracking in Mlisp2" Proceedings of the third IJCAI, 1973.

Stefik, M. "Inferring DNA Structures from Segmentation Data" Artificial Intelligence 11 (1978).

Sussman, G.J. and Steele, G.L. "Constraints - A Language For Expressing Almost-Hierarchical Descriptions" Artificial Intelligence 14(1980), 1-39.

Terry, C.S. (ed.) "The Four-voice Chorals of J.S. Bach" Oxford University Press, 1964.

Turk, A.W. "Compiler Optimizations for the WAM" Proc. 3rd ICLP, 1986.

Voda, P.J. "Choices in, and Limitations of, Logic Programming" Proc. third ICLP, 1986.

Warren, S.H., Auslander, M.A., Chaitin, G.J., Chibib, A.C., Hopkins, M.E., and MacKay, A.L. "Final Code Generation in the PL.8 Compiler" research report RC11974, IBM Thomas J. Watson Research Center, 1986.

Chorale no. 68