Horn Logic Denotations and Their Applications[1]

Gopal Gupta

Laboratory for Logic, Databases, and Advanced Programming
Department of Computer Science
New Mexico State University
Box 30001/CS, Las Cruces
New Mexico, USA 88003
http://www.cs.nmsu.edu/lldap

Abstract. In spite of decades of work, the practical impact of programming language semantics (denotational semantics) has been limited. Our thesis is that a major contributing factor to this lack of practical impact is the declarative notation used for expressing the semantics, namely, the λ-calculus. We propose to use Horn Logic (and eventually Constraint Logic) instead of the λ-calculus to express denotational semantics. This simple change leads to many practical applications, most notably to automatic program verification and automatic generation of compilers from semantic specifications. These Horn Logic denotations and their applications are discussed at length in this paper.

1 Introduction

There has been decades of work in programming language semantics, however, the practical impact of this area has been limited [40]. The reason is the different types of semantics that have been advocated—operational semantics, denotational semantics, and axiomatic semantics—that use different notations and that are designed for different types of users (language designers, implementors, programmers, etc.). Also, these different semantics are specified in a non-executable or only partially executable notation. Often, as Schmidt notes [40], formal semantics (especially denotational semantics) is mired in complex mathematics: "domain theory, intuitionistic type theory, category theory, linear logic, process algebra, continuation-passing style, or whatever" [40]. To quote him further: "formal semantics has fed upon increasing complexity of concepts and notation at the expense of *calculational clarity.*" "General users desire semantics definitions with strong *calculational flavor.*" We believe that this lack of "calculational flavor" has severely hindered the practical applications of programming language semantics. Schmidt offers the following challenge for researchers working in semantics:

[1] The author has been supported by NSF grants CCR 96-25358, CDA-9729848, EIA 98-10732, HRD 98-00209, HRD 96-28450, and INT 95-15256, and by a grant from the Fullbright Foundation.

> "A challenge for semantics writers is the following: design a calculational semantics for a significant subset of, say, Java, that can be learned and applied by first-year university students to debug their programs. Perhaps such a semantics will use graphical state and small-step transitions upon the state for its calculations, but calculation upon program properties—that is, a variation on predicate transformer semantics—is also a possibility. In any case, it is crucial that the formulation be "mathematical" in the sense that denotational semantics was first termed mathematical: there must be an underlying consistent proof theory. And of course, the semantics should be some form of extension of BNF."

In this paper we present a simple solution to providing this "calculational clarity" to formal semantics. Our solution has all the characteristics that Schmidt requires a "popular" semantics to possess. Our simple solution [14] is to change the notation used for expressing denotational semantics: instead of the traditional λ-calculus, we propose to use *Horn Logic* (and eventually constraint logic [21]) as the language for specifying semantics. Horn logic is the basis of logic programming and the Prolog language [42]. By using Horn logic as the language for writing semantics, all three semantics—operational, denotational, and axiomatic—can be expressed in the same notation that is also executable, leading to myriad applications:

- Given that the syntax of a language, $\mathcal{L}$, can be expressed as a Definite Clause Grammar [42], the syntax of $\mathcal{L}$ can also be specified as a Horn Logic Program. Because the syntax and semantics of $\mathcal{L}$ are both expressed in Horn Logic, both are executable—the syntax specification yields a parser for $\mathcal{L}$, the semantics specification yields the back-end of an interpreter for $\mathcal{L}$.
- Given this executable syntax and semantic specification, various operational semantics for $\mathcal{L}$ can be obtained by choosing an appropriate Horn Logic selection function. Choosing an operational semantics yields an interpreter for $\mathcal{L}$ (according to that operational semantics).
- Partial evaluation of the interpreter (using a partial evaluator, such as Mixtus [36]) w.r.t. to a given program, $\mathcal{P}$ (written in $\mathcal{L}$), yields "compiled code" for $\mathcal{P}$. Provably correct compiled code is thus automatically obtained from the semantic specification.
- By using a bottom-up evaluation strategy (or a *tabling-based* evaluation strategy [6,37]), the fixpoint of the denotation of a program written in $\mathcal{L}$ can be computed and used for verification purposes.
- The postconditions and preconditions can be inserted in the program's Horn logical denotation and evaluated; by choosing a precondition carefully, and turning it into a *generator*, certain properties of interest of the program can be verified.
- By using evaluation engines other than Prolog, such as those that are tabling-based [6,37], and by choosing abstract semantics, abstract inter-

preters [10] for a language $\mathcal{L}$ can be automatically obtained from the abstract semantic specification of $\mathcal{L}$.
- By choosing a suitable semantic algebra, and using partial evaluation, abstract machines can be derived and studied for the language in question.
- A formal theory of *semantic porting* is also obtained. If a program written in a language $\mathcal{L}_1$ is to be ported to language $\mathcal{L}_2$, then the denotational semantics of $\mathcal{L}_1$ can be given in terms of $\mathcal{L}_2$. The interpreter obtained by using Horn logic for specifying the syntax (as a DCG) and semantics, is effectively a (formal, provably correct) language translation system.
- Our approach can be used for building executable software specifications. A software system can be thought of as an evaluator of its input language. The denotational semantic specification of this input language is a specification of the software system, and if this specification is executable (as is the case if Horn Logic is used), then a (rapid) prototype implementation of the software is obtained. The executable specification can be used for obtaining efficient implementation as well as for verification.
- Semantics of parallel languages is easily given, as logic programs naturally encapsulate parallelism. Additionally, parallelizing compilers can also be derived from *parallel semantics* of the language.
- By generalizing Horn Logic to Constraint Logic, more complex software systems, e.g., real-time systems, can be elegantly modeled and verified.

The two principal advantages of switching to Horn Logic are: (i) both the syntax and semantic specification are executable, yielding an interpreter; and, (ii) The logical denotation of a program can also be used for verification. Most of the applications listed above follow from one of the two cases.

2 Logical Denotations

Denotational semantics [38,39,18] of a language has three components:
- *syntax*: specified as a context free grammar;
- *semantic algebra*: these are the basic domains along with associated operations; meaning of a program is expressed in terms of these basic domains.
- *valuation function*: these are mappings from patterns of parse trees and semantic algebras to values in the basic domains in the semantic algebra.

Traditional denotational definitions express syntax in the BNF format, and the semantic algebras and valuation function in the λ-calculus. However, a disadvantage of this approach is that while the semantic algebra and the valuation functions could be easily made executable, syntax checking and generation of parse trees cannot. A parser has to be written (or generated) to do syntax checking and generate parse trees. These parse trees will then be processed by the valuation functions to produce the program's denotation. These two phases constitute an interpreter for the language being defined.

An interpreter for a language can be thought of as a specification of its operational semantics, however, using traditional notation (BNF and λ-calculus) it has to be obtained in a complex way.

The reason λ-calculus has been traditionally used for specifying the semantic algebras and valuation functions component of denotational semantics is because when it was first proposed by Scott, functional programming (with its formal basis in λ-calculus) was the only declarative formalism that was available. A declarative computational formalism is needed since the semantics is supposed to be mathematical (declarative), i.e., it should have a "consistent proof theory" [40]. However, today there is another declarative formalism that is available, namely, logic programming (with its formal basis in Horn logic, a subset of predicate logic). The additional advantage that logic programming possesses, among others, is that even syntax can be expressed in it at a very high level, and *a parser for the language is immediately obtained from the syntax specification*. Moreover, generation of parse trees requires a trivial extension to the syntax specification. (The parsing and parse tree generation facility of logic programming is described in almost every logic programming textbook, under the heading Definite Clause Grammars). The semantic algebras and valuation functions are also expressed in Horn logic programming quite easily, as relations (or predicates) subsume functions. The semantic algebra and valuation functions are executable, and can be used to obtain executable program denotation. A significant consequence of this is that the fixpoint of a program's denotation can be executed bottom-up (assuming that it is finite), which can then be used for verification. Thus, verification can also be done in the framework of Horn Logic.

Thus, given a language, both its syntax and semantics can be directly specified in logic programming. This specification is executable using any standard logic programming system. What is noteworthy is that different operational models will be obtained both for syntax checking and semantic evaluation by employing different execution strategies during logic program execution. For example, in the syntax phase, if left-to-right, Prolog style, execution rule is used, then recursive descent parsing is obtained. On the contrary, if a *tabling-based* [6] execution strategy is used then chart parsing is obtained, etc. Likewise, by using different evaluation rules for evaluating the semantic functions, strict evaluation, non-strict evaluation, etc. can be obtained. By using bottom-up or tabled evaluation, the fixpoint of a program's denotation can be computed, which can be used for verification and structured debugging of the program.

2.1 Discussion

Denotational semantics expressed in a Horn Logic notation is executable. So is denotational semantics expressed in the λ-calculus notation. However, Horn-logic expressed semantics allows for fixed points of programs to be computed much more intuitively, simply, and efficiently than, we believe, in the case of the λ-calculus. There is a whole body of literature and implemented systems (e.g., tabling based systems such as XSB [6]) for computing fixpoints of logic programs because of their applicability to deductive databases [37,25]. Due to this reason, semantics based verification (and program debugging) can be much more easily performed in the Horn Logic denotational framework than using the λ-calculus based denotational framework, as is shown later. This becomes much more prominent when we generalize Horn Logic to Constraint Logic. This generalization makes specification and verification of very complex systems, e.g., real-time systems considerably easier [12] (it should be noted that the denotational specification and verification of real-time systems using the traditional λ-calculus approach will be quite cumbersome and complex).

One could argue that the denotation or mathematical meaning of a program (which denotational semantics attempts to captures) exists independently of the notation used for expressing it, just as integers exist independent of their representation (base 2, base 10, or whatever). Indeed denotations exist independent of the notation used, however, if we wish to use the denotation (meaning) of a program for some practical purpose, such as automatic derivation of efficient compiled code, verification of program properties, etc., then it has to be expressed symbolically in some form or the other for processing. *The symbolic notation in which the denotation is expressed will determine how easy or difficult this processing is going to be.* To complete the analogy with integers, the binary representation of integers is the most efficient form for realization on computers. Another analogy can be given from the field of Algorithms. In theory, an algorithm exists independently of the notation used to express it. However, in practice, an algorithm expressed in a high-level language is a lot easier to code, understand, and modify compared to one expressed in, say, an assembly language. Our thesis thus is that the notation used for expressing denotation has a tremendous impact on how many further (practical) uses it can be put to, and the ease with which it can be put to these uses.

Denotational semantics expressed as Horn Logic satisfies all the criteria laid out by Schmidt. Horn Logic has a consistent proof theory, and hence various automatic systems can be built for further processing Horn Logic Denotations, e.g., automatic generators of compiled code (including automatic generators of parallel compiled code), automatic program property verifiers, etc. The BNF spirit of denotational semantics is maintained since the meaning is still expressed as maps between parse tree patterns (abstract syntax) and "meaning spaces" (semantic algebras). A strong calculational flavor is present, as Horn Logic is executable, which permits both execution and verification.

Thus, given the Horn Logic Denotational Semantic of a language $\mathcal{L}$, the Horn Logic Denotation of a program written in $\mathcal{L}$ can be computed. This denotation is executable and can be put to many uses which include implementation (obtaining interpreters and compilers) and verification.

2.2 An Example of Logical Denotational Semantics

Consider a very simple subset of a Pascal like language that contains assignment statement, if-then-else, and while-do statement. To keep matters simple, assume that the only possible variable names allowed in the program are `w`, `x`, `y` and `z`, and that the only data-type allowed is integers. Assume, again for simplicity, that constants appearing in the program are only 1 digit long. The context free grammar of this language is given in Figure 1. This BNF is easily transformed into a definite clause grammar (DCG) by a simple change in syntax [42] (plus removal of left-recursion, if a Prolog system is going to be used for its execution).

```
Program ::= C.
C ::= C1;C2 |
      loop while B C endloop while |
      if B then C1 else C2 endif |
      I := E
E ::= N | Identifier | E1 + E2 |
      E1 - E2 | E1 * E2 | (E)
N ::= 0 | 1 | 2 | ... | 9
Identifier ::= w | x | y | z
```

Fig. 1: BNF grammar

```
             SYNTAX:
program(p(X)) --> comm(X), [.].
comm(X) --> comm1(X).
comm(comb(X,Y)) --> comm1(X),[;],comm(Y).
comm1(assign(I,E)) --> id(I),[:=],expn(E).
comm1(ce(X,Y,Z)) --> [if],expn(X),[then], comm(Y),
                        [else], comm(Z), [endif].
comm1(while(B,C)) --> [loop, while], bool(B),
                    comm(C), [endloop, while].
expn1(id(X)) --> id(X).
expn1(num(X)) --> n(X).
expn1(e(X)) --> ['('],expn(X),[')'].
expn(X) --> expn1(X).
expn(add(X,Y)) --> expn1(X),[+],expn(Y).
expn(sub(X,Y)) --> expn1(X),[-],expn(Y).
expn(multi(X,Y)) --> expn1(X),[*],expn(Y).
bool(equal(X,Y)) --> expn(X),[=],expn(Y).
bool(greater(X,Y)) --> expn(X),[>],expn(Y).
bool(less(X,Y)) --> expn(X),[<],expn(Y).
id(x) --> [w].        id(x) --> [x].
id(y) --> [y].        id(z) --> [z].
n(0) --> [0].         ....         n(9) --> [9].
```

Fig. 2: DCG for the BNF above

An extra argument has been added in which the parse-tree is synthesized. The DCG is a logic program, and when executed, a parser is automatically obtained. This parser parses a program in this simple language and produces a parse tree for it. This DCG is shown in Figure 2. Thus, the query to parse the program for computing the value of y^x and placing it in variable z:

```
?- program(P, [z,=,1,;, w,=,x,;,
               loop, while,w,>,0,
                  z,=,z,*,y,;, w,=,w,-,1,
               endloop, while], []).
```

will parse that program as syntactically correct, and produce the parse tree shown below:

```
P = p(comb(assign(z,num(1)),
    comb(assign(w,id(x)),while(
       greater(id(w),num(0)),
          comb(assign(z,multi(id(z),id(y))),
          assign(w,sub(id(w),num(1))))))))
```

The denotational semantics can be defined next, by expressing the semantic algebra and the valuation functions as logic programs. In the semantic definition, we assume that the input is initially found in variables `x` and `y`. The answer is computed and put in variable `z`. The semantic algebra consists simply of the `store` domain, realized as an association list of the form `[(Id, Value) ...]` with operations for creating, accessing, and updating the store, and is given below as a logic program:

```
SEMANTIC ALGEBRA 1:
initialize_store([]).
access(Id,[],0). %return 0 if uninitialized
access(Id,[(Id,Val)|_ ],Val).
access(Id,[_|R],Val) :- access(Id,R,Val).
update(Id,NewV,[],[(Id,NewV)]).
update(Id,NewV,[(Id,_)|R],[(Id,NewV)|R]).
update(Id,NewV,[P|R],[P|R1]) :- update(Id,NewV,R,R1).
```

Next, the valuation functions, that impart meaning to the language are specified, again as logic programs. These valuation functions, or valuation predicates, relate the current store, a parse tree pattern whose meaning is to be specified, and the new store that results on executing the program fragment specified by the parse tree pattern. These valuation predicates are shown in Figure 3; the very first valuation predicate takes the two input values, that are placed in the store locations corresponding to `x` and `y`. Once the syntax, semantic algebras, and valuation functions are defined as a logic program, an interpreter is immediately obtained. Now for both parsing and interpreting the program, define the logic program in Figure 4. The predicate `main` in Figure 4 is the denotation of the exponentiation program.

If the above syntax and semantics rules are loaded in a logic programming system and the query `?- main(5,2,A)` for computing the value of 2^5 posed, then the value of `A` will be computed as `32`. Notice that switching to logic programming for specifying denotational semantics results in a complete interpreter. Additionally, the fixpoint of this denotation can be computed and used, for example, for verification and debugging purposes (see later).

```
prog_eval(p(Comm), Val_x, Val_y, Output) :- initialize_store(Store),
      update(x, Val_x, Store, Mst), update(y, Val_y, Mst, Nst),
      comm(Comm, Nst, Pst), access(z, Pst, Output).
comm(comb(C1, C2), Store, Outstore) :- comm(C1, Store, Nstore),
                          comm(C2, Nstore, Outstore).
comm(while(B, C), Store, Outstore) :-
        (bool(B, Store) -> comm(C, Store, Nstore),
           comm(while(B, C), Nstore, Outstore); Outstore=Store).
comm(ce(B, C1, C2), Store, Outstore) :- (bool(B, Store) ->
           comm(C1, Store, Outstore); comm(C2, Store, Outstore)).
comm(assign(I, E), Store, Outstore) :-
       expr(E, Store, Val), update(I, Val, Store, Outstore).
expr(add(E1, E2), Store, Result) :- expr(E1, Store, Val_E1),
       expr(E2, Store, Val_E2), Result is Val_E1+Val_E2.
expr(sub(E1, E2), Store, Result) :- expr(E1, Store, Val_E1),
               expr(E2, Store, Val_E2), Result is Val_E1-Val_E2.
expr(multi(E1, E2), Store, Result) :- expr(E1, Store, Val_E1),
               expr(E2, Store, Val_E2), Result is Val_E1*Val_E2.
expr(id(X), Store, Result) :- access(X, Store, Result).
expr(num(X), _, X).
bool(greater(E1, E2), Store) :- expr(E1, Store, Eval1),
             expr(E2, Store, Eval2), Eval1 > Eval2.
bool(less(E1, E2), Store) :- expr(E1, Store, Eval1),
             expr(E2, Store, Eval2), Eval1 < Eval2.
bool(equal(E1, E2), Store) :- expr(E1, Store, Eval),
             expr(E2, Store, Eval).
```

Fig. 3: Valuation Predicates (with Semantic Algebra 1)

In the semantics above, we assumed that the store is maintained as an association list (SEMANTIC ALGEBRA 1), which is passed around as an argument. This does not exactly model imperative languages in which the memory store is treated as a global entity. Given that logic programs can support global data structures through their database facility (the `assert` and `retract` built-ins [42]), it is better to model the store as a collection of dynamic facts manipulated using `assert` and `retract`. If we decide to adopt this point of view, *the only change that will take place will be in the store algebra*, which transforms to:

```
main(ValX, ValY, A) :-
    program(P, [z,=,1,;, w,=,x,;,
                  loop, while,w,>,0,
                       z,=,z, *, y, ;,
                       w,=,w,-,1,
                  endloop,while], []),
    prog_eval(P,ValX,ValY,A).
```

Fig. 4: Program Denotation

```
SEMANTIC ALGEBRA 2 (globalized):
access(Var,Val) :- (store(Var,Val) -> true; Val = 0).
update(Var,Val) :- retractall(store(Var,_)),
          assert(store(Var,Val)).
```

Now, both the input and the output store arguments can be eliminated in the semantic valuation predicates. While we do use impure features of logic programming in this globalized semantic algebra, it should be noted that these features are restricted to the semantic algebra only. The advantage, however, is that the store argument that threads through the definition of valuation predicates is eliminated.

2.3 Discussion

Note that in the denotational specification, the meaning of the while program was given recursively, rather than using the traditional fixpoint approach [38,39]. This recursive meaning of the while loop preserves the "calculational aspect" of our logical denotational semantics. If one is interested in computing the fix points, then one can compute the fixpoint for the whole program's denotation (rather than compute it for pieces of the program and mix it with the denotation as is done in traditional λ-calculus based approach). In fact, we'd argue that one feature that hinders the "calculational clarity" of traditional λ-calculus based denotational semantics is the mixing of fixpoints and regular functions. As a consequence, the resulting denotation is quite complex, and difficult to reason with and process automatically. Specifying the semantics of the `while` loop recursively will have advantages in automatic compilation, as we will see shortly.

Note also that our semantic definition on two occasions uses impure Prolog: (i) the if-then-else `(p->q;r)` of Prolog has a hidden cut; and the arithmetic built-in `is`. Regarding (i), in most languages (especially in languages we are most interested in, namely, imperative languages and domain specific languages) the condition `p` is deterministic (and, hence, the hidden cut is inconsequential). In such a case, a (bottom-up) declarative semantics can be given to `(p->q;r)`. Thus, for all practical purposes, the if-then-else is declarative, so that the mathematical nature of the logical denotational semantic is not lost. Regarding (ii), the `is/2` arithmetic operator is not declarative, however, if we switch to arithmetic constraints [21] (and constraint logical denotations) to express arithmetic, then this declarativeness is restored.

A concern that one might have is that traditional denotational definitions make extensive use of higher order functions, which do not fit logically in the framework of logic programming. However, it should be noted that higher order functions are not indispensable for expressing denotations (a language without higher order functions can express all computable functions). The use of higher-order functions just makes the denotations more compact. In fact, this compactness and terseness introduced due to higher-order functions sometimes leads to difficult-to-understand semantic specification. Also, the presence of higher-order functions makes automatic processing of denotations (for verification, or compiler generation) extremely difficult. Thus, even though logic programming does not support higher order functions/predicates logically, it is adequate for specifying denotational semantics [48], including continuation semantics. The effect of *continuation semantics* is easily achieved, for example, by keeping list of commands still to be executed as an argument of the valuation predicates [38,14].

The main advantage of denotational semantics is that there exist powerful proof rules for reasoning about program equality and program properties: the principal of extensionality and fixed-point induction [38]. It would appear that these would have to be abandoned on switching to Horn Clause Denotations. However, this is not true. The principle of extensionality and fixed point induction are independent of the notation used for coding the denotation. Two predicates are the same if their fixpoints are identical (if the fixpoints are identical then essentially, for every possible input combination both predicates evaluate to true, which is essentially the principal of extensionality). Likewise, fixpoint induction applies equally well to Horn Logic Denotations. The fixpoint semantics of Horn logic programs is expressed using the T_p operator [25] (which is essentially the same as the functionals used in defining fixpoints of functions). To prove a property $\mathcal{P}$ for a predicate `p` one will have to show that $\mathcal{P}$ holds for $T_p \uparrow 0$ (which is essentially equivalent to the base case of fixpoint induction where one has to show that the property holds for $\lambda n.\bot$), and then show that if $\mathcal{P}$ holds for $T_p \uparrow n$ then it also holds for $T_p \uparrow (n+1)$ (which essentially carries out the induction step of the fixpoint induction). The fixpoint theory of logic programs is an extensively researched subject in the deductive database and logic programming community [45,3,37,6]. Efficient fixpoint computation engines such as XSB exist [6], and have been used for verification and model checking [33].

It is possible that one could dismiss Horn Logic Denotations as "Prolog coded interpreters," however, the uses to which these "Prolog coded interpreters" can be put to are many. If nothing else, our work demonstrates how such a "Prolog coded interpreter" can be systematically derived using a denotational approach and easily used for facilitating the task of verification/debugging, automatically obtaining efficient implementation, compilation, abstract interpretation, interoperation, etc. In fact, use of denotational definitions for the purpose of proving properties of programs is virtually non-existent, though this is one area where denotational semantics should be extensively used. The use of semantics for automatically proving properties of programs is one major research direction our research hopes to highlight [12,15,33].

One could argue that Horn Logic Denotations really represents operational semantics and not (declarative) denotational semantics. Two points could be made regarding this argument: (i) *implementational denotational semantics* have been proposed in the past as a means of incorporating operational concepts into denotational semantics [32]. One could argue that implementational denotational semantics are best represented in Horn Logic. (ii) A Horn Logic specification is declarative, in that any selection function [25] can be chosen to evaluate a Horn logic program. As we will see, for some applications (e.g., compilation) SLD resolution is better suited, while for certain others (e.g., verification) bottom-up evaluation is more appropriate.

3 Provably Correct Compilation

We next show the first application of a program's logical denotation: automatic generation of compiled code using simple techniques such as partial evaluation. Producing a true compiler generator has been a long quest for researchers in compiler theory and programming language semantics. While research in parser generators is very mature, research in back-end generators is not. It is well known in partial evaluation [20] that compiled code for a program $\mathcal{P}$ written in language $\mathcal{L}$ can be obtained by partially evaluating $\mathcal{P}$ w.r.t. the interpreter for $\mathcal{L}$. We have already obtained an interpreter for the language from its denotational specification. Removal of the semantic algebra for the store from our definition, followed by partial evaluation of the interpreter w.r.t. the program for computing y^x, results in "compiled" code. Our goal is to treat the semantic algebra operations as primitives, and hide their implementation from the partial evaluator. Using the Mixtus partial evaluation system from SICS [36], the program that results after partially evaluating the query `?- main(5,2,A)` from the previous section (using SEMANTIC ALGEBRA 1) is shown in Figure 5.

```
main(5,2,A) :-
      initialize_store(B),
      update(a,5,B,C),
      update(b,2,C,D),
      update(z,1,D,E),
      access(x,E,F),
      update(w,F,E,G),
      commandwhile(G,H),
      access(z,H,A).
commandwhile(A,B) :-
     (access(w,A,C),
      0<C -> access(z,A,D),
         access(y,A,E),
         F is D*E,
         update(z,F,A,G),
         access(w,G,H),
         I is H-1,
         update(w,I,G,J),
         commandwhile(J,B)
       ; B=A ).
```

Fig. 5: Compiled code

The resulting program is very similar to compiled code. Essentially, a series of memory `access`, memory `update`, arithmetic and comparison operations are left, that correspond to `load`, `store`, arithmetic, and comparison operations of a machine language. The while-loop, whose meaning was expressed using recursion, will (always) partially evaluate to a *tail-recursive* program. These tail-recursive calls are easily converted to iterative structures using jumps. Note that the update and access operations are also parameterized on the store name (in contrast, load and store operations of any machine architecture do not take the whole store as an argument). This parameter can easily be eliminated through globalization [38] (in fact, if we rewrote our valuation predicates using the globalized SEMANTIC ALGEBRA 2, given earlier, and then partially evaluated the interpreter, the store arguments will disappear; Thus, one could use ALGEBRA 2 for compilation, and ALGEBRA 1 for other applications where declarative purity is important). Thus, compiled machine code is just a few simple transformation steps away, and provably correct compiled code is obtained simply and automatically. Note that we can be confident that compiled code is correct only if we can prove that the partial evaluator is correct; the correctness of the partial evaluator needs to be proven only once.

Logic programming languages have been widely regarded as highly suitable for building compilers. In [49] Warren shows how the various phases of a compiler writing can be implemented as logic programs. The discussion here goes one step further than Warren's ideas, and shows how a compiler can be obtained from a language specification with little effort. Automatic compilation via partial evaluation is perhaps most useful for obtaining compilers and experimenting with implementations of *domain specific languages* [9,31] (see later).
Partial evaluation can also be used to obtain *parallel* code from the *parallel semantics* of an imperative language (e.g., Fortran). For obtaining parallel code, Horn logic has to be generalized to Constraint Logic. The idea is that if a parallel semantic specification is given for the language, then parallel compiled code can be obtained from this semantic specification via partial evaluation. Parallelism in imperative languages (e.g., Fortran) arises when *independent* iterations of a DO-loop are executed in parallel. Checking for independence of iterations requires solving linear Diophantine equations [50]. Linear Diophantine equations are nothing but linear arithmetic constraints, and hence independence conditions are easily and straightforwardly expressed in the constraint logical denotational framework. Partial evaluation of this parallel semantics (e.g., using Mixtus [36]) yields parallel code. Note that the parallel code obtained is provably correct. Details can be found in [13].

```
program
integer length, width, area, i
integer dimension (100) arr
length = 12
width = 3
do i = 1,8
arr[2*i] =length+1
length = arr[i+1]
enddo
end program
```

Fig. 6: A Fortran Program

For illustration purposes, we partially evaluate (using Mixtus) the parallel semantics of a simple Fortran like language (that includes variable declarations, DO loops, and single dimensional arrays) with respect to the program above (Figure 6). The parallel compiled code obtained is shown below:

```
COMPILED CODE output by MIXTUS:
go1 :- create_store,
       update(length, 0),
       update(width, 0),
       update(area, 0),
       update(i, 0),
       update(length, 12),
       update(width, 3),
       execute_body1(1).
execute_body1(A) :-
           (A>8 -> true  ;
            par_begin(task1, task2),
             start_task(task1),
                ( A=7 ->
                    wait(4) ;
                    true ),
                ( A=5 ->
                    wait(3) ;
                    true ),
                ( A=3 ->
                    wait(2);
                    true ),
                access(length, B),
                C is B+1,
                D is 2*A,
                E is D,
                update_arr(arr, E, C),
                F is A,
                G is F+1,
                access_arr(arr, G, H),
                update(length, H),
                ( A=4 ->
                    signal(7) ;
                    true ),
                ( A=3 ->
                    signal(5) ;
                    true ),
                ( A=2 ->
                    signal(3) ;
                    true ),
             end_task(task1),
             start_task(task2),
               I is A+1,
               execute_body1(I),
             end_task(task2),
            par_end).
```

Due to lack of space, we only show the final generated code and not the actual parallel semantics. Observe how array dependences are captured as dependencies between loop iterations that synchronize using `wait` and `signal`. Note that `par_begin`, `par_end`, `signal`, `wait`, `end_task`, `start_task` are constructs introduced in the semantic algebra of the parallel semantics (not shown here). Note that `execute_body1` is a tail-recursive (parallel) function, and hence can be converted to an iterative loop. Note also that the code is obtained from a globalized parallel semantics (i.e., the parallel semantics uses SEMANTIC ALGEBRA 2); it does not have a memory argument threading through.

4 Program Denotation & Verification

Axiomatic semantics is perhaps the most well-researched technique for verifying properties of programs. Axiomatic semantics which is traditionally expressed in first order logic, can also be expressed in Horn logic (or at least quite a large subset of axiomatic semantics can be expressed), as Horn logic is a subset of first order logic. In Axiomatic Semantics [19] preconditions and postconditions are specified to express conditions under which a program is correct. The notation $(P)C(Q)$ states that if the predicate P is correct before execution of command C, then Q must be correct afterwards. The pre-conditions and post-conditions can be expressed in Horn logic. The post-conditions of a program are theorems with respect to the denotation of that program and the program's pre-conditions [38,39]. Given that the denotation is expressed in Horn logic, the pre-conditions can be incorporated into this denotation, and then the post-conditions can be executed as queries w.r.t this extended program denotation, effectively checking if these post-conditions are satisfied or not. In effect, *model checkers* [8] can be specified and generated automatically. By generalizing Horn logic to constraint logic, real-time systems can also be specified and implemented [12] and parallelizing compilers obtained [13].

One way to prove correctness is to show that given the set of all possible state-configurations, S, that can exist at the beginning of the command C, if P holds for a state-configuration $s \in S$, then Q holds for the state-configuration that results after executing the command C in s. If the denotation is a logic program, then it is possible to generate all possible state-configurations. However, the number of such state-configurations may be infinite. In such a case, the precondition P can be specified in such a way that it acts as a *finite* generator of all possible state-configurations. A model checker is thus obtained from this specification. This model checker can be thought of as a debugging aid, since a user can obtain a program's denotation, add preconditions to it and then pose queries to verify the properties that (s)he things should hold.

We next give an example. Consider the program for computing y^x, whose logical denotation was given earlier. If we were to add the preconditions and the postconditions to the programs, we will obtain the annotated program in Figure 7. In Figure 7, `I(x)` means `x` is an integer. We assume that a variable appearing in the precondition is an input variable, and cannot be modified in the program. If the preconditions `I(x)` and `I(y)` could be turned into generators for integer values of `x` and `y`, then we can verify the correctness of the postcondition for all possible `<x,y>` pairs. Theoretically, `I(x)` and `I(y)` will produce infinite number of values, making the number of `<x,y>` pairs that can serve as input to this program infinite. However, if the preconditions were such that the state-configuration satisfying them are finite in number (i.e., preconditions act as finite generators), then this verification can be automatically done by encoding the preconditions and the postconditions in Horn logic.

```
(I(x) & I(y) & y >= 0)
z := 1 ; w := x;
while w > 0 do
     z := z * y ;
     w := w - 1;
od
(z = y**x)
```

Fig. 7: Annotated Program

Suppose we modified our precondition to state that `x` and `y` are numbers between 0 and 10, then the postcondition `z = y**x` can be immediately verified. The verification is done, by encoding the preconditions as `between(0,10,ValX)`, `between(0,10,ValY)` that acts as a generator for `ValX` and `ValY`, which are the values with which the program is interpreted. The program is further augmented with the negation of the post-condition as shown (negation ensures that the entire search space is explored, a substitute for computing the fixpoint of the program):

```
?- between(0,10,VarX), between(0,10,VarY),    %precondition
   main(VarX,VarY,Out),                       %interpret the program
   Z is VarY**VarX, Out \== Z.                %negated postcondition
```

where `between` is defined as:

```
    between(I, J, I) :- I =< J.
    between(I, J, K) :- I < J, I1 is I+1, between(I1, J, K).
```

The above query will fail, proving that if $0 \leq X, Y, \leq 10$, then indeed this program works correctly. Essentially, the logical denotation together with the precondition as (finite) generators provides an easy way of generating all possible program state-space sequences. The post-condition is verified for each such sequence. For large programs, the search space may be enormous. However, a little better efficiency can be achieved by partially evaluating the above query and then running the resulting program (essentially, checking for the post-condition on the compiled program).

The set of all possible state-space sequences for model-checking can also be generated by using a logic programming system based on bottom-up evaluation or on tabling. In this sense, our ideas provide a formal framework for deductive model-checking as espoused in [33]. Essentially, by using logi-

cal denotations, deductive model-checkers can be automatically specified and implemented.
Our approach also produces the insight, namely, that model checking is essentially constraint solving. Let's use the CLP(FD) [44] notation for the previous query[2]:

```
?- VarX :: [1..10], VaryY :: [1..10],  %preconditions.
   main(5,2,A),                        %call to the program
   Z = VarY**VarX, A =\= Z             %negated postcondition
   indomain(VarX), indomain(VarY).     %generators
```

This will lead to a considerably more efficiency of verification, since finite-domain constraint solving can explore the search space more efficiently [44].

While the example given may sound simple, the fact that we can verify certain properties (even though with certain restriction) is of considerable importance. It shows that a logical denotational semantics can be used for verifying and debugging program. Essentially, the Horn Logical denotation axiomatizes a program w.r.t. the semantics of the language the program is written in. This axiomatized program can then be used for verification. Indeed this is an answer to Schmidt's challenge that we "design a calculational semantics for a significant subset of, say, Java, that can be learned and applied by first-year university students to debug their programs."

When we generalize Horn Logic to Constraint Logic more interesting applications become possible. For example, real-time systems can be modeled and verified/debugged using our approach. A real-time system is essentially a recognizer of a sequence of timed-event. A sequence of timed-event is correct if the individual events occur in a certain order (syntactic correctness) and the time at which these events occur satisfy the time-constraints laid out by the real-time system specification (semantic correctness). The syntax and semantics of a real-time system can be specified using constraint logic denotations [12] (the key insight is that the specification of a real-time system is a semantic specification of its corresponding timed-language [12]). Essentially, the time constraints are modeled as constraints over real numbers [21]. The semantic algebra models the state, which consists of the global time (wall-clock time), the valuation predicates are maps from sequence of events to the global time. This constraint logical denotational specification is executable and can be used for verifying interesting properties of real-time systems, e.g., safety (for instance, if we design a real-time system for a railroad gate controller, we want to make sure that at the time a train is at the gate, the gate can never be open). In the real-time system modeled, we don't exactly know when an event is actually going to occur, all we know is the relationship between the time at which different events took place. Thus, the exact time

[2] The notation `X::[1..10]` specifies the finite domain of variable `X` to be the set $\{1..10\}$ [44].

at which each event took place cannot be computed from the constraint logical denotation. However, the constraints laid out in the denotation together with constraints that the safety property enforces can be solved to check for their consistency (essentially, the constraints laid out by the real-time system should entail the constraints laid out by the properties to be verified, if the property indeed holds). More details of how the denotational approach has been applied to modeling, verification and debugging of real-time systems, can be found elsewhere [12]. We include a short example below.

Let us consider the following problem. An automatic controller is designed to handle a gate at a railroad crossing. The system is composed of three entities, a gate-controller, the gate itself, and the train (the example is adapted from [1]). The train is modeled by the *timed-automata* [1] shown in figure 8(i). It recognizes four different events: (i) `approach` denoting approach of the train to the gate; (ii) `in` denoting that the train is at the gate; (iii) `out` denoting that the train has just left the gate; and, (iv) `exit` denoting that the train has left the gate area. The controller is modeled by the timed-automata in figure 8(ii); it recognizes the `approach` and `exit` events described above, together with the two events: (i) `lower` denoting starting of the lowering of the gate; and, (ii) `raise` denoting starting of the raising of the gate.

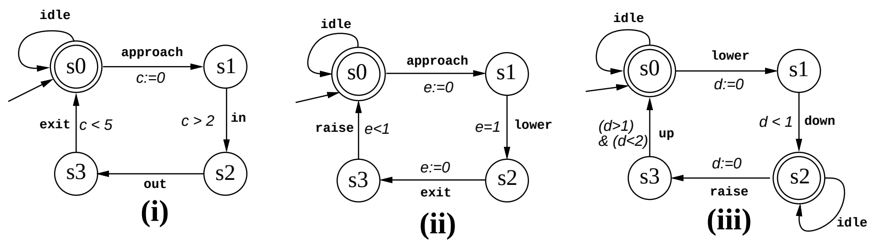

Fig. 8. Train, Controller, and Gate automata

Finally, the gate is modeled by the timed-automata in figure 8(iii). Time constraints force the train to employ at most 5 units of time to cross the gate area, with a speed which should allow 2 units of time to the gate to lower before the arrival of the train. The controller should be able to react in less than one unit of time to the approach of the train, and the gate should employ no more than a unit of time to completely lower or raise the gate.
This real-time system can be specified using the denotational approach. Essentially, the correct sequence of events is captured by a context free grammar, while the timing constraints are captured as part of the semantics using constraints over reals. The complete denotational specification of the system in the figure is given below. Note that because we are dealing with an automata, the syntax recognition is done by a transition table, and the timing constraints are attached to the transitions; Also, the semantic algebra we need is for modeling the global clocks and is also fused with the single

specification; thus, the syntax and semantics phase are modeled in one single specification. However, if we were dealing with *timed push down automata* than we will have to have separate syntax and semantic phase [12].

```
train(s0,approach,s1,T1,T2,T3) :- T3 = T1.
train(s1,in,s2,T1,T2,T2) :- T1 - T2 > 2.
train(s2,out,s3,T1,T2,T2).
train(s3,exit,s0,T1,T2,T2) :- T1 - T2 < 5.
train(X,lower,X,T1,T2,T2).
train(X,down,X,T1,T2,T2).
train(X,raise,X,T1,T2,T2).
train(X,up,X,T1,T2,T2).
```

```
gate(s0,lower,s1,T1,T2,T1).
gate(s0,lower,s1,T1,T2,T1).
gate(s1,down,s2,T1,T2,T2)
     :- T1 - T2 < 1.
gate(s2,raise,s3,T1,T2,T1).
gate(s3,up,s0,T1,T2,T2)
     :- T1-T2 > 1, T1-T2 < 2.
gate(X,approach,X,T1,T2,T2).
gate(X,in,X,T1,T2,T2).
gate(X,out,X,T1,T2,T2).
gate(X,exit,X,T1,T2,T2).
```

```
cntl(s0,approach,s1,T1,T2,T1).
cntl(s1,lower,s2,T1,T2,T2)
       :- T1 - T2 = 1.
cntl(s2,exit,s3,T1,T2,T1).
cntl(s3,raise,s0,T1,T2,T2)
       :- T1-T2 < 1.
cntl(X,in,X,T1,T2,T2).
cntl(X,out,X,T1,T2,T2).
cntl(X,up,X,T1,T2,T2).
cntl(X,down,X,T1,T2,T2).
```

Once these executable specifications of the train, controller, and the gate is obtained, we need to compose them together, and give the semantics of the complete system. The semantics of the complete system is given as a list of $\langle \epsilon, \tau \rangle$ pair, where ϵ is an event and τ the time that event took place. This composition of the three systems is done in the specification below:

```
driver([],S0,S1,S2,T,T0,T1,T2,[]).
driver([X|S],S0,S1,S2,T,T0,T1,T2,[(X,T)|R]):-
   train(S0, X, S00, T, T0, T00),
   gate(S1, X, S10, T, T1, T10) ,
   cntl(S2, X, S20, T, T2, T20) , TA > T,
   driver(S,S00,S10,S20,TA,T00,T10,T20,R).
```

Once the composed system has been specified, we have an executable specification of the composite real-time system. Note that an implementation of the composite system can also be automatically derived through suitable transformations. Note that if the event stream is non-terminating, then the driver will execute forever. The driver terminates when the event stream terminates.

The composite specification can be used for verifying various global properties. For example, we may want to verify the safety property that when the train is at the gate, the gate is always closed. These properties are specified by the designer, and ensure correctness of the real-time system specification. We use the axiomatic semantics based framework discussed earlier to perform this verification. We use pre-conditions to put restrictions on the event list to ensure finiteness, and then the properties of interest (post-conditions) can be verified. The net effect obtained is that of (deductive) model-checking.

Thus, our queries to the program will be of the form:

```
pre_condition(X),
driver(X, ...),
not post_condition(X)
```

where `post_condition` is the verification condition, while `pre_condition` is the condition imposed on the input to ensure finiteness. This query should fail, if the `post_condition` holds true. The `pre_condition` should be such that it generates all sentences that satisfy it. For example, if we want to check that the event `in` (train is in) never occurs before event `down` (gate is down), then the pre-condition is that the `in` or the `down` events must occur between two `approach` events. This pre-condition can be expressed so that it can act as a generator of all possible strings that start with an `approach` and end in `approach`. Integrating this pre-condition in the driver we get:

```
driver(_,[],_,_,_,_,_,_,_,[]) :-
driver(N,[X|S],S0,S1,S2,T,T0,T1,T2,[(X,T)|R]) :-
     train(S0,X,S00,T,T0,T00),
     gate(S1,X,S10,T,T1,T10) ,
     contr(S2,X,S20,T,T2,T20) ,
     TA > T, (X = approach ->
                (N = 0 -> M = 1; Rest = []);
                 M = N),
     driver(M,S,S00,S10,S20,TA,T00,T10,T20,R).
```

The driver thus acts as a generator, generating all possible strings that begin with `approach` and end in `approach` and that will be accepted by the automata. Now a property can be verified by calling the driver with uninstantiated input, and checking that the negated property does not hold for every possible meaning of the automata. Suppose, we want to verify that when the train is at the crossing, the gate must be down. This boils down to the following fact: in every possible meaning of a single run of the real-time system, the event `down` must occur before the event `in`. The negated property will be that the event `in` occurs before `down`. Thus, the query:

```
?- driver(0,X,s0,s0,s0,0,0,0,0,R), append(A, [(down,_)|_], R),
          append(_, [(in,_)|_], A).
```

will fail when run on a constraint logic programming system (we used the CLP(R) [21] system). The `append` program is the standard logic program for appending two lists.

Likewise, if we want to verify that the gate will be down at least 4 units of time. That is, an up cannot follow down within 4 time units, then we pose the following query:

```
?- driver(0,s0,s0,s0,0,0,0,0,X,R), append(A, [f(up,T2) |_], R),
        append(_, [f(down,T1)|_], A), T2 - T1 < 4.
```

The above query will fail. Using our system one can also find out the minimum and the maximum amount of time the gate will be closed by posing the following query:

```
?- driver(0,s0,s0,s0,0,0,0,0,X,R), append(A, [f(up,T2) |_], R),
     append(_, [f(down,T1)|_], A), N < T2 - T1 < M, M > 0, N > 0.
```

We obtain the answer `M < 7, N > 1`. This tells us that the minimum time the gate will be down is 1 units, and the maximum time it will be down is 7 units. Other properties of the real-time system can similarly be tested. *The ability to compute values of unknowns is a distinct advantage of a logic programming based approach and is absent from most other approaches used for verifying real-time systems.*

A major weakness of our approach is that we have to make sure that the denotation of a program is finite. If it is not, then we have to impose pre-conditions that ensure this. A popular approach to verifying an infinite state system is to *abstract* it (so that it becomes finite) while making sure that enough information remains in the abstraction so that the property of interest can be verified. The technique of abstraction can be easily adapted in our logical denotational approach: (i) one can give an abstract (logical denotational) semantics for the language, and then run tests on the resulting abstract denotation obtained, using the approach described above. (ii) we can use abstract interpretation tools built for logic programming to abstract the concrete denotation and use that for verifying the properties; in fact, work is in progress in our group to use non-failure analysis of constraint logic programs [4] to verify properties of real-time systems (see also Section 7.2).

5 Specification, Implementation and Verification of DSL Programs

In this section we discuss how our logical, denotational approach can be applied to specification, implementation, and verification of software systems, especially to programs written in Domain Specific Languages. The intuition for using logical denotational semantics is the following: a software system that interacts with its outside world can be thought of as a system that processes programs written in its input language. An executable denotational specification of this input language is essentially an executable specification of the software system. Thus, if we use logical denotational semantics for writing the semantics of the input language of a software system, then indeed we obtain an executable specification of the software system. Many large software systems define some kind of input language that has to be used to interact with them and to use them. Such languages are called *domain specific languages* [31,9], and have been the focus of much attention recently. Using our approach, giving the semantics of a DSL, essentially, gives an executable specification of the software system implementing the DSL. What is more, efficient implementations can be derived for this DSL, by choosing the right abstraction for the semantic algebra, and partially evaluating a DSL program to operations in the semantic algebra. Finally, properties of the software system can be automatically verified, by adding preconditions and postconditions to the denotation of a DSL program and executing them, in the way described in the previous section. We illustrate this application of our logical denotational semantics through an example: we consider the input language (a DSL) of a file-editor, and show how giving the logical denotational semantics of this DSL yields an executable specification of the file editor, and how we can verify certain properties that a file-editor should satisfy (e.g., modifying one file doesn't change other files).

```
program(session(I,S)) --> [edit], id(I),
      [cr], sequence(S).
sequence(seq(quit)) --> [quit].
sequence(seq(C,S)) -->
      command(C), [cr], sequence(S).
command(command(newfile)) --> [newfile].
command(command(forward)) --> [moveforward].
command(command(backward)) --> [moveback].
command(command(insert(R))) -->
      [insert],record(R).
command(command(delete)) --> [delete].
id(identifier(a)) --> [a].
id(identifier(b)) --> [b].
id(identifier(c)) --> [c].
record(rec(0)) --> [0].
....
record(rec(9)) --> [9].
```

Fig. 9: DCG for the Command Language

Consider a simple file editor which supports commands for opening a file, inserting and deleting records, moving forward and backward in the file, and closing the file. A specification for this editor can be given by giving a denotational semantics of its input command language. The editor supports the following commands: edit I (open the file whose name is in identifier I), newfile (create an empty file), forward (move file pointer to next record), backward (move file pointer to previous record), insert(R) (insert record whose value is in identifier R), delete (delete the current record), and quit (quit the editor, saving the file in the file system). The syntax of the editor language is given as a DCG in Figure 9. Our example is adapted from [38]. The DCG is slightly extended, so that it produces a parse tree during the parsing process. For simplicity we assume that there are only three possible file names a, b, and c, and that the records inserted consist of single digit numbers. Note that cr stands for a carriage return, inserted between each editor command. The above DCG specification when loaded in a Prolog system, automatically produces a parser.

We next give the semantic algebras (Figure 10) for each of the domains involved: the file store (represented as an association list of file names and their contents) and an open file (represented as a pair of lists; the file pointer is assumed to be currently at the first record of the second list). The semantic algebra essentially defines the basic operations used by the semantic valuation functions for giving meanings of programs.

```
%Define Access and Update Operations
access(Id,[(I,File)|_],File).
access(Id,[(I,File)|Rest],File1) :-
        (I = Id -> File1 = File;
                access(Id,Rest,File1)).
update(Id,File,[],[(Id,File)]).
update(Id,File,[(Id,_)|T],[(Id,File)|T]).
update(Id,File,[(I1,F1)|T],[(I1,F2)|NT]) :-
       (Id=I1 --> F2 = File, NT = T;
            F2 = F1, update(Id,File,T,NT).
%Operations on Open File representation
newfile(([],[])).
copyin(File,([],File)).
copyout((First,Second),File):-
         reverse(First,RevFirst),
         append(RevFirst,Second,File).
forwards((First,[X|Scnd]),([X|First],Scnd)).
forwards((First,[]),(First,[])).
backwards(([X|First],Scnd),(First,[X|Scnd])).
backwards(([],Scnd),([],Scnd)).
insert(A,(First,[]),(First,[A])).
insert(A,(First,[X|Y]),([X|First],[A|Y])).
delete((First,[_|Y]),(First,Y)).
delete((First,[]),(First,[])).
at_first_record(([],_)).
at_last_record((_,[])).
isempty(([],[])).
```

Fig. 10: Semantic Algebra for the File System

The semantic valuation predicates that give the meaning of each construct in the language are given next (Figure 11). These semantic functions are mappings from parse-tree patterns and a global state (the file system) to domains (file system, open files) that are used to describe meanings of programs. The above specification gives both the declarative and operational semantics of the editor, and as discussed earlier, can serve both as an implementation as well as be used for verification.

```
prog_val(session(identifier(I),S),FSIn,FSOut) :-
    access(I,FSIn,File), copyin(File,OpenFile),
    seq_val(S,OpenFile,NewOpenFile),
    copyout(NewOpenFile,OutFile),
    update(I,OutFile,FSIn,FSOut).
seq_val(seq(quit),InFile,InFile).
seq_val(seq(C,S),InFile,OutFile) :-
                comm_val(C,InFile,NewFile),
                seq_val(S,NewFile,OutFile).
comm_val(command(newfile),_,OutFile) :-
    newfile(OutFile).
comm_val(command(moveforward),InFile,OutFile) :-
    (isempty(InFile) -> OutFile = InFile;
     (at_last_record(InFile) -> OutFile=InFile;
                     forwards(InFile,OutFile))).
comm_val(command(moveback),InFile,OutFile) :-
    (isempty(InFile) -> InFile = OutFile;
         (at_first_record(InFile) ->
    InFile = OutFile; backwards(InFile,OutFile))).
comm_val(command(insert(R)),InFile,OutFile) :-
    record_val(R,RV), insert(RV,InFile,OutFile).
comm_val(command(delete),InFile,OutFile) :-
    (isempty(InFile) ->
     InFile = OutFile; delete(InFile,OutFile)).
record_val(R,R).
```

Fig. 11: Valuation Predicates

Using a logic programming system, the above specification can serve as an interpreter for the command language of the editor, and hence serves as an implementation of the editor. Thus, the above semantic definition is an executable specification of an editor. Although editors are interactive programs, for simplicity, we assume that the commands are given in batches (interactive programs can also be handled by modeling rest of the unknown commands through Prolog's unbound variables: we omit the discussion to keep the presentation simple). Thus, if the editor is invoked and a sequence of commands issued, then assuming that we start with an unspecified file system (modeled as the unbound variable, `Fin`), the resulting file system after executing all the editor commands can be obtained by posing the query:

```
?- Comms = [edit,a,cr,newfile,cr,insert,1,cr,insert,2,cr,delete,
   cr,moveback,cr,insert,4,cr,insert,5,cr,delete,cr,quit]),
   program(Tree,Comms,[]),     %produce parse tree
   prog_val(Tree,Fin,Fout).  %execute commands
```

The final resulting file-system will be:

```
Fout = [(a,[rec(1),rec(4)])| _B ].
```

The output shows that the final file systems contains the file `a` that contains 2 records, and the previously unknown input file system (represented by Prolog's anonymous variable `_B`, aliased to `Fin`). The key thing to note is that in our logical denotational framework, a specification is very easy to write as well as easy to modify. This is because of the declarative nature of the logic programming formalism used and its basis in denotational semantics. It is also easy to verify the specification, thanks to use of Horn Logic.
Given the executable implementation of the file-editor, and a program in its command language (a DSL), we can partially evaluate it to obtain a more efficient implementation of the program. The result of partially evaluating the file-editor specification, w.r.t. previous command-language program, is shown in Figure 12. Partial evaluation translates the editor command language program to a series of instructions that call operations defined in the semantic algebra. This series of instructions look a lot like "compiled" code. More efficient implementations of the editor can be obtained by implementing these semantic algebra operations in an efficient way, e.g., using a more efficient language, e.g., C, C++, or Java, instead of using logic programming.

```
access(a, Fin, C),
copyin(C, _),
newfile(D),
insert(rec(1), D, E),
insert(rec(2), E, F),
( isempty(F) -> G=F
; delete(F, G)),
( isempty(G) -> H=G
; at_first_record(G) ->
       H=G
; backwards(G, H)),
insert(rec(4), H, I),
insert(rec(5), I, J),
( isempty(J) -> K=J
; delete(J, K)),
copyout(K, L),
update(a,L,Fin,Fout).
```

Fig. 12: Compiled code

Given that our specification is executable, we can run it to check its soundness and to test that it meets our expectations. Consider the specification of the file editor. Under the assumption that the file system is finite and that the pool of possible records is also finite, we can verify, for instance, that every editing session consisting of an insertion followed by a deletion leaves the original file unchanged, by posing the following query:

```
?- true,        %Null pre-condition
   program(A,[edit,X,cr,insert,Y,cr,delete,cr,quit],[]),
   prog_val(A,F,G),
   F ≠ G.       %negated post-condition
```

The above query states that for all possible files `X` and for all possible records `Y` that can be inserted, the resulting file-system is different from the original file system, and this query should fail, if indeed the result of one insertion and one deletion leaves the file system unchanged. The backtracking mechanism of logic programming goes through all possible values for variables `X` and `Y` (finiteness is hence important and was coded in the specification by limiting the range of file names and record values; alternatively, this finiteness could be enforced in the precondition) and finds that in every case `F = G` holds, and thus finally the whole query fails because the final call asserts that F $\neq$ G. More complex inferences are possible on this semantic definition. For example, we can verify that if an editing session is open for a certain file, then this editing session will affect no other file [15].

6 Semantic Porting

Logical denotations also provide a formal theory of *porting* or *language filtering*. Porting can involve migrating a software from one machine/operating system to another (e.g. moving an application from a Sun Sparc running Solaris to a PC running Linux), or migrating a system written in one notation to another on the same machine (e.g. porting a database written in Oracle SQL to IBM DB2). It is the latter type of porting, that we are interested in. Essentially, this type of porting involves development of a semantic translation system to translate programs written in a language $\mathcal{L}_s$ to another language $\mathcal{L}_t$ (such a system is called a *filter*). Specification of a filter can be seen as an exercise in semantics. Essentially, the meaning or semantics of the language $\mathcal{L}_s$ can be given in terms of the constructs of the language $\mathcal{L}_t$. This meaning consists of both syntax and semantic specifications. If these syntax and semantic specifications are executable, then the specification itself acts as a translation system, providing a provably correct filter. The task of specifying the filter from $\mathcal{L}_s$ to $\mathcal{L}_t$ consists of specifying the DCG grammar for $\mathcal{L}_s$ and the appropriate valuation predicates which are essentially maps from parse tree patterns of $\mathcal{L}_s$ to parse tree patterns of $\mathcal{L}_t$.

Consider the translation of *Braille Nemeth Math* [28], a Braille-based language notation for the blind, to LaTeX. To obtain such a translator one can specify the semantics of the Braille Nemeth Math notation (a formal language) in terms of LaTeX. (such a translator can help the sighted instructors in grading the assignment/papers of blind students since these students write answers to their homework in Nemeth Braille which the instructors can't read). We next give the specification of a filter for translating Braille Nemeth Math code used for expressing *polynomials with polynomial powers* to LaTeX. Polynomial powers means that each variable can be raised to another polynomial: for example, $x^{x^2+1}+5$ (note that in Nemeth Braille this will be coded as `x^x^^2^+1"+5`; the number of `^` indicates the exponent level and `"` the base line expression, hence the Nemeth Braille expression language is not context free). Assume, for simplicity, that the only variable names allowed are `x`, `y`, and `z`. Consider the grammar for such polynomials that also produces parse trees. We first give the syntax of Nemeth Braille for polynomials with polynomial powers as a DCG, then we give the semantics of the parse trees produced in terms of LaTeX. The resulting specification is executable and provides the filter.

```
SYNTAX:
exp(e(X)) --> term(X).
exp(e(T,O,E)) --> term(T), op1(O), exp(E).
term(t(X)) --> digit(X).
term(t(X)) --> vari(X).
term(t(V,H,T)) --> vari(V), hats(H), term(T).
op1(op(H,+)) --> hats(H), [+].
op1(op('"',+)) --> ['"'], [+].
op1(op(+)) --> [+].
hats([^]) --> [^].
hats([^|L]) --> [^], hats(L).
vari(x) --> [x].  vari(y) --> [y].  vari(z) --> [z].
digit(0) --> [0].  digit(1) --> [1].  digit(2) --> [2].
digit(3) --> [3].  digit(4) --> [4].  digit(5) --> [5].
digit(6) --> [6].  digit(7) --> [7].  digit(8) --> [8].
digit(9) --> [9].

SEMANTICS:
sexp(e(X),INL,L) :- sterm(X,INL,ONL,L1),
                        getlist(ONL, Bl), append(L1,Bl,L).
sexp(e(T,O,E),NL,L) :- sterm(T,NL,OL, L1),
           sop(O,OL,NL1), Close_Braces is OL - NL1,
           getlist(Close_Braces, Bl), sexp(E,NL1,L2),
           append(L1,Bl,Lp), append(Lp,[+|L2],L).
sop(op(H,+),_,Ct) :- shats(H,Ct).
sop(op('"',+),_,0).
sop(op(+),CL,CL).
shats([^],1).
shats([^|L],Ct) :- shats(L,Ct1), Ct is Ct1+1.

sterm(t(X),OL,OL,[X]).
sterm(t(V,H,T),Ct, OL, [V,^,'{'|R]) :- Ct1 is Ct+1,
                       shats(H,Ct1), sterm(T,Ct1,OL,R).
getlist(0,[]).
getlist(N,['}'|L]) :- N > 0, N1 is N-1, getlist(N1,L).
```

The grammar for Braille Nemeth code is given as a DCG. Expressing it as a DCG results in a parser immediately. The parser also produces the parse trees for the Braille Nemeth code. The denotations (meaning) of Braille Nemeth code parse tree patterns are expressed as logic programs which are maps from Nemeth math parse trees and the global state (the exponent level) to LaTeX math (the LaTeX expression is assembled as a list of symbols; it will perhaps be better to produce a term representing the LaTeX expression). The resulting program acts as an automatic translation system from Braille Nemeth code to LaTeX.

Thus, if we load this program on a logic programming system and pose the query for translating `x^x^^2^+5"+6` Braille Nemeth code to LaTeX, the answer obtained is shown below:

```
| ?- exp(T,[x,^,x,^,^,2,^,+,5,'"',+,6],[]),sexp(T,0,L).

L = [x,^,'{',x,^,'{',2,'}',+,5,'}',+,6],
T = e(t(x,[^],t(x,[^,^],t(2))),op([^],+),
                        e(t(5),op('"',+),e(t(6)))) ?
```

The answer in `L` corresponds to the LaTeX expression `x^{x^{2}+5}+6` (i.e., $x^{x^2+5}+6$).

Note that semantic filtering gives us another way of building provably correct compilers: a compiler is a semantic filter from a source language to a target machine language [35]. Another application of the logical denotational semantics based semantic porting is in *interoperability* of databases: both the query programs and data defined under one type of database can be translated to programs and data of another different database.

7 Other Applications

7.1 Derivation of Abstract Machines

Our approach can also be used to derive and experiment with different abstract machines for a given language. This is based on the observation that compilation using partial evaluation can be done to whatever degree we wish. We could first define our semantics at a very high level by defining the semantic algebras at a very high level. Partial evaluation can then be used to compile programs to this abstract machine. This semantics could be further refined, by giving a finer level semantics for this high level semantic algebra. The program compiled to this higher level algebra, can be partially evaluated further with respect to the interpreter obtained for this lower level semantics, to obtain an even lower level compiled code. This can be repeated further (i.e., still finer semantics can be defined). For example, consider the definition of the Prolog language itself: the semantics of the language can be defined in such a way, that on partial evaluation WAM [47] like compiled code is obtained. However, we could further refine this semantics, so that, instead, on partial evaluation native code is obtained. This is indeed under investigation, and our approach is being applied to Prolog to automatically generate compilers that compile to both WAM as well as native code [16].

7.2 Specification of Abstract Interpreters

So far we have been specifying concrete semantics in our logical denotation. The semantics specified can also be abstract. The resulting logic program acts as an abstract interpreter. Program properties can be then be verified using the pre-condition/post-condition approach described earlier. To obtain abstract semantics one has to only specify abstractions for semantic algebras. Recall that a semantic algebra consists of a set along with operations that apply to the elements of this set. An abstract semantic algebra can be obtained by abstracting this set as well as the concrete operations on this set. The abstraction function should be properly chosen in accordance to the criterion laid out in [10]. Once the semantic algebra has been abstracted, the valuation function should be appropriately modified as well, essentially replacing the calls to concrete semantic algebra functions by abstract ones. If the abstracted semantic algebras are finite then one can guarantee that the abstract semantics of the language in question is also finite. This follows from the fact that valuation functions are maps from parse trees and semantic algebras to semantic algebras. If the semantic algebras are finite, then the co-domains of the valuation functions are also finite. From this it follows that given a program `P` written in the language being studied, the fixpoint of `P`'s abstract denotation will be finite. The abstract denotation can be used to verify properties of programs. We omit details due to lack of space [17].

This denotational semantics based approach to abstract interpretation gives a computational interpretation to observations such as those made by Schmidt, namely, that data-flow analysis is model-checking of the abstract interpreter [41]. The preceding ideas are related to automatic generation of abstract interpreters for Prolog [30] (though our approach can be applied to any arbitrary language).

Furthermore, given a program to be analyzed and the denotational abstract semantics of the language the program is written in, then the abstract interpreter can be partially evaluated w.r.t. program. The checking of program properties can be done on this *abstract compiled code*. This approach provides a formal frameworks for efforts such as those where abstract compilers were developed to make the process of abstract interpretation based analysis faster [43]. It also automates the process of deriving abstract compiled code from the language specification.

We hope that our work on generating abstract interpreters from abstract semantics will spur research in use of abstract interpretation for program verification (most use so far has been for static analysis in order to obtain faster compiled code).

7.3 Using Existing Logic Programming Technology

Once the denotation of a program is obtained as a logic program, all the analysis tools that have been developed for logic programming can be applied to analyze the properties of the logical denotation. For example, abstract interpretation based analyzers can be applied to the compiled program to check if two operations are independent [2]. Program termination analyzers [24] can be applied to check if the program will terminate. Non-failure analysis

[4] can be applied to check for non-failure of the logical denotation, etc. At present, analysis tools developed for constraint logic programming [2] are being applied to analyze the denotations of Fortran programs to detect parallelism (as an alternative to work presented in [13]), while non-failure analysis is being used to aid in verification of real-time systems.

8 Related Work

Work on programming language semantics has been ongoing for decades. However, use of programming language semantics for practical applications has been very limited, as Schmidt notes [40]. Most work in practical use of semantics has been in automatic derivation of sequential compilers. Many systems have been developed for automatically generating interpreters and compilers from semantic specification [27,46,11,29,22,5]. However, because the syntax is specified as a (non-executable) BNF and semantics is specified in the λ-calculus, this automatic generation process is very cumbersome. The lack of "calculational clarity" in the traditional denotational semantics notation is the main reason why application of semantics based approaches to practical problems has been hindered. In contrast, our logical denotation approach imparts the much needed "calculational view." In fact, this simple change of approach, we believe, can have considerable impact on applications and use of programming language semantics. Most of the applications of denotational semantics outlined in this paper (automatic generation of model checkers, language filters, sequential compilers, parallelizing compilers, abstract interpreters, etc.) are possible in the existing frameworks found in the literature, however, they are too cumbersome to realize and hence not explored.

Using Horn logic for expressing denotational semantics has been considered by Stepney [35], however, she does not realize the immense potential that this simple switch of notations brings about. Stepney is solely interested in obtaining provably correct compilers. However, this provably correct compiler is specified by the user in her approach and not automatically generated as in our framework. In Stepney's approach, the compiler is specified by giving the meanings of program constructs in terms of machine instructions (i.e., the valuation functions map parse trees to machine instructions). Consel has independently applied denotational semantics and partial evaluation to the specification and implementation of domain specific languages [9]. He does consider verification in the same work, but this verification has to be done manually via user supplied proofs. In contrast, in our framework, verification can be done automatically. Also, Fritzson has considered deriving data-parallel code from denotational specification of a language, but once again the procedures involved are quite complex and cumbersome [34]. Miller has proposed using operational semantics to derive efficient implementations [26], while we are interested in deriving abstract machines from denotational specifications. Satluri and Fleck [7] have considered using logic programs for specifying the semantic actions associated with production rules of an attribute grammar. They are mainly concerned with providing a Horn-logic based operational semantics, and do not consider declarative denotational semantics at all.

9 Conclusions

In this paper we presented a Horn logic and constraint based framework for denotational semantics. The switch to logic programming provides a "calculational flavor" to formal semantics, and leads to many interesting applications. The most interesting aspect of this framework is that denotational specification once written can be quickly debugged and verified as well as compiled code automatically obtained. Thus, denotational semantics becomes an experimental tool, that can be used by software writers and language designers to experiment with software design and language design respectively. At present, several practical applications of the ideas presented in the paper are being pursued [12,13,15,17,16,23] by the author's research group.

Acknowledgments

Thanks to Enrico Pontelli for his collaboration on several of the applications discussed in this paper, to Shameem Akhter and Enrico Pontelli for helping with coding of the examples, to Art Karshmer, Sandy Geiger, and Chris Weaver for bringing the Braille to LaTeX translation problem to my attention. Thanks to Pieter Hartel of Southampton University for bringing Stepney's work to my attention. Thanks also to students in my Programming Language Semantics class at New Mexico State on whom this logic programming approach to Denotational Semantics was first tried.

References

1. R. Alur and D. Dill. The Theory of Timed Automata. *Theoretical Computer Science*, 126, 1994.
2. M. Garcia de la Banda, M. Hermenegildo, et al. Global Analysis of Constraint Logic Programs. In *ACM Trans. on Prog. Languages and Systems*, Vol. 18, Num. 5, pages 564-615, ACM, 1996.
3. S. K. Das. Deductive Databases and Logic Programming. Addison-Wesley. 1992.
4. S. Debray, P. Lopez-Garcia, and M. Hermenegildo. Non-failure Analysis for Logic Programs. In *International Conference on Logic Programming*. MIT Press, 1997.
5. D.F. Brown et al. ACTRESS: an action semantics directed compiler generator. In *Proc. 4th Int'l Conf. on Compiler Construction*. Springer LNCS 641, pp. 95-109. 1992.

6. W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs, *In JACM 43(1):20-74.*
7. S. Satluri and A. C. Fleck. Semantic Specification using Logic Programs. In *Proc. N. American Conf. on Logic Programming* 1989, MIT Press. pp. 772-791.
8. E. M. Clark, E. A. Emerson, A. P. Sistla. Automatic Verification of finite-state Concurrent Systems Using Temporal Logic Specification. In *ACM TOPLAS*, 8(2), 1986.
9. C. Consel. Architecturing Software Using a Methodology for Language Development. In *Proc. 10th Int'l Symp. on Prog. Lang. Impl., Logics and Programs (PLILP)*, Sep. 1998, Springer LNCS 1490, pp. 170-194.
10. P. Cousot, R. Cousot, "Abstract Interpretation: A Unified Model for Static Analysis of Programs for Construction or Approximation of Fix-points," In *Conference Record of the 4th ACM POPL,* pp. 238-252, 1977.
11. Thierry Despeyroux. Executable specification of static semantics. In *Semantics of Data Types*, Springer LNCS 173. pp. 215-234. 1984.
12. G. Gupta, E. Pontelli. A Constraint-based Denotational Approach to Specification and Verification of Real-time Systems. In *Proc. IEEE Real-time Systems Symposium*, San Francisco, pp. 230-239. Dec. 1997.
13. G. Gupta, E. Pontelli, R. Felix-Cardenas, A. Lara, Automatic Derivation of a Provably Correct Parallelizing Compiler, *In Proceedings of International Conference on Parallel Processing*, IEEE Press, Aug, 1998, pp. 579-586.
14. G. Gupta. Horn Logic Denotations and Their Applications. Internal Memo. Jan 1995.
15. G. Gupta, E. Pontelli. Specification, Verification, Composition, and Interoperation of Complex Software Systems: A Logical Denotational Approach. Technical Report. New Mexico State University. Dec. 1997.
16. G. Gupta, E. Pontelli, H. Guo, L. Zhu. Automatic Generation of a WAM-compiler and a Native Code Compiler for Prolog. New Mexico State University. Working paper.
17. G. Gupta, E. Pontelli. Horn Logical Denotational Framework for Abstract Interpretation. New Mexico State University. Working paper.
18. C. Gunter. Programming Language Semantics. MIT Press. 1992.
19. C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. of the ACM.* Vol. 12. pp. 576-580, 1969.
20. N. Jones. Introduction to Partial Evaluation. In *ACM Computing Surveys.* 28(3):480-503.
21. J. L. Lassez and J. Jaffar. Constraint logic programming. In *Proc. 14th ACM POPL*, 1987.
22. P. Lee. Realistic Compiler Generation. The MIT Press, Cambridge, MA, 1989.
23. A. Karshmer, G. Gupta, S. Geiger, C. Weaver. A Framework for Translation of Braille Nemeth Math to LaTeX: The MAVIS Project. In *Proc. ACM Conference on Assistive Technologies*, ACM Press, pp. 136-143, Mar. 1998.
24. N. Lindenstrauss, Y. Sagiv. Automatic Termination Analysis for Logic Programs. In *Proc. International Conference on Logic Programming*, 1997. pp. 63-77.
25. J.W. Lloyd. Foundations of Logic Programming. Springer Verlag. 2nd ed. 1987.
26. J. Hannan and D. Miller. From Operational Semantics to Abstract Machines. In *Proc. Lisp and Functional Programming Conference*, 1990.
27. P.D. Mosses. Compiler Generation using Denotational Semantics. In *Math. Foundations of Computer Science*, Springer LNCS 45, pages 436-441, 1976.

28. A. Nemeth. The Braille-Nemeth Math Code. American Printing House for the Blind. 1972 Revision. Louisville, Kentucky.
29. F. Nielson and H. R. Nielson. Two level semantics and code generation. *Theoretical Computer Science*, 56(1):59-133. 1988.
30. C. R. Ramakrishnan, S. Dawson, and D. Warren. Practical Program Analysis Using General Purpose Logic Programming Systems: A Case Study. In *Proc. ACM Conf. on Programming Language Design and Implementation*. 1996.
31. C. Ramming. Editor, *Proceedings of the Usenix Conference on Domain-Specific Languages*, October 1997, Santa Barbara, California, USA.
32. M. Raskovsky, Phil Collier. From Standard to Implementational Denotational Semantics. In *Semantics Directed Compiler Generation*. Lecture Notes in Computer Science 94. Springer Verlag. pp. 94-139.
33. Efficient Model Checking using Tabled Resolution. Y.S. Ramakrishnan, C.R. Ramakrishnan, I.V. Ramakrishnan et al. In *Proceedings of Computer Aided Verification (CAV'97)*. 1997.
34. J. Ringstrom, P. Fritzson, M. Pettersson. Generating an Efficient Compiler for a Data Parallel Language from a Denotational Specification. In Lecture Notes in Computer Science, Vol. 786, pp. 248-260. 1994.
35. S. Stepney. High Integrity Compilation. Prentice Hall. 1993.
36. D. Sahlin. An Automatic Partial Evaluator for Full Prolog. Ph.D. Thesis. 1994. Royal Institute of Technology, Sweden. (Software available from www.sics.se)
37. K. Sagonas, T. Swift, and D. S. Warren. XSB as an efficient deductive database engine. In *Proc. SIGMOD International Conf. on Management of Data*, 1994.
38. D. Schmidt. *Denotational Semantics: a Methodology for Language Development*. W.C. Brown Publishers, 1986.
39. D. Schmidt. Programming language semantics. In CRC Handbook of Computer Science, Allen Tucker, ed., CRC Press, Boca Raton, FL, 1996. Summary version, ACM Computing Surveys 28-1 (1996) 265-267.
40. D. Schmidt. On the Need for a Popular Formal Semantics. *Proc. ACM Conf. on Strategic Directions in Computing Research,* Cambridge, MA, June 1996. ACM SIGPLAN Notices 32-1 (1997) 115-116.
41. D. Schmidt. *Dataflow Analysis is Model Checking of an Abstract Interpreter*. In *Proc. ACM POPL'98*.
42. L. Sterling & S. Shapiro. The Art of Prolog. MIT Press, '94.
43. J. Tan and I-P. Lin. Compiling Dataflow Analysis of Logic Programs. In *Proc. ACM Conf. on Programming Language Design and Implementation*. SIGPLAN Notices, 27(7), 1992.
44. P. Van Hentenryck. *Constraint Handling in Prolog*. MIT Press, 1988.
45. S. Abiteboul, R. Hull, V. Vianu. Foundation of Databases. Addison-Wesley, 1995.
46. M. Wand. Semantics-directed Machine Architecture. In *ACM POPL*. pp. 234-241. 1982
47. D.H.D. Warren. An Abstract Instruction Set for Prolog. Tech. Note 309, SRI Int'l, '83.
48. D.H.D. Warren. Higher Order Extensions to Prolog: Are They Needed? *Machine Intell.*, 10:441-454.
49. D.H.D. Warren. Logic Programming for Compiler-writing. *Software Practice and Experience*, 10, pp. 97-125. 1979.
50. H. Zima, B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, '91.