Towards Provably Correct Code Generation via Horn Logical Continuation Semantics*

Qian Wang[1], Gopal Gupta[1], and Michael Leuschel[2]

[1] Department of Computer Science
University of Texas at Dallas, Richardson, TX 75083-0688 USA
Email: {qxw015000,gupta}@utdallas.edu
[2] Department of Electronics and Computer Science
University of Southampton, Southampton, UK S017 1BJ
Email: mal@ecs.soton.ac.uk

Abstract. Provably correct compilation is an important aspect in development of high assurance software systems. In this paper we explore approaches to provably correct code generation based on programming language semantics, particularly *Horn logical semantics*, and partial evaluation. We show that the *definite clause grammar (DCG)* notation can be used for specifying both the syntax and semantics of imperative languages. We next show that continuation semantics can also be expressed in the Horn logical framework.

1 Introduction

Ensuring the correctness of the compilation process is an important consideration in construction of reliable software. If the compiler generates code that is not faithful to the original program code of a system, then all our efforts spent in proving the correctness of the system could be futile. Proving that target code is correct w.r.t. the program source is especially important for high assurance systems, as unfaithful target code can lead to loss of life and/or property. Considerable research has been done in this area, starting from the work of McCarthy [18]. Most efforts directed at proving compiler correctness fall into three categories:

– Those that treat the compiler as just another program and use standard verification techniques to manually or semi-automatically establish its correctness (e.g., [3]). However, even with semi-automation this is a very labour intensive and expensive undertaking, which has to be repeated for every new language, or if the compiler is changed.

* The authors have been partially supported by NSF grants CCR 9900320, INT 9904063, and EIA 0130847, by the Department of Education and the Environmental Protection Agency, and by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-38059 ASAP project.

– Those that *generate* the compiler automatically from the mathematical semantics of the language. Typically the semantics used is denotational (see for example Chapter 10 of [23]). The automatically generated compilers, however, have not been used in practice due to their slowness and/or inefficiency/poor quality of the code generated.
– Those that use program transformation systems to transform source code into target code [16, 20]. The disadvantage in this approach is that specifying the compiler operationally can be quite a lengthy process. Also, the compilation time can be quite large.

In [6] we developed an approach for generating code for imperative languages in a provably correct manner based on partial evaluation and a type of semantics called *Horn logical semantics*. This approach is similar in spirit to semantics-based approaches, however, its basis is Horn-logical semantics [6] which possesses both an operational as well as a denotational (declarative) flavor. In the Horn logical semantics approach, both the syntax and semantics of a language is specified using Horn logic statements (or pure Prolog).

Taking an operational view, one immediately obtains an interpreter of the language $\mathcal{L}$ from the Horn-logical semantic description of the language $\mathcal{L}$. The semantics can be viewed dually as operational or denotational. Given a program $\mathcal{P}$ written in language $\mathcal{L}$, the interpreter obtained for $\mathcal{L}$ can be used to execute the program. Moreover, given a partial evaluator for pure Prolog, the interpreter can be *partially evaluated* w.r.t. the program $\mathcal{P}$ to obtain compiled code for $\mathcal{P}$. Since the compiled code is obtained automatically via partial evaluation of the interpreter, it is faithful to the source of $\mathcal{P}$, provided the partial evaluator is correct. The correctness of the partial evaluator, however, has to be proven only once. The correctness of the code generation process for *any* language can be certified, provided the compiled code is obtained via partial evaluation. Given that efficient execution engines have been developed for Horn Logic (pure Prolog), partial evaluation is relatively fast. Also, the declarative nature of the Horn logical semantics allows for language semantics to be rapidly obtained.

In this paper, we further develop our approach and show that in Horn logical semantics not only the syntax but also the semantics can be expressed using the definite clause grammar notation. The semantics expressed in the DCG notation allows for the store argument to be naturally (syntactically) hidden. We show that continuation semantics can also be expressed in Horn logic. Continuation semantics model the semantics of imperative constructs such as *goto statements*, *exception handling mechanisms, abort*, and *catch/throw constructs* more naturally. We also show that continuation semantics expressed as DCGs can be partially evaluated w.r.t. a source program to obtain "good quality" target code.

In this work we use partial evaluation to generate target code. Partial evaluation is especially useful when applied to interpreters; in this setting the static input is typically the object program being interpreted, while the actual call to the object program is dynamic. Partial evaluation can then produce a more efficient, specialized version of the interpreter, which can be viewed as a compiled version of the object program [5].

In our work we have used the LOGEN system [14]. Much like MIXTUS, LOGEN can handle many non-declarative aspects of Prolog. LOGEN also supports *partially static* data by allowing the user to declare custom "binding types." More details on the LOGEN system can be found elsewhere [14]. Unlike MIXTUS, LOGEN is a so-called *offline* partial evaluator, i.e., specialization is divided into two phases: (i) A *binding-time analysis* (*BTA* for short) phase which, given a program and an approximation of the input available for specialization, approximates all values within the program and generates annotations that steer (or control) the specialization process. (ii) A (simplified) *specialization phase*, which is guided by the result of the *BTA*.

Because of the preliminary BTA, the specialization process itself can be performed very efficiently, with predictable results (which is important for our application). Moreover, due to its simplicity it is much easier to establish correctness of the specialization process.

Finally, while our work is motivated by provably correct code generation, we believe our approach to be useful to develop "ordinary" compilers for domain specific languages in general [8].

2 Horn Logical Semantics

The denotational semantics of a language $\mathcal{L}$ has three components: (i) *syntax specification:* maps sentences of $\mathcal{L}$ to parse trees; it is commonly specified as a grammar in the BNF format; (ii) *semantic algebra:* represents the mathematical objects whose elements are used for expressing the meaning of a program written in the language $\mathcal{L}$; these mathematical objects typically are sets or domains (partially ordered sets, lattices, etc.) along with associated operations to manipulate the elements of the sets; (iii) *valuation functions:* these are functions mapping parse trees to elements of the semantic algebras.

Traditional denotational definitions express syntax as BNF grammars, and the semantic algebras and valuation functions using λ-calculus. In Horn Logical semantics, Horn-clauses (or pure Prolog) and constraints[1] are used instead to specify all the components of the denotational semantics of programming languages [6]. There are three major advantages of using Horn clauses and constraints for coding denotational semantics.

First, the syntax specification *trivially and naturally* yields an executable parser. The BNF specification of a language $\mathcal{L}$ can be quite easily transformed to a *Definite Clause Grammar* (DCG) [24]. The syntax specification[2] written in the DCG notation serves as a parser for $\mathcal{L}$. This parser can be used to parse

[1] Constraints may be used, for example, to specify semantics of languages for real-time systems [7].

[2] A grammar coded as a DCG is syntax specification in the sense that various operational semantics of logic programming (standard Prolog order, tabled execution, etc.) can be used for execution during actual parsing. Different operational semantics will result in different parsing algorithms (e.g., Prolog in recursive descent parsing with backtracking, tabled execution in chart parsing, etc.).

programs written in $\mathcal{L}$ and obtain their parse trees (or syntax trees). Thus, the syntactic BNF specification of a language is easily turned into *executable syntax* (i.e., a parser). Note that the syntax of even context sensitive languages can be specified using DCGs [6].

Second, the semantic algebra and valuation functions of $\mathcal{L}$ can also be coded in Horn-clause Logic. Since Horn-clause Logic or pure Prolog is a declarative programming notation, just like the λ-calculus, the mathematical properties of denotational semantics are preserved. Since both the syntax and semantic part of the denotational specification are expressed as logic programs, they are both executable. These syntax and semantic specifications can be loaded in a logic programming system and executed, given a program written in $\mathcal{L}$. This provides us with an interpreter for the language $\mathcal{L}$. In other words, the *denotation*[3] of a program written in $\mathcal{L}$ is executable. This executable denotation can also be used for many applications, including automated generation of compiled code.

Third, non-deterministic[4] semantics can be given to a language w.r.t. resources (e.g., time, space, battery power) consumed during execution. For example, some operations in the semantic algebra may be specified in multiple ways (say in software or in hardware) with each type of specification resulting in different resource consumption. Given a program and bounds on the resources that can be consumed, only some of the many possible semantics may be viable for that program. Resource bounded partial evaluation [2] can be used to formalize resource conscious compilation (e.g., energy aware compilation) [26] via Horn Logical semantics.

Horn-logical semantics can also be used for automatic verification and consistency checking [6, 7]. We do not elaborate any further since we are not concerned with verification in this paper.

The disadvantage of Horn logical semantics is that it is not denotational in the strict sense of the word because the semantics given for looping constructs is not compositional. The fix operator used to give compositional semantics of looping constructs in λ-calculus cannot be naturally coded in Horn logic due to lack of higher order functions. This, for example, precludes the use of structural induction to prove properties of programs. However, note that even though the semantics is not truly compositional, it is declarative, and thus the fix-point of the logic program representing the semantics can be computed via the standard T_P operator [17]. Structural/fix-point induction can then be performed over this T_P operator to prove properties of programs. Note that even in the traditional λ-calculus approach, the declarative meaning of the fix operator (defined as computing the limit of a series of functions) is given outside the operational framework of the λ-calculus, just as the computation of the $\mathrm{fix}(\mathrm{T}_P)$ in logic programming is outside the operational framework of Horn Clause logic. For partial evaluation, the operational definition of fix, i.e., $\mathrm{fix}(\mathrm{F}) = \mathrm{F}(\mathrm{fix}\ \mathrm{F})$, is used.

[3] We refer to the denotation of a program under the Horn-logical semantics as its *Horn logical denotation.*

[4] Non-deterministic in the logic programming sense.

In [6] we show how both the syntax and semantics of a simple imperative language (a simple subset of Pascal whose grammar is shown in Figure 1) can be given in Horn Logic. The Horn logical semantics, viewed operationally, automatically yields an interpreter. Given a program P, the interpreter can be partially evaluated w.r.t. P to obtain P's compiled code.

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

Fig. 1: BNF grammar

A program and its corresponding code generated via partial evaluation using the LOGEN system [14] is shown below. The specialization time is insignificant (i.e., less than 10 ms). Note that the semantics is written under the assumption that the program takes exactly two inputs (found in variables `x` and `y`) and produces exactly one output (placed in variable `z`). *The definitions of the semantic algebra operations are removed, so that unfolding during partial evaluation will stop when a semantic algebra operation is encountered.* The semantic algebra operations are also shown below.

```
z = 1;
w = x;
loop while w > 0
  z = z * y ;
  w = w - 1
end while.
```

```
main(A, B, C) :-
initialize_store(D),
update(x, A, D, E),
update(y, B, E, F),
update(z, 1, F, G),
access(x, G, H),
 update(w, H, G, I),
 while_eval__1(I, J),
 K=J,
 access(z, K, C).
```

```
while_eval__1(A, B) :-
  access(w, A, C),
  ( C>0 ->
     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),
      while_eval__1(J, B),
     ;   B=A ).
```

```
SEMANTIC ALGEBRA:
 initialize_store([(x,0),(y,0),(z,0),(w,0)]).
 access(Id,[(Id,Val)|_ ],Val).
 access(Id,[_|R],Val) :-
                access(Id,R,Val).
```

```
update(Id,NV,[(Id,_)|R],[(Id,NV)|R]).
update(Id,NewV,[P|R],[P|R1]) :-
                  update(Id,NewV,R,R1).
```

Notice that in the program that results from partial evaluation, only 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 partially evaluate to a *tail-recursive* program. These tail-recursive calls are easily converted to iterative structures using jumps in the target code.

Though the compiled code generated is in Prolog syntax, it looks a lot like machine code. A few simple transformation steps will produce actual machine code. These transformations include replacing variable names by register/memory locations, replacing a Prolog function call by a jump (using a goto) to the code

for that function, etc. The code generation process is provably correct, since target code is obtained automatically via partial evaluation. Of course, we need to ensure that the partial evaluator works correctly. However, this needs to be done only once. Note that once we prove the correctness of the partial evaluator, compiled code for programs written in any language can be generated as long as the Horn-logical semantics of the language is given.

It is easy to see that valuation predicate for an iterative structure will always be tail-recursive. This is because the operational meaning of a looping construct can be given by first iterating through the body of the loop once, and then recursively re-processing the loop after the state has been appropriately changed to reflect the new values of the loop control parameters. The valuation predicate for expressing this operational meaning will be inherently tail recursive.

Note also that if a predicate definition is tail recursive, a folding/unfolding based partial evaluation of the predicate will preserve its tail-recursiveness. This allows us to replace a tail recursive call with a simple jump while producing the final assembly code. The fact that tail-recursiveness is preserved follows from the fact that folding/unfolding based partial evaluation can be viewed as algebraic simplification, given the definitions of various predicates. Thus, given a tail recursive definition, the calls in its body will be expanded in-place during partial evaluation. Expanding a tail-recursive call will result in either the tail-recursion being eliminated or being replaced again by its definition. Since the original definition is tail-recursive, the unfolded definition will stay tail recursive. (A formal proof via structural induction can be given [25] but is omitted due to lack of space.)

3 Definite Clause Semantics

Note that in the code generated, the `update` and `access` operations are parameterized on the memory store (i.e., they take an input store and produce an output store). Of course, real machine instructions are not parameterized on store. This store parameter can be (syntactically) eliminated by using the DCG notation for expressing the valuation predicates as well.

All valuation predicates take a store argument as input, modify it per the semantics of the command under consideration and produce the modified store as output [6]. Because the semantic rules are stated declaratively, the store argument "weaves" through the semantic sub-predicates called in the rule. This suggests that we can express the semantic rules in the DCG notation. Thus, we can view the semantic rules as computing the *difference* between the output and the input stores. This difference reflects the effect of the command whose semantics is being given. Expressed in the DCG notation, the store argument is (syntactically) hidden away. For example, in the DCG notation the valuation predicate

```
command(comb(C1, C2), Store, Outstore) :-
          command(C1, Store, Nstore),
          command(C2, Nstore, Outstore).
```

is written as:

```
command(comb(C1, C2)) --> command(C1), command(C2).
```

In terms of difference structures, this rules states that the difference of stores produced by `C1; C2` is the "sum" of differences of stores produced by the command `C1` and `C2`. The rest of the semantic predicates can be rewritten in this DCG notation in a similar way.

```
main(U,V,A) -->
      update(x,U),
      update(y,V),
      update(z,1),
      access(x,F),
      update(w,F),
      while_eval__1,
      access(z,A).
while_eval__1 -->
     (access(w,C),
      {0<C} ->
         access(z,D),
         access(y,E),
         {F is D*E},
         update(z,F),
         access(w,H),
         {I is H-1},
         update(w,I),
         while_eval__1
       ; []).
```

```
main:            while:
 store x U        load w C
 store y V        skipgtz C
 store z 1        jump else
 load x F         load z D
 store w F        load y E
 jump while       mul D E F
end:              store z F
 load z W         load w H
                  sub1 H I
                  store w I
                  jump while
                 else:
                  noop
                  jump end
```

Fig. 2. Partially evaluated semantics and its assembly code

Expressed in the DCG notation, the semantic rules become more intuitively obvious. In fact, these rules have more natural reading; they can be read as simple rewrite rules. Additionally, now we can partially evaluate this DCG w.r.t. an input program, and obtain compiled code that has the store argument syntactically hidden. The result of partially evaluating this DCG-formatted semantics is shown to the left in Figure 2. Notice that the store argument weaving through the generated code shown in the original partially evaluated code is hidden away. Notice also that the basic operations (such as comparisons, arithmetic, etc.) that appear in the target code are placed in braces in definite clause semantics, so that the two store arguments are not added during expansion to Prolog. The constructs appearing within braces can be regarded as the "terminal" symbols in this semantic evaluation, similar to terminal symbols appearing in square brackets in the syntax specification. In fact, the operations enclosed within braces are the primitive operations left in the residual target code after partial evaluation. Note, however, that these braces can be eliminated by putting wrappers around the primitive operations; these wrappers will have two redundant store

arguments that are identical, per the requirements of the DCG notation. Note also that since the LOGEN partial evaluator is oblivious of the DCG notation, the final generated code was cast into the DCG notation manually.

Now that the store argument that was threading through the code has been eliminated, the access/update instructions can be replaced by load/store instructions, tail recursive call can be replaced by a jump, etc., to yield proper assembly code. The assembly code that results in shown to the right in figure 2. We assume that inputs will be found in registers `U` and `V`, and the output will be placed in register `W`. Note that `x, y, z, w` refer to the memory locations allocated for the respective variables. Uppercase letters denote registers. The instruction `load x Y` moves the value of memory location `x` into register `Y`, likewise `store x Y` moves the value of register Y in memory location `x` (on a modern microprocessor, both `load` and `store` will be replaced by the `mov` instruction); the instruction `jump label` performs an unconditional jump, `mul D E F` multiplies the operands `D` and `E` and puts the result in register `F`, `sub1 A B` subtracts 1 from register A and puts the result in register B, while `skipgtz C` instruction realizes a conditional expression (it checks if register `C` is greater than zero, and if so, skips the immediately following instruction).

Note that we have claimed the semantics (e.g., the one given in section 3) to be denotational. However, there are two problems: (i) First, we use the `(p->q;r)` construct of logic programming which has a hidden cut, which means that the semantics predicates are not even declarative. (ii) second, the semantics is not truly compositional, because the semantics of the while command is given in terms of the while command itself. This non-compositionality means that structural induction cannot be applied.

W.r.t. (i) note that the condition in the `->` always involves a relational operator with ground arguments (e.g., `Bval = true`). The negation of such relational expressions can always be computed and the clause expanded to eliminate the cut. Thus, a clause of the form

```
p(..) :- (Bval = true -> q(...); r(...))
```

can be re-written as

```
p(..) :- Bval = true, q(...).
p(..) :- Bval = false, r(...).
```

Note that this does not adversely affect the quality of code produced via partial evaluation.

W.r.t. (ii), as noted earlier, program properties can still be proved via structural induction on the T_P operator, where P represents the Horn logical semantic definition.

Another issue that needs to be addressed is the ease of proving a partial evaluator correct given that a partial evaluator such as LOGEN [14] or Mixtus [22] are complex pieces of software. However, as already mentioned, because of the offline approach the actual specialization phase of LOGEN is quite straightforward and should be much easier to prove correct. Also, because of the predictability

of the offline approach, it should also be possible to formally establish that the output of LOGEN corresponds to proper target code.[5]

Note that because partial evaluation is done until only the calls to the semantic algebra operation remain, the person defining the semantics can control the type of code generated by suitably defining the semantic algebra. Thus, for example, one can first define the semantics of a language in terms of semantic algebra operations that correspond to operations in an abstract machine. Abstract machine code for a program can be generated by partial evaluation w.r.t. this semantics. This code can be further refined by giving a lower level semantics for abstract machine code programs. Partial evaluation w.r.t. this lower level semantics will yield the lower level (native) code.

4 Continuation Semantics

So far we have modeled only direct semantics [23] using Horn logic. It is well known that direct semantics cannot naturally model exception mechanisms and `goto` statements of imperative programming languages. To express such constructs naturally, one has to resort to continuation semantics. We next show how continuation semantics can be naturally expressed in Horn Clause logics using the DCG notation. In the definite clause continuation semantics, semantics of constructs is given in terms of the *differences of parse trees* (i.e., difference of the input parse tree and the continuation's parse tree) [25]. Each semantic predicate thus relates an individual construct (difference of two parse trees) to a fragment of the store (difference of two stores). Thus, semantic rules are of the form:

```
command(C1, C2, Program, S1, S2) :- ...
```

where the difference of C1 and C2 (say ΔC) represents the command whose semantics is being given, and the difference of S1 and S2 represents the store which reflects the incremental change (ΔS) brought about to the store by the command ΔC. Note that the `Program` parameter is needed to carry the mapping between labels and the corresponding command. Each semantic rule thus is a stand alone rule relating the difference of command lists, ΔC, to difference of stores, ΔS. *If we view a program as a sequence of difference of command lists then its semantics can simply be obtained by "summing" the difference of stores for each command.* That is, if we view a program P as consisting of sequence of commands:

$P = \Delta C_1 + \Delta C_2 + \ldots + \Delta C_n$

then its semantics S is viewed as a "sum" of the corresponding differences of stores:

$S = \Delta S_1 \oplus \Delta S_2 \oplus \ldots \oplus \Delta S_n$

and the continuation semantics simply maps each ΔC_i to the corresponding ΔS_i. Note that $\oplus$ is a non-commutative operator, and its exact definition depends on how the store is modeled. Additionally, continuation semantics allow for cleaner,

[5] E.g., for looping constructs, the unfolding of the (tail) recursive call has to be done only once through the recursive call to obtain proper target code.

more intuitive declarative semantics for imperative constructs such as exceptions, catch/throw, goto, etc. [23].

Finally, note that the above continuation semantics rules can also be written in the DCG notation causing the arguments `S1` and `S2` to become syntactically hidden:

```
command(C1, C2, Program) --> ...
```

Below, we give the continuation semantics of the subset of Pascal considered earlier after extending it with statement labels and a `goto` statement. Note that the syntax trees are now represented as a list of commands. Each command is represented in the syntax tree as a pair, whose first element is a label (possibly null) and the second element is the command itself. Only the valuation functions for commands are shown (those for expressions, etc., are similar to the one shown earlier).

```
prog_eval([], _, _, 0) --> []
prog_eval(CommList, Val_x, Val_y, Output) -->
   update(x, Val_x), update(y, Val_y),
   command_eval(CommList,cont([],[]), CommList), access(z, Output).

command_eval([],[],_Program) --> [].
command_eval([],cont(CommList,Cont),Program)-->
   command_eval(CommList,Cont,Program).
command_eval([Comm|CommList],Cont,Program)-->
   comm_eval(Comm,CommList,Cont,NCommList,NCont,Program),
   command_eval(NCommList,NCont,Program).

comm_eval([(_,abort)|_],_Comm,_Cont,[],[],_Program) --> [].
comm_eval((Label,while(B,LoopBody)),OldRest,OldCont,[],[],Program)
     --> bool_while_eval(B,LoopBody,
               cont([(Label,while(B,LoopBody))|OldRest],
                        OldCont), OldRest,OldCont,Program).
comm_eval((_,ce(B,C1,C2)),OldRest,OldCont,[],[],Program) -->
   bool_eval(B,C1,cont(OldRest,OldCont),C2,cont(OldRest,OldCont),Program).
comm_eval((_,ce(B,C1)),OldRest,OldCont,[],[],Program) -->
   bool_eval(B,C1,cont(OldRest,OldCont),OldRest,OldCont,Program).
comm_eval((_,jmp(ID)),_OldRest,_OldCont,JumpList,cont([],[]),Program)-->
   {find_label(ID,Program,JumpList)}.
comm_eval((_,assign(id(I), E)),OldRest,OldCont,OldRest,OldCont,_Program)
    --> expr(E, Val), update(I, Val).

bool_while_eval(Cond,C1,C1Cont,C2,C2Cont,Program) -->
   bool_eval(Cond,C1,C1Cont,C2,C2Cont,Program).
bool_eval(greater(E1, E2),C1,C1Cont,C2,C2Cont,Program)
    --> expr(E1, Eval1), expr(E2, Eval2),
          ({Eval1 > Eval2} -> command_eval(C1,C1Cont,Program) ;
                              command_eval(C2,C2Cont,Program)).
  /*the code for lesser(E1,E2) and equal(E1,E2) is very similar*/
```

The code above is self-explanatory. Semantic predicates pass command continuations as arguments. The code for `find_label/3` predicate is not shown. It looks for the program segment that is a target of a `goto` and changes the current continuation to that part of the code.

Consider the program shown below to the left in Figure 3. In this program segment, control jumps from outside the loop to inside via the goto statement. The result of partially evaluating the interpreter (after removing the definitions of semantic algebra operations) obtained from the semantics w.r.t. this program (containing a `goto`) is shown in the figure 3 to the right. Figures 4 shows another instance of a program involving goto's and the code generated by the LOGEN partial evaluator by specialization of the definite clause continuation semantics shown above.

```
//source code
z = 1;
w = x;
goto label;
loop while w > 0
    z = z * y ;
    label: w = w - 1
endloop while;
z = 8;
z = 7.
```

```
//generated code
interpreter(A, B, C) -->
    update(x, A),
    update(y, B),
    update(z, 1),
    access(x, D),
    update(w, D),
    access(w, E),
    {F is E-1},
    update(w, F),
    fix1,
    access(z, C).
```

```
fix1 -->
    ( access(w, A),
        {0<A} ->
        access(z, B),
        access(y, C),
        {D is B*C},
        update(z, D),
        access(w, E),
        {F is E-1},
        update(w, F),
        fix1
    ;   update(z, 8),
        update(z, 7)
    ).
```

Fig. 3. Example with a jump from outside to inside a while loop

Note that a Horn logical continuation semantics can be given for any imperative language in such a way that its partial evaluation w.r.t. a program will yield target code in terms of access/update operation. This follows from the fact that programs written in imperative languages consist of a series of commands executed under a control that is explicitly supplied by the programmer. Control is required to be specified to a degree that the continuation of each command can be uniquely determined. Each command (possibly) modifies the store. Continuation semantics of a command is based on modeling the change brought about to the store by the continuation of this command. Looking at the structure of the continuation semantics shown above, one notes that programs are represented as lists of commands. The continuation of each command may be the (syntactically) next command or it might be some other command explicitly specified by a control construct (such as a goto or a loop). The continuation is modeled in the semantics explicitly, and can be explicitly set depending on the control construct. The semantics rule for each individual command computes

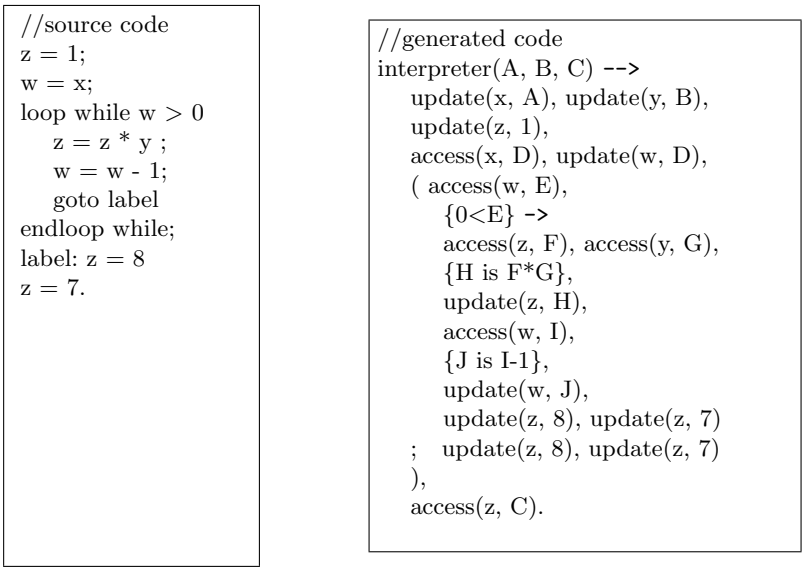

Fig. 4. Example with a jump from inside to outside a while loop

the changes made to the store as well as the new continuation. Thus, as long as the control of an imperative language is such that *the continuation of each command can be explicitly determined*, its Horn logical continuation semantics can be written in the DCG syntax. Further, since the semantics is executable, given a program written in the imperative language, it can be executed under this semantics. The execution can be viewed as unfolding the top-level call, until all goals are solved. If the definitions of the semantic algebra operations are removed, then the top-level call can be simplified via unfolding (partial evaluation) to a resolvent which only contains calls to the semantic algebra operations; this resolvent will correspond to the target code of the program.

It should also be noted that the LOGEN system allows users to control the partial evaluation process via annotations. Annotations are generated by the BTA and then can be modified manually. This feature of the LOGEN system gives considerable control of the partial evaluation process—and hence of the code generation process—to the user. The interpreter has to be annotated only once by the user, to ensure that good quality code will be generated.

5 A Case Study in SCR

We have applied our approach to a number of practical applications. These include generating code for parallelizing compilers in a provably correct manner [6], generating code for controllers specified in Ada [13] and for domain specific languages [8] in a provably correct manner, and most recently generating code in a provably correct manner for the Software Cost Reduction (SCR) framework.

The SCR (Software Cost Reduction) requirements method is a software development methodology introduced in the 80s [9] for engineering reliable software systems. The target domain for SCR is real-time embedded systems. SCR has been applied to a number of practical systems, including avionics system (the A-7 Operational flight Program), a submarine communication system, and safety-critical components of a nuclear power plant [10].

We have developed the Horn logical continuation semantics for the complete SCR language. This Horn logical semantics immediately provides us with an interpreter on which the program above can be executed. Further, the interpreter was partially evaluated and compiled code was obtained. The time taken to obtain compile code using definite clause continuation semantics of SCR was an order of magnitude faster than a program transformation based strategy described in [16] that uses the APTS system [20], and more than 40 times faster than a strategy that associates C code as attributes of parse tree nodes and synthesizes the overall code from it [16].

6 Related Work

Considerable work has been done on manually or semi-mechanically proving compilers correct. Most of these efforts are based on taking a specific compiler and showing its implementation to be correct. A number of tools (e.g., a theorem prover) may be used to semi-mechanize the proof. Example of such efforts range from McCarthy's work in 1967 [18] to more recent ones [3]. As mentioned earlier, these approaches are either manual or semi-mechanical, requiring human intervention, and therefore not completely reliable enough for engineering high-assurance systems. "Verifying Compilers" have also been considered as one of the grand challenge for computing research [11], although the emphasis in [11] is more on developing a compiler that can verify the assertions inserted in programs (of course, such a compiler has to be proven correct first).

Considerable work has also been done on generating compilers automatically from language semantics [23]. However, because the syntax is specified as a (non-executable) BNF and semantics is specified using λ-calculus, the automatic generation process is very cumbersome. The approach outlined in this paper falls in this class, except that it uses Horn logical semantics which, we believe and experience suggests, can be manipulated more efficiently. Also, because Horn logical semantics has more of an operational flavor, code generation via partial evaluation can be done quite efficiently.

Considerable work has also been done in using term rewriting systems for transforming source code to target code. In fact, this approach has been applied by researchers at NRL to automatically generate C code from SCR specification using the APTS [20] program transformation system. As noted earlier, the time taken is considerably more than in our approach. Other approaches that fall in this category include the HATS system [27] that use tree rewriting to accomplish transformations. Other transformation based approaches are mentioned in [16].

Recently, Pnueli et al have taken the approach of verifying a given run of the compiler rather than a compiler itself [21]. This removes the burden of maintaining the compiler's correctness proof; instead each run is proved correct by establishing a refinement relationship. However, this approach is limited to very simple languages. As the authors themselves mention, their approach "seems to work in all cases that the source and target programs each consist of a repeated execution of a single loop body ..," and as such is limited. For such simple languages, we believe that a Horn logical semantics based solution will perform much better and will be far easier to develop. Development of the refinement relation is also not a trivial task. For general programs and general languages, it is unlikely that the approach will work.

Note that considerable work has been done in partially evaluating meta-interpreters for declarative languages, in order to eliminate the interpretation overhead (see, for example, [19, 1]). However, in this paper our goal is to generate assembly-like target code for imperative languages.

7 Conclusions

In this paper we presented an approach based on formal semantics, Horn logic, and partial evaluation for obtaining provably correct compiled code. We showed that not only the syntax specification, but also the semantic specification can be coded in the DCG notation. We also showed that continuation semantics of an imperative language can also be coded in Horn clause logic. We applied our approach to a real world language—the SCR language for specifying real-time embedded system. The complete syntax and semantic specification for SCR was developed and used for automatically generating code for SCR specifications. Our method produces executable code considerably faster than other transformation based methods for automatically generating code for SCR specifications.

Acknowledgments

We are grateful to Constance Heitmeyer and Elizabeth Leonard of the Naval Research Labs for providing us with the BNF grammar of SCR and the safety injection program as well as for discussions, and to the anonymous referees.

References

1. A. Brogi and S. Contiero. Specializing Meta-Level Compositions of Logic Programs. Proceedings LOPSTR'96, J. Gallagher. Springer-Verlag, LNCS 1207.
2. S. Debray. Resource bounded partial evaluation. PEPM 1997. pp. 179-192.
3. A. Dold, T. Gaul, W. Zimmermann Mechanized Verification of Compiler Backends Proc. Software Tools for Technology Transfer, Denmark, 1998.
4. S. R. Faulk. State Determination in Hard-Embedded Systems. Ph.D. Thesis, Univ. of NC, Chapel Hill, NC, 1989.

5. Y. Futamura. Partial Evaluation of Computer Programs: An approach to compiler-compiler. *J. Inst. Electronics and Comm. Engineers, Japan*. 1971.
6. G. Gupta "Horn Logic Denotations and Their Applications," *The Logic Programming Paradigm: A 25 year perspective*. Springer Verlag. 1999:127-160.
7. G. Gupta, E. Pontelli. A Constraint-based Denotational Approach to Specification and Verification of Real-time Systems. In *Proc. IEEE Real-time Systems Symposium*, pp. 230-239. Dec. 1997.
8. G. Gupta, E. Pontelli. A Logic Programming Framework for Specification and Implementation of Domain Specific Languages. In *Essays in Honor of Robert Kowalski*, 2003, Springer Verlag LNAI,
9. K. L. Henninger. Specifying software requirements for complex systems: New techniques and their application. IEEE Trans. on Software Engg. 5(1):2-13.
10. C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated Consistency Checking of Requirements Specifications. ACM TOSEM 5(3). 1996.
11. C. A. R. Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. J.ACM, 50(1):63-69. Jan 2003.
12. N. Jones. Introduction to Partial Evaluation. In *ACM Computing Surveys*. 28(3):480-503.
13. L. King, G. Gupta, E. Pontelli. Verification of BART Controller. In *High Integrity Software*, Kluwer Academic, 2001.
14. M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline specialisation in Prolog using a hand-written compiler generator. *Theory and Practice of Logic Programming*, 4(1):139–191, 2004.
15. M. Leuschel, B. Martens, and D. De Schreye. Controlling Generalization and Polyvariance in Partial Deduction of Normal Logic Programs. ACM Transactions on Programming Languages and Systems (TOPLAS), 20(1):208-258.
16. E. I. Leonard and C. L. Heitmeyer. Program Synthesis from Requirements Specifications Using APTS. Kluwer Academic Publishers, 2002.
17. J. Lloyd. Foundations of Logic Programming (2nd ed). Springer Verlag. 1987.
18. J. McCarthy and J. Painter. Correctness of a Compiler for Arithmetic Expressions. MIT AI Lab Memo, 1967.
19. S. Owen. Issues in the Partial Evaluation of Meta-Interpreters. Proceedings Meta'88. MIT Press. pp. 319–339. 1989.
20. R. Paige. Viewing a Program Transformation System at Work. *Proc. Programming Language Implementation and Logic Programming*, Springer, LNCS 844. 1994.
21. A. Pnueli, M. Siegel, E. Singerman. Translation Validation. *Proc TACAS'98*, Springer Verlag LNCS, 1998.
22. D. Sahlin. An Automatic Partial Evaluator for Full Prolog. Ph.D. Thesis. 1994. Royal Institute of Tech., Sweden. (available at www.sics.se)
23. D. Schmidt. *Denotational Semantics: a Methodology for Language Development*. W.C. Brown Publishers, 1986.
24. L. Sterling & S. Shapiro. The Art of Prolog. MIT Press, '94.
25. Q. Wang, G. Gupta, M. Leuschel. Horn Logical Continuation Semantics. UT Dallas Technical Report. 2004.
26. Q. Wang, G. Gupta. Resource Bounded Compilation via Constrained Partial Evaluation. UTD Technical Report. Forthcoming.
27. V. L. Winter. Program Transformation in HATS. *Software Transformation Systems Workshop*, '99.