ParaGraph : an interactive environment for parallelizing FORTRAN programs

Bernard Dion, Laurent Angeli, Angel Bravo Lastra

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

ParaGraph

An interactive environment for parallelizing FORTRAN programs

Bernard DION
Laurent ANGELI
Angel BRAVO LASTRA

N° 1920
MAI 1993

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel

ParaGraph

An interactive environment
for parallelizing FORTRAN programs

Un environnement interactif
de parallélisation de programmes FORTRAN

Bernard DION, Laurent ANGELI, Angel BRAVO LASTRA

CERICS et INRIA
Sophia Antipolis

April 23, 1993

Abstract

This report presents the implementation of PARAGRAPH, a parallelizer for F, a subset of the FORTRAN language, in the CENTAUR system. Using TYPOL, the dynamic semantics of the language has been formally defined, dependence graphs have been computed and a standard set of transformations has been specified. An interactive user interface based on the SOPHTALK message based system has then been experimented.

Résumé

Ce rapport présente l'implémentation de PARAGRAPH, un paralléliseur pour le langage F, un sous-ensemble du langage FORTRAN, à l'aide du générateur d'environnements interactifs CENTAUR. La sémantique dynamique du langage F a été spécifiée formellement en TYPOL, ainsi que le calcul des graphes de dépendances. Un ensemble de transformations classiques a finalement été spécifié. Une interface utilisateur interactive, réalisée en SOPHTALK, a été exprérimentée.

Key Words: Interactive Programming Environments, Formal Semantics, Parallelization, Dependence Graphs, FORTRAN.

Contents

List of Figures

1 Introduction

1.1 Motivation

The goal of the research that is presented in this report was to realize an interactive parallelizer for a subset of the FORTRAN language, using the facilities provided by **CENTAUR** [Centaur92] [Jacobs92], a generator of interactive programming environments[1].

As this had been done for the ESTEREL language [Bertot89] [Saint91], we wanted to provide the CENTAUR research project with a non-trivial example, that could help in understanding the issues in two main directions:

- The effectiveness of using the **TYPOL** language for the specification of the dynamic semantics, the calculation of dependence graphs, and the realization of a standard set of transformations, as described, for example, in [Zima91] [2].
- The use of **SOPHTALK** [Clément92], a message based system, in experimenting various ways of user interaction with the parallelizer.

The result of this project is the **PARAGRAPH** prototype which is an interactive parallelizer for the **F** language, a subset of FORTRAN 77, augmented with a few constructs taken from FORTRAN 90 and FORTRAN PCF, as it can be seen on the example of Figure 1.

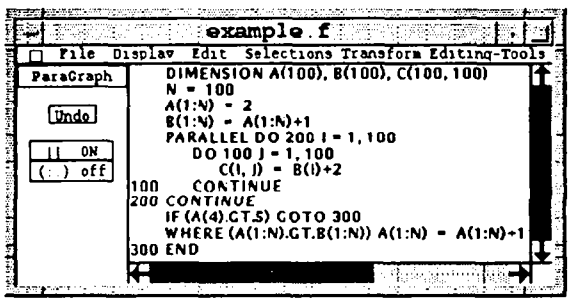

Figure 1: *An example F program*

The precise syntax and semantics of **F** are defined in the next chapter.

[1]This project was partially funded by the "Ministère de la Recherche et de la Technologie" under contract 90.S.0960.

[2]In this respect respect, we build upon the work that was done in the PIAF prototype [Giboulot92], which used LISP and the Virtual Tree Processor primitives of CENTAUR to implement similar transformations. We firmly hope that, going to a language such as TYPOL, which has been specially designed to describe the semantics of programming languages, we have a chance to be able to prove, *in the long term*, that the transformations are correct.

1.2 The CENTAUR framework and the issues

As it is the case for most generators of interactive programming environments (see, for example, [Teitelbaum81][Reps88]), the **CENTAUR** system is organized around an internal representation which has the form of an *abstract syntax tree*. This internal form can be manipulated by various *tools* which we can classify as follows:

- *Syntactic analysis tools*, based on the **METAL** formalism which is used to define the concrete and abstract syntax of languages.
- *Decompilation tools*, based on the **PPML** formalism which is used to specify pretty-printing of abstract syntax trees.
- *Semantic analysis tools*, based on the **TYPOL** formalism which is an implementation of Natural Semantics [Clément85]. TYPOL is used to develop such tools as type checkers, interpreters, debuggers, compilers, parallelizers ... .

The integration of these tools in the framework of an interactive programming environment will be achieved using the **SOPHTALK** message based system. **SOPHTALK** will be used to maintain the coordination of multiple views of programming objects, such as the sequential and parallel forms of a given program, or a program view and its associated dependence graph.

Using **CENTAUR** to develop a programming environment, we will be interested by the following properties:

- ease of expression, and readability,
  concerning the language in which the tools are written,
- possibility to prove the correctness of the tools,
- performance of the various engines which are used in running the tools.

With respect to the first two points, we will show how rule-based languages provide us with great help. The **METAL** , **PPML** and **TYPOL** formalisms all obey to a *declarative style* of programming. It is certainly not yet the case of **SOPHTALK** which provides us with a set of fairly low level LISP primitives that have to be used in a more traditional *imperative style*.

The last point, that is performance, will include a necessary discussion on the *incrementality* of the tools. For example, the **PPML** engine has the property of being incremental. This means that, in the case the abstract syntax tree associated to a **CENTAUR** editor window is modified, only the parts of the window that have been effectively modified are redrawn. Incrementality becomes even more important in the case of *distributed environments*, as it is the case with **CENTAUR**: the various tools communicate by using networking facilities, and it is necessary to minimize the volume of data that is transmitted throughout the network.

1.3 Organization of the report

In section 2, we present the definition of the **F** language. This includes a specification of syntax, pretty-printing, static and dynamic semantics. The animation that we implemented in the evaluator will be a starting point for a discussion of incrementality.

In section 3, we present the issues related to parallelization. We start by describing the preliminary analysis phase, whose main objective is the calculation of dependences graphs. On the basis of these graphs, we demonstrate, on a small example, how program slices can be used in the context of program maintenance. We then show how these graphs are exploited for the usual transformations involved in the parallelization of programs. Finally, we initiate a discussion on the topic of incremental semantic evaluation.

In section 4, we present two models of user interface. In the first one, transformations are triggerred by menus, and the various versions of a program are presented in different views which are coordinated by a network. In the second one, transformations are triggered by clicking directly on program pieces, and they are performed in-place, within a single editor view.

In section 5, we present an evaluation of the prototype, mainly from the point of view of the implementor, and directions for future research.

2 The definition of the F language

2.1 Defining the syntax using METAL

Within **CENTAUR**, we define a language's grammar mainly by its *abstract syntax*, which consists of nodes, called *operators*, and node types, called *phyla*. An operator is defined by its name and the phyla of its descendents. A phylum is defined by its name and the set of operators it may contain.

In **F**, we have considered the `assign`, `if`, `where`, `do`, `goto`, `allocate` and `deallocate` instructions. Any instruction may have a label. A `do` statement has a type which is either sequential or parallel.

```
chapter INSTRUCTIONS
      stm_s  -> LAB_STM * ... ;
      stm  -> LABEL STM ;
      empty  -> implemented as SINGLETON ;
      where  -> EXP ASSIGN ;
      if  -> EXP STM_S STM_S ;
      do  -> LOOP_TYPE INT VAR SEXP STM_S ;
      goto  -> INT ;
      assign  -> EXP EXP ;
      allocate  -> ARRAYS_DECL ;
      deallocate  -> ID_S ;
      STM_S  ::= stm_s ;
      LAB_STM  ::= stm ;
      LABEL  ::= int empty ;
      ASSIGN  ::= assign ;
      STM  ::= assign where if do goto allocate deallocate ;
end chapter ;
```

In another chapter of the METAL definition of **F**, we find the syntax of expressions. As we can see it below, basic data types are `int` and `logical`.

```
chapter EXPRESSIONS
      sexp_s  -> SEXP * ... ;
      slice  -> EXP EXP EXP ;
      binexp  -> EXP BINOP EXP ;
      unexp  -> UNOP EXP ;
      binop  -> implemented as STRING ;
      unop  -> implemented as STRING ;
      array_ref  -> VAR SEXP_S ;
      id  -> implemented as IDENTIFIER ;
      int  -> implemented as INTEGER ;
      logical  -> implemented as STRING ;
      logical_s  -> LOGICAL * ... ;
      SEXP_S  ::= sexp_s ;
      SEXP  ::= EXP slice ;
      EXP  ::= id int logical array_ref binexp unexp ;
      BINOP  ::= binop ;
      UNOP  ::= unop ;
      VAR  ::= id empty ;
      INT  ::= int ;
      LOGICAL  ::= logical ;
end chapter ;
```

In **F**, names can be simple variables or (multi-dimensional) array references. Expressions can be unary or binary; one has chosen to have a single operator `unop` and a single operator `binop` in order to minimize the number of rules to

write in the semantics analysis phase, where most treatments do not depend on which precise operator we are dealing with. Slices which have initial, final and increment expressions are used for specifying the control of do loops, or a reference to an array slice.

2.2 Specifying pretty-printing rules using PPML

The next step in the language definition is the specification of decompilation rules using the **PPML** formalism. For a given abstract syntax, such as the one of **F**, it is possible to specify several decompilers. For example, one might want to obtain a different decompilation in a **CENTAUR** editor window (a ctedit) or in a text file.

A pretty-printer is made up of rules having the form:

pattern — *format*

where *pattern* indicates which abstract syntax operators a rule recognizes, and *format* describes the corresponding layout in terms of a *box language*. Patterns are tried in the order they appear in the definition, and the first one that matches is applied. A box may contain terminals, variables or other boxes. Variables represent recursive calls to the pretty-printer. A box is typed by an *operator* which specifies how its elements will be aligned. For example, the `<h>` operator concatenates elements horizontally, and the `<v>` operator vertically.

We now present the decompilation rules for the `INSTRUCTIONS` chapter of the above **METAL** definition.

```
chapter INSTRUCTIONS

   stm_s(**stms) -> [<v> (**stms)];

   stm(empty, *stm) -> [<h> *stm];

   stm(*label, *stm) -> [<h> [<vm> in class = label :  *label] *stm];

   #if(*exp, stm_s(*stm), stm_s()) where
    *stm in {stm(empty, assign(*v, *e)), stm(empty, goto(*l))} ->
      [<h> in term class = key-word : [<h> "IF" " (" *exp ") " *stm]];

   #if(*exp, *stms1, stm_s()) ->
       [<v>
        in term class = key-word : [<v tab,0> [<hv 1> "IF" *exp "THEN"]
                                              *stms1]
        [<h> in term class = key-word : "END IF"]];

   #if(*exp, *stms1, *stms2) ->
       [<v>
        in term class = key-word :
         [<v 0,0>
          [<v tab,0> [<hv 1> "IF" *exp "THEN"] *stms1]
          [<v 0,0>
           in term class = key-word : [<v tab,0> "ELSE" *stms2]
           [<h> in term class = key-word : "END IF"]]]];

   do(*type, *label1, *id, slice(*exp1, *exp2, *exp3), stm_s(**body, *stm))
    where *stm in {stm(*label2, nothing)} ->
       [<v>
        in term class = key-word :
         [<v>
          [<v tab,0> [<hv 0> if *type in {loop_type 'par'}
                             then [<h> "PARALLEL "]
                             end if
                             "DO " in class = label : [<h> *label1] " " *id
                             " = " *exp1 ", " *exp2 step-context::*exp3]
                     [<v> (**body)]]
          [<h> *stm in class = key-word : "CONTINUE"]]];

   goto(*int) ->
       [<h> in class = key-word : "GOTO " in class = label : [<h> *int]];

   #where(*exp, *assign) ->
       [<h> in term class = key-word :
                  [<h> "WHERE" " (" *exp ") " *assign]];

   assign(*var, *exp) -> [<hv 1,2,0> *var "=" *exp];

   allocate(*arrays) ->
       [<h> in term class = key-word : "ALLOCATE" " ( " *arrays " )"];

   deallocate(*ids) ->
       [<h> in term class = key-word : "DEALLOCATE" " ( " *ids " )"];

   nothing -> [<h> ""];

end chapter;
```

Let us make the following remarks on the above definition:

- The three rules for the **if** statement allow us to obtain the following presentation, going from the more specific to the more general:

```
IF (X.EQ.3) X = X + 1

IF (X.EQ.3) THEN
   Y = 4
   Z = 5
END IF

IF (X.EQ.3) THEN
   Y = 4
ELSE
   Y = 5
END IF
```

- As it would be the case with FORTRAN, we wanted labels to appear in column 1, and statements to appear in columns 7 to 72:

```
          DO 100 I = 1, N
             A(I) = 0
100       CONTINUE
```

  This is contradictory with the principle of having nested rectangular boxes, and it had to be dealt with in an *ad-hoc* fashion in **PPML**, using the <vm> operator.

```
stm(*label, *stm) ->
            [<h> [<vm> in class = label :  *label]
                 *stm];
```

- In the *format* part of the rule, we may assign resource names to tokens, so that they are printed according to font and color resource values which can be found in an external resource file. As an example, consider the goto rule:

```
goto(*int) ->
      [<h> in class = key-word : "GOTO "
           in class = label : [<h> *int]];
```

  and the corresponding entries in the resource file:

```
*visual-color*formalism-F*class-label.foreground:\
        RoyalBlue3
*visual-color*formalism-F*class-label.font:\
        -*-times-bold-i-*-*-12-*-*-*-*-*-*-*
*visual-color*formalism-F*class-key-word.foreground:\
        SeaGreen
*visual-color*formalism-F*class-key-word.font:\
        -*-helvetica-bold-o-normal-*-12-*-*-75-*-*-*-*
```

2.3 Specifying the semantics in TYPOL

2.3.1 Static semantics and the type-checker

The type-checker is used to construct a symbol table of a **F** program, and to verify the various semantic restrictions of the language, as shown in Figure 2.

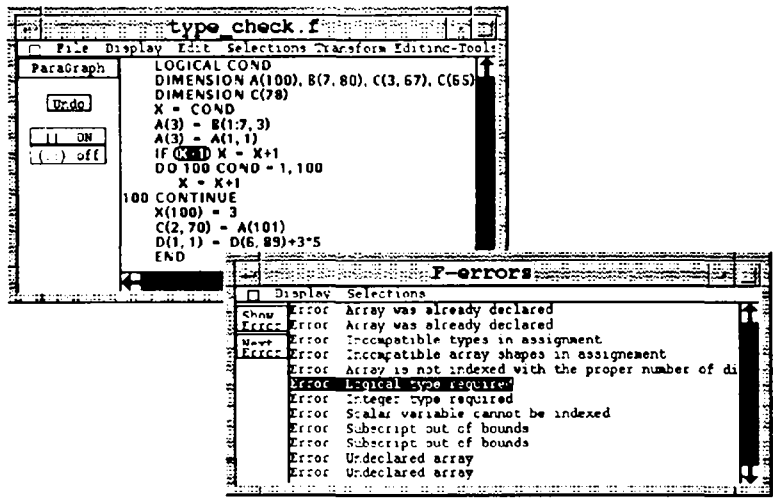

Figure 2: *Type checking of the tc program*

The symbol table is described in **METAL**.

```
chapter SYMBOL_TABLE
     symtab  -> TEMPORARY_INDEX VARIABLE_S ;
     variable_s  -> VARIABLE * ... ;
     variable  -> VAR TYPE ;
     type  -> DATA_TYPE SHAPE ;
     int_type  -> implemented as SINGLETON ;
     logical_type  -> implemented as SINGLETON ;
     scalar  -> implemented as SINGLETON ;
     array_shape  -> INT DIMENSION_S ;
     dimension_s  -> DIMENSION * ... ;
     undef_dimension  -> implemented as SINGLETON ;
     SYMTAB  ::= symtab ;
     VARIABLE_S  ::= variable_s ;
     VARIABLE  ::= variable ;
     TYPE  ::= type ;
     DATA_TYPE  ::= int_type logical_type ;
     SHAPE  ::= scalar array_shape ;
     DIMENSION_S  ::= dimension_s ;
     DIMENSION  ::= slice undef_dimension ;
end chapter ;
```

We have to note that the description of the symbol table had to be kept in the same `F.metal` file as the language definition. This is because, at this point in time, it is not possible to reference an external language in a **METAL** definition

file. As it is now the case in **PPML** when we specify pretty-printers, we would like to have the possibility to extend a language definition without having to change the original file. The type-checker, written in **TYPOL**, constructs a symbol table and then performs the necessary checks. The symbol table is finally attached as an *annotation* to the root of the program tree, as it will later be needed by the parallelizer.

```
set type_check is
decls_type_check(()variable_s ⊢ DECLARATIONS → vars)  &
stms_type_check(vars ⊢ STATEMENTS → vars')
-------------
⊢    DECLARATIONS
     STATEMENTS
     END ;
        do sendannot(subject, 'symbol_table', symtab(0, vars'));
end type_check;
```

Let us consider how one can verify that the condition in an **if** statement is a scalar value of type **logical**. This is defined in the **stm_type_check** set of rules, where the conclusions we want to establish are of a form which is governed by the following judgement:

```
VARIABLE_S |- STM -> VARIABLE_S
```

This simply says that we will type check a statement, starting with an initial symbol table, and produce an eventually modified symbol table, considering the fact that, in **F** as in FORTRAN, undeclared variables may appear anywhere in the program text.

```
set stm_type_check is
judgement VARIABLE_S ⊢ STM → VARIABLE_S;

exp_type_check(vars ⊢ CONDITION → TYPE, vars')
& logical_scalar_type_check(s(subject, 1), TYPE)
& stms_type_check(vars' ⊢ STATEMENTS1 → vars'')
& stms_type_check(vars'' ⊢ STATEMENTS2 → vars''')
-------------
vars
   ⊢ IF COND THEN
        STATEMENTS1
     ELSE
        STATEMENTS2
     END IF
        →
        vars''' ;

...
end stm_type_check;
```

The verification that the **CONDITION** is a scalar value of type logical is performed by the following set of rules.

```
set logical_scalar_type_check is
judgement (path, TYPE);

(_, type(logical_type(), scalar())) ,

(EXP, type(DATA_TYPE, scalar())) ,
        provided DATA_TYPE ≠ logical_type();
        do report(EXP, "type_check", "F", 4);

(EXP, type(logical_type(), array_shape(_, _))) ;
        do report(EXP, "type_check", "F", 5);

(EXP, type(DATA_TYPE, array_shape(_, _))) ;
        provided DATA_TYPE ≠ logical_type();
        do report(EXP, "type_check", "F", 6);
end logical_scalar_type_check;
```

We assume the following error messages are kept in the appropriate **type_check.mess** file:

```
...
4  "Error: Logical type required"
5  "Error: Scalar type required"
6  "Error: Logical scalar type required"
...
```

The error reporter is able to select a badly typed expression through the **s(subject, int 1)** *path* in the **if** rule, since this is a path from the root of the program tree to the **CONDITION** node.

2.3.2 Dynamic semantics and the evaluator

The dynamic semantics of **F** was then specified in **TYPOL**. The execution of a **F** program is fairly simple to describe. We chose to characterize it by the effect of the program on the memory, which we have defined in **METAL** in the following way:

```
chapter MEMORY
   memory  -> VAR_S ARRAY_S ;
   array_s  -> ARRAY * ... ;
   array  -> VAR INT INT_S INT VALUE_S ;
   var_s  -> ASSOC * ... ;
   assoc  -> VAR VALUE ;
   MEMORY  ::= memory ;
   ARRAY_S  ::= array_s ;
   VAR_S  ::= var_s ;
   ARRAY  ::= array ;
   ASSOC  ::= assoc ;
end chapter ;
```

The memory associates scalar or array variables to values. In the case of a (multi-dimensional) array, values are kept in a linear list, together with the necessary dope vector. Values may be of type `int` or `logical`. We now consider the **TYPOL** description of the semantics of a **F** statement. This is defined in the `stm_eval` set of rules, where the conclusions we want to establish are of the form:

```
MEMORY |- STM -> MEMORY;
```

This simply says that we will consider the state of memory before and after execution of a statement. Let us have a look at the rules for the `do` statement. In **F** as in FORTRAN, we have to consider two cases: the `do` loop is either active or inactive, depending on the values of the loop parameters. We will then use the auxiliary sets of rules `active_loop` and `inactive_loop`, in order to distinguish between these two cases.

Therefore, the first rule, which applies when the loop is active, says that the body of the loop is executed for the initial value of the loop parameter, and that it is then executed for the remaining values of the loop parameter. The second rule, which applies when the loop is inactive, says that the loop parameter has to be assigned a value.

```
set stm_eval is
judgement MEMORY ⊢ STM → MEMORY;

stm_eval(m' ⊢ ID = INIT_PARAM → m'') & stms_eval(m'' ⊢ BODY → m''') &
stm_eval(m'''
        ⊢ DO LABEL ID = INIT_PARAM + INCR_PARAM, TERM_PARAM, INCR_PARAM
             BODY
          END DO
             →
             m'''')
-------------------
m
  ⊢ DO LABEL ID = EXP1, EXP2, EXP3
       BODY
    END DO
       →
       m'''' ;
       provided active_loop(m
                              ⊢ DO LABEL ID = EXP1, EXP2, EXP3
                                   BODY
                                END DO
                                   →
                                   INIT_PARAM, TERM_PARAM, INCR_PARAM, m');

stm_eval(m' ⊢ ID = INIT_PARAM → m'')
-------------------
m
  ⊢ DO LABEL ID = EXP1, EXP2, EXP3
       BODY
    END DO
       →
       m'' ;
       provided inactive_loop(m
                              ⊢ DO LABEL ID = EXP1, EXP2, EXP3
                                   BODY
                                END DO
                                   →
                                   INIT_PARAM, m');

...
end stm_eval;
```

The above definition is, of course, recursive. Now considering the translation that is made from **TYPOL** to PROLOG, we have to realize that this is a space-consuming fashion of implementing loops, since all iterations are kept on the PROLOG stack, making provision for an eventual back-track[3]. However, it is possible to specify that the evaluation a **TYPOL** sequent has to be done in a *functional* mode, that is without any possibility of back-track, by using the **once** construct. In our case, we could re-write the first rule as follows:

[3] Although extremely costly, this is a powerful mechanism. In the case of an evaluator, for example, one can imagine executing the program in reverse order, undoing the effect of instructions on the memory state. Such experiments have been done, using the TYPOL generic debugger.

```
stm_eval(m' ⊢ ... → m'') & stms_eval(m'' ⊢ BODY → m''') &
once(stm_eval(m'''
                ⊢ DO LABEL ID = INIT_PARAM ; INCR_PARAM, TERM_PARAM, INCR_PARAM
                    BODY
                  END DO
                    →
                    m''''))
-------------------
m
   ⊢ DO LABEL ID = EXP1, EXP2, EXP3
       BODY
     END DO
       →
       m'''' :
       provided active_loop(...);
```

The only difficult point in the semantics definition of **F** is the specification of the **goto** statement. Gotos are handled via the notion of *continuation*. Execution basically has two modes, depending on the continuation value c:

- $c = 0$, normal execution of statements,
- $c \neq 0$, execution of a goto statement, that is search for the label of value c by traversing the rest of the program tree, and going back to the beginning of the program if the label is not found.

Following the abstract interpretation model of Cousot [Cousot77], the execution of a **F** program now consists in getting to a fixpoint, where two conditions are met:

- we are at the end of the program,
- and the continuation is equal to zero.

At this point, the memory has reached its final state. This is described by the following rules

```
set eval is
judgement MEMORY ⊢ PROG → MEMORY.

decls_eval(m ⊢ DECLARATIONS → m') & stms_eval(m', int: 0 ⊢ STATEMENTS → m'', c) &
fixpoint(STATEMENTS, m'', c → m''')
-------------------
m
   ⊢    DECLARATIONS
        STATEMENTS
        END
        →
        m''' :

   set fixpoint is
   judgement (STATEMENTS, MEMORY, CONTINUATION → MEMORY);

   (_, m, int: 0 → m) .

   stms_eval(m, int: c ⊢ STATEMENTS → m', c') & fixpoint(STATEMENTS, m', c' → m'')
   -------------------
   (STATEMENTS, m, int: c → m'') :
           provided c ≠ 0;
   end fixpoint;
end eval;
```

It now remains to modify the original **stm_eval** set of rules in order to take into account the value of the continuation:

```
set lab_stm_eval is
judgement MEMORY, CONTINUATION ⊢ LAB_STM → MEMORY, CONTINUATION;

stm_eval(m, int: 0 ⊢ STM → m', c')
---------------------------------
m, int: 0 ⊢ STM → m', c' :

stm_eval(m, int: c ⊢ STM → m', c')
---------------------------------
m, int: c ⊢ STM → m', c' :
        provided c ≠ 0;

stm_eval(m, int: 0 ⊢ STM → m', c')
---------------------------------
m, LABEL ⊢ LABEL STM → m', c' :

stm_eval(m, int: LABEL ⊢ STM → m', c')
---------------------------------
m, int: LABEL ⊢ LABEL' STM → m', c' :
        provided LABEL ≠ LABEL';
end lab_stm_eval;
```

```
set stm_eval is
judgement MEMORY, CONTINUATION ⊢ STM → MEMORY, CONTINUATION;

m, int: 0 ⊢ GOTO LABEL → m, LABEL :

exp_eval(m ⊢ EXPRESSION → VALUE, m') & value_set(m' ⊢ ID = VALUE → m'')
---------------------------------
m, int: 0 ⊢ ID = EXPRESSION → m'', int: 0 :

m, int: c ⊢ ID = EXPRESSION → m, int: c :
        provided c ≠ 0;
end stm_eval;
```

We finally consider the possibility of animating the execution of a **F** program in the following way:

- the instruction that is currently executed should be selected,
- the memory values that are changed by an **assign** statement should also be selected.

This is demonstrated in Figure 3.

The first point, that is selecting the current instruction, is quite easy to achieve. In the **TYPOL** rule concerning the execution a labelled statement, we add a **do** part, which is executed after the corresponding rule has been proved. In this **do** part, we merely call the LISP function **select_subject_in_program** with an argument which is the **subject** of the rule, that is the program subtree on which the rule has been applied.

```
stm_eval(m, int: 0 ⊢ STM → m', c')
---------------------------------
m, int: 0 ⊢ STM → m', c' :
        do redisplay(subject, 'select subject_in_program');
```

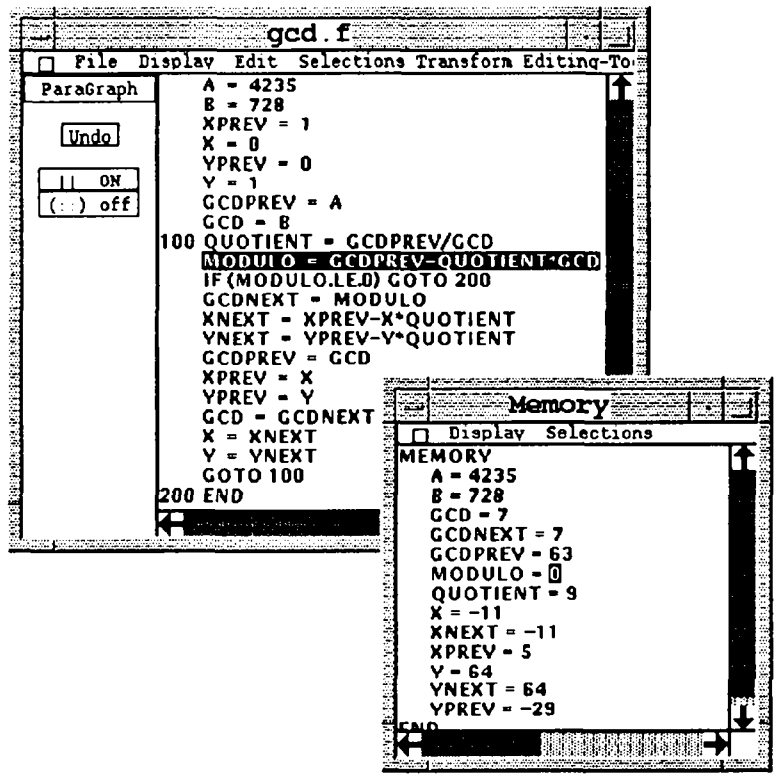

Figure 3: *Animation of the gcd program*

Now on the LISP side, we have:

```
(de select_subject_in_program (subject)
;-------------------------
    (when *animation-requested*
        ({ctedit}:update-tree *program-ctedit* '{current} subject)
        ({ctedit}:scroll-to-selection *program-ctedit* '{current})
        ({ctedit}:incremental-redisplay *program-ctedit*)
    )
)
```

The second point, that is selecting memory values as they change, raises a more difficult question which is due to the fact that, in the current implementation of **CENTAUR**, we manipulate two kinds of objects:

- abstract syntax trees, which are *LISP objects* handled by the VTP (Virtual Tree Processor) component of **CENTAUR**. The memory which is displayed in a **CENTAUR** editor window is such a LISP object.
- terms manipulated when proving a **TYPOL** specification, which are *PROLOG objects*. The memory which we have to consider while evaluating a **F** program is such a PROLOG object.

Therefore, when a memory value is updated in the evaluator, we first have to convert the PROLOG term into a LISP object. This is done as follows by the `sendtree` predicate:

```
set redisplay_memory is
judgement (MEMORY);

sendtree(lisp: 'memory_object', MEMORY)  &  withsubject(lisp: 'memory_object':  display() ⊢ MEMORY))
_____
(MEMORY) :

   set display is
   judgement ⊢ MEMORY;

   ⊢ memory :
            do redisplay(subject, 'redisplay_memory');
   end display;
end redisplay_memory;
```

Now on the LISP side, we have the problem of dealing with two different memory objects, the old one and the new one. We do not want to simply replace the old memory tree by the new one, because this would have a flashing effect in the memory window. In fact, we want to benefit from the incremental redisplaying facility of the **PPML** engine. Therefore, we start by doing an incremental change of the old tree by the new one[4]. This is achieved by the `{variable}:incremental-change` function which starts by computing the difference between the old and new trees by using the `{tree}:diff` primitive of **CENTAUR**. `{tree}:diff` returns a *modification path*, which is then applied to the old tree in order to get the new one, using the `{path}:do-modif` primitive. `{variable}:incremental-change` finally returns the modification path. Once this is done, we just need to redisplay the memory window, while selecting the modified parts, using the `{path}:show-modif-on-new` primitive.

[4] Based on this experiment, we suggested using the `{variable}:incremental-change` primitive instead of `{variable}:change` within the Centaur system itself.

```
(de redisplay-memory (new-memory-tree)
;-------------------
  (when *animation-requested*
    (lets ((variable ({ctedit}:variable *memory-ctedit*))
           (old-memory-tree ({variable}:root variable))
           (selection ({ctedit}:find-selection *memory-ctedit* '{current}))
           (modifications
               ({variable}:incremental-change
                    variable old-memory-tree new-memory-tree)))
           ({selection}:change
               selection
               ({path}:show-modif-on-new modifications))
           ({ctedit}:update-select
               *memory-ctedit*
               ({ctedit}:find-select *memory-ctedit* '{current}))
           ({node-object}:emit
               variable '{variable}:variable:selection-modified '{current})
    )
  )
)

(de {variable}:incremental-change (variable old-tree new-tree)
;-------------------------------
   (let ((modifications ({tree}:diff old-tree new-tree))
         (path ({tree}:path old-tree)))
        ({path}:for-all-leaf modifications
             (lambda (leaf type)
                (selectq type
                     (c
                       ({tree}:super-erase ({leaf}:new leaf)))
                     (i
                       ({tree}:super-erase ({leaf}:new leaf))))))
        ({path}:do-modif modifications old-tree)
        ({path}:multiply path modifications)
        ({variable}:update-selections variable path)
        ({variable}:add-modif variable path)
        modifications
    )
)
```

3 Analysis and transformations

3.1 Introduction

The main purpose of the **PARAGRAPH** prototype is to perform the parallelization or vectorization of **F** programs. The list of the transformations we have implemented is the following:

- loop parallelization,
- loop vectorization,
- loop distribution,
- if conversion,
- scalar expansion,
- loop fusion,
- loop splitting,
- loop peeling.

All of these are described in [Zima91].

We assume that type-checking has been performed as indicated in the previous chapter, and that the symbol table is attached as an annotation to the root of the program tree, as it will be needed at the time we have to generate new variables.

Before performing transformations, we will go through a preliminary phase, where the control structure of the program will be analyzed, and dependence graphs will be computed.

3.2 Preliminary analysis of programs

3.2.1 Control flow analysis

We have chosen to do a minimal amount of work with respect to the analysis of the control structure of **F** programs.

The only treatment we perform consists in attaching to every **goto** statement of a program an annotation which is a *path* to its destination. Moreover, for every statement of the program, we attach an annotation which contains the list of labels which are either defined or referenced within the statement, as this will be needed when transforming the flow of control of programs, for example when doing if-conversions[5].

3.2.2 Reaching definition analysis

We now come to the problem of data flow analysis. Following Zima [Zima91], and many others, we will consider so-called *monotone data flow systems*, whose

[5] We must admit that, in the context of a real tool, there would be a necessity to compute a *control flow graph* whose node would be basic blocks and whose edges would correspond to transfer of control between basic blocks.

solution can be found by an iterative algorithm[6]. Data flow information associated to each program node can be obtained by propagation through the control flow graph. We start with an initial value for each solution set, and we repeat the treatment until a fixpoint is reached.

We first consider the problem of computing the *reaching definitions* of a statement. Let v be a variable of a program, we will say that a statement S of this program will be a *definition* of v if it is an assignment to this variable and we will say that a statement S *uses* the variable v if, during the execution of S, the value of v may be read. We now want to compute, for every usage of a *scalar* variable in a program, the definitions of this variable that can reach this usage. Those will be the definitions for which there exists an execution path in the program flow graph which is *definition-free* with respect to the given variable.

The reaching definitions graph can be described in **METAL** as follows[7]:

```
chapter REACHING_DEFINITIONS
    reaching  -> RDEF * ... ;
    rdef  -> VAR  PATH_S ;
    path_s  -> PATH * ... ;
    rd_graph -> RD_ENTRY * ... ;
    rd_entry -> PATH RD ;
    RD  ::= reaching ;
    RDEF  ::= rdef ;
    PATH_S  ::= path_s ;
    RD_GRAPH :: rd_graph ;
    RD_ENTRY ::= rd_entry ;
end chapter ;
```

A `rd_graph` is a list of `rd_entries`. Each `rd_entry` contains a `path`, which refers to a statement we are considering, and list of `rdefs` which are the reaching definitions associated to this statement. A reaching definition consists of a variable name and a list of `paths` to the definition statements that have to be considered for that variable.

The following set of **TYPOL** rules computes the reaching definition graph `rdg` of a given program.

[6] We could also describe the resolution of these problems as an abstract interpretation [Cousot77].

[7] This is not quite the way it appears in our **METAL** file, for the reason that it is not possible to reference the phylum PATH, which belongs to the predefined PSP formalism, within our metal file. We then had to define the phyla RD_GRAPH and RD_ENTRY within the **TYPOL** code, which is possible. We must admit that this is not a satisfactory solution, and that **METAL** should be more modular that it is.

```
set reaching_definitions_analysis is
judgement ⊢ PROG:

fixpoint(empty_graph0, 0rd_graph, _ ⊢ STATEMENTS → rdg, rd)
---------------------------------------------------------
⊢   DECLARATIONS
    STATEMENTS
    END ;
      do annot_reaching(rdg'):

  set fixpoint is
  judgement RD_GRAPH, RD_GRAPH, RD ⊢ STATEMENTS → RD_GRAPH, RD:

  rdg, rdg, rd ⊢ STMS → rdg, rd :

  stms_reaching(rdg', (;reaching ⊢ STATEMENTS → rdg'', rd'')  &
  fixpoint(rdg', rdg'', rd'' ⊢ STATEMENTS → rdg''', rd''')
  ---------------------------------------------------------
  rdg, rdg', rd' ⊢ STATEMENTS → rdg''', rd''' :
          provided rdg ≠ rdg':
  end fixpoint;
end reaching_definitions_analysis;
```

A fixpoint will be reached when we have done an iteration on the program tree, and no new definition is added to the current reaching definitions graph. This is when the first rule of the set `fixpoint` applies. Once a fixpoint has been reached, the reaching definitions of a statement which correspond to variables that are effectively used within that statement are attached as annotations to the statement node. This is the role of the set `annot_reaching`.

Let us now consider some of the rules that deal with individual statements, as the program list of statements is examined:

```
set stm_reaching is
judgement RD_GRAPH, RD ⊢ STM → RD_GRAPH, RD;

update_node_reaching(subject, rdg, rd, rdg', rd')
& reaching_init(), rd', rdef(V, 0, (subject, rd'')path_s))
-----------
rdg, rd ⊢ V = EXPRESSION → rdg', rd'' ;

update_node_reaching(subject, rdg, rd, rdg', rd')
& stms_reaching(rdg', rd' ⊢ STATEMENTS1 → rdg'', rd'')
& stms_reaching(rdg'', rd' ⊢ STATEMENTS2 → rdg''', rd''')
& rd'''' = rd'' ∪ rd'''
-----------
rdg, rd
   ⊢ IF CONDITION THEN
        STATEMENTS1
     ELSE
        STATEMENTS2
     END IF
        →
        rdg''', rd'''' ,

update_node_reaching(subject, rdg, rd, rdg', rd')
& getannot(subject, 'destination', DESTINATION)
& update_node_reaching(DESTINATION, rdg', rd', rdg'', ⊃)
-----------
rdg, rd ⊢ GOTO LABEL → rdg'', ()reaching ;

...
end stm_reaching;
```

The role of the `update_node_reaching(subject, rdg, rd, rdg', rd')` premise, which appears in each rule, is to update the set of reaching definitions associated to the given statement, by adding definitions which where not yet known. We then have a specific treatment for each statement. The `assign` statement has the effect of killing the definitions of the variable being assigned to. For the `if` statement, we first have to consider `STATEMENTS1` and `STATEMENTS2`, and then we have to perform the union of the sets we obtain from the two branches of the `if`. For the `goto` statement, we have to do an `update_node_reaching` on the destination node of the `goto`.

Using this information, we were able to specify two basic transformations, which we will not further describe in this report:

- *constant propagation*, which allows to detect when a reference to a variable can be replaced by a constant value. This was coupled to the *simplification* of expressions and statements (both `2+J+3` and `DO 100 I = 3,2 ...` can be simplified). Together, constant propagation and simplification can be repeatedly applied until a fixpoint is reached.
- *dead code removal*, which consists in removing assignment statements which define a value which will never be used. The algorithm that we used is organized as follows:
  - we first mark statements that are useful
    (again, this amounts to solving a monotone data flow system).
  - we then remove the statements
    which have not been marked in the previous step.

It appears that both of these transformations will be quite useful in the context of parallelization. This is because, in many circumstances, we generate expressions which can be simplified, or code which is in fact useless.

3.2.3 Dependence analysis

The dependence equation

The parallelization of a program is achieved by relaxing the order in which its statements are executed. Of course, we are authorized to change the order of execution of statements only in the case this does not change the semantics of the program; therefore we will be interested in the *dependence relation* between pairs of statements.

More precisely, considering two statements S and S' of a program, such that S is executed before S', we will say that there exists:

- a *true dependence* from S to S', if S defines the value of a variable v that is used by S',
- an *anti dependence* from S to S', if S references a variable v which is defined by S',
- an *output dependence* from S to S', if S and S' both define the value of the same variable v.

Our main interest will be the parallelization of program loops of the following form:

```
                DO L1   I1(= It1)  =  t1(= tt1),  u1(= ut1)
                DO Lm   Im(= Itm)  =  tm(= ttm),  um(= utm)
                    DO Lm+1   Im+1 = tm+1,  um+1
                    DO Ln     In = tn,  un
S:                      A(f(I)) = ...
Ln                  CONTINUE
Lm+1                CONTINUE
                    DO Ltm+1   Itm+1 = ttm+1,  utm+1
                    DO Ltn'    Itn' = ttn',  utn'
S':                     ... = ... A(ft(It)) ...
Ltn'                CONTINUE
Ltm+1               CONTINUE
Lm              CONTINUE
L1              CONTINUE
```

In order to determine which statement, of S or S' is executed first, we will have to consider two iteration vectors of the form $\vec{I} = (I_1, ..., I_n)$ and $\vec{It} = (It_1, ..., It_{n'})$.

We now have to establish the dependence relation in the case S and S' are references to the same array, A in the above example. In the context of **PARAGRAPH**, we will be interested in solving the dependence equation only in the case subscript expressions are linear functions in the do variables and

the loop bounds are known integer constants. Therefore, in the general case, we will consider the following system of equations:

$$f(\vec{I}) = a_0 + \sum_{j=1}^{n} a_j I_j$$
$$f'(\vec{I'}) = b_0 + \sum_{j=1}^{n'} b_j I'_j$$
(with a_j, $0 \leq j \leq n$; b_j, $0 \leq j \leq n'$: integer constants)
($a_0 + \sum_{j=1}^{n} a_j i_j - (b_0 + \sum_{j=1}^{n'} b_j i'_j) = 0$: the dependence equation)
(i_j, $t_j \leq i_j \leq u_j$, $1 \leq j \leq n$; i'_j, $t'_j \leq i'_j \leq u'_j$, $1 \leq j \leq n'$: the unknowns)

For example, if we have the following program:

```
DO I=0,10
   DO J=0,10
      S:  ... A( 2 * I+ J ) ...
      S': ... A(-I + 2 * J - 21 ) ...
   END DO
END DO
```

We will have:

$(m = 2;\ n = n' = m;\ a_0 = 0;\ a_1 = 2;\ a_2 = 1;\ b_0 = -21;\ b_1 = -1;\ b_2 = 2)$
($2i_1 + i_2 - (-21 - i'_1 + 2i'_2) = 0$: the dependence equation)
(i_1, i_2, i'_1, i'_2, $0 \leq i_1$, $i'_1 \leq 10$; $0 \leq i_2$, $i'_2 \leq 10$: the unknowns)

The dependence tests

In the case the subscript expressions to be considered contain at most one do variable, that is:

$$f(\vec{I}) = a_0 + a_k I_k$$
$$f'(\vec{I'}) = b_0 + b_k I'_k$$
(with $k \in [1 : m]$ and a_0, a_k, b_0, b_k: integer constants)
($a_k i_k - b_k i'_k = b_0 - a_0$: the dependence equation)
($i_k, i'_k, t_k \leq i_k \leq u_k$, $t'_k \leq i'_k \leq u'_k$: the unknowns)

The above system of equation will be solved using the *separability test* whose detailed description can be found in [Zima91]. The separability test has the advantage of being an exact test; that is we will be able to determine the existence of solutions to the given system of equation, and to characterize them, when they exist.

In the more general case, we will apply:

- the *gcd test*, which is a necessary condition for dependence, since it ignores the region where the do variables are defined.
- the *Banerjee test*, which is also a necessary condition for dependence, since solutions when they are proved to exist, may be either integer or real.

The description of both of these tests can be found in [Banerjee88]. Within the context of **PARAGRAPH**, we have not investigated further the topic of dependence tests. The interested reader can refer to [Eisenbeis92] for a description of much more general algorithms for solving integer linear programming problems in the context of dependence analysis.

The PARAGRAPH implementation

A dependence graph will be associated to all **do** statements of a **F** program which contain **assign** and **where** statements only. The nodes of this graph are the **do** statements which are numbered 1, 2, ... and the edges represent dependences. This is described by the following **METAL** specification:

```
chapter DEPENDENCE_GRAPH
      dg  ->  EDGES_S ;
      dependence_s  -> DEPENDENCE * ... ;
      edges_s  -> EDGE_S * ... ;
      edge_s  -> EDGE * ... ;
      edge  -> INT DEPENDENCE ;
      dependence  -> EXP EXP KIND DIRECTION_VECTOR SOLUTION ;
      solution  -> SOLUTION_SET INT INT ;
      direction_vector_s  -> DIRECTION_VECTOR * ... ;
      direction_vector  -> THETA * ... ;
      theta  -> implemented as STRING ;
      dependence_kind  -> implemented as STRING ;
      strong_sep_solution_set  -> EXP ;
      weak_sep_solution_set  -> VAR EXP EXP INT INT ;
      DG  ::= dg ;
      DIRECTION_VECTOR  ::= direction_vector ;
      DIRECTION_VECTOR_S  ::= direction_vector_s ;
      THETA  ::= theta ;
      DEPENDENCE ::= dependence;
      SOLUTION  ::= solution empty ;
      SOLUTION_SET  ::= strong_sep_solution_set weak_sep_solution_set ;
end chapter ;
```

A dependence graph is a list of lists of **edge_s**. The source statement number of an **edge** node of such a list is implicitly given by the position of the **edge_s** node in the **edge_s** list. For example,

```
edges_s[edge_s[edge(2, ..., ...)  ...]  ...]
```

means that there is a dependence from statement numbered 1 to statement numbered 2.

A dependence is characterized in the following way:

```
dependence -> EXP EXP KIND DIRECTION_VECTOR SOLUTION ;
```

The first two expressions are the subscript expression that caused the dependence. **KIND** is the dependence kind, i.e. a *true*, *anti* or *output* dependence. The direction vector specifies the relationship between the pair of iteration vectors that had to be considered. Each component of a direction vector will be a value $\theta \in \{<, =, >, *\}$. $*$ means that the respective ordering of the loop variable values is not defined.

The **solution** of a dependence system of equations is also kept within this data structure in the case it is known. This will be the case, in particular, when the separability test is applicable. For example, if we consider the **weak** separability case [Zima91], the solution set is characterized in the following way:

```
solution  -> INT INT_OR_EMPTY SOLUTION_SET ;
weak_sep_solution_set  -> VAR EXP EXP INT INT ;
SOLUTION_SET  ::= weak_sep_solution_set ;
```

The **INT**s sons of a **solution** node are the minimum and maximum dependence distance, that is the minimum and maximum distance between the loop variable values. The solution set in the case of weak separability is characterized as follows:

$$\{(I, I') \mid I = EXP_1(t),\ I' = EXP_2(t),\ INT_1 \leq t \leq INT_2\}$$

All dependences are computed as a **F** program is read in a CENTAUR editor window. Dependence graphs are attached as annotations to **do** statements. They can be visualized by using the graph displaying package of **CENTAUR** [Lehors92], as in Figure 4[8].

Let us now explain how these three different views (program, graph and message) are produced. In fact, they all correspond to a specific decompilation the same abstract syntax tree, which is the internal representation the original source program, annotated with dependence graphs.

Using **PPML**, we can define pretty printers other than the standard F pretty printer used to decompile the abstract syntax tree in a program view. We will use this facility for decompilation in the graph view and the dependences view.

We have defined a *graph* F pretty-printer which decompiles the **dg** nodes attached to **do** loops into a LISP function which can be loaded and evaluated in order to display the graph in the dependence graph view. Such an (unreadable) LISP function can be seen in Figure 5, where we have changed the pretty-printer to *graph*, using the *Set Pretty Printer* command of the *Display* menu bar.

Using the same technique, we have defined a *dependences* F pretty-printer which produces the messages that we can see in Figure 4 by decompilation of the **solution** and (**strong** or **weak**)**solution_set** nodes we have described above.

[8] Although this cannot be shown in this report, colors are used to establish correspondences between:

- nodes in the graph view and statements in the program view,
- edges in the graph view and expressions in the program view,
- the currently selected edge and the message explaining the dependence in the dependences view.

Colors are also used to distinguish between true, anti and output dependences.

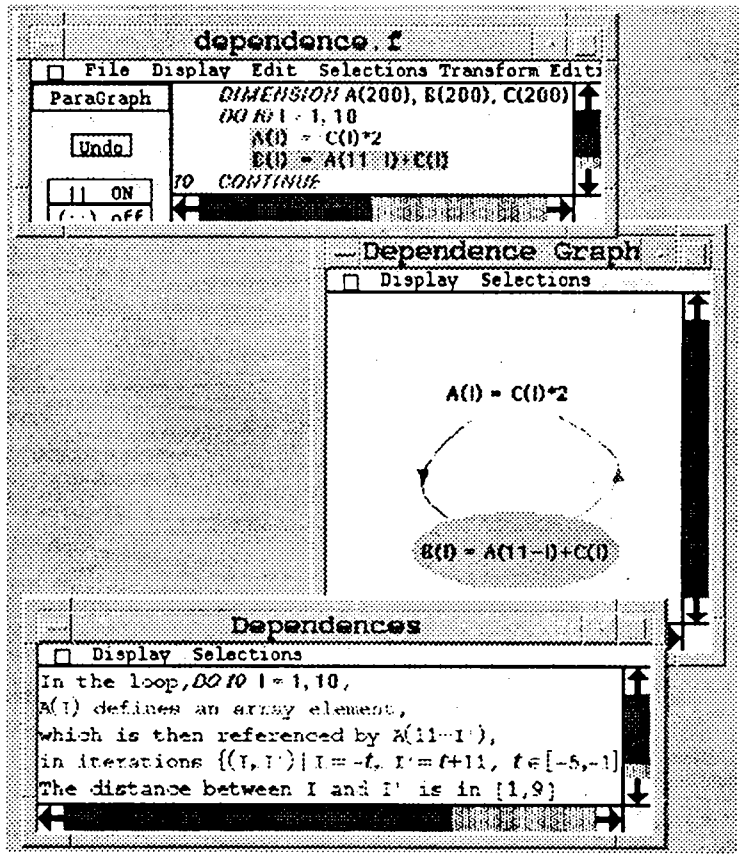

Figure 4: *Displaying dependence graphs*

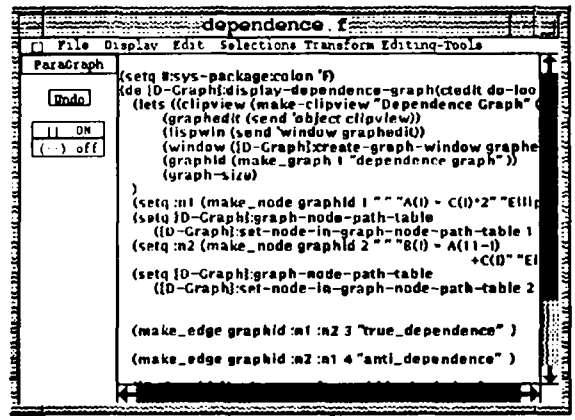

Figure 5: *The LISP decompilation of a do loop*

3.3 Program slices and there applications

Before we look at the utilization of dependence graphs for parallelizing programs, let us consider some other benefits that we can obtain from such an analysis. Following [Weiser84] and [Reps92], we believe that *slices* can be used for many different applications which are quite relevant in software engineering, in particular when one is concerned with *reverse* engineering.

Given a program point, and a variable v, a slice isolates the program parts that contribute to the evaluation of v at this point. The semantics of program slices has been studied by Reps and Yang [Reps92]. They are the basis for visualization tools. Moreover, Horwitz and Reps [Horwitz88-b] have shown that they can be used in order to automatically integrate several variants of a given program.

Let us take an example of the use of slicing in **F**. The **gcd** program computes:

- the greatest common divisor GCD of two numbers A and B,
- two coefficients X and Y such that $GCD = A * X + B * Y$.

In Figure 6, we have taken a slice of the **gcd** program with respect to the variable GCD, by using the *Slice* command of the *Display* menu bar, after having selected the GCD variable in the program view.

Of course, we see that the corresponding slice still computes the value of GCD, without computing the coefficients X and Y. Using functionalities that we will describe in detail in the next chapter, we can either evaluate the slice, or the source program. In either case, the animation of the code will be such that one can follow the evaluation both in the source code and in the slice. It is obvious that such a mechanism can greatly help a programmer, or a maintainer, at looking at a program from a specific point of view, when he tries to understand what it does, or when he debugs it.

The implementation of such a facility was quite obvious. We just had to use the same technique as for the *dead code removal* transformation, specifying

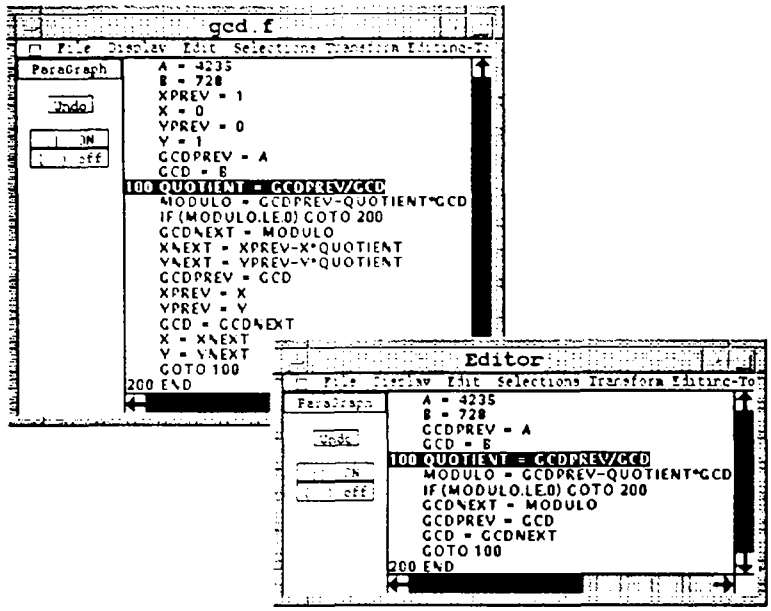

Figure 6: *A slice of the gcd program*

the variable with respect to which we want to take a slice, so that we mark as useful all statements which participate to its evaluation.

```
set sliced_program is
useful_stms_from_node(subject, #use((variable)var_use, [array_use), _, _)  &
marked_useful_stms(- STATEMENTS)  &
removed_useless_stms(STATEMENTS => STATEMENTS')  &
stms_static_eval(STATEMENTS' => STATEMENTS'')
---------
variable
   |-      DECLARATIONS
           STATEMENTS
           END
           =>
               DECLARATIONS
               STATEMENTS'
               END .
end sliced_program.
```

The above work should naturally be extended to deal with array references by using dependence graphs, as we used reaching definitions for scalar variables.

Other uses of dependence graphs have been proposed. For example, Jackson [Jackson91] uses them to prove or disprove simple properties of programs, such the dependence of a given result on all the components of an array.

3.4 Transformation of programs

3.4.1 Parallelization and vectorization

Parallelization

The first step in the parallelization of a do statement is the computation of the *strongly connected components* of its dependence graph. Connected components are then sorted in topological order, and typed as *sequential* or *parallel*, with respect to their corresponding loops nest. Finally, components are regrouped following there type in order to generate code with parallel loops which are as large as possible.

As usual, the relevant data structure to perform this work, a *typed dependence graph*, has been specified in METAL :

```
chapter TYPED_DEPENDENCE_GRAPH
    typed_dependence_graph -> COMPONENT * ... ;
    component -> INT_S COMPONENT_TYPE
                        SEQUENTIAL_FATHERS PARALLEL_FATHERS ;
    component_type  -> LOOP_TYPE * ... ;
    fathers  -> INTS_S * ... ;
    COMPONENT -> component ;
    COMPONENT_TYPE  ::= component_type
    SEQUENTIAL_FATHERS ::= fathers ;
    PARALLEL_FATHERS  ::= fathers ;
    LOOP_TYPE ::= parallelizable vectorizable sequential ;
end chapter ;
```

Lets us now consider the example of Figure 7, its associated dependence graph and the strongly connected components of this graph, as they appear in Figure 8. Each component is either:

- a group constituted of one or more instructions for which there is a *dependence cycle*,
- a group constituted of a single instruction without any dependence cycle.

The components are then sorted in topological order, following the above criterion, as it can be seen in Figure 9.

Components are finally regrouped, so that the generated parallel loops are as big as possible. This is done by observing the following rules:

1. The type of regroupment, sequential or parallel, is determined by the type of the first component which is introduced in the group.
2. A component may be added in a group only if it is of the same type as the group type.
3. A component may be added in a group only if all its sequential or parallel fathers have been already dealt with.
4. A component may be added in a parallel group only in the case none of its sequential fathers is within this group, so that there is no sequential dependence within a parallel loop.

The original loop to parallelize:

```
DO 100 I = 2,200
  1:  C(I) = B(I) + A(I)
  2:  B(I) = C(I) + G(I-1)
  3:  G(I) = B(I-1)
  4:  A(I) = A(I-1)
  5:  D(I) = E(I) * 3
  6:  F(I) = A(I) + F(I)
100 CONTINUE
```

Its dependence graph:

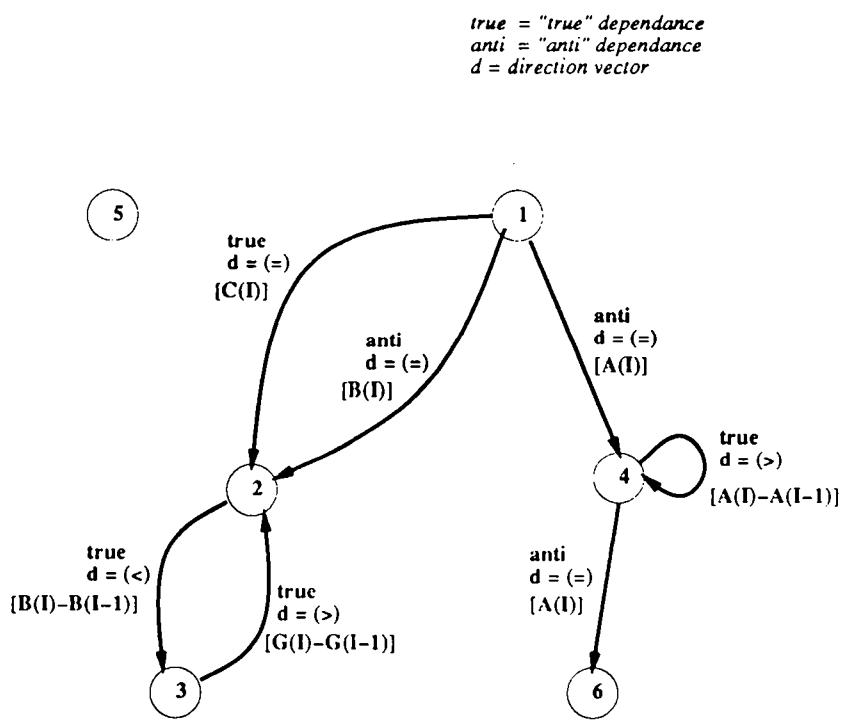

Figure 7: *The original loop and its dependence graph*

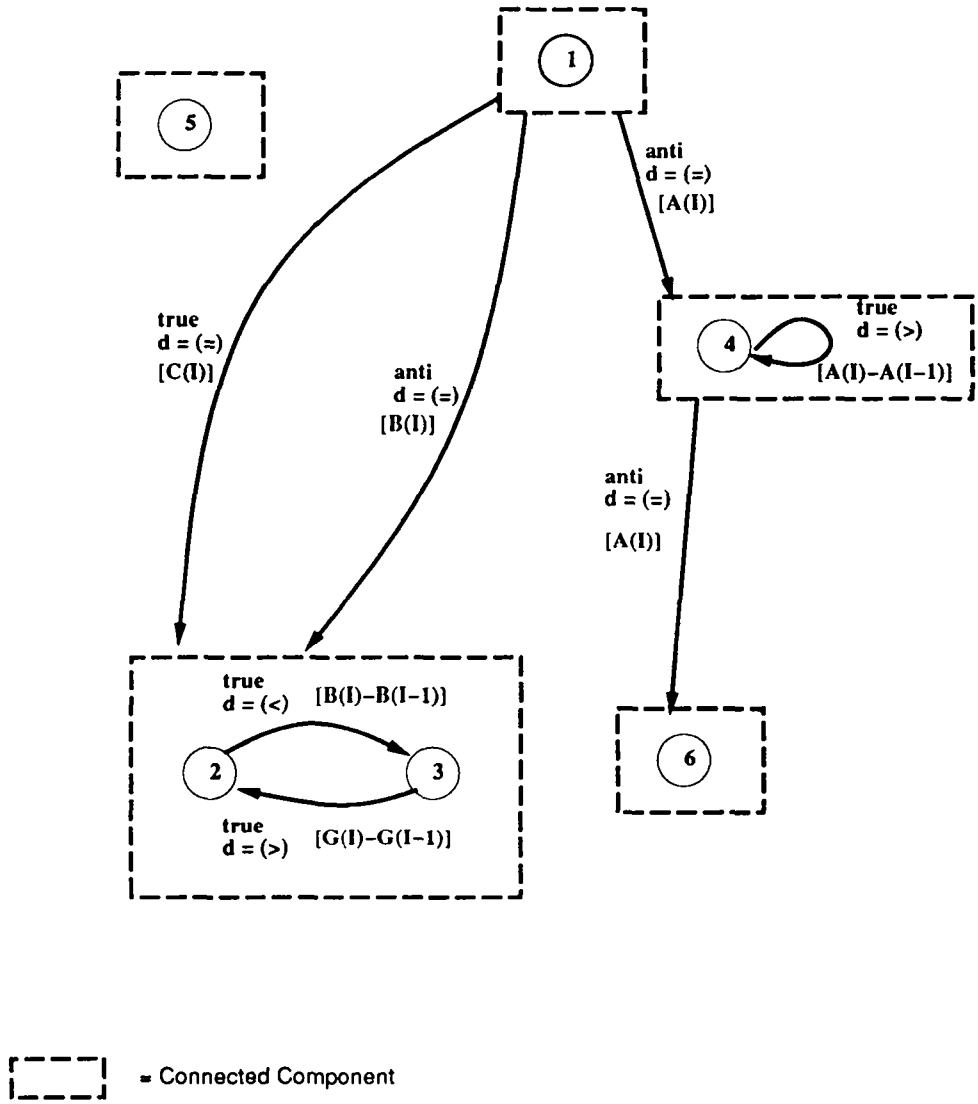

Figure 8: *The strongly connected components*

Component	Type	Sequential fathers	Parallel fathers
(1)	parallel	()	()
(2,3)	sequential	()	(1)
(4)	sequential	()	(1)
(5)	parallel	()	()
(6)	parallel	()	(4)

Figure 9: *The typed sorted strongly connected components*

Following the above rules, we obtain the regroupment of Figure 10.

Finally, it remains to generate the parallel code by recuperating the instructions in each of the groups and embedding them in **parallel do** or **do** statements according to there type. This process is usually called *loop distribution*, and it is performed by the following set of rules:

```
set loop_distribution is
()typed_dependence_graph, LABELS ⊢ _, _, _, _, _, LOOPS → LOOPS, LABELS ;

generate_loop(COMPONENT, COMPONENTS, LABELS
                ⊢ LABEL, LOOP_ID, INIT_PARAM, TERM_PARAM, BODY
                    →
                    COMPONENTS'', LABEL', LOOP, LABELS')
& loop_distribution(COMPONENTS', LABELS'
                    ⊢ LABEL', LOOP_ID, INIT_PARAM, TERM_PARAM, BODY, LOOPS
                        →
                        LOOPS', LABELS'')
_____
(COMPONENT.COMPONENTS)typed_dependence_graph, LABELS
  ⊢ LABEL, LOOP_ID, INIT_PARAM, TERM_PARAM, BODY, LOOPS
      →
      (LOOP.LOOPS')stm_s, LABELS'' ;
end loop_distribution;
```

Using this process, we generate the following parallel code:

```
      PARALLEL DO 100 I = 1,200
         C(I) = B(I) + A(I)
         D(I) = E(I) * 3
100   CONTINUE
      DO 101 I = 1,200
         A(I) = A(I-1)
         B(I) = C(I) + G(I-1)
         G(I) = B(I-1)
101   CONTINUE
      PARALLEL DO 102 I = 1,200
         F(I) = A(I) + F(I)
102   CONTINUE
```

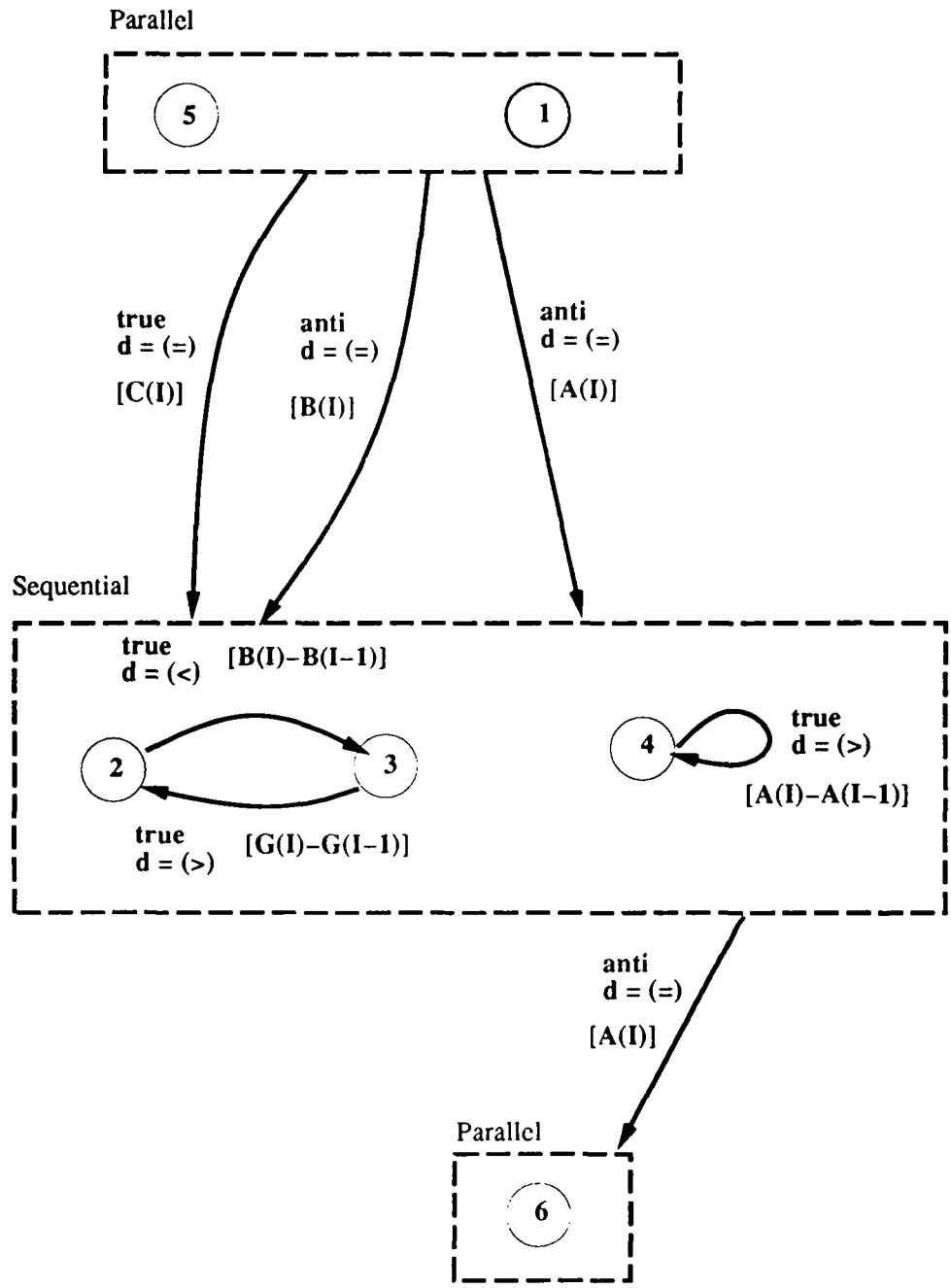

Figure 10: *Regroupment of connected components*

Component	Type	Sequential fathers	Parallel fathers
(1)	vectorizable	()	()
(2,3)	sequential	()	(1)
(4)	sequential	()	(1)
(5)	vectorizable	()	()
(6)	vectorizable	()	(4)

Figure 11: *The typed sorted strongly connected components*

Vectorization

The principle that is used for vectorization is close to the one used for parallelization. Connected components of the dependence graph are first computed. They are then typed as *vectorizable* or *sequential*. Let us consider again the loop of Figure 7.

Components of the dependence graph are typed as follows:

- ***sequential***, if the component contains more than one instruction, or if the component contains a single instruction which is not vectorizable,
- ***vectorizable***, if the component contains a single instruction which is vectorizable.

The list of sequential and parallel fathers is computed as in the case of parallelization. The typed sorted strongly connected components of the dependence graph are shown in Figure 11.

These typed components are regrouped in the same fashion as in the case of parallelization, with the difference that vectorizable components are not regrouped since each of them generates a single (vector) `assign` or `where` instruction. Figure 12 shows such a regroupment.

We are now ready to generate the vectorized code for the above loop:

```
C(1:200) = B(1:200) + A(1:200)
D(1:200) = E(1:200) * 3
DO 100 I = 1,200
   A(I) = A(I - 1)
   B(I) = C(I) + G(I-1)
   G(I) = B(I-1)
100 CONTINUE
F(1:200) = A(1:200) + F(1:200)
```

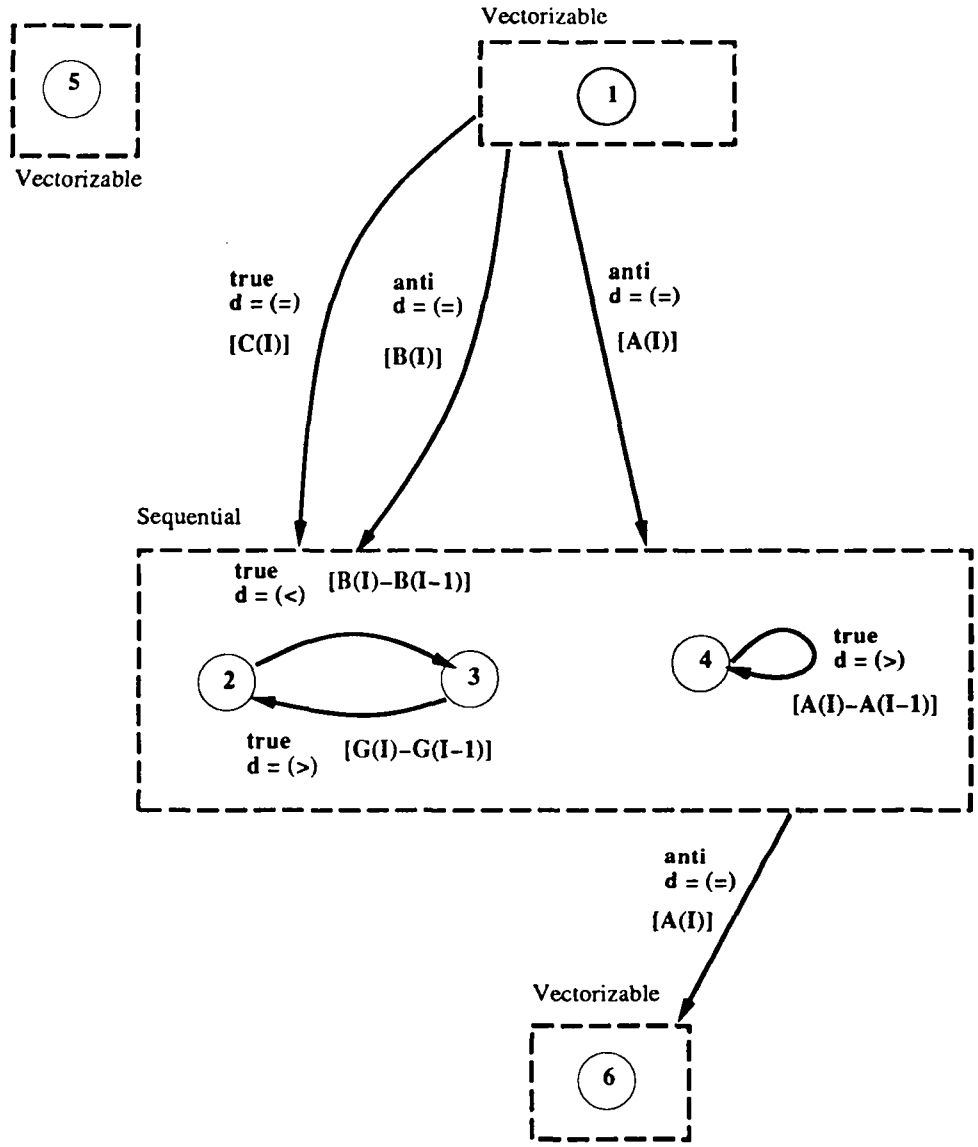

Figure 12: *Regroupment of connected components*

3.4.2 Scalar expansion

In addition to vector or parallel code generation, we have implemented a number of standard program transformations which can be used to augment the size of the parallel regions. As an example of these transformations, we now consider *scalar expansion*.

In a number of cases, anti-dependences can be removed by expanding a scalar into an array whose components are all initialized with its value, thus breaking cycles in the dependence graph. Let us consider the example of Figure 13. We can see that, due to the presence of scalar variables K and T in the body of the do loop, there are several cycles within the corresponding dependence graph.

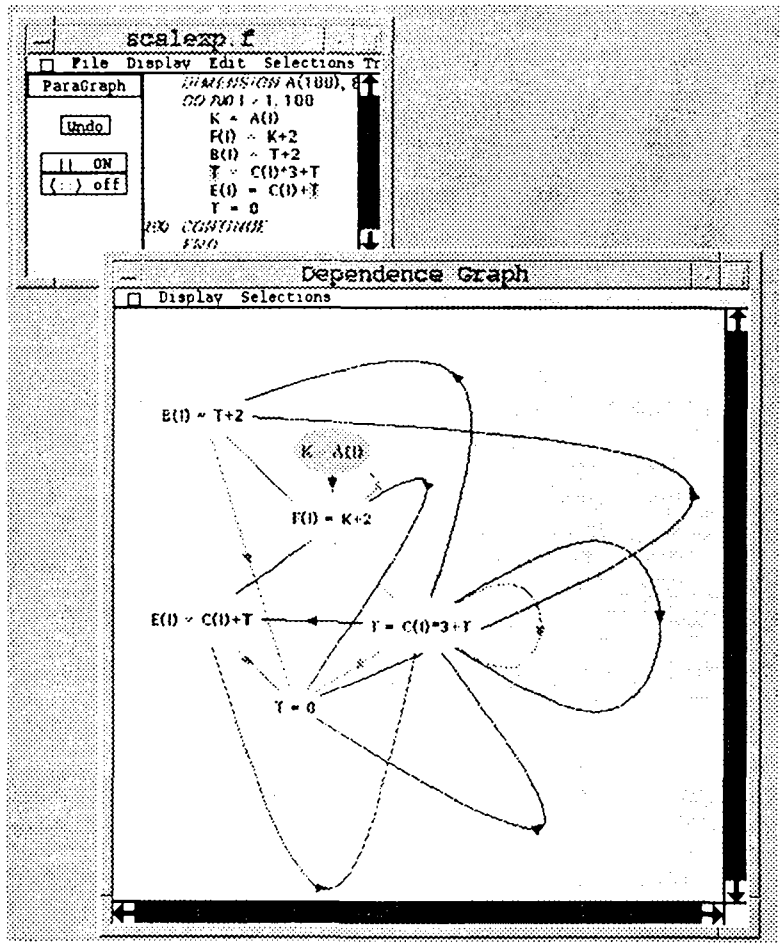

Figure 13: *The original loop before scalar expansion*

Since scalar expansion is a transformation that consumes space in memory,

we decided to perform it under control of the user, who can choose the scalar variables to expand into arrays, as it can be seen in Figure 14. We found it very important that the user can make a choice based on a cost-benefits analysis. The user is therefore presented with the resulting parallel code in a separate program view. He can then accept the transformation, or modify the initial choice of scalar variables to expand.

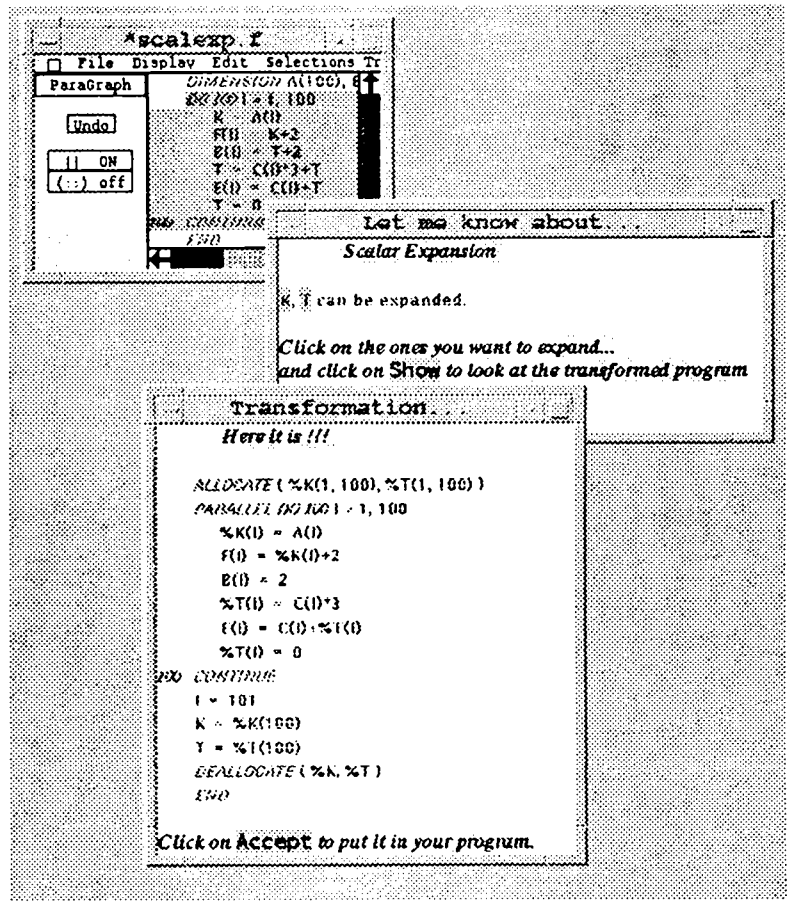

Figure 14: *The choice of scalar variables to expand into arrays*

The user makes his final choice by clicking on the resulting loop he desires. By selecting variables K and T, we obtain the result of Figure 15, and we can verify that the dependence graph corresponding to the transformed loop does not contain cycles any more.

The transformation itself is performed by the following sets of rules, where we see that all occurences of the SCALAR variable are replaced by **NEW_ARRAY(ID)** in the case a definition of this variable occurs before its uses, or by **NEW_ARRAY(ID-1)** before the definition and **NEW_ARRAY(ID)** after the definition if uses of the vari-

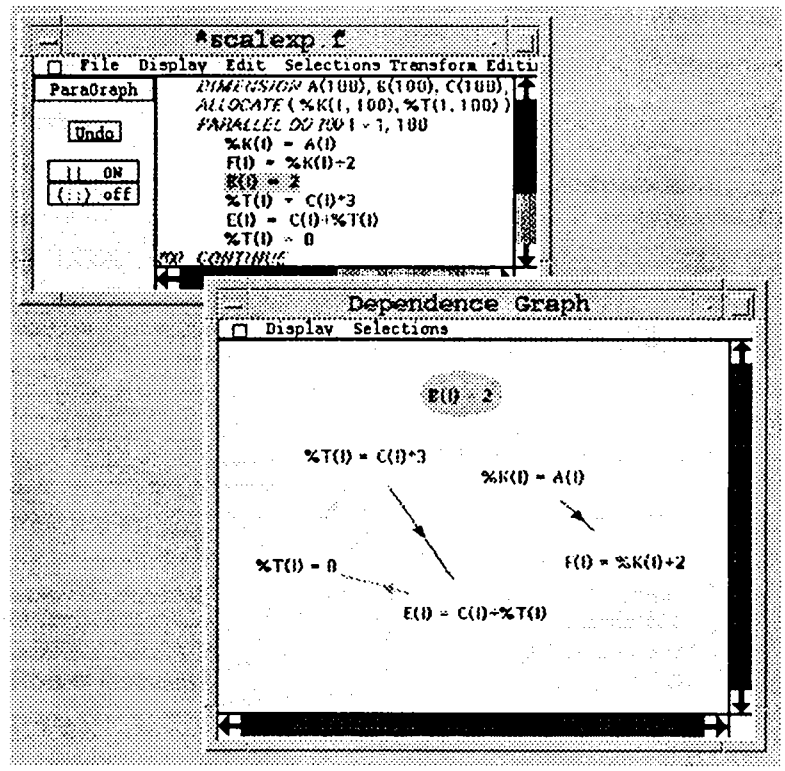

Figure 15: *The parallelized program after scalar expansion*

able occur before a definition.

```
set scalar_expanded is
new_array(SCALAR ⊢ I1, I2 → NEW_ARRAY(BOUNDS))
& replaced_VAR_in_STMS(STATEMENTS : SCALAR \ NEW_ARRAY(ID) => STATEMENTS')
---------
DEF, Int: 0, SCALAR
   ⊢ STATEMENTS, ID, I1, I2 → NEW_ARRAY(BOUNDS), ()stm_s, STATEMENTS' ;

new_array(SCALAR ⊢ I1, I2 → NEW_ARRAY(BOUNDS))
& replaced_VAR_in_STMS(STATEMENTS : SCALAR \ NEW_ARRAY(ID) => STATEMENTS')
---------
Int: DEF, Int: USE, SCALAR
   ⊢ STATEMENTS, ID, I1, I2 → NEW_ARRAY(BOUNDS), ()stm_s, STATEMENTS' ;
        provided DEF < USE;

I1'=I1-1
& new_array(SCALAR ⊢ Int: I1', I2 → NEW_ARRAY(BOUNDS))
& scalar_expanded_with_use_first(SCALAR, NEW_ARRAY, STATEMENTS, Int: 1
                                    ⊢ Int: DEF, ID → STATEMENTS')
---------
Int: DEF, Int: USE, SCALAR
   ⊢ STATEMENTS, ID, Int: I1, I2
        →
        NEW_ARRAY(BOUNDS), (NEW_ARRAY(I1') = SCALAR)stm_s, STATEMENTS' ;
        provided DEF ≥ USE;
end scalar_expanded;
```

```
set scalar_expanded_with_use_first is
_, _, ()stm_s, _ ⊢ _, _ → ()stm_s ;

replaced_VAR_in_STM(STATEMENT : SCALAR \ NEW_ARRAY(ID -    1) => STATEMENT')
& N'=N+1
& scalar_expanded_with_use_first(SCALAR, NEW_ARRAY, STATEMENTS, Int: N'
                                    ⊢ Int: DEF, ID → STATEMENTS'')
---------
SCALAR, NEW_ARRAY, (LABEL STATEMENT.STATEMENTS)stm_s, Int: N
   ⊢ Int: DEF, ID → (LABEL STATEMENT'.STATEMENTS')stm_s ;
        provided N < DEF;

replaced_VAR_in_EXP(EXP : SCALAR \ NEW_ARRAY(ID -    1) => EXP')
& replaced_VAR_in_STMS(STATEMENTS : SCALAR \ NEW_ARRAY(ID) => STATEMENTS')
---------
SCALAR, NEW_ARRAY, (LABEL SCALAR = EXP.STATEMENTS)stm_s, Int: N
   ⊢ Int: DEF, ID → (LABEL NEW_ARRAY(ID) = EXP'.STATEMENTS')stm_s ;
        provided N=DEF;
end scalar_expanded_with_use_first;
```

We will not further detail in this report the other transformations that have been performed in the same spirit as scalar expansion.

3.5 A discussion of incrementality

An important question is naturally raised when doing such analysis and transformations: what happens if the original program is (slightly) modified? At the time being, all of this work is redone, which might take a few seconds, even on the smallest examples. On the other hand, in many cases, the structure of the whole program stays pretty much the same. The answer to that is incrementality of the semantics evaluation.

Incremental evaluators have already been proposed for Attributed Grammars by Reps [Reps84] and others, or for a restricted class of Natural Semantics

Specifications by Attali [Attali92] [9]. However, and at the moment, this problem cannot be handled in a satisfactory manner for general **TYPOL** programs.

Following Horwitz [Horwitz92], we tried to use the concept of a slice to obtain incremental evaluation in the case of the control flow graph construction, which in our case amounts to resolving `goto` statements.

```
set incremental_resolution_of_gotos is
sliced(label_defs_refs
        ⊢   DECLARATIONS
            STATEMENTS
            END)
& resolution_of_gotos(      DECLARATIONS
                            STATEMENTS
                            END)
_____
label_defs_refs
   ⊢    DELARATIONS
        STATEMENTS
        END ;
end incremental_resolution_of_gotos.
```

Given a program which has been modified by using the editor, the above **TYPOL** rule is automatically invoked on the whole program, with a `label_defs_refs` argument which represents the list of labels which have been defined or referenced within the new version of the program part which was modified.

The role of the `sliced` premise is to slice off the parts of the program which do not reference labels which are defined in `label_defs_refs` and which do not define labels which are referenced in `label_defs_refs`. Slicing is simply performed by putting a specific annotation on these nodes. Once this is done, the usual `resolution_of_gotos` is achieved, ignoring tree nodes which have been sliced-off.

As shown by Zadeck [Zadeck83], the work we have done for control flow analysis can adapted to deal with a certain class of data flow problems. However such a method suffers from the fact that slice derivation is not automatic, in the sense that one has to write a specific `sliced` set of rules for every new problem.

We certainly think it is worthwhile investigating any approach that could produce automatically (or semi-automatically) the incremental version of any **TYPOL** program. Following private discussions we had with T. Teitelbaum and G. Kahn, consier for example a set of rules that takes an arbitrary list of integers as its input and that produces a sorted list as its output. It is very easy to show that the the incremental version of this set of rules, when one considers the addition of an element in an initially sorted list, simply is a set of rules that performs the insertion of an integer into a sorted list. Doing this automatically requires a mechanism that does *partial evaluation* of **TYPOL** programs. In our case, the modifications we would have to consider, would be replacements of program sub-trees.

[9] in the case a functional evaluation is sufficient

4 The user interface

4.1 Introduction

With respect to the user interface of **PARAGRAPH**, we have chosen to experiment with two different, and non-exclusive, operating modes:

- in the first case, the user starts with an editor window containing the source program and produces a transformed version of his program in another window by issuing a command in a pull-down menu. From this on, the two program views have to be coordinated. For example, it should be possible to see how a given statement has been transformed by just selecting it in the source program view. Or, conversely, to establish the correspondence between a generated statement and the corresponding source statement(s). This is demonstrated in Figure 16.

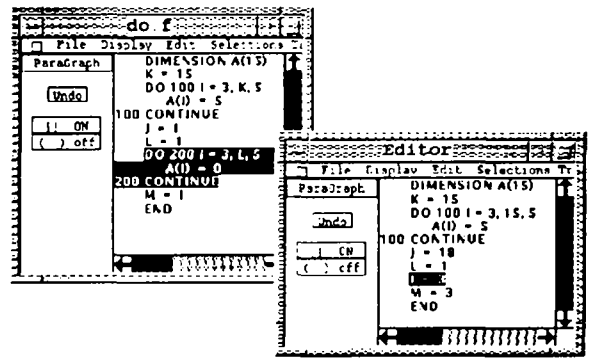

Figure 16: *The correspondence between source and generated code*

- in the second case, the user directly interacts with the source program view. The parts of the program that are susceptible of being transformed (parallelized) are automatically selected using a special color, as it can be seen in Figure 17. A transformations can then simply be triggered by a (middle-button) mouse-click on one of those program parts. The transformed expression or statement then goes back in the same program view.

Let us now describe a network implementation of the first type of user interface.

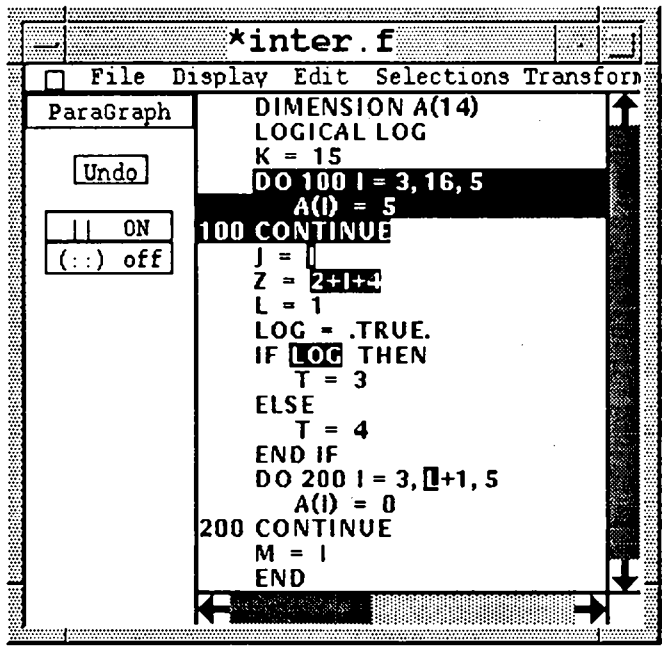

Figure 17: *The automatic selection of transformable code*

4.2 The connection of program views through a network

4.2.1 Motivation

We believe that the main tasks of an interactive environment are to provide:

- coordinated *views* of programming objects,
  as a natural extension of a *make* functionality [Feldman],
- ways to derive objects from other objects,
  by using appropriate transformations.

Following recent progress in software engineering, we have chosen to implement these functionalities by using the **SOPHTALK** component of CENTAUR [Clément92], a message-based mechanism that allows us to separately describe the *transformations* that can be performed interactively under control of the user on a given object, and the necessary *communication* that has then to take place in order to update the other objects.

Rather than forcing the user to know related objects, the modified object announces events by broadcasting them into a network. Those messages will only reach the ears of those objects that have to know them. A listening object can then react to a message by performing some transformation of its internal state, in an asynchronous fashion, and by issuing its own messages.

For example, assume program P' is derived from program P by performing some transformation, and the original program P is modified by using an editor. Then one has to proceed as follows:

- a new version of P' should be computed
  by re-applying the same transformation.
- P' should be redisplayed.

Of course, the efficiency of such a process may strongly depend on the possibility to achieve those two tasks in an incremental fashion.

4.2.2 The design of a F network

In this section, we present the design of a **F** network that allows to perform the parallelization of **F** programs in an interactive setting. Views of programs will be presented to users in a textual form, or in the form of dependence graphs. Coordination will be achieved between programs and corresponding dependence graphs and between the various versions of a given program obtained by applying transformations.

The network is structured as a tree corresponding to the construction of the various versions of a **F** program, as transformations are applied, as it can be seen in Figure 18. Each program version will be associated to a *F-Network* that allows to access its own state, as well as to the *F-Networks* of its descendents and to the *F-network* of its parent. Moreover, the *F-Network* needs to be informed when its corresponding *ctedit* has been modified. This is achieved by encapsulating this same *ctedit* in a *F-Spy* node that we include in the standard Centaur objects network.

Whenever the tree contained in the father *ctedit* is modified, for example by the editor, the *F-Spy* node will be informed by receiving a *modified* message and

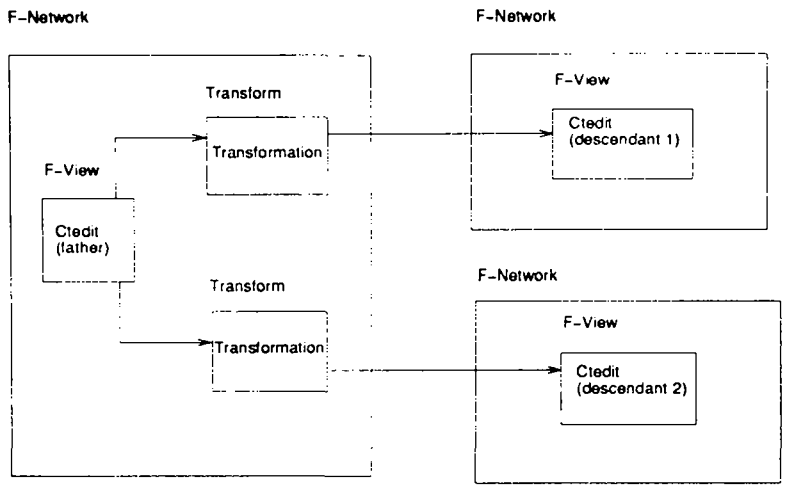

Figure 18: *A global view of the F network*

will cause the corresponding *F-View* node to broadcast a propagate-modified-tree message on its local bus. In reaction to this message, each Transform node will permit to re-apply its Transformation in order to allow the updating of the descendents ctedits. This will in fact be done by the descendents *F-View* nodes, when they receive a *update-modified-tree-and-down* message.

A detailed description of the network can be seen in Figure 19 that exhibits communication between a *ctedit* and its descendents, and in Figure 20 for communication between a dependence graph view and a program view.

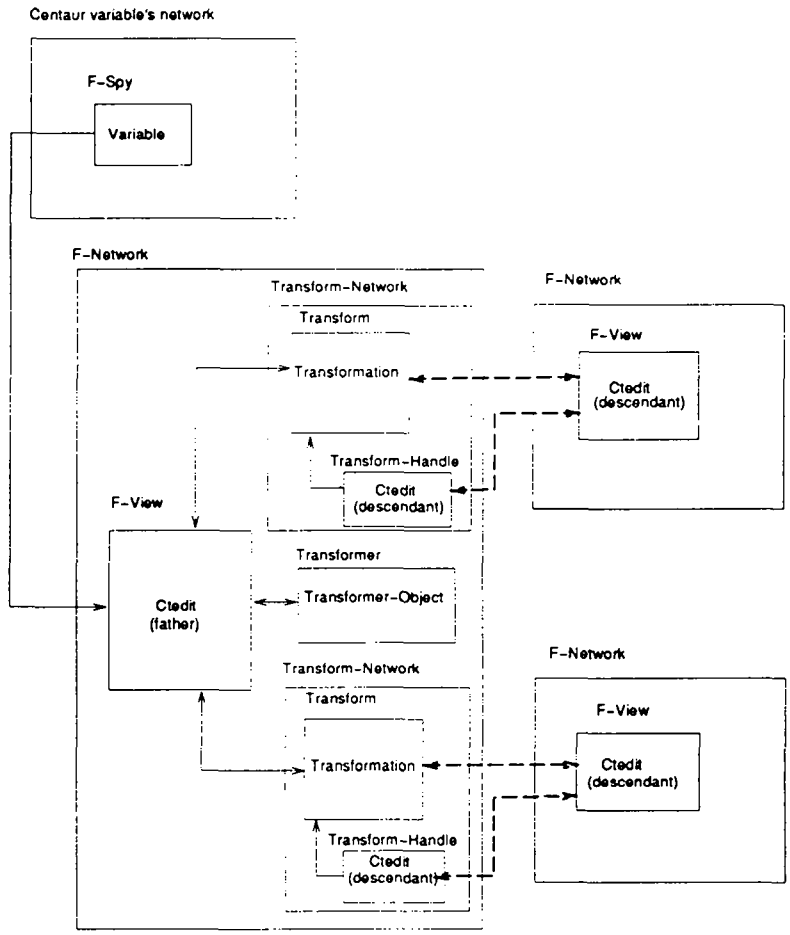

Figure 19: *A detailed view of the F network*

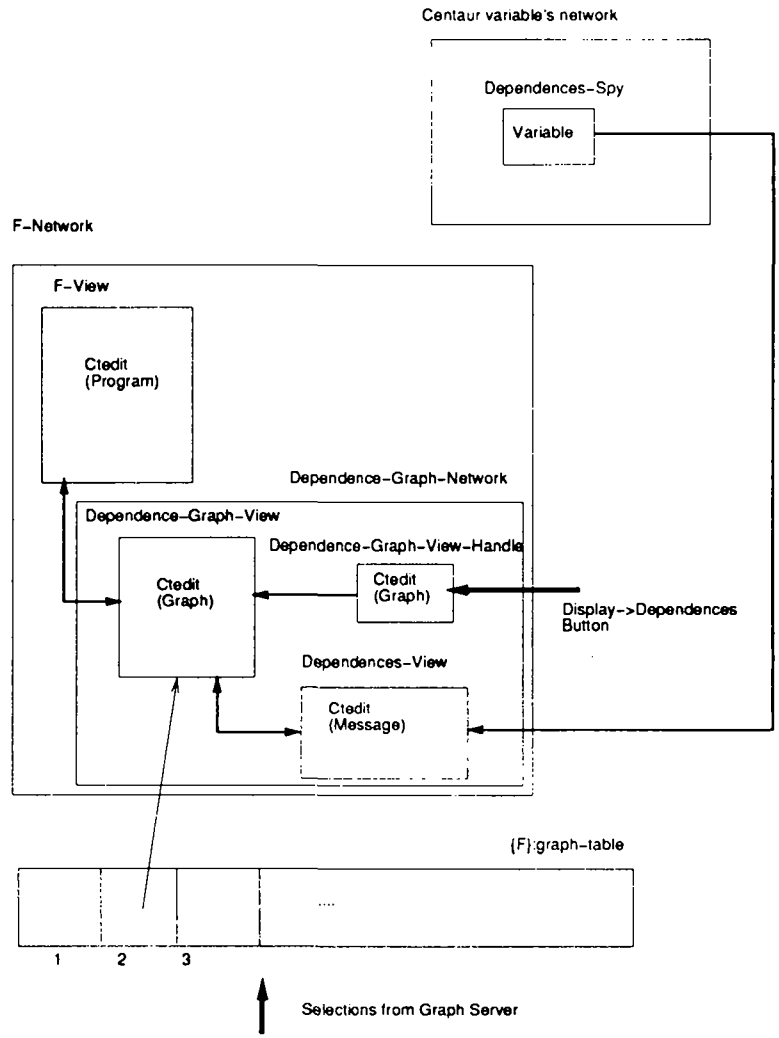

Figure 20: *Communication between a graph view and a program view*

4.2.3 Implementation of selection propagation

As an example, we now consider the **SOPHTALK** implementation of *selection propagation*. Let us start by creation of the necessary node types (**F-Spy**, **F-View**, ...). A node type is defined by its name and its protocol, *i.e.* the name of the messages it can receive (input messages) and send (output messages).

```
({node}:declare
    F-Spy
       (... selection-modified)
       ())

({node}:declare
    F-Network
       ()
       ())

({node}:declare
    F-View
       (update-modified-selection-and-propagate-down
        update-modified-selection-and-propagate-up
        ...)

       (propagate-modified-selection-downwards
        ...))

({node}:declare
    Transform-Network
       (propagate-modified-selection-downwards
        ...)
       (update-modified-selection-and-propagate-down
        update-modified-selection-and-propagate-up
        ...))

({node}:declare
    Transform
       (propagate-modified-selection-downwards
        propagate-modified-selection-upwards
        ...)
       (update-modified-selection-and-propagate-down
        update-modified-selection-and-propagate-up
        ...))

({node}:declare
     Transform-Handle
       ()
       (propagate-modified-selection-upwards))
```

As program views are created, we assume nodes are organized as described in the network of Figure 19, and objects are encapsulated in the right nodes:

- *ctedit* in *F-view* and *Transform-Handle* nodes,
- *transformation* in *Transform* nodes.

We now assume a user clicks on some expression displayed in a F program view. As we placed a spy in the CENTAUR general network of all editor windows, the `{F-Spy}:selection-modified` method is invoked. In the case we find that a `Transform-handle` node exists, we know that the editor window is not the root of a transformation tree. We therefore have to propagate the modification of selection upwards by emitting the `propagate-modified-selection-upwards` message from the `Transform-handle` node. Moreover, in all cases, we emit the `propagate-modified-selection-downwards` message so that descendents of the current editor window, if any, can update there own selection.

```
(de {F-Spy}:selection-modified (node port selection-name)
;---------------------------
    (lets ((ctedit ({node}:object node))
           (tree ({variable}:root ({ctedit}:subject ctedit)))
           (path ({ctedit}:selection-path ctedit selection-name))
           (transform-handle-node
                 ({network}:transform-handle ctedit)))
          (when transform-handle-node ;; current node is not root
              ({node}:emit
                  transform-handle-node
                  'propagate-modified-selection-upwards
                  selection-name
                  tree
                  path))
          ({node}:emit                ;; to descendents if any
              ({network}:f-view ctedit)
              'propagate-modified-selection-downwards
              selection-name
              tree
              ({ctedit}:selection-path ctedit selection-name))))
```

where `{network}:f-view` and `{network}:transform-handle` are node access functions defined as follows:

```
(de {network}:f-view (ctedit)
;-------------------
    (when ({node-object}:atomp ctedit 'F-View)
          (car ({node-object}:nodes ctedit 'F-View))))


(de {network}:transform-handle (ctedit)
;----------------------------
    (when ({node-object}:atomp ctedit 'Transform-Handle)
          (car ({node-object}:nodes ctedit 'Transform-Handle))))
```

Let us now consider propagation up the tranformation tree, the message **propagate-modified-selection-upwards** which was emitted by the **Transform-Handle** node is received by the associated **Transform** node, which will itself emit a **update-modified-selection-and-propagate-up** message to the parent **F-view** node. This message has two arguments which are:

- a modification path which was computed by applying **{tree}:diff** between the son and the parent editor window associated program trees,
- selection-path corresponding to the selection in the son editor window.

```
(de {Transform}:propagate-modified-selection-upwards
    (node port selection-name tree path)
;-----------------------------------------------
    (lets ((transformation ({node}:object node))
           (modification-path
                ({Transformation}:upwards-modification-path
                      transformation tree)))
          ({node}:emit
              node
              'update-modified-selection-and-propagate-up
              selection-name
              modification-path
              path)))
```

When the **F-view** node receives such a message, it has to update its selection according to the modification path and the selection path. In the case the editor window is not the root of a transformation tree, and the selection has effectively been modified, (**old-path** not **nil**), the selection modification has to be propagated upwards.

```
(de {F-View}:update-modified-selection-and-propagate-up
    (node port selection-name modification-path new-path)
;-----------------------------------------------------
    (lets ((ctedit ({node}:atomp node))
           (transform-handle-node ({network}:transform-handle ctedit))
           (old-path ({ctedit}:updated-modified-selection-going-up
                         ctedit selection-name modification-path new-path)))
          (when (and old-path transform-handle-node)
                ({node}:emit
                    transform-handle-node
                    'propagate-modified-selection-upwards
                    selection-name
                    ({variable}:root ({ctedit}:subject ctedit))
                    old-path))))
```

Propagation of a selection downwards is handled in the same fashion by the propagate-modified-selection-downwards and update-modified-selection-and-propagate-do messages.

```
(de {Transform}:propagate-modified-selection-downwards
    (node port selection-name tree path)
;-------------------------------------------------
    (lets ((transformation ({node}:object node))
           (modification-path ({Transformation}:downwards-modification-path
                                transformation tree)))
          ({node}:emit
              ({network}:f-view ({Transformation}:son-ctedit transformation))
              'update-modified-selection-and-propagate-down
              selection-name
              modification-path
              path)))

(de {F-View}:update-modified-selection-and-propagate-down
    (node port selection-name modification-path old-path)
;-------------------------------------------------------
    (lets ((ctedit ({node}:atomp node))
           (new-path ({ctedit}:updated-modified-selection-going-down
                       ctedit selection-name modification-path old-path)))
          (when new-path
                ({node}:emit
                    node
                    'propagate-modified-selection-downwards
                    selection-name
                    ({variable}:root ({ctedit}:subject ctedit))
                    new-path))))
```

4.3 Transformation by clicking

4.3.1 Motivation

We now comment on the second type of user interaction we have experimented, that is *transformation by clicking*. Following the work that has been presented in [Théry92] for mechanical theorem proving, we wanted to address the real problem of a user of such a tool: he wants to parallelize his program, and he may not know how to do it. Therefore, the menu approach of the previous section is not satisfactory.

In the *transformation by clicking* approach, the program exhibits the sub-expressions that can be transformed. The user can trigger transformations by a mouse click. Moreover, if there is a choice to make, as we have seen it in the case of scalar expansion, the user is presented with the expression resulting from a proposed choice. To summarize it up, that main idea is that one makes choices on the basis of the *result*, and not on the basis on a sequence of transformations to apply to a sub-expression. Finally, we had to implement a possibility to *undo* transformations, as the user may estimate that he went the wrong way.

Of course, we encountered the same problems as in theorem proving. First of all, in order to exhibit possible areas of the program that can be transformed, we have have to apply transformations in a reasonable, but *predefined*, order[10], and there might exist a different ordering which would give better results. Next, the undo facility we have described has the drawback of *not being parallel*. That means that in order to undo some transformations, one has to undo all the transformations that had been done in-between.

In order to illustrate this approach, we now describe the implementation of undo, which has been proposed by L. Théry, and used in **PARAGRAPH**.

4.3.2 The implementation of undo

The CENTAUR system, through the notion of a *variable*, provides the basic mechanisms to be able to track the modifications applied to an abstract syntax tree, so that these modifications can be undone. We merely had to define an abstraction which could manipulate *sequences* of such modifications in the same spirit, as a mouse click can trigger several elemental transformations of a programm tree. Again, this implementation uses the **SOPHTALK** primitives.

An undo node is attached to a variable by (`{variable}:link-undo variable`) From now on this variable is spyed. If ones wants to consider a given sequence of modifications as an undo step, the variable has to emit a `modification-step` message which will be handled by `{variable}:undo:modification-step`. If undo of a modification step has to be performed, because the user ckicked on the *undo* button, the variable corresponding to the program view has to emit the `undo-modification-step` message which will be handled by `variable:undo:undo-modification-step`.

[10] otherwise, this would be computationaly untractable.

5 Conclusion

We have seen in this report that a simple prototype of a parallelizer for a subset of FORTRAN has been easily developed in a reasonably short time frame.
We want to insist on the following positive points:

- The clarity of programs that have been written in rule-based languages such as **METAL**, **PPML**, and **TYPOL**.
- The ease of debugging of such programs.
- The quality of the generic user interface (edition, selections, incremental redisplay).
- The essential role of networks in the development of a truly interactive environment.

We also found out directions where future work is needed:

- The time and space requirements of **TYPOL** evaluation should be greatly improved in the case it is to be used in a production quality environment. Some directions are actually pursued:
  - use of an efficient PROLOG compiler,
  - functional evaluation, in the case back-tracking is not needed,
  - translation into attributed grammars specifications.
- The fact that the semantics of transformations has been formally specified with a rule-based language should make it possible to use tools and interfaces such as the ones presented in [Théry92] to mechanically prove that they preserve the semantics of programs.
- The relative difficulty of manipulating graphs in **TYPOL** shows the interest there is in developing a formalism that can be used for pattern matching in graphs, as this has been done by Grundman in [Grundman90].
- Working in an interactive environment based on a distributed system clearly shows the need for incrementality of the semantic evaluation. As we have seen, it seems very promising to examine the partial evaluation of **TYPOL** programs to achieve this goal. Moreover, the partial evaluator could very well be assisted by the interactive provers described in [Théry92].
- The use of networks for expressing the control of the interactive environment was considered to be at the state of the art. However, there is an imperative need for a higher level description of control. We need to describe dependence relations between programming objects, this could be done in a much more declarative style.
- Such an environment should much more open, in the same way this has been done by Kajler for Computer Algebra Systems in [Kajler92]. One should be able to solve a system of equation by using an external tool, defined as a server. One could also connect **PARAGRAPH** to some other parallelizer to benefit from transformations that are not directly available. Another direction is to let an experimented user specify transformations

without accessing the internal data structures. This implies the development of a pattern matching language that would be more elementary than TYPOL, and of an interface to the dependence graph data base.

Acknowledgements

We want to thank Gilles Kahn for having provided us with the opportunity to realize this piece of work, and for having inspired many of the choices we made.

We also want to thank the members of the CROAP project, especially I. Attali, S. Dissoubray, F. Montagnac, J. Bertot, Y. Bertot, T. Despeyroux, L. Hascoet, V. Prunet, L. Rideau-Gallot and L. Théry which have provided invaluable help and advice throughout the past year at INRIA.

References

[Attali92] I. Attali, J. Chazarain, S. Gilette "Incremental Evaluation of Natural Semantics Specifications" CNRS-I3S, Research report, n. 92-20, February 1992.

[Balasundaram89] V. Balasundaram, K. Kennedy, U. Kremer, K. McKinley, "The Parascope Editor: An Interactive Parallel Programming Tool", Conference Proceedings Supercomputing'89, IEEE-ACM, pp. 540-550, Reno, November 1989.

[Banerjee88] U. Banerjee, "Dependence Analysis ofr Supercomputing", Kluwer Academic Publishers, Norwell, Massachussets, 1988.

[Bertot89] Y. Bertot, "Implementation of an interpreter for a parallel language in CENTAUR" INRIA, Research Report, n. 1076, August 1989.

[Centaur92] I. Jacobs, *ed.*, "The Centaur 1.2 Manual", INRIA-Sophia-Antipolis, March 1992.

[Clément85] D. Clément, J. Despeyroux, T. Despeyroux, L. Hascoët, G. Kahn, "Natural Semantics on the Computer", INRIA, Research Report, n. 416, June 1985.

[Clément92] D. Clément, V. Prunet, F. Montagnac, "Integrated Software Components: a paradigm for Control Integration", Proceedings of the Software Development Environments and CASE technology Symposium, Springer-Verlag, 1991.

[Cousot77] P. Cousot, R. Cousot, "Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints", Proceedings of the 4th ACM Conference on the Principles of Programming Languages, 1977.

[Dehbonei88] B. Dehbonei, G. Memmi, "VELOUR: a new vectorizing compiler prototype", Conference Proceedings ICS'88, 1988.

[Eisenbeis92] C. Eisenbeis, J. C. Sogno, "A General Algorithm for Data Dependence Analysis", INRIA, Research Report, n. 1699, May 1992.

[Feldman] S. Feldman, "UNIX make".

[Giboulot92] M.C. Giboulot, G. Popovitch, F. Thomasset, "PIAF - user guide", in GIPE 2, ESPRIT project, Fourth review report, vol. 2, February 1992.

[Grundman90] D. Grundman, "Graph Transformations and Program Flow Analysis", University of California, Berkeley, Research Report, n. UCB/CSD91/620, December 1990.

[Horwitz88-a] S. Horwitz, J. Prins, T. Reps, "On the Adequacy of Program Dependence Graphs for Representing Programs", Conference Proceedings ACM POPL'88, San Diego, January 1988.

[Horwitz88-b] S. Horwitz, J. Prins, T. Reps, "Integrating Non-Interfering Versions of Programs", Conference Proceedings ACM POPL'88, San Diego, January 1988.

[Horwitz92] S. Horwitz, T. Reps, "The Use of Program Dependence Graphs in Software Engineering", Invited conference, Conference Proceedings IEEE- ICSE'92, Melbourne, May 1992.

[Jacobs92] I. Jacobs, L. Rideau-Gallot, "A CENTAUR Tutorial", Reasearch Report, INRIA, n. 140, July 1992.

[Jackson91] D. Jackson, "Aspect: An Economical Bug-Detector", Conference Proceedings IEEE-ICSE'91, Austin, July 1991.

[Kuck72] D.J. Kuck, Y. Muraoka, S.C. Chen, "On the Number of Operations Simultaneously Executable in Fortran-like Programs and their Resulting Speed-up", IEEE Transactions on Computers C-21, pp. 1293-1310, December 1972.

[Kajler92] N. Kajler, "CAS/PI: a Portable and Extensible Interface for Computer Algebra Systems" Conference Proceedings of ACM ISSAC'92, Berkeley, California, July 1992.

[Lehors92] A. Le Hors, "Graph: A Directed Graph Displaying Server", in GIPE 2 ESPRIT project, 4th Review Report, Workpackage 4, January 1992.

[Leisure85] B.R. Leisure, "The Parafrase Project's Fortran Analyzer. Major Module Documentation", Technical report CSRD-504, University of Illinois at Urbana-Champain.

[Reps84] T. Reps, "Generated Language Based Environments", ACM Doctoral Dissertation Award, M.I.T. Press, Cambridge, Massachussets, 1984.

[Reps88] T. Reps, T. Teitelbaum, "The Synthesizer generator : a system for constructing language-based editors", Springer-Verlag, 1988.

[Reps92] T. Reps, W. Yang, "The Semantics of Program Slicing", to be published.

[Saint91] J. B. Saint, "AZUR: un environnement pour ESTEREL sous CENTAUR", Doctoral thesis, University of Paris VII, December 1991.

[Teitelbaum81] T. Teitelbaum, T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", Communications of ACM, 24(9), pp. 563-573, September 1981.

[Théry92] L. Théry, G. Kahn, Y. Bertot, "Real Theorem Provers deserve Real User-Interfaces", Proceedings of the 5th ACM SIGSOFT Symposium on Software Development Environments, Washington, December 1992.

[Weiser84] M. Weiser, "Program Slicing", IEEE Transactions on Software Engineering SE-10(4), pp. 352-357, July 1984.

[Zadeck83] K. Zadeck, "Incremental Data Flow Analysis in a Structured Program Editor" Ph.D. Thesis, Rice University, October 1983.

[Zima88] H. Zima, H.J. Bast, M. Gerndt, "Superb: A tool for semi-automatic MIMD/SIMD parallelization", Parallel Computing, Vol. 6, pp. 1-18, 1988.

[Zima91] H. Zima, with B. Chapman, "Supercompilers for Parallel and Vector Computers", ACM Press, New York, 1991.

Unité de Recherche INRIA Sophia Antipolis
2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

Unité de Recherche INRIA Lorraine Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)
Unité de Recherche INRIA Rennes IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Rocquencourt Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399