PARALLELIZATION BY SEMANTIC DETECTION OF REDUCTIONS

P.JOUVELOT

ABSTRACT

One of the best known techniques to compile sequential programs for multiprocessors is to detect the so called **reduction operations** . An example of such an operation is the sum of vector elements which can be evaluated under a pyramidal scheme using the associativity property of addition. A method to detect such operations in a PASCAL or FORTRAN-like programming language is presented. This detection and the corresponding modifications to the source programs are considered as non-standard denotational interpretations of the abstract syntax tree of the object programs. A by-product of this paper is to show how a denotational specification of a non-trivial application can directly lead to a running prototype, using here the ML programming language as an "executable specification language".

MASI, Université PARIS-VI

1. INTRODUCTION

We propose to detect reduction operations (**redop**), e.g. sum of vector elements or scalar product, in order to improve the "parallelism degree" of programs. For instance, our aim is to translate a program such as :

```
sp := 0 ; n := 1000 ;
for i to 100 do [
      a[i+n] := 2*a[i] ;
      sp := sp + a[i]*b[i] ;
      ]
```

whose complexity for the computation of **sp** is *O(N)*, into a new program exhibiting explicitly the redop on **sp** which could be evaluated in *O(log N)* on a multiprocessor architecture :

```
sp := 0 ; n := 1000 ;
for i to 100 do
      a[i+n] := 2*a[i] ;
sp := for_red i to n eval (sp + a[i]*b[i]) ;
```

where the keyword **for_red** signals to the compiler code generator the possibility of a reduction optimization. Note that this transformation has to be based on semantic considerations (in particular on the value of **n**) to assert that there is no conflict between the different iterations of the loop body. A simple pattern-matching on standard instructions schemes is here totally inadequate.

Of course, we could yet improve the program parallelism. For example, the **for** loop could be replaced by a parallel-loop. But that is outside the scope of this paper (see [Jou85a]). Once again, note that we need semantic informations to parallelize this simple loop : the common techniques proposed in the litterature such as [All83a] or [Kuc79a] are generally unable to cope with such situations.

In a first section, we shall introduce more precisely the language abstract syntax which allows reduction expressions. Then, after a general overview of our analysis method, we shall describe its essential part : the symbolic evaluator. At last, we shall give the description of the major non-standard interpretations which form the basis of our technique. The reader is supposed to have a minimal knowledge of denotational techniques as described in [Gor79a] ,[Ten76a] or [Sto77a] , and to be familiar with high-order applicative languages such as "Pure Lisp", Scheme [Abe85a] or ML. The later will be used here as a specification language [Jou85b] .

2. ABSTRACT SYNTAX

We shall present rapidly the abstract syntax (in ML BNF-like form [Gor79b] ,[Cou85a]) of a simplified version of the ALL programming language [Jou85a] , enhanced here to allow reduction operations. The concrete syntax is given in comments.

```
rectype Exp =   con_to_exp of int |                    % k %
                ide_to_exp of Ide |                    % i %
                tab_to_exp of Ide # Exp |              % t[e] %
                unopr of O1pr # Exp |                  % o1 e1 %
                binopr of Exp # O2pr # Exp |           % e1 o2 e2 %
                cond_exp of Exp # Exp # Exp            % if e1 then e2 else e3 %

rectype Com =   nop |                                  % nop %
                assign of Ide # Exp |                  % i:=e %
                tab_asg of Ide # Exp # Exp |           % i[e1]:=e2 %
                if_then_else of Exp # Com # Com |      % if e then c1 else c2 %
                for of Ide # Exp # Com |               % for i to e do c %
                list_com of Com list |                 % c1;...;cn %
                for_red of Ide # Ide # Exp # Exp       % i1:=for_red i2 to e1
                                                         eval e2 %
```

where **Ide** is the domain for identifiers, **O2pr** for binary operators and **O1pr** for unary ones. Since this syntax is quite usual, we shall only insist on the most original point, i.e, the last command (reduction).

Its intuitive semantics is the following. In **i1:=for_red i2 to e1 eval e2** , **e2** is an expression which, given **i2** (considered here as a locally bound variable), can be seen as a function of **i1** and free variables depending on **i2**. Hence, we can (meta)write **e2** as **e2(i1,l(i2))**. The idea is to apply the function **e2** to successive **i1** values when **i2** evolves between 1 and **e1**'s initial value. The result is eventually bound to **i1**. This semantics is quite similar to ML functionnal **itlist** .

The detection of such an iteration operation is worthwhile only if **e2(i1,l)** is associative because it allows a pyramidal evaluation scheme. For example, **e2(e2(e2(i1,l),l'),l'')** needs (on a Von-Neuman sequential machine) three steps to be evaluated. If **e2** is associative, we can rearrange this expression to permit a two-step evaluation : **e2(e2(i1,l),e2(l',l''))**. Of course, this abstract syntax tree modification is effective only if at least two parallel processors are simultaneously available. This kind of parallel operator is quite common in the VLSI world. For instance, we can compare **for_red** to the "census operator" of[Ull84a] or to the "beta function" of the Connection Machine[Hil85a] .

More formally, we give below a direct denotational semantics for this reduction instruction (in an ML-like notation). Moreover, it introduces some notations we shall use later.

```
typeabbrev Value = int
typeabbrev Env = Ide -> Loc list
typeabbrev Store = Loc -> Value

% E:Exp -> Env -> Store -> Value
  C:Com -> Env -> Store -> Store %

let C (for_red(i1,i2,e1,e2):Com) (r:Env) (s:Store) =
   let max = E e1 r s and
      ad_i1, ad_i2 = hd (r i1), hd (r i2) in
   letrec roll s =
      let i2 = s ad_i2 in
      if i2 < max+1 then
         let new_i1 = E e2 r (expand s (i2+1,ad_i2)) in
         roll (expand s (new_i1,ad_i1))
      else s in
   expand (roll (expand s (1,ad_i2))) (s ad_i2,ad_i2)
```

where **expand f (b,a)** is a function which, when applied to **x**, gives **b** if **x** equals **a** and else the result of **f**'s application to **x**. **E** and **C** are the standard semantics for Expressions and Commands evaluation. They are quite straightforward despite a rather original choice for **Env**(ironments) domain : each identifier (array or simple variable) corresponds to a **Loc**(ation) list in order to ease array's treatment.

This semantics makes sense whether the underlying function is associative or not. However, it is only in the case of associative reduction that parallel evaluation is possible. Consequently, this instruction may be written by the user under his own responsability (with, if possible, some check by the vectoriser/paralleliser) or by the tool we are presenting.

3. METHOD OVERVIEW

The basic idea is to define a functional **P** whose main duty is to analyze loops in order to detect redops (reduction operations) : its type is **Com -> Assert -> Niv -> Com # Assert** . **Com** is the syntactic domain of language commands. Each program command is labelled by its *level* which is a member of the **Niv** domain. In practice, this level is an index (implemented by an integer list and hidden under an abstract data type) which numbers the nodes of the program abstract syntax tree. The **Assert** domain is the lattice of functions which map every program level to a pre-condition called a *claim* (from the **Cla** domain). We have already proposed in[Jou85a] some methods to associate a claim to every node of the abstract syntax tree and, then, an assert function to every program. Let us simply recall here that a claim is a logical formula (pratically in Presburger's arithmetics [Suz77a]) which gives information on scalars possible values. For instance, before the loop of our example program, the claim is **(s=0) and (n=1000)** .

The functional **P** takes three parameters : a command **c** (Com), a tree of claims **h** (Assert) and the level **n** (Niv) of **c**. It returns as a result a pair built from a "more parallel" command (indeed, with explicit reductions) and an assertion function updated accordingly

(due to abstract syntax tree transformations). The algorithm's fundamental is based on Bernstein's conditions computation [Ber66a] for the whole set of program commands. Let us recall that Bernstein's conditions claim that two commands are parallely executable if their shared variables are only accessed in read mode. During this computation, the assertion function enables a more precise specification of the *areas* (domain **Exp # Cla**) being manipulated by the program both in read and write modes. For instance, the area used in write mode by the first command of the loop body of our example program is **(a[i+n],n=1000 and 1<=i<=100)** .

The basic point of our method is a symbolic evaluator [Dan82a] which determines the function or denotation equivalent to a loop body. If this function owns such fine properties as Bernstein's conditions satisfaction and associativity, we shall prune (i.e. change to **nop**) every instruction of the loop which uses the variables concerned by such reductions. At last, we simply shall declare the reduction operations on loop exit (see our example program).

4. SYMBOLIC EVALUATION

To test for the presence of reduction, we need to consider the transformation denoted by the whole loop body. For instance, we do not want to miss the summation in

```
for i to 1000 do [
      s := a[i] ;
      ....
      s := s+b[i] ;
      ]
```

Further more, due to conditional instructions, there may be multiple paths in the loop body ; we wish to explore all these paths in order to detect cases like

```
for i to 1000 do
      if a[i]>0 then x := x+a[i] ;
```

The principle of our solution is to associate to each scalar variable a *symbolic value* (Svalue) which is a list of pairs whose components are a *symbolic environment* (or context) and an expression. Loosely speaking, a symbolic environment (Senv) is a conjunction of logical symbolic values and represents a path in a tree of conditional instructions. When the initial store is such that all expressions in the symbolic context evaluate to **true**, then, at the end of the loop body, the corresponding scalar variable will be bound to the value of the expression of its symbolic value. The binding **Ide -> Svalue** is called *symbolic state* (Sstate). In our ML-like notation, these notions could be defined with concrete types (to allow recursive domain definitions)

```
rectype Svalue = sval of (Senv # Exp) list
  and Senv    = senv of Svalue list

typeabbrev Sstate = (Ide,Svalue) func
```

where **func** is an here unspecified polymorphic abstract data type which, by instantiation, defines arbitrary functions (here from the **Ide** domain to **Svalue** one). We use such an

implementation for symbolic states since we need to be able to retrieve quite particuliar informations on these functions (e.g. its domain) : this is impossible with simple ML lambda abstractions.

As advocated by [Don78a] , this symbolic evaluator can be seen as a non-standard interpretation of ALL abstract syntax. To achieve this, we introduce two functions **Es** and **Cs** which are the symbolic equivalents of **E** and **C**.

```
Es:Exp -> Senv -> Sstate -> Svalue
Cs:Com -> Senv -> Sstate -> Sstate
```

4.1. Expressions

The semantic equations of symbolic evaluations are given by :

```
let Es (con_to_exp k) (r:Senv) (s:Sstate) = sval [r, con_to_exp k]
```

A constant's Svalue in the Senv **r** is a LIST (in Lisp sense) of a pair built from the symbolic environment and the constant.

```
let Es (ide_to_exp i) r s =
   let sval re_list = apply s i in
   sval (map (\senv ri,ei.senv (r @ ri),ei) re_list)
```

where @ is the list concatenation operator (APPEND). First, we get **i**'s Svalue by application (**apply**) of the Sstate **s**. For each pair **(senv ri,ei)** of domain **Senv # Exp** , we simply add (in fact, that yields a "logical and") the current symbolic environment **r**. This iteration is implemented by applying (via **map**) a lambda-expression (\) to the Svalue.

```
let Es (binopr(e1,o2,e2)) r s =
   let sval r1e1_list, sval r2e2_list = Es e1 r s, Es e2 r s in
   sval (itlist (\(senv ri,ei) re_l.
                re_l @ (map (\senv rj,ej.senv (ri @ rj @ r),O2 o2 (ei,ej))
                            r2e2_list))
               []
               r1e1_list)
```

The binary operator's case is almost evident. We have to create all possible combinations (by **itlist**) of **r** with the symbolic environments of **e1** and **e2**'s Svalues. Then, for each couple **(senv ri,ei)** and **(senv rj,ej)** , we apply the **o2** operator to the two Exps **ei** and **ej**. The **O2** functional is the here unspecified denotational semantic evaluator for binary operators. Since the treatment of unary operators is quite similar, we feel free not to give it here.

```
let Es (tab_to_exp(i,e)) r s =
   let sval re_list = Es e r s in
   map (\senv rj,ej.senv (r @ rj),tab_to_exp(i,ej)) re_list
```

As the treatement of arrays is special (see below), all we have to do is to apply the constructor **tab_to_exp** to all possible expressions of **e** Svalue in the extended Senv.

```
let Es (cond_exp(e1,e2,e3)) (senv r) s =
   let r1, r1' = Es e1 r s, Es (unopr(non,e1)) r s in
   let sval r2e2_list = Es e2 (senv (r1.r)) s and
      sval r3e3_list = Es e3 (senv (r1'.r)) s in
   sval (r2e2_list @ r3e3_list)
```

where '.' is the ML equivalent of the CONS operator on lists. We evaluate **e2** in a symbolic environment where **e1** is true and **e3** in an environment where **e1** is false. The symbolic value of the conditional expression is the concatenation of the two results.

4.2. Commands

We are now able to cope with commands : they are seen as symbolic state's transformers.

```
let Cs (nop) r s = s
let Cs (tab_asg _) r s = s
```

By principle, we don't make symbolic evaluation for arrays : the reason is that we don't need it. However, would it be absolutely necessary, a possible solution could be to use conditionnal expressions to sweep array's elements as proposed in [Don78a] . Let us point out that this restriction can lead to incorrect Svalues for some variables, say **i**. But, in that case, we are sure that **i** uses an array element which is modified by the command set currently analyzed. In consequence, Bernstein's conditions aren't verified : the parallelization will be inhibited and so, **i** Svalue is useless.

```
let Cs (assign(i,e)) r s = expand s (Es e r s,i)
```

An assignment leads to current symbolic state change with binding between **i** and **e**'s Svalue. **Expand** is a constructor of the **func** abstract data type.

```
let Cs (if_then_else(e,c1,c2)) (senv r) s =
   let ds = domain s and
      R = senv r in
   let r1, r2 = senv (Es e R s.r), senv (Es (unopr(non,e)) R s.r) in
   let update r = itlist (\i s'.expand s' (Es (ide_to_exp i) r s,i)) s ds in
   let s1, s2 = update r1, update r2 in
   let s'1, s'2 = Cs c1 r1 s1, Cs c2 r2 s2 in
   let evol i = not (apply s'1 i = apply s1 i) or not(apply s'2 i = apply s2 i)in
   itlist (\i f.
         expand f (let sval v1, sval v2 = apply s'1 i, apply s'2 i in
                  sval (v1 @ v2),i))
      s
      (filter evol ds)
```

The main problem with the symbolic evaluation of a conditional instruction **c** is introduced by the variables which are conditionally modified in **c** (i.e. identifiers assigned only in one branch). For instance, if **i** is only modified in **c1**, we have to take care not to forget that **i** Svalue is unchanged in **c2**, i.e. when **not(e)** appears in the symbolic environment. That's

why we evaluate each instruction **c1** and **c2** in a symbolic state adapted to each branch : **s1** (resp.**s2**) binds each identifier of **s**'s domain with its Svalue in an environement which takes into account the fact that **e** (resp.**not(e)**) is true when **c1** (resp.**c2**) is executed. To avoid cluttering Svalues by unmodified identifiers, we use these branch evaluation results only when (**filter**) an assignement is effectively present, either in **c1**, or in **c2** (**evol**).

let Cs (list_com l) r s = rev_itlist (\c.Cs c r) s l

where **rev_itlist** compose the commands, considered as symbolic state transformers.

We have not evoked the action induced on symbolic states by loops : that is deliberate. To introduce it, we would have to face the undecidable problem of fix point approximations [Cou78a] . Moreover, for our purpose, we can overcome this difficulty. The only symbolic evaluations needed to detect reductions will be done inside loop bodies. We only have to limit ourselves to the innermost loop : we don't chase reductions into nested loops. As a consequence, there is no need to study **for_red** influence on symbolic states.

4.3. Correction

The correction proof of the preceding algorithm is simple, though tedious. We shall only give here the main lemmas.

First, by induction on program text, we can easily show that :

> *For each Sstate s created by Cs and for each identifier i in s's domain, there is a unique Senv of i Svalue (considered here as a list in Senv # Exp) which is true.*

Or, more clearly, for a given initial memory state, there exists a unique path (via conditional branchs) taken by a single computation. We are then able to prove the main theorem which translates informally to :

> *For an identifier i and a command c, let i' be the symbolic value of i after symbolic evaluation of c. Then, i' value in an initial memory state s0 is equal to the value of i in a state memory obtained after standard execution of c from s0. The main condition is that i' doesn't use an array element modified in c.*

Note that this last restriction will be a direct consequence of Bernstein's conditions verification.

5. REDUCTION PARALLELIZATION

5.1. The Method

The **P** interpretation is formally defined below :

```
let P (for(i,e1,c1)) h n =
   let n1 = app_niv n 1 in
   let ie_list = recur c1 h n1 in
   let Cr,hr = com_prune c1 (map fst ie_list) h n1 and
      C_red = map (\j,e.for_red(j,i,e1,e)) ie_list in
   let C' = if Cr = nop then
              assign (i,binopr(e1,plus,con_to_exp 1))
            else
              for(i,e1,Cr) in
   let new_c = list_com (C'.C_red) in
   let new_h = H new_c n (apply hr n) in
   new_c, adapt hr n1 new_h
```

Some points need here to be emphasized. First, if the loop level is **n**, **c1**'s one is by definition **n1** (here, **app_niv** is an operator of the abstract data type Niv). Second, we shall see just later that the **recur** function (whose type is **Com -> Assert -> Niv -> (Ide # Exp)** list) tries to detect the reductions embedded in **c1** and returns the corresponding list of identifiers and expressions. With this output, **com_prune** (**Com -> Ide list -> Assert -> Niv -> Com # Assert**) (see likewise below) has to change to **nop** each instruction which is concerned by some reduction. Moreover, **com_prune** modifies accordingly the claims tree in the affected part of the program. So, we shall not be obliged to recompute the whole Assert function after the reduction detection pass. This point is important if we remember that our technique has, eventually, to be included in a full parallelizer.

Then, if the whole loop body shrank to **nop** during this operation, we have only to return the **C_red** list of reduction operations with the correct update of the loop counter (here **i**). If there remains at least one useful command after reduction elimination, we simply have to add the pruned loop to the preceding reduction list. Let us point out that, in order to simplify our presentation, we supposed that every loop is at least executed once : this could be easily fixed.

Note that the here unspecified function **adapt** has to take into account the potential claims induced by redops and, moreover, to correctly manage the assertion function domain. To achieve this incremental update, we use a functional **H** (**Com -> Niv -> Cla -> Assert**) already presented in[Jou85a] which labels every node of an abstract tree with its valid pre-condition.

The other defining equations of **P** (on other commands) are straightforward : we just have to propagate **P** until a loop on which the preceding treatement is applied. Don't forget that we also have to check that this loop doesn't embed yet another inner loop : this is a "context condition" on program syntactic structure.

5.2. Reduction detection

This is the purpose of the **recur** function introduced in the preceding section.

```
let recur c h n =
   let m, l = M c h n, L c h n and
      in_cla = apply h n in
   let m_ids = mapfilter (\ide_to_exp i,_.i) m in
   let V = union (mapfilter (\ide_to_exp i,_.i) l) m_ids in
   let new_s = Cs c (senv []) (mk_Sstate V) in
   mapfilter (\j.
      let sval jval = apply new_s j and
         Lp = L (fst (com_prune c [j] h n)) h n in
      if reducible j new_s &
        forall (\senv r,e.
              let A_Senv = itlist (\e.union (A e in_cla)) []
                                  (map snd (flat (map (\sval r.r) r))) in
              not (inter A_Senv m) &
              not (inter (A e in_cla)
                         (filter (\e,_.case e of
                                    (ide_to_exp i).not (i=j) |
                                    _.true)
                                 m)) &
              not (inter [ide_to_exp j,true_cla] Lp))
             jval then
          j,Sval_to_exp jval
       else fail)
      m_ids
```

The two functionals **L** and **M** enable us to determine what are the accessed (**L**) and modified (**M**) area lists for a given command : they both use the functional **A** which gives the accessed area list for a given expression. They are left unspecified here : their specifications as sketched in [Jou85a] are moreover fairly simple.

We determine the set of accessed (**l**) and modified (**m**) areas of the loop body **c** and the in-coming claim **in_cla**. Using **m**, we denote by **m_ids** the identifiers which appear in commands left hand sides : they are the identifiers potentially involved in a redop. **V** is then the set of all identifiers appearing in **c**. It is used to define (via **mk_Sstate**) a symbolic initial state which binds every identifier **j** to the Svalue **(sval ([senv [],ide_to_exp j))** . From this initial Sstate, a symbolic evaluation of **c** is performed, yielding a new Sstate **new_s**.

Our redop detection is made by analysis of the Svalue of every identifier of **m_ids** in the terminal Sstate. Each couple (identifier **j**, expression of **j**'s Svalue) represents a reduction if the following properties are true :

- **j** is modified in the loop body (that is why we iterate on **m_ids** by **mapfilter**),
- **j** Svalue at the end of the loop body (**new_s**) has to be **reducible** , i.e. the corresponding function (cf.section 1) has to be associative,
- there is no collision (**inter**) between **m** and the different areas (given by **A_Senv**) involved in the symbolic environments of **j** Svalue,
- there is no collision between **m** and **j** Svalue's expressions, except **j** itself,
- **j** is not read elsewhere in the loop body,

The **Sval_to_exp** operation transforms a Svalue into a syntactic expression (which is generally a conditional one, using symbolic environments as tests).

We may consider differents possibilities to test (in **reducible**) whether an identifier has, in a given symbolic state, a symbolic value which is compatible with a reduction (associativity). The more general one would be based on functional analysis or general pattern-matching [McI85a] . But, for our goal, a much simpler solution was adopted, which is however sufficiently powerful to take care of the great majority of usual code. Beside constant values, we only try to detect additive (resp. multiplicative) operations : in that case, all we have to do is to check that, for each Senv, the corresponding Exp minus (resp.divided by) the identifier is independant of it. This strategy is implemented in the **reducible** function but is not given here because of its triviality.

5.3. Pruning of the abstract syntax tree

The only function yet to be specified is **com_prune** ; its purpose is to clear the abstract syntax tree of all the instructions converted to reductions. For elementary commands, it is obvious :

```
let com_prune (assign(i,e)) l h n =
   if mem i l then (nop,h) else (assign(i,e),h)
let com_prune (nop) l h n = nop,h
let com_prune (tab_asg(i,e1,e2)) l h n = tab_asg(i,e1,e2),h
```

We simply eliminate the instructions whose left member is present in the list l of reduced identifiers.

```
let com_prune (list_com lc) l h n =
   letrec prune L h n1 =
      if null L then ([],h)
      else let C',h' = com_prune (hd L) l h n1 and
               n2 = inc_niv n1 in
           if C' = nop then
              prune (tl L) (translate h' (is_after n2) (dec_niv n1)) n1
           else let L'',h'' = prune (tl L) h' n2 in
                (C'.L'',h'') in
   let n1 = app_niv n 1 in
   let L',h' = prune L h n1 in
   if null L' then
      (nop, retract h' n1)
   else if length L' = 1 then
      (hd L',translate h' (is_after (app_niv n1 1)) (pull_up n1))
   else (list_com L',h')
```

where **retract f x** returns a function (from the **func** abstract data type) where **x** is unbound. The list case is sligthly more complex and uses an internal auxiliary function **prune** : it has to deal with a list **L** whose first element is at level **n1**. The argument is by induction on the list length. If **L** is empty, we simply returns ([],**h**). If not, we prune the first command (level **n1**) of **L**. If the resulting code **C'** is **nop** , we iterate the process on the tail of **L** : the assertion function is simply "left-shifted" by **translate** . If **C'** is not completely removed, we take the same assertion function to study the tail of **L**. Afterward, all we have to do is to insert **C'** at the head of the returned code.

Using **prune** , the semantics of a command list treatement is easy to understand. Let us just point out the need for the test on **L'** length. If it is 1, the constructor **list_com** is useless : so we return the head of **L'** and the function **h'** where all the nodes whose level is "under" **(hd L')**'s one have been shifted up.

The functions **translate**, **is_after**, **pull_up** and **dec_niv** may be caracterized, on the representation domain of Niv, by the following axioms where *n,n',m,m'* are levels and *a,b* are integers (used in the representation of levels) :

t n = true => h n = translate h t g (g n)
t n = false => h n = translate h t g n
a >= b => is_after (n.b) (n.a.n') = true
pull_up (n.a) (n.a.n') = n.n'
|n| = |m|+1 => dec_niv n (m.x.m') = m.(x-1).m'

where '.' denotes the concatenation in the representation domain of Niv and |*n*| the length of *n*'s representation.

```
let com_prune (if_then_else(e,c1,c2)) l h n =
   let n1, n2 = app_niv n 1, app_niv n 2 in
   let c'1,h'1 = com_prune c1 l h n1 in
   let c'2,h'2 = com_prune c2 l h'1 n2 in
   if (c'1 = nop) & (c'2 = nop) then
      (nop,retract (retract h'2 n2) n1)
   else (if_then_else(e,c'1,c'2),h'2)
```

For the conditional statement, the principle is quite similar to the previous one.

The correction of the **com_prune** function is a direct consequence of the following "invariant form" theorem :

For every function h of Assert, command c and level n,
If c',h' = com_prune c h n, then :
domain h - mk_domain c n = domain h' - mk_domain c' n

where **domain h** gives the inital domain of h and **mk_domain c n** returns the set of levels of the abstract syntax tree of **c** if its root's level is **n**. The proof is by induction on **c** and uses only the preceding axioms.

6. CONCLUSION

We presented a method to detect reduction operations which can be efficiently implemented on multiprocessors or specific hardware. The core of the method is a symbolic evaluator completely specified here. Beside simple reduction detection, the approach taken uses semantic informations which may be automatically extracted from the program source.

The result is that the specification of the whole process is given in a "denotational"-like framework and shows the power of this "syntax-directed" paradigm for design of program manipulation programs [Nie85a] . Moreover, using adequate tools, this approach directly (and for free) gives a running prototype as soon as complete specification is defined.

A prototype, implemented in ML, is being developed at the MASI laboratory in order to validate this reduction detection technique.

Ackwnoledgments

I wish to thank P.Feautrier for his helpful comments during the preparation of this paper.

References

Abe85a. Abelson and Sussman, *Structure and Interpretation of Computer Programs,* MIT Press (1985).

All83a. J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of Control Dependence to Data Dependence," *POPL83*, ACM (1983).

Ber66a. A. J. Bernstein, "Analysis of Programs for Parallel Processing," *IEEE Trans. on Elec. Comp.* **15**, pp.757-763 (October 1966).

Cou85a. G. Cousineau, "The ML Handbook," *(draft INRIA)* (Mai 1985).

Cou78a. P. Cousot, *Methodes Iteratives de Construction de Points Fixes d'Operateurs Monotones sur un Treillis, Analyse Semantique des Programmes,* U.S.M.G, Grenoble (1978).

Dan82a. R. B. Dannenberg and G. W. Ernst, "Formal Program Verification Using Symbolic Execution," *IEEE Trans.on Soft.Eng.* **8** (Janvier 1982).

Don78a. V. Donzeau-Gouge, "Utilisation de la semantique denotationelle pour l'etude d'interpretations non-standard," *Rapport de Recherche 273,* INRIA (Janvier 1978).

Gor79a. M. J. C. Gordon, *The Denotational Description of Programming Languages,* Springer-Verlag (1979).

Gor79b. M. J. C. Gordon and R. Milner, *Edinburgh LCF,* Springer-Verlag (1979).

Hil85a. Hillis, *The Connection Machine,* MIT Press (1985).

Jou85a. P. Jouvelot, *Evaluation semantique des Conditions de Bernstein,* MASI (Mars 1985).

Jou85b. P. Jouvelot, *ML : Un Langage de Maquettage ?,* AFCET (Octobre 1985).

Kuc79a. D. J. Kuck and D. A. Padua, *High-Speed Multiprocessors and their Compilers,* IEEE (1979).

McI85a. K. McIsaac, "Pattern Matching Algebraic Identities," *SIGSAM Bull.* **19** (Mai 1985).

Nie85a. F. Nielson, "Program Transformations in a Denotational Setting," *TOPLAS,* ACM (Juillet 1985).

Sto77a. J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory,* MIT Press (1977).

Suz77a. N. Suzuki and D. Jefferson, *Verification Decidability of Presburger Array Programs,* 1977.

Ten76a. R. D. Tennent, "The Denotational Semantics of Programming Languages," *CACM* **19** (Aout 1976).

Ull84a. J. D. Ullman, *Computational Aspects of VLSI,* Computer Science Press (1984).