Automatic Generation of Provably Correct Parallelizing Compilers

Gopal Gupta, Enrico Pontelli, Amado Lara-Rodriguez, Roberto Felix-Cardenas
Laboratory for Logic, Databases, and Advanced Programming
New Mexico State University
Las Cruces, NM 88003, USA
{gupta,epontell,arodrigu,rfelix-c}@cs.nmsu.edu;
http://www.cs.nmsu.edu/lldap

Abstract

We show how parallelizing compilers can be automatically derived from denotational definitions of programming languages. In our approach, the denotational definition is expressed using *definite clause grammars* (syntax specification) and *Horn Logic* or *Constraint Logic* (semantic specification). The conditions for executing two or more statements in parallel (e.g., GCD test, Banerjee test, or exact test) are included as part of the (parallel) denotational semantics of the language. Solutions of diophantine equations, needed for parallelizing DO loops, can be expressed in constraint logic as well, and are thus easily incorporated in our denotational framework. This parallel denotational specification of the language is executable, and thus automatically yields a parallel interpreter. This interpreter can be *partially evaluated* w.r.t. a given program to obtain parallel compiled code. In addition, the various syntactic and semantic restructuring transformations that have been proposed to expose more parallelism in sequential programs can also be expressed in our denotational framework. Partial evaluation of a program w.r.t. this interpreter will result in parallel code that is specialized to a given architecture. Automatic derivation of these parallelizing compilers becomes possible because of use of Horn Logic, instead of the traditional λ-calculus, for expressing denotational semantics. The major advantage of our approach is that the parallelizing compilers derived are provably correct, since they are automatically generated from language specifications.
Keywords: Parallelizing Compiler, Denotational Semantics, Constraint Logic Programming, Partial Evaluation, Verification.

1 Introduction

In this paper we present a denotational semantics based framework for deriving parallelizing compilers from language specifications. Denotational semantics is a well-established methodology for language design and specification [18, 19, 6]. We automatically derive parallel compiled code from the specification of a language (the language considered in this project is Fortran, though any imperative language can be used). Essentially, we obtain a parallelizing compiler automatically. The compiler thus obtained is provably correct. Two statements in a program can be parallelized if they are *independent*. We assume that the definition of independence is made part of the language specification, i.e., the notion of independence employed is part of the semantic specification of the language. Using *Horn logic* (or *Constraint-logic*) based denotational semantics [5], we are able to derive parallel code from this extended semantics through partial evaluation. This becomes possible because the use of Horn/Constraint logic renders *semantic specification executable* [5].

Traditionally, in imperative languages such as Fortran, parallelism is extracted by executing the DO loops in parallel. Different iterations of the DO loop can be executed in parallel if certain independence conditions are satisfied. These independence conditions boil down to non-existence of solutions of diophantine equations [25, 24]. These diophantine equations arise from equating array references. The diophantine equations can be seen as constraints over the index variables of the DO loops, and thus are expressible and solvable in

Constraint-logic. Thus the framework of Horn/Constraint-logic is powerful enough to express semantics of computations as well as for capturing the notions of independence by solving these equations using exact or approximate techniques.

Using the implementation technology developed for solving constraints [22, 7], independence tests such as the I test [16] are also automatically incorporated. More advanced tests can also be constructed on the fly during program execution. An additional, advantage of our approach is that constraint solving needed to detect independence can be done in parallel using a parallel constraint logic programming system [14, 23]. Partial evaluators for Horn Logic are written in Horn Logic languages (i.e., logic programming languages) themselves. It is well known that parallelism can be automatically exploited from logic programs [14, 3]. Thus, the partial evaluation to obtain the parallel code can be done in parallel itself. In effect, *this means that compilation of the program to parallel code is done in parallel.* So even if the exact test is used to solve the diophantine equations that arise, parallelism exploited should keep the time for compilation quite reasonable.

Our denotational approach also provides a semantic framework for restructuring transformations. Restructuring transformations are essentially semantics preserving transformations that are applied to programs. These transformations can be expressed as mappings from one semantic domain to another, and are expressible in the denotational framework. Recall that denotational semantics expresses semantics of a language as maps from parse trees to values in the *semantic algebra* domains. These mappings provide a powerful enough framework for expressing these transformations.

Our Horn/Constraint logic based denotational framework can be put to further use. The problem of scheduling parallel tasks on a given architecture is NP complete. Constraint logic is a good framework for specifying and solving such problems (powerful heuristics are automatically applied to considerably narrow the search space whenever possible). Given that our denotational specification is based on a constraint logic framework, the parallel architecture on which the program is to run can be included as part of the semantic specification. Parallel code tailored to the particular architecture can then be automatically generated from this specification through partial evaluation.

Our work makes several contributions: (i) it shows how constraint logic can be usefully employed to solve many problems encountered in building parallelizing compilers; (ii) it presents a denotational semantics based framework which can be used for specifying parallel computations, restructuring loop transformations, as well as task scheduling for a given architecture; (iii) it shows how provably correct parallelizing compilers can be derived from the semantic specification of a language.

The rest of the paper is organized as follows: Section 2 presents introduction to denotational semantics, and Horn/Constraint Logic based denotational semantics. Section 3 shows how a logical denotational specification can be used to automatically derive provably correct code. Section 4 shows how the I test gets incorporated in our denotational definition through the use of finite domain constraint solving. Section 6 discusses other applications of our framework. We assume that the reader is familiar with denotational semantics (e.g., [18]), parallelizing compilers (e.g., [24]), and constraint logic programming (e.g., [22]). In the rest of the paper we will use the terms Horn Logic and Constraint Logic interchangeably. By Horn Logic we mean Logic Programming (or pure Prolog). Constraint Logic is a generalization of Horn Logic which includes constraints and constraint solving [7].

2 Denotational Semantics and Horn Logic Denotations

Denotational semantics [18, 19] of a language has three components:

- syntax: typically 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 the global state (which itself is modeled as a semantic algebra) 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 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 are then 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 convoluted way. Thus, the syntax and semantics component are specified in different (perhaps non-executable) notations, inhibiting useful applications of the specification.

These disadvantages disappear if the formalism of logic programming (with its formal basis in Horn logic, a subset of predicate logic) is used. 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, termed Definite Clause Grammars, is well known [21]). The semantic algebras and valuation functions are also expressed in Horn logic programming quite easily, as relations subsume functions (and relational programming or logic programming subsumes functional programming). The semantic specification obtained using Horn Login has many other uses, e.g., for verification [5, 4].

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 breadth-first strategy (e.g., *tabling-based,* as in XSB Prolog [1]) 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.

Thus, by switching to Horn Logic, *the denotational specification of a language also becomes its operational specification.* An interpreter for the language can be obtained for free. Consider $\mathcal{LD}$, the logical denotation of a program $\mathcal{P}$ written in language $\mathcal{L}$. The bottom-up semantics of $\mathcal{LD}$, gives the declarative (denotational) semantics of the program $\mathcal{P}$, while the top-down SLD resolution semantics [11] of $\mathcal{LD}$ gives its operational semantics (i.e., an interpreter for $\mathcal{L}$). Using a suitable engine (e.g., a tabling-based one [1]), the declarative semantics of the program $\mathcal{P}$ can be computed as a set of input-output pairs.

Consider a very simple subset of Fortran 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 as:

```
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
```

This BNF is easily transformed into a definite clause grammar (DCG) by a simple change in syntax [21] (plus removal of left-recursion, required for recursive descent parsing). The DCG is a logic program, and when executed, a parser is automatically obtained. Additionally, an extra argument has been added in which the parse-tree is synthesized. This DCG is shown below:

```
SYNTAX:
program(p(X)) --> command(X), [.].

command(X) --> command1(X).
command(comb(X,Y)) --> command1(X),[;],command(Y).
command1(assign(I,E)) -->  id(I),[=],expression(E).
command1(ce(X,Y,Z)) -->    [if],expression(X),[then],command(Y),[else],command(Z), [endif].
command1(while(B, C))  --> [loop, while], bool(B), command(C), [endloop, while].

expression1(id(X)) -->     id(X).
expression1(num(X)) -->    n(X).
expression1(e(X)) -->      ['('],expression(X),[')'].
expression(X) -->          expression1(X).
expression(add(X,Y)) -->   expression1(X),[+],expression1(Y).
expression(sub(X,Y)) -->   expression1(X),[-],expression1(Y).
expression(multi(X,Y)) --> expression1(X),[*],expression1(Y).

bool(equal(X,Y)) -->       expression(X),[=],expression(Y).
bool(greater(X,Y)) -->     expression(X),[>],expression(Y).
bool(less(X,Y)) -->       expression(X),[<],expression(Y).

id(x) --> [w].      id(x) --> [x].     id(y) --> [y].      id(z) --> [z].
n(0) --> [0].       n(1) --> [1].      n(2) --> [2].       n(3) --> [3].       n(4) --> [4].
n(5) --> [5].       n(6) --> [6].      n(7) --> [7].       n(8) --> [8].       n(9) --> [9].
```

The above grammar specification is a logic program, which serves as a parser for the language being described. Thus, the query to parse the program for computing the value of y^x and placing it in the 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:

```
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 placed in the variable `z`.

The semantic algebra consists simply of the `store` domain, realized as an association list of the form `[(Identifier, 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).     %if not initialized, return 0
    access(Id, [(Id,Val)|_], Val).
    access(Id, [_|R], Val) :- access(Id,R,Val).
    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 below; the very first valuation predicate takes the two input values, that are placed in the store locations corresponding to `x` and `y`.

```
SEMANTICS: VALUATION PREDICATES
prog_eval(p(Comm), Val_x, Val_y, Output) :-
        initialize_store(Store),
        update(x, Val_x, Store, Midstore),  update(y, Val_y, Midstore, Nextstore),
        command(Comm, Nextstore, Newstore),
        access(z, Newstore, Output).
command(comb(C1, C2), Store, Outstore) :-
        command(C1, Store, Newstore),  command(C2, Newstore, Outstore).
command(while(B, C), Store, Outstore) :-
        (boolean(B, Store) -> command(C, Store, Newstore),
                              command(while(B, C), Newstore, Outstore);
                              Outstore=Store).
command(ce(B, C1, C2), Store, Outstore) :-
        (boolean(B, Store) -> command(C1, Store, Outstore);
                              command(C2, Store, Outstore)).
command(assign(I, E), Store, Outstore) :-
        expression(E, Store, Val), update(I, Val, Store, Outstore).
expression(add(E1, E2), Store, Result) :- expression(E1, Store, Val_E1),
                                     expression(E2, Store, Val_E2), Result is Val_E1+Val_E2.
expression(sub(E1, E2), Store, Result) :- expression(E1, Store, Val_E1),
                                     expression(E2, Store, Val_E2), Result is Val_E1-Val_E2.
expression(multi(E1, E2), Store, Result) :- expression(E1, Store, Val_E1),
                                     expression(E2, Store, Val_E2), Result is Val_E1*Val_E2.
expression(id(X), Store, Result) :-  access(X, Store, Result).
expression(num(X), _, X).
boolean(greater(E1, E2), Store) :-   expression(E1, Store, Eval1),
                expression(E2, Store, Eval2), Eval1 > Eval2.
boolean(less(E1, E2), Store) :-      expression(E1, Store, Eval1),
                                     expression(E2, Store, Eval2), Eval1 < Eval2.
boolean(equal(E1, E2), Store) :-     expression(E1, Store, Eval),
                                     expression(E2, Store, Eval).
```

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:

```
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).
```

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. Thus, the Horn Logic approach gives us a quick and effortless way of obtaining the interpreter for a language from its denotational specification.

In the semantics above, we assumed that the store is maintained as an association list, 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 [21]), 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:

```
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.

3 Provably Correct Compilation

It is well known in partial evaluation [8] 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 the Swedish Institute of Computer Science [17], the program that results after partially evaluating the query `?- main(5,2,A)` from the previous section (using SEMANTIC ALGEBRA 1) is the following:

```
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),
        commandwhilegreateridw1(G,H),
        access(z,H,A).
commandwhilegreateridw1(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),
                    commandwhilegreateridw1(J,B)
            ;       B=A
        ).
```

Note the similarity of this resulting program 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, partially evaluates to a *tail-recursive* program. These tail-recursive calls are easily converted to iterative structures using jumps. Partial evaluation of the while-loop will always generate tail-recursive programs (chiefly because the recursive specification of the semantics of the while-loop itself is tail-recursive and the way partial evaluators work). 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 [18] (using `assert` and `retract` discussed above). Thus, compiled machine code is just one or two very simple transformation steps away. Thus, provably correct compiled code is trivially obtained. Note that we can be confident of the correctness of the compiled code only if we can prove that the partial evaluator is correct; the correctness of the partial evaluator, however, can be proved once and for all.

3.1 Derivation of a Parallelizing Compiler

A denotational definition of a language $\mathcal{L}$ is a complete specification of the syntax and semantics of $\mathcal{L}$. A parallel semantics can also be given to the language, within the same framework. For instance, in Fortran programs, parallelism is mainly exploited in the DO loops. Iterations of a DO loop can be executed in parallel only if certain conditions are satisfied. These conditions essentially boil down to checking if linear diophantine equations do not have integer solutions within the range of loop limits [25, 24]. These conditions (the non-existence of integer solutions to the diophantine equations) can be made part of the semantics of the DO loops. These conditions will be evaluated during partial evaluation. During this evaluation, if no solutions for these diophantine equations are found, then parallel code will be generated. If solutions are indeed found, then code for parallelizing dependent iterations is generated. We give an example next.

Consider a slightly larger subset of Fortran with declarations, DO loops, and (single dimensional) arrays. The grammar of this subset can be straightforwardly coded as the DCG below (details such as structure of identifiers, etc., are omitted for brevity):

```
program(prog(O,X))  -->
                  [program], [cr],
                    specification_part(O), execution_part(X),
                  [end], [program], [cr].
specification_part(spec_part(DC)) -->
                  declaration_construct_list(DC).

declaration_construct_list(DC) -->
                  type_declaration_stmt(DC), [cr].
declaration_construct_list((DC,DCL)) -->
                  type_declaration_stmt(DC), [cr],
                  declaration_construct_list(DCL).
declaration_construct_list([]).
type_declaration_stmt(type_decl(TS,[EDL])) -->
                  type_spec(TS),
                  simple_identifier(EDL).
type_declaration_stmt(type_decl(TS,dimension(AS),[EDL])) --> type_spec(TS), [dimension], ['('],
                                                 array_spec(AS), [')'], simple_identifier(EDL).
type_spec(type_spec(integer)) --> [integer].
array_spec(array_spec(ASL)) --> array_spec_list(ASL).
array_spec_list(PN) -->         positive_number(PN).
array_spec_list((PN,ASL)) -->   numeral(PN), [','], array_spec_list(ASL).

execution_part(exec_part(EPC)) --> execution_part_construct(EPC).
execution_part_construct(exec_part_const(EPL)) --> execution_part_list(EPL).
execution_part_list(AS) -->       action_stmt(AS), [cr].
execution_part_list([]).
execution_part_list((AS,EPL)) --> action_stmt(AS), [cr], execution_part_list(EPL).

action_stmt(do(I,Z1,Z2,EPC)) -->
                  [do],
                  simple_identifier(I), [=], expression(Z1), [','], expression(Z2), [cr],
                  execution_part_construct(EPC), [enddo].
action_stmt(assign(I,E)) --> identifier(I), [=], expression(E).

expression_list([X|R]) -->   expression(X),[','],expression_list(R).
expression_list([E]) -->     expression(E).
identifier_list([]).
identifier_list([I|R]) -->   identifier(I),[','],identifier_list(R).
expression(E) --> expr4(E).
expr4(E) --> expr5(E).
expr4(plus(E1,E2)) --> expr5(E1),['+'],expr4(E2).
expr5(E) --> expr6(E).
expr5(times(E1,E2)) --> expr6(E1),['*'],expr5(E2).
expr6(E) --> expr7(E).
expr6(E) --> ['('],expr4(E),[')'].
expr7(V) --> numeral(V); identifier(V).

identifier(I) -->  array_identifier(I); simple_identifier(I).
array_identifier(array(I,EL)) --> simple_identifier(ident(I)), ['('], expression_list(EL), [')'].
```

The (parallel) semantics can be next given. For each grammar rule in the Definite Clause Grammar, there is one semantic valuation function. These semantic functions are predicated on parse trees and the global state and compute the meaning of the program in terms of domains defined in the semantic algebra. Before we define the semantic valuation function, let us define the semantic algebras needed for the parallel semantics. Mainly, we need the semantic algebra for the store, similar to semantic algebra 2 in example above. Note that a global store is very important for parallel programs, since parallel tasks may access the store concurrently.

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

update_array(Name,E,V) :- retractall(store(Name,E,_)), assert(store(Name,E,V)).
access_array(Name,E,V) :- (store(Name,E,V) -> true; V = 0).
```

Next we define the various semantic functions. These semantic functions specify the meaning of each construct in the language. The most significant is the (parallel) semantics of the DO loop construct and is discussed next. Code for some of the valuation predicates have been omitted for simplicity.

```
program(prog(Spec,Exec)) :-
           process_declarations(Spec), execute_commands(Exec).
execute_commands(exec_part(exec_part_const(Seq))) :-
           process_commands(Seq,[]).
process_commands((C1,C2),S) :-
           command(C1,S), process_commands(C2,S).
process_commands(C,S) :-
           command(C,S).

command(assign(I,E),S) :-
           expr_val(E,V1,S), update_location(I,V1,S).
command(do(ident(I),E1,E2,exec_part_const(Seq)),S) :-
           expr_val(E1,V1,S), expr_val(E2,V2,S), collect_exp(Seq,Left,Right),
           create_equations(Left,Right,Eqs), solutions(Eqs,Sols), execute_body(V1,V2,Seq,Sols,I,S).

/*Begin: auxiliary functions used by semantic function of DO*/
execute_body(Start,End,Seq,Sols,I,S) :-
         ( Start > End -> true;
                   par_begin(task1,task2),
                   task1(task1,Sols,Start,Seq,I,S), task2(task2,End,Start,Seq,Sols,I,S),
                   par_end).

task1(T1,Sols,Value,Seq,I,S) :-
                   start_task(T1), enforce_deps_big(Sols,Value),
                   process_commands(Seq,[(I,Value)|S]), enforce_deps_small(Sols,Value),
                   end_task(T1).
task2(T2,End,Start,Seq,Sols,I,S) :-
                   start_task(T2),
                   Start1 is Start + 1, execute_body(Start1,End,Seq,Sols,I,S),
                   end_task(T2).

enforce_deps_big([],_).
enforce_deps_big([(D1,D2)|Rest],Value) :-
            max(D1,D2,D3), min(D1,D2,D4),
            (D3 = D4 -> true;
                        (Value = D3 -> wait(D4); true)),
            enforce_deps_big(Rest,Value).
enforce_deps_small([],_).
enforce_deps_small([(D1,D2)|Rest],Value) :-
```

```
            min(D1,D2,D3), max(D1,D2,D4),
            (D3 = D4 -> true;
                        (Value = D3 -> signal(D4); true)),
            enforce_deps_small(Rest,Value).
/*end: auxiliary functions used by semantic function of DO*/

/*begin: auxiliary function used by assign command*/
update_location(array(Name,[Express]),Value,S) :-
  expr_val(Express,E1,S), update_array(Name,E1,Value).
update_location(J,Value,_) :-
  (J = ident(I) -> update(I,Value);
   J = array(_,_) -> fail; update(J,Value)).
/*end: auxiliary function used by assign command*/

expr_val(plus(E1,E2),Val,S) :-
  expr_val(E1,V1,S), expr_val(E2,V2,S), Val is V1+V2.
expr_val(times(E1,E2),Val,S) :-
  expr_val(E1,V1,S), expr_val(E2,V2,S), Val is V1*V2.
expr_val(integer(N),N,_).
expr_val(array(Name,[Arg]),Val,S) :-
           expr_val(Arg,Pos,S), access_array(Name,Pos,Val).
expr_val(ident(Name),Val,S) :-
           (find_local(Name,Val,S) -> true; access(Name,Val)).

find_local(Name,Val,[(Name1,Val2)|R]) :-
           (Name =  Name1 -> Val = Val2; find_local(Name,Val,R)).
```

Consider the valuation predicate for the DO loop, shown below:

```
command(do(ident(I),E1,E2,exec_part_const(Seq)),S) :-
  expr_val(E1,V1,S), expr_val(E2,V2,S),
  collect_exp(Seq,Left,Right), create_equations(Left,Right,Eqs), solutions(Eqs,Sols),
  execute_body(V1,V2,Seq,Sols,I,S).
```

The first argument is the parse tree for the DO loop construct. The parse tree contains the index variable (`ident(I)`), the starting value (`E1`), ending value (`E2`), and the parse tree of the loop body (`Seq`). For simplicity we assume in the semantics that the increment to the index variable is always 1. The start value expression and the end value expressions are evaluated. A local store is maintained by each iteration, where the value of the index variable is kept. This is needed since all iterations are being executed in parallel. The final value of the index variable should be copied into the main store at the end of the DO loop execution, however, for simplicity in our semantics we discard the local stores at the end of the execution of the DO loop. The body of the DO loop is examined, and the array subscript expressions are collected (`collect_exp` goal). The subscript expressions are equated to generate all the diophantine equations that need to be solved to check for independence of loops (`create_equations`). These diophantine equations are then solved (in the goal `solutions`) and the list of solutions is computed in `Sols`. If there are no solutions, this list is empty, otherwise it contains pairs of the form `(n, m)`. Each pair is a solution to one of the diophantine equations. If `(n, m)` is a solution to the equations, then it means that iteration `n` must be executed before iteration `m`. This list is passed to the predicate `execute_body` that specifies the parallel semantics of the DO loop. Note that in our case we assume that the `solutions` predicate solves the diophantine equation exactly, taking the lower and upper limits of the loop into account. We could use any independence test that we choose. For instance, we could employ the GCD test, an approximate test, if we wish to be conservative in exploiting parallelism. Likewise, Banerjee test, or any other desired approximate or exact test could be used. The `execute_body` function is reproduced below.

```
execute_body(Start,End,Seq,Sols,I,S) :-
```

```
  ( Start > End -> true;
                   par_begin(task1,task2),
                   task1(task1,Sols,Start,Seq,I,S), task2(task2,End,Start,Seq,Sols,I,S),
                   par_end).
task1(T1,Sols,Value,Seq,I,S) :-
                   start_task(T1),
                   enforce_deps_big(Sols,Value),
                   process_commands(Seq,[(I,Value)|S]),
                   enforce_deps_small(Sols,Value),
                   end_task(T1).
task2(T2,End,Start,Seq,Sols,I,S) :-
                   start_task(T2),
                   Start1 is Start + 1, execute_body(Start1,End,Seq,Sols,I,S),
                   end_task(T2).
```

The `execute_body` function is a recursive function whose arguments are the starting value, ending value, the parse tree of the body of the DO loop, the (possibly empty) list of solutions to the diophantine equations, the current value of the index variable, and the local store to be used by that iteration. The last argument is the output store produced by the body of the DO loop. If the value of the index variable exceeds the end value expression, `execute_body` terminates. Otherwise, the execution of the current iteration is begun in parallel with rest of the iterations after updating the index variable in the local store of the current iteration with the new value. Maintaining a local store where the current value of the index variable for that particular iteration is stored is, we believe, the best way to model the actual execution semantics of the parallel DO loop. Appropriate `wait` and `signal` operations should be called, given the solutions of the diophantine equations. If `(n, m)` is present as a solution, then iteration `n` at the end of its execution should call `signal(m)`, indicating that iteration `m` can now proceed. Likewise, iteration `m` should call the `wait(n)` instruction before it begins execution to ensure that execution of iteration `n` is over. This is exactly what is specified in the `enforce_deps_small` and `enforce_deps_bigs` predicates. Parallelism is indicated using the `parbegin/parend` construct. The tasks to be executed in parallel inside a `parbegin/parend` parallel construct are identified by the `start_task/end_task` construct. Consider the following simple program below (the program is represented as a list of tokens; `cr` stands for carriage return):

```
example([program,cr,
 integer,length,',',width,',',area,cr,
 integer,i,cr,
 integer,dimension,'(',100,')',arr,cr,
 length,=,12,cr,
 width,=,3,cr,
 do,i,=,1,',',8,cr,
 arr,'(',2,*,i,+,0,')',=,length,+,1,cr,
         length,=,arr,'(',1,*,i,+,1,')',cr,
 enddo,cr,
 end,program,cr]).
```

Calling the parser routine for this program: `?- example(Prog),program(ParseTree,Prog,[]).` yields the following parse tree:

```
X = prog(program,spec_part((type_decl(type_spec(integer),(length,width,area)),
      type_decl(type_spec(integer),i),type_decl(type_spec(integer),
      dimension(array_spec(integer(100))),arr))),
      exec_part(exec_part_const((assign(ident(length),integer(12)),
      assign(ident(width),integer(3)),do(ident(i),integer(1),
      integer(8), exec_part_const((assign(array(arr,[plus(times(integer(2),
      ident(i)),integer(0))]),plus(ident(length),integer(1))),
      assign(ident(length),array(arr,[plus(times(integer(1),ident(i)),
      integer(1))]))))))))),end,program).
```

If par_begin/par_end are implemented (so that the tasks enclosed within them are executed in parallel) along with wait and signal, then in fact a parallel interpreter is obtained. A call to the semantic functions will execute the above program in parallel, and produce results.

Let us now partially evaluate the parallel interpreter w.r.t. this Fortran program above. We will remove the semantic algebra for the store and the array domain, so that evaluation stops when a store or an array operation is reached. The diophantine equations are solved during partial evaluation (the solutions predicate is defined in our semantics so as to solve the diophantine equations exactly). The diophantine equation generated is $2i_1 = i_2 + 1$. Note that the variables i_1 and i_2 are constrained by the loop limits, i.e., $1 \leq i_1 \leq 8, 1 \leq i_2 \leq 8$ and $i_1 \neq i_2$. These constraints can also be included in the semantics (in the definition of the solutions/4 predicate). Solving these constraints, the list of solutions obtained is: [(4,7),(3,5),(2,3)]. The partial evaluator produces the following code, shown below. Notice that each call in the code below essentially corresponds to a machine instruction. The compiled code produced has iterations numbers 7, 5, and 3 waiting for iterations 4, 3, 2 respectively. Note that all recursive procedures (e.g., executed_body1) are tail-recursive (end_task and par_end are really markers), and hence can be automatically transformed into iterative loops. The empty goal true translates to noop, which can be optimized away. Thus, machine code can be now obtained by simple mechanical transformations.

```
go1 :-
        create_store,
        update(length, _),
        update(width, _),
        update(area, _),
        update(i, _),
        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_array(arr, E, C),
               F is A,
               G is F+1,
               access_array(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
    ).
```

Consider the same program as above, except that the array expressions involved are $2 * i_1 + 1$ and $2i_2$. The diophantine equation $2 * i_1 + 1 = 2 * i_2$ has no integer solutions. In this case, no `waits` or `signals` are generated, as all iterations can be executed in parallel. This can be seen in the compiled code obtained through partial evaluation that is shown below (only the code for `execute_body1` is shown).

```
execute_body1(A) :-
        (   8<A ->
              true
        ;   par_begin(task1, task2),
              start_task(task1),
                access(length, B), C is B+1,
                D is 2*A, E is D, update_array(arr, E, C),
                F is A, G is F+1, access_array(arr, G, H),
                update(length, H),
              end_task(task1),
              start_task(task2),
                I is A+1,
                execute_body1(I),
              end_task(task2),
              par_end
        ).
```

Note that in the discussion above, the partial evaluator applied to the parallel interpreter (obtained from the semantic specification of the language) serves as a parallelizing compiler.

4 The I test

As shown above, the conditions for independence of loops are included as part of the parallel semantics of the language. If we have a partial evaluator for a constraint logic programming language, then these constraints can be evaluated during partial evaluation. The constraints framework needed for this purpose is that of CLP(FD) [22]. The CLP(FD) framework is capable of directly solving constraints of the form: $a * i_1 + b * i_2 = c, k_1 \leq i_1, i_2, \leq k_2$ (several other types of constraints can also be solved, but these do not concern us here). CLP(FD) solves these constraints assuming the variables involved have finite domains. Thus, the diophantine equations of the above form are solved with domain of the variables i_1, i_2 set to $k_1..k_2$. If we use the CLP(FD) framework, and we use a partial evaluator for CLP(FD) then we can obtain parallel compiled code in a way similar to the above, the only difference will be that the constraints will be solved using the implementation mechanisms used in CLP(FD) (in the example in the previous section, a simple solver for diophantine equations was also included in the semantic specification, in the `solutions` predicate, since Mixtus does not support constraints solving).

An interesting thing happens when CLP(FD) is used: *the I test* [16], as well as the *Direction Vector I test* [15], is incorporated automatically. This is because the mechanisms used in the *I* tests have long been known in the constraint programming community and are incorporated in the constraint solving systems (see page 87 in [22]). Consider a linear equation of the form:

$$a_1x_1 + ... + a_nx_n = c$$

where c and x_is are unknown but whose ranges are known (in CLP(FD) terminology, they are termed *finite domain variables*). Assume, for simplicity, that a_is are natural numbers ($a_i \geq 0$). Suppose the range of c is $[min..max]$, where min, max are integers. Recall that both in parallelizing compilers as well as in CLP(FD) we are only interested in integer solutions of the above linear equation. Suppose the finite domain of each x_i is $[min_i..max_i]$. Then, the following holds $a_ix_i \geq low_i$ and $a_ix_i \leq high_i$, where

$$low_i = min - (\sum_{k=1,k\neq i}^{n} (a_k max_k)) \qquad high_i = max - (\sum_{k=1,k\neq i}^{n} (a_k min_k))$$

Thus, the new domain of x_i can be computed as $[low_i..high_i]$. If the original domain of x_i, namely, $[min_i..max_i]$ does not overlap with the new computed domain $[low_i..high_i]$, then the constraints are unsolvable. If they do overlap, then the domain of x_i can be narrowed to the intersection of the two ranges. Computing integer solutions with this narrowed domain is considerably more efficient. The above constraint solving facility is present in most constraint solving systems (e.g., CHIP [22] and ALICE [10]). This technique has been used in the parallelizing compiler community as well and is known as the I-test [16].

If there are further constraints on x_i, then these will be used in constraint programming systems to narrow the domains of x_i's further. Thus, if constraints imposed by direction vectors were included, then these constraints will just be included as new constraints in a CLP(FD) system and used during finite domain constraint solving. This extended constraint solving mechanism is known by the name of *Direction Vector I* test in the parallelizing compiler community [15]. The above discussion is easily extended to the case where a_i's are integers (the way low_i and hi_i are computed will change).

If we used a CLP(FD) system to execute our semantics (CLP(FD) systems are Prolog systems extended with finite domain constraint solving [22]), then the *I test* will automatically be included. What is more, other advanced forms of constraint solving that are found in CLP(FD) systems will also be available. Additionally, CLP(FD) systems can be implicitly parallelized, thus, not only the parallelized program can be run in parallel, the parallelization process itself can be run in parallel.

5 A Denotational Framework for Restructuring Transformations

Restructuring transformations can also be expressed in our denotational semantics framework. A restructuring transformation specifies how one or more DO loop (call the set of loops S_1) that has a specific structure is to be transformed to one or more loops (call this set of loops L_2), such that the semantics of the program is preserved. The transformation is done with the hope that the transformed set of loops S_2 will expose more parallelism.

Denotational semantics serves as a good uniform framework for specifying these restructuring transformations. A restructuring transformation can be thought of as transforming the parse tree of the set of loops S_1 to the parse tree of the set of loops S_2. In the denotational framework, parse tree of S_2 is regarded as the meaning of the parse tree of S_1. Thus, the semantic algebra employs domains that range over parse trees, and the valuation predicates give the denotation of a given loop structure in terms of the transformed loop structure expressed using these domains. Given that semantics is executable in our logical denotational framework, the semantic specification of the transformation yields a program transformer automatically.

The dependence information that usually needs to be preserved during restructuring transformation can be computed from the parse trees, and included as part of the semantic specification.

The intended framework is as follows. Loop restructuring transformations are specified as described above. Given a program, the transformer maps this program to transformed code. The transformed code is partially evaluated (in the manner described in previous sections) to obtain optimized parallel compiled code. We next take the example of *Index Set Splitting Transformation* and show how it can be expressed in this denotational framework. The Index Set Splitting transformation splits the index set of a loop into two parts in order to remove a conditional statement. For example, consider the loop

```
DO I=1,100
  IF 2*I+4 > 60
    THEN A[I-20] = I + 2  ELSE A[I] = I + 5
  ENDIF
ENDDO
```

This loop can be transformed into the code shown below, and the condition check avoided:

```
DO I=1,28
    A[I] = I + 5
ENDDO
DO I=29,100
   A[I-20] = I + 2
ENDDO
```

This transformation can be symbolically described in our denotational framework as shown below. Essentially, we employ the power of logical variables of logic programming to express patterns of parse-trees. We assume that the `Condition` is an affine expression [24] involving only loop indices and symbolic constants, of the form `a*I + b > c`, or of the form `a*I+b < c` (we place this restriction only for simplicity of presentation, the most general case in which the condition is an arbitrary conjunction/disjunction of affine expressions can be easily handled). The predicate `affine_expression` takes the parse tree of this condition as an argument and produces the index value around which the loop is to be split. It also computes if the condition is true in the first part of the split loop or the second part. The code for `affine_expression` `split` and `combine` predicates is obvious, and is not included for simplicity.

```
command(do(ident(i),E1,E2,Stmt),S) :-
       split(Stmt,Stmt1,Stmt2,Stmt3),
       Stmt2 = if(Conditon,Then-part,Else-part),
       affine_expression(Condition,SplitIndex,Direction),
       combine(Stmt1,Then-part,Stmt3,B1),
       combine(Stmt1,Else-part,Stmt3,B2),
       (Direction = first -> Body1 = B1, Body2 = B2;
                     Body1 = B2, Body2 = B1),
       S = (do(ident(i),E1,SplitIndex,Body1),
            do(ident(i),SplitIndex+1,E2,Body2)).
```

Most other restructuring transformations can be specified in a similar way.

6 Other Applications

Our denotational framework is powerful enough to incorporate scheduling as well as data distribution.

- Given an architecture, its symbolic description can be made available as an argument to the semantics. The semantics specification can be extended to compute dependencies as well. The dependency information and the architectural description can be employed while specifying the semantics of different language constructs to generate "optimal" code (e.g., parallel code generated to minimize communication between processors). Alternatively, transformations can be specified that take into account the architectural description, and transform a given programming structure to another structure that contains architecture specific execution information (e.g., it is annotated with information regarding where each subpart of the structure is to be executed; for example, in case of DO loop, the processor where each iteration should be executed is determined). Note that the problem of optimally mapping tasks to an architecture is NP-complete, and thus is conveniently expressed in the CLP framework, and hence in our framework. The CLP framework, as mentioned earlier, automatically triggers efficient heuristics to narrow the search space as much as possible. Partial evaluation of this extended semantics which contains task scheduling information will yield parallel code tailored to the architecture.

- From a program's denotation we know the data structures that the program is going to use. Given an architectural description (as an input to the semantics), we can compute optimal data-distribution. Once again the dependency information will have to be taken into account. Partial evaluation will generate data allocation that is "optimal" (e.g., minimizes communication) for the given architecture. As discussed earlier, optimization problems are easily specified in the CLP framework [22], and given that the data-distribution problem is NP complete, it can be naturally specified and handled in our framework.

7 Conclusion and Related Work

In this paper we developed a framework for automatic generation of provably correct parallelizing compilers, provably correct parallel compiled code, and provably correct software system for applying restructuring transformations. As is obvious from the previous sections, Horn Logic (or Constraint Logic) denotations are powerful devices that can be used to specify (parallel) semantics of a language. The semantics has a computational interpretation and yields a parallel interpreter. This parallel interpreter can be partially evaluated w.r.t. a given program to *automatically* obtain parallel code for the program. Thus, a parallelizing compiler is automatically obtained. Because the parallel interpreter and the parallel code is automatically obtained, both the compiler and the compiled code are provably correct (assuming that the partial evaluator is correct, but its correctness has to be proved only once). For many critical applications (e.g., in medicine or defense), correctness of the parallel code is important. Our constraint logic denotational framework can also be used to specify all sorts of restructuring transformation (quest for a uniform framework for restructuring transformations has been an important goal in parallelizing compiler research, e.g., see [9, 13]).

Previous work in semantics based study of parallelizing compiler includes the work of Parsons [12] and Felleisen [2]. Parson's work (i) gives an operational semantics based on graph rewriting for program dependence graphs, and, (ii) shows the equivalence between this rewriting semantics and the operational semantics of the program. This is applied for study of restructuring transformations. This is somewhat similar to our approach, except that we give a declarative denotational semantics (which yields an operational semantics). Graph rewriting is somewhat like partial evaluation. Our framework is more general, in that it can be used for specifying transformations, specifying independence, as well as generating parallel code.

Research has also been done in developing unified frameworks for restructuring transformations [13, 9]. In contrast to their approach, our framework is based on denotational semantics. A major advantage of our approach is that the same framework suffices for most aspects of compilation and transformation of parallel programs. To the best of our knowledge, no framework has been developed in the past that can derive efficient parallel code from the semantic specification of a language.

References

[1] W. Chen and D. S. Warren. Tabled Evaluation with Delaying for General Logic Programs, *In JACM 43(1):20-74.*

[2] M. Felleisen and R. Cartwright. A Semantic Foundation for Program Optimization. Internal Report, Rice Univ. 1988.

[3] G. Gupta, M. Carlsson (eds). High Performance Implementation of Logic Programming Systems, *Journal of Logic Programming*, 1996.

[4] G. Gupta and E. Pontelli. A Constraint-based Approach for Specification and Verification of Real-time Systems. In *Proc. IEEE Real-time Systems Symposium*, Dec. 1997.

[5] G. Gupta. Horn Logic Denotations and Their Applications. Technical Report, NMSU, Dec. 1997.

[6] C. Gunter. Programming Language Semantics. MIT Press. 1992.

[7] J. Jaffar and M.J. Maher. Constraint Logic Programming. *Journal of Logic Programming*, 19/20, 1994.

[8] N. Jones. Introduction to Partial Evaluation. In *ACM Computing Surveys*. 28(3):480-503.

[9] W. Kelly, W. Pugh. A Unifying Framework for Iteration Reordering Transformations. University of Maryland, Technical Report CS-TR-3430, 1995.

[10] J-L. Lauriere. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence*, 10(1):29-127, 1978.

[11] J.W. Lloyd. Foundations of Logic Programming. Springer Verlag. 2nd ed. 1987.

[12] R. Parsons. A Rewriting Semantics for Program Dependence Graphs. 16th POPL, ACM Press, 1989.

[13] W. Li and K. Pingali. A Singular Loop Transformation Framework based on Nonsingular Matrices. In *International Journal of Parallel Programming*, 22(2), 1994.

[14] E. Pontelli, G. Gupta. Parallel Symbolic Computation with ACE. In *Annals of AI and Mathematics*. (to appear in Dec. 97). (Summary in http://www.cs.nmsu.edu/lldap/ace/ace.html).

[15] K. Psarris, X. Kong, D. Klappholz. The Direction Vector I Test. *IEEE Transactions on Parallel and Distributed Systems*, 4(11), 1993.

[16] K. Psarris, X. Kong, D. Klappholz. The I Test: An Improved Dependence Test for Automatic Parallelization and Vectorization. *IEEE Transactions on Parallel and Distributed Systems*, 2(3), 1991.

[17] D. Sahlin. An Automatic Partial Evaluator for Full Prolog. Ph.D. Thesis. 1994. Royal Institute of Technology, Sweden. (Software available from www.sics.se)

[18] D. Schmidt. *Denotational Semantics: a Methodology for Language Development*. W.C. Brown Publishers, 1986.

[19] D. Schmidt. Programming language semantics. In CRC Handbook of Computer Science, CRC Press, Boca Raton, FL, 1996. Summary version, ACM Computing Surveys 28-1 (1996) 265-267.

[20] 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.

[21] L. Sterling & S. Shapiro. The Art of Prolog. MIT Press, '94.

[22] P. Van Hentenryck. *Constraint Handling in Prolog*. MIT Press, 1988.

[23] P. Van Hentenryck. Parallel Constraint Satisfaction in Logic Programming. *Proc. Int. Conference on Logic Programming*, MIT Press, 1989.

[24] M. Wolfe. *High Performance Compiler for Parallel Computing*, Addison Wesley, 1996.

[25] H. Zima, B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press, '91.