Science of Computer Programming 15 (1990) 15-54
North-Holland

USING REWRITING TECHNIQUES TO PRODUCE CODE GENERATORS AND PROVING THEM CORRECT

Annie DESPLAND

LIFO, Université d'Orléans, BP 6759, 45067 Orléans Cedex 2, France; and INRIA, Domaine de Voluceau, BP 105, 78153 Le Chesnay Cedex, France

Monique MAZAUD

INRIA, Domaine de Voluceau, BP 105, 78153 Le Chesnay Cedex, France

Raymond RAKOTOZAFY

LIFO, Université d'Orléans, BP 6759, 45067 Orléans Cedex 2, France; and INRIA, Domaine de Voluceau, BP 105, 78153 Le Chesnay Cedex, France

Communicated by M.-C. Gaudel
Received June 1987
Revised January 1989

Abstract. A major problem in deriving a compiler from a formal definition is the production of correct and efficient object code. In this context, we propose a solution to the problem of code-generator generation.

Our approach is based on a target machine description where the basic concepts used (storage classes, access modes, access classes and instructions) are hierarchically described by tree patterns. These tree patterns are terms of an abstract data type. The program intermediate representation (input to the code generator) is a term of the same abstract data type.

The code generation process is based on access modes and instruction template-driven rewritings. The result is that each program instruction is reduced to a sequence of elementary machine instructions, each of them representing an instance of an instruction template.

The axioms of the abstract data type are used to prove that the rewritings preserve the semantics of the intermediate representation.

Introduction

A compiler, in order to produce code, needs full knowledge of not only the syntax and the semantics of the source language but also of the structure of the target machine and the semantics of its instruction set. Therefore, each combination of source and target languages needs its own compiler. The profusion of new languages and computers leads to a constant demand for new compilers. Considerable research effort has been put into making the compiler construction as modular and automatic as possible. At present, automatic production of parsers from a BNF-like specification of grammars is widely known and used.

As the properties of the source and the target languages are quite different, it is necessary to introduce an intermediate representation (IR).

Numerous works have developed techniques and tools for the implementation of retargetable code generators. Such tools should combine portability features and ease of writing for the compiler writer. Moreover, they must make it possible to:

- clearly separate the description of a general technique from its application to a particular machine;
- describe and achieve various code generation subtasks without imposing any particular ordering on them;
- accept as input a description which is easily deduced from the handbook of the machine.

The use of formal description of a target machine within a compiler writing system has given rise to several techniques. Some use a table-driven description of the instruction set, others use templates for each IR primitive for which code has to be produced.

In the past few years, the code selection problem has received a lot of attention and has been relatively successfully dealt with by Cattell [3], Graham and Glanville [12, 13], Ganapathi and Fischer [7], and Ganzinger and Giegerich [8].

In contrast, the storage allocation phase of code generation seems to have been neglected with the exception of the PQCC system [17, 18], which gives a partial solution to this problem.

In all these works, the IR and the target machine specification are of a quite different nature:

- The Graham-Glanville approach is similar to the methods used for table-driven syntax analysis, and a target machine description is used to derive a parser. The overall structure of the code selection algorithm is similar to the one of an LR(1) parser, which works on the linearized string of an intermediate tree and produces a sequence of instructions. It is necessary to design an IR for each combination of source and target languages.
- Cattell uses TCOL (a universal tree language) for the IR. From a complete specification of the instruction set processor, tree templates are built which depict instructions. The code is emitted during a recursive top-down traversal of the IR tree.
- Giegerich [10] describes the target machine as a signature of an algebraic abstract data type. The code generation is achieved by the production of a term of this abstract data type.
- Aho and Ganapathi [1] propose to represent the target machine instructions as tree rewriting rules, each of them consisting in a replacement node, a tree template, a cost and an action. The code selection is done using a combination of efficient tree-pattern matching with dynamic programming. This can be done using the tree manipulation language called Twig, implemented by Tjiang [21]. Register allocation is done by user-specified actions.

The main drawback of the systems above is that the input IR cannot be parameterized by the features of both the source language and the target machine. The solution

proposed here overcomes this. It produces, as in the Perluette system [5, 9], a compiler from a specification in three parts: a source language definition, a target language definition and the description of the implementation choices (see Fig. 1). The source and target language definitions are algebraic data types (ADTs), the implementation choices define a mapping from the source ADT to the target ADT.

The target machine description is hierarchically structured in three levels [6]:

- storage description (sets of available locations, i.e. registers, memory locations, stacks) as storage classes;
- addressing modes description (various ways to access locations) as access modes;
- instruction set description.

Generally speaking, the addressing mode semantics depends on the position (source or destination) of the corresponding operand. That is the reason why we have included the position in the definition of the concept of an access mode

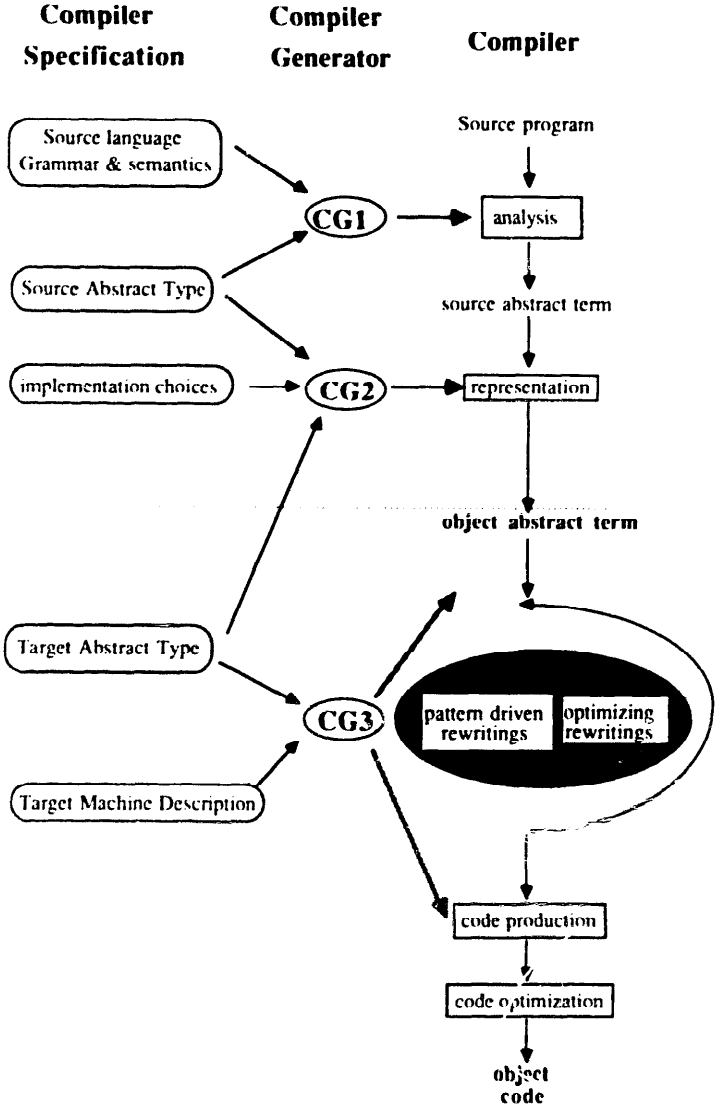

Fig. 1

template. Since an operand of an instruction may accept several access modes, the access modes are assembled in access classes. So the access classes are instruction parameters. The semantics of access modes and instructions is given as terms of the target data type. Once the correct IR term is produced, the code generation will proceed in four steps (omitting optimization tasks):

Step 1. *Access mode templates and instruction templates pattern matching.* The bottom-up matching process of the IR is carried out until the IR is identified to instruction templates. In the context of an instruction template, operand subterms are matched with access mode templates. When it succeeds, the subterm is replaced by a representative tree of the access mode recognized. In the sequel, this representative is called "canonical form". When it fails, an inner subterm, matching an access mode template, is replaced by a temporary resource. This requires the insertion of elementary universal store trees of temporary results in temporary storages. The result is an IR whose leaves are instantiated access modes in "canonical form". In order to recognize an instruction template instance, it is necessary to verify that the access modes of the IR belong to the corresponding access classes of the instruction template. At this point of the rewriting process, the IR is in its "canonical form".

We use universal store more efficiently than Cattell [3], because in our approach, temporaries are explicitly managed and have their own operators. This allows us to achieve the correct assignment of storage locations and an easy choice of the right mnemonic (for example, the choice between MOVE.B, MOVE.W and MOVE.L on the MC68000 is thus possible).

Step 2. *Binding.* Each temporary access mode is bound with the list of the allowed access modes.

Step 3. *Register assignment.* According to the available storages, first we pick for each temporary the storage class related to the cheapest access mode of the preceding set. Second, we choose the name of an available location of this storage class for the temporary.

Step 4. *Code production.* Now, the IR is a tree built up with instances of instruction templates rooted by the sequence operator. As each instruction template is decorated with the corresponding assembly instruction mnemonic, the code emission can be simply achieved by a top-down sequential tree traversal of these instances.

All the above steps concern the IR tree. In particular, the allocation of storage locations transforms the tree leaves but has no immediate effect on code production: unlike Cattell, the possibility of choosing the right mnemonic is postponed until the end. Moreover, in our system, and contrary to the PQCC, the temporaries generated by the algorithm are set clearly apart and have their own operators. This enables them to be explicitly addressed. Optimizations specified by tree rewritings can take place at any time. Code production constitutes the ultimate phase of the global rewriting process of the IR tree.

This solution allows a modular multi-pass code generation process with the different optimization tasks inserted as rewritings according to the compiler writer's

wishes. Such an approach confers modularity and independence to our system as in the MUG2 system [8], while conserving the descriptive features of the Cattell and Graham-Glanville approaches.

Our solution to automatic production of code generators is complete in the sense that it tackles correctly both the code selection and the storage allocation problems.

This paper is divided into three sections. The first one presents the two specifications required as input to the code-generator generator: a formal specification of the target machine and the IR description. Section 2 deals with the code-generator generator itself: first, definitions of tree templates associated with the target machine components are given, then the IR rewriting algorithm (which precedes the code emission phase) is developed. Finally, the looping and blocking problems are discussed. Section 3 deals with the correctness proof of the code generator produced.

The basic concepts developed here are illustrated by MC68000 examples throughout the paper.

1. Inputs to the code-generator generator

1.1. Specification of the target machine

We provide a language to specify the instruction set processor of a target machine. The basic concepts used are described by specific constructs of the language: storage bases, storage classes, value classes, access modes, access classes and instructions.

Each construct is defined by properties such as the size of the associated addressable units or the semantics of the occurrence of the construct. This semantics is expressed using a term of the target abstract data type and takes into account this size.

As the occurrences of a construct are related to the size of the addressable units, their semantic descriptions are nearly identical [6]. A solution proposed to deal with large algebraic specifications [2] is the use of parameterization and instantiation mechanisms. Such mechanisms fit very well to our machine specification language.

The compiler writer can factor some instances of a given construct in a generic pattern followed by the possible values of the generic parameters of the pattern. The system derives from this declaration as many occurrences of the construct as there are sizes of addressable units associated to it. The instantiation mechanism is bound to a name generation mechanism.

Throughout the paper the following notations will be used:

- Let n be a name and L a variable. If L is instantiated by v, then $n!L$ builds the name n_v.
- $\langle S\ V \rangle$ means that V is a constant or a variable of sort S.
- All keywords of the language are in bold letters in the following examples.

1.1.1. Storage structuring

A component of the physical storage doesn't represent the same operand depending on the size associated to the operation applied to this operand. For instance, an access to a register may designate a byte operand, a word operand or a long-word operand. So, we define two fundamental concepts: *storage base* and *storage class*. A storage base is defined as a set of smallest addressable units of physical storage. For a given storage base, the compiler writer must describe as many classes as there are ways to gather storage base elements to represent logical storage units.

A storage class occurrence is characterized by the following properties:

- its denotation,
- its storage base viewed as an attribute (in the attributed grammars meaning) of a storage class,
- the dereference operation that can be applied to this storage class, i.e. the access operation to the contents of an element of this storage class.

In the machine description language, the storage class construct is described using a predefined keyword for each of these properties.

The MC68000 has two kinds of registers: the data registers dedicated to data values and address registers dedicated to addresses. Thus, the compiler writer must declare the two following storage bases:

```
Storage_base DREG                          —Data registers
  Set is {DREG[k]  where k in 0..31}
End

Storage_base AREG                          —Address registers
  Set is {AREG[k]  where k in 0..13}
End
```

Let us consider the storage classes related to the data register storage base. Since an access to a data register may represent an access to a byte operand, a word operand or a long-word operand, the compiler writer must declare three storage classes respectively: the "dregister_B", the "dregister_W" and "dregister_L" storage classes:

```
Storage_class
  Denotation (dregister_B Dk)
  Attributes
    $Base ≈ DREG
  Deference_operation
    cont_of_dreg_B((dregister_B Dk))
  Symbolic_notation
    Dk is DREG[4*4k]  where k in 0..7
End
```

Storage_class
Denotation ⟨dregister_W D_k⟩
Attributes
$Base = DREG
Dereference_operation
cont_of_dreg_W(⟨dregister_W D_k⟩)
Symbolic_notation
D_k **is** DREG [4*k..4*k + 1] **where** k **in** 0..7
End

The compiler writer must declare similarly the dregister_L storage class.

Notice that these three declarations are quite similar. In this context, the use of the parameterization and instantiation mechanisms is especially useful. In order to avoid the repetition of similar declarations, the compiler writer can declare a generic pattern of a *data register* storage class using generic names. The instance part of the declaration includes the information needed by the name generation mechanism to build the effective names.

Storage_class
Denotation ⟨dregister!size D_k⟩
Attributes
$Base = DREG
Dereference_operation
cont_of_dreg!size(⟨dregister!size D_k⟩)
Symbolic_notation
D_k **is** DREG[4*k..4*k + length − 1] **where** k **in** 0..7
Instances
size in {B, W, L}
case size **is**
B: length **is** 1
W: length **is** 2
L: length **is** 4
end case
End

From this pattern, the system deduces the three preceding descriptions of effective storage classes.

The specification of the address register class is similar to that of the data register class. The only difference is that the byte access to an address register is not available.

Storage_class
Denotation ⟨aregister!size A_k⟩
Attributes
$Base = AREG
Dereference_operation
cont_of_areg!size(⟨aregister!size A_k⟩)

```
Symbolic_notation
  A_k is AREG[2*k..2*k + length - 1]   where k in 0..7
Instances
  size in {W, L}
  case size is
    W: length is 1
    L: length is 2
  End case
End
```

The system deduces the description of the "aregister_W" and "aregister_L" storage classes from this pattern.

1.1.2. Values classes

On any machine, there are instructions whose operands are immediate constants. Their value is generally stored in the address space of the instruction. We define value classes corresponding to allowed ranges for the operands to get homogeneous description. We suppose that there exists a value base named INTEGER including all possible value classes. In the same way, we suppose that there exists a label value base named LABEL.

On the MC68000, an immediate value operand may be a signed integer represented by a byte, a word or a long-word value.

```
Value_class
  Denotation ⟨value!size val⟩
  Attributes
    $Base = INTEGER
  Symbolic_notation
    val in -2**(length - 1)..2**(length - 1) - 1
  Instances
    size in {B, W, L}
    case size is
      B: length is 8
      W: length is 16
      L: length is 32
    End case
End
```

1.1.3. Access modes

Let us consider an assignment statement of A to B, we shall state in the sequel that A is the source operand and B the destination of the assignment. In an instruction context, an operand is designated by an addressing mode. Whereas an addressing mode in source position designates the contents of a storage, it designates the storage itself in destination position.

A particular machine has several addressing modes. For a given addressing mode of the machine, the compiler writer must define as many access modes as there are associated storage classes.

An access mode pattern is specified by:

- a canonical form, representative of the access mode, including its name and its parameters; these parameters are formal storage or value classes;
- its related attributes: length, format and costs;
- templates that describe the access path to the corresponding operand when it is respectively in destination position and in source position; they are introduced by the keywords **dst** and **src** and defined by a term of an abstract data type.

As for the storage class construct, the compiler writer can define a formal access mode.

Among the numerous addressing modes of the MC68000, let us consider the *data register* addressing mode. Its associated storage classes are byte data register, word data register, long-word data register. Thus the compiler writer define a generic access mode pattern "dreg_am!size" parameterized by the generic data register "dregister!size" defined in Section 1.1.1. The instances define the effective *data register* access modes with their appropriate storage class:

```
Access_mode                          —Data register access modes
  Canonical_form
    dreg_am!size
      (⟨dregister!size reg⟩)
  Attributes
    $length = size                   —Length of the addressable unit
    $fmt = ~reg~                     —Assembly language format
    ...
  Templates
    dst = ⟨dregister!size reg⟩       —Access path in destination position
    src = cont_of_dreg!size(dst)     —Access path in source position
  Instances
    size in {B, W, L}
End
```

When an access mode template in source position is defined by the dereference operation (here "cont_of_dreg!size") applied to the term which defines the access mode template in destination position, the compiler writer can use the abbreviation **dst** to denote the access mode template in destination position. For example, using the preceding definition, the system derives the definition of the "dreg_am_B" access mode:

```
Access_mode
  Canonical_form
    dreg_am_B(⟨dregister_B reg⟩)
```

```
  Attributes
    $length(dreg_am_B(⟨dregister_B reg⟩)) = B
    $fmt(dreg_am_B(⟨dregister_B reg⟩)) = ~reg~
    ...
  Templates
    dst(dreg_am_B(⟨dregister_B reg⟩))
      = ⟨dregister_B reg⟩
    src(dreg_am_B(⟨dregister_B reg⟩))
      = cont_of_dreg_B(⟨dregister_B reg⟩)
End
```

The *indirect with displacement* access mode has instances which depend on the size of the location indirectly accessed in source position:

```
Access_mode
  Canonical_form                          —Indirect with displacement
    disp_am!size(⟨aregister_L reg⟩        —access modes
                 ,⟨value_W val⟩)
  Attributes
    $length = size                        —Length of the addressable unit
    $fmt = ~val(reg)~                     —Assembly language format
    ...
  Templates
    dst = index!size                      —Access path in destinatition
          (cont_of_areg_L                 —position
             (⟨aregister_L reg⟩)
           ,⟨value_W val⟩)
    src = cont_of_laddress!size(dst)      —Access path in source position
  Instances
    size in {B, W, L}
End
```

1.1.4. Access classes

The operands of an instruction are access classes which are defined as sets of access modes. An access class can be also specified by a generic pattern including the instantiation of its elements. There are as many instances of a generic access class as there are possible sizes of operands.

```
Access_class
  ⟨All_access!size AM⟩
    = dreg_am!size(⟨dregister!size reg⟩)   where size in {B, W, L}
    = areg_am!size(⟨aregister!size reg⟩)   where size in {W, L}
    = ...
End
```

1.1.5. Instructions

An instruction may be characterized by the following properties:

- the access classes defining the operands to which the instructions apply;
- its related attributes: format, i.e. the syntax in the assembly language, length, ...;
- the template describing what is performed by the instruction (it is a term of an abstract data type).

Nearly every instruction of the target machine may be applied to the different lengths of its operands. In order to avoid the repetition of such descriptions, the compiler writer specifies a pattern of an instruction and its instances.

Let us consider the *move* instruction which corresponds to an assignment operation. The size of the instruction may be specified to be a byte, a word or a long word. We get the following specification:

```
Instruction
  move!size(⟨All_access!size AM1⟩
            ,⟨Altdata_access!size AM2⟩)
  Attributes
    $length = size
    $fmt = ~MOVE.$length
              $fmt(⟨All_access!size AM1⟩)
              ,$fmt(⟨Altdata_access!size AM2⟩)~
  Template
    assign!size(src(⟨All_access!size AM1⟩)
              ,dst(⟨Altdata_access!size AM2⟩))
  Instances
    size in {B, W, L}
End
```

The value of the attribute format (**$fmt**) of an access class is synthetized from the value of the corresponding attribute of this access mode. The *add* instruction presented below adds an operand in a data register to an operand in memory location and puts the result in the data register:

```
Instruction
  add!size(⟨All_access!size AM1⟩
           ,⟨Dregister_access!size AM2⟩)
  Attributes
    $length = size
    $fmt = ~ADD.$length
             $fmt(⟨All_access!size AM1⟩)
               ,$fmt(⟨Dregister_access!size AM2⟩)~
  Template
    assign!size
      (add!size (src(⟨Dregister_access!size AM2⟩)
```

```
                    , src((All_access!size AM1)))
          , dst((Dregister_access!size AM2)))
    Instances
      size in {B, W, L}
  End
```

1.2. Intermediate representation

1.2.1. Introduction

Let us summarize the specification process of the code generator for a given target machine. The machine is specified by the description of available locations, access modes and instructions in the form presented above. As the locations are defined, names of sorts are associated to them. The access modes and instructions are specified by operators on locations. Thus, a set of operations with corresponding domains and co-domains can be defined. Therefore the specification of the code generator includes two parts:

- the target machine description,
- the target abstract data type related to the target machine.

In the framework of a compiler writing system which works in three steps (see Fig. 1), the intermediate form input of the code generator is a target abstract term. It is the representation of a source abstract term which is the semantic value associated to the source program. The source term corresponding to the semantic value of a statement is a term of type *modification* as defined in [9]. The representation of a modification of the source abstract data type is a modification of the target abstract data type.

The translation of the semantic value of the source program in terms of semantic value of the target program is specified and proved [9] to be the representation of one abstract data type into another. The translation of a source abstract term into a target abstract term is a rewriting corresponding to this representation. The basic IR for the code generation process is the tree corresponding to the abstract target term. The code generation process can be considered as a rewriting process such that, at the end of the rewritings, each son of the root is an instance of an instruction template.

1.2.2. Abstract data type related to the target machine

In order to emit the code related to an instruction, it is necessary to recognize an instance of an instruction template. As in the syntactic analysis process, an automaton is derived automatically. It is used by a target-machine-independent kernel which consists of an analyzer and modules of rewriting and tree construction. The target term, input to the code generation process, belongs to the target abstract

data type. The specification of the target abstract data type given by the compiler writer includes:

- a set of names of sorts,
- a signature, i.e. a set of names of operations with their corresponding domains and co-domains,
- a set of axioms.

Sorts are defined from the machine description elements such as storage classes and value classes.

The description of the signature of the target machine **ADT** is factored in the same way that the description of the constructs in the machine specification language, using the name generation mechanism:

```
ADT target_machine
  Sort
    laddress!size                          —Address
    , aregister!size if size in {W, L}     —Address register
    , dregister!size                       —Data register
    , value!size                           —Immediate value operand
    , data_label!size                      —Data label
    , code_label!size if size in {B, W}    —Code label
    ...
  Union of sorts represent basic concepts such as memory locations and
  operands:
    cell!size = Union(aregister!size                  —Memory location
                    , dregister!size
                    , laddress!size)
    operand_src!size = Union(value!size               —Contents of memory
                           , data_label!size          —location or contents
                           , laddress!size)           —of register
    operand_dst!size = Union(data_label!size
                           , laddress!size
                           , aregister!size
                           , dregister!size
                           , code_label!size)
    operand_src = Union(operand_src_B
                      , operand_src_W
                      , operand_src_L)
    operand_dst = Union(operand_dst_B
                      , operand_dst_W
                      , operand_dst_L)
```

Operations

The operations of the target abstract type include:

- the usual arithmetic and logical operations:
  (value!size, value!size) → value!size:add!size
  (value!size, value!size) → value!size:sub!size
  ...
- the operations of sort modification that represents instructions:
  (operand_src!size, cell!size) → modification:assign!size
  (modification, modification) → modification:seq
  ...
- the operations describing access path to memory cells such as
  - the indexing operation which adds an offset to an address,
  - the access to the contents of an address (byte, word and long word),
  - the access to the contents of a data register or an address register:
    (laddress_L, value_W) → laddress!size:index!size
    (laddress!size) → operand_src!size:cont_of_laddress!size
    (aregister!size) → operand_src!size:cont_of_areg!size
    (dregister!size) → operand_src!size:cont_of_dreg!size

Instances

size in {B, W, L}

End target_machine

2. The code-generator generator

2.1. Introduction

The IR given as input to the code generation process is a sequence of modifications. Modifications M_i of the sequence are dealt with one after the other, trying to match each of them with an instruction template (because each instruction template corresponds exactly to one assembly instruction). Generally speaking, the modification M_i is not an instance of an instruction template. In that case, the code generation process transforms the tree M_i into a sequence of trees representing intermediate computations, followed by an instance of an instruction template.

Let us consider now the pattern matching process of a modification M_i with an instruction template. It is necessary to recognize the sons S_{ij} of M_i as operands of an instruction template, i.e. as elements of access classes. As an access class is a set of access modes, each son S_{ij} of M_i must be recognized first as an access mode.

When a subtree S_{ij} matches an access mode, it is replaced in M_i by a representative tree of the access mode which is the "canonical form" of the access mode. When the match fails, the largest subtree H_{ij} of S_{ij}, starting from the leaves, that matches an access mode is considered. It computes an intermediate operand which is stored in a temporary resource. The subtree H_{ij} in S_{ij} is replaced by the reference to the temporary resource.

2.2. Templates and canonical forms

2.2.1. Access mode templates

From the specification of access modes, the system deduces two families of trees: one for the access modes in source position, the other one for the access modes in destination position. In order to emit the corresponding code, it is necessary to store the format; this is done by means of the attribute *&format.* By convention, attribute names are preceded by "&".

For each tree template of these families, the system builds a tree corresponding to the related "canonical form" (see Fig. 2).

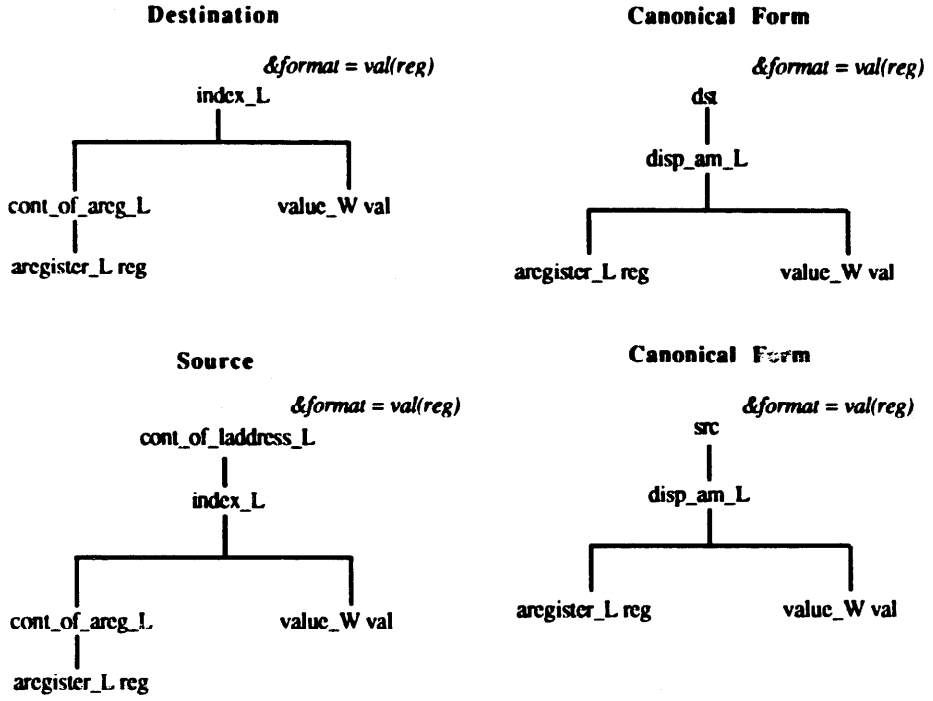

Fig. 2

2.2.2. Instruction templates

An instruction template is a tree (decorated with the code attribute), the parameters of which are either access modes in "canonical form" or access classes. The templates associated with a *move* instruction and with an *add* instruction are given in Fig. 3.

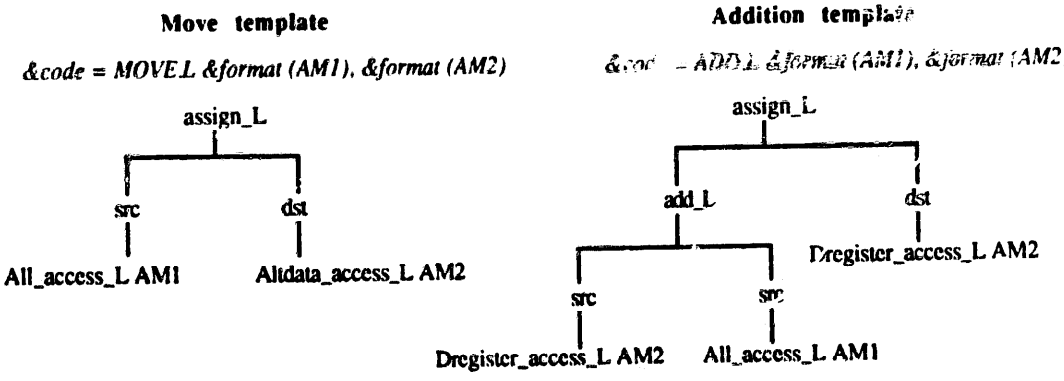

Fig. 3

2.3. Specification for intermediate representation rewritings

2.3.1. Universal operations and temporaries

When the matching of a subtree of the IR with an access mode template fails, the computation of an intermediate operand must be designed. Such a computation is stored in a temporary location. In order to perform this, the following sort and associated operation are provided by the system:

- the sort "temporary" with its related access mode: "temporary_am" and its dereference operation "contents_of",
- the universal store operation "Univ_assign" which allows to store a source operand in a temporary location.

The store of an intermediate operand is represented by a tree that must be rooted in sequence with the rewritten tree M_i (see Section 2.1), in order to preserve the whole calculus. The "Univ_seq" operator is provided by the system for that purpose.

In order to be able to produce code for trees involving universal operators, it is necessary to rewrite them into instances of instruction templates. That is the reason why the compiler writer must specify equivalence declarations between the machine operators and the universal operators. This is declared by the compiler writer in an interface between the machine specification and the code-generator writing system. During the binding step, the temporary access modes are bound to the access modes selected for the management of temporary locations.

The equivalence declarations for the MC68000 are specified below:

```
Univ_seq:seq
Univ_assign:assign!size   where size in {B, W, L}
temporary_am(⟨temporary temp⟩):
  areg_am!size(⟨aregister!size temp⟩)          where size in {W, L}
  | dreg_am!size(⟨dregister!size temp⟩)        where size in {B, W, L}
  | relative_am!size(⟨data_label!size temp⟩)   where size in {B, W, L}
```

The universal sequence operator "Univ_seq" is bound to the sequence operator "seq" of the target data type. The universal store operator "Univ_assign" is bound to different assign operators of the target data type (assignment on byte, word and long-word operands). The access mode on temporaries is bound to all access modes allowing direct access to an *address register*, to a *data register* or relative access to the memory. They are given in the preference order according to their cost; the binding step takes this order into account.

```
ADT universal use target_machine
  Sort
    temporary
  Operations
    (temporary)                  → operand_src:contents_of
    (operand_src, temporary)     → modification:Univ_assign
    (modification, modification) → modification:Univ_seq
```

Axioms
Univ_seq($U_1, \ldots, U_p$, Univ_seq($V_1, \ldots, V_n$))
= Univ_seq($U_1, \ldots, U_p, V_1, \ldots, V_n$)
where $U_1, \ldots, U_p$ **and** $V_1, \ldots, V_n$: modification
End universal

The universal sequence operator has the associativity property. Axioms dealing with the canonical form of the temporary access modes are defined in the following section.

2.3.2. Abstract data type of the canonical term

The axioms associated to the sort "operand" express the definition of the canonical form of the access modes. There is, however, a slight difficulty with the register access mode axioms. For such access modes, it is not possible to directly derive the axioms from the definition of the access paths in the two positions because the system would not be confluent. That is the reason why it is necessary to take into account the context of an instance of a register access mode. It is done using the context of the modification.

ADT canonical_form **use** target_machine, universal
Sort
operand!size = **Union**(operand_src!size, operand_dst!size)
operand = **Union**(operand_B, operand_W, operand_L)
Operations
(operand) → operand_src : src
(operand) → operand_dst : dst
(aregister!size) → operand!size : areg_am!size
(aregister!size) → operand!size : areg_ind_am!size
(aregister_L, value_W) → operand!size : disp_am!size
...
(temporary) → operand : temporary_am
Axioms
source_operand_context
(cont_of_areg!size(⟨aregister!size reg⟩)) **is true**
⇒ cont_of_areg!size(⟨aregister!size reg⟩)
= src(areg_am!size(⟨aregister!size reg⟩))
destination_operand_context(⟨aregister!size reg⟩) **is true**
⇒ ⟨aregister!size reg⟩
= dst(areg_am!size(⟨aregister!size reg⟩))
source_operand_context
(cont_of_laddress!size
(index!size(cont_of_areg_L(⟨aregister_L reg⟩
, ⟨value_W val⟩))))
is true

```
        ⇒ cont_of_laddress!size
            (index!size(cont_of_areg_L(⟨aregister_L reg⟩)
                                        ,⟨value_W val⟩))
          = src (disp_am!size(⟨aregister_L reg⟩
                                ,⟨value_W val⟩))
      destination_operand_context
        (index!size(cont_of_areg_L(⟨aregister_L reg⟩),⟨value_W val⟩))
          is true
        ⇒ index!size(cont_of_areg_L(⟨aregister_L reg⟩)
                                        ,⟨value_W val⟩)
          = dst(disp_am!size(⟨aregister_L reg⟩,⟨value_W val⟩))
      ...
      source_operand_context(contents_of(⟨temporary temp⟩)) is true
        ⇒ contents_of(⟨temporary temp⟩)
          = src(temporary_am(⟨temporary temp⟩))
      destination_operand_context(⟨temporary temp⟩) is true
        ⇒ ⟨temporary temp⟩
          = dst(temporary_am(⟨temporary temp⟩))
      instruction_context
        (Univ_assign
          (src(can), dst(temporary_am(⟨temporary temp⟩)))) is true
        ⇒ contents_of(⟨temporary temp⟩) = src(can)
      where
        can:operand
    Instances
      size in {B, W, L}
  End canonical_form
```

The new modifications appearing in the rewritten tree deal with the universal store operator and the sequence operator. As the "Univ_assign" operator applies on an access mode in source position which is stored in a temporary in destination position, the axiom describing the semantics of "Univ_assign" has a meaning only in that case. From now on, the target abstract data type will be the union of the abstract data type related to the target machine and of the universal operators and the abstract data type of the canonical form.

2.4. Intermediate representation rewritings

2.4.1. Introduction

The rewriting algorithm belongs to the target-machine-independent kernel of the code generator. At present this algorithm applies to an IR corresponding to a sequence of modifications.

For each IR term the rewriting algorithm needs to know the boundary where the access mode pattern matching can stop and where the instruction pattern matching

can begin. For this purpose, we define a partition of instruction templates into instruction classes that have the same boundary.

The leaves of instruction templates are denotation of access classes (they can be considered as typed variables) or access modes. Two templates of an instruction class can only differ on the type of the variables. We are interested in the "canonical representative" of such a class, i.e. a tree in which all the sons of src and dst are replaced by variables where each variable has a single occurrence. If we consider all the addition instruction templates, the canonical representative of the instruction class is:

assign!size(add!size(src(X), src(Y)), dst(Z))

2.4.2. The rewriting rules of a modification

Each term of the IR corresponding to a modification is rewritten until the pattern matching with an instruction template succeeds.

The strategy of application of the rewriting rules is strongly connected with the notion of canonical representative of an instruction class which defines the context of research of access modes.

The algorithm of Section 2.4.3 describes the strategy of applications of the rewriting rules.

2.4.2.1. Introduction The transformation rules of the IR are not rewriting rules in the common sense of term rewriting systems [14].

Usually a rewriting rule is a formula of the form

$$\lambda \rightarrow \rho.$$

If t is a term filtered by λ, i.e. there exists a minimal substitution σ such that $\sigma(\lambda) = t$, then t is rewritten into t' such that $t' = \sigma(\rho)$.

There are major differences in the way our system applies rewriting rules. The first one is that the filtering process by a pattern is replaced by the composition of filtering processes with patterns belonging to different sets of patterns. The second one is that the variables appearing in the pattern terms have restricted definition domains. In a certain extend, they are "sorted" variables. Thus when filtering, it is necessary to verify if a subterm related to a variable belongs to the definition domain of the variable. So, we get as in [16] a notion of positive/negative conditional rewritings.

Each transformation rule consists of:

- the description of the input pattern defined in an operational way by the composition of the filtering processes on the different sets of patterns;
- the conditions that must be verified by the variable instances during the filtering process;
- the pattern of the output tree.

The description of the first two points are interleaved.

2.4.2.2. Different kinds of rewriting rules

Notation.

- Let AM_{source} be the ordered set of access mode patterns in source position.
- Let $AM_{destination}$ be the ordered set of access mode patterns in destination position.
- Let I be the ordered set of instruction patterns.
- Let IC be the ordered set of instruction class patterns.
- Let can(A) denote the canonical form of the access mode whose pattern is A.
- Let TR be the set of access mode patterns (in canonical form) that are defined by the compiler writer as possible rewritings of the temporary access mode.
- Let CA be the set of access classes. Any element of CA is defined as a set of access modes.
- Let INST(X) denote the set of instances of the patterns belonging to the set X.
- Let $t/[t_i = a]$ denote the tree t' obtained by replacement of the subtree t_i in t by the tree a.
- All the sets of patterns defined here are ordered.
- In all the following rules, the search for a pattern of a set of patterns that matches a term is done by trying the patterns of the set one after the other with respect to the set ordering.

Definition

(1) A substitution pair $\langle x, e\rangle$ consisting of a variable x and of a term e is said to pertain to variable x.

(2) A substitution $\sigma = \{\langle x_i, e_i\rangle \mid 1 \leq i \leq n\}$ is a finite set of substitution pairs pertaining to distinct variables. We ignore pairs such as $\langle x, x\rangle$ in σ.

Rule R1. *When matching a tree t with an instruction class pattern, the source and destination position contexts are set according to the position nodes in the instruction class pattern. If there is a subtree t_i of t which is an instance of an access mode pattern in the right position, then t_i is replaced in t by the instantiated canonical form of the access mode pattern. The instruction class and access mode patterns are matched with respect to the ordering of the two sets* IC *and* AM.

Let t be the tree to transform. Suppose there exist $T \in$ IC and a substitution σ_{IC} such that

$$\sigma_{IC} = \{\langle \Delta_i, t_i\rangle \mid \Delta_i \in \mathrm{Var}(T), 1 \leq i \leq n\} \quad \text{with } \sigma_{IC}(T) = t.$$

Then, if there exists a subtree t_i of t such that $\langle \Delta_i, t_i\rangle \in \sigma_{IC}$ and if the context of Δ_i in T is the position

$$\mathrm{pos} \in \{\text{source, destination}\},$$

and there exist $A \in AM_{pos}$ and a substitution σ_{AM} such that $\sigma_{AM}(A) = t_i$, then we

define the rewriting of the tree t by:

$$t \to t/[t_i = \sigma_{AM}(\text{can}(A))].$$

Rule R2. *When matching a tree t with an instruction class pattern, if we find a subtree t_i of t which is not an instance of an access mode pattern, then, starting from the leaves of t_i, we look for the biggest subtree t_{ij} of t_i which is an instance of an access mode in source position. A universal assignment tree is built whose source operand is the instantiated canonical form of the access mode corresponding to t_{ij} and whose destination operand is the reference to a new temporary location. The subtree t_{ij} is replaced in t_i by the temporary access mode in source position applied to this new temporary location. The universal assignment tree and the rewritten tree are rooted by a universal sequence operator.*

Let t be the tree to transform. Suppose there exist $T \in \text{IC}$ and a substitution σ_{IC} such that

$$\sigma_{IC} = \{\langle \Delta_i, t_i \rangle \mid \Delta_i \in \text{Var}(T), 1 \leq i \leq n\} \quad \text{with } \sigma_{IC}(T) = t.$$

Also, suppose there exists t_i such that $\langle \Delta_i, t_i \rangle \in \sigma_{IC}$ and, if the context of Δ_i in T is the position pos, and for all $A \in \text{AM}_{\text{pos}}$ there exists no substitution μ such that $\mu(A) = t_i$, then, if there exist:

- a largest subtree t_{ij} starting from the leaves of t_i,
- $B \in \text{AM}_{\text{source}}$,
- a substitution γ such that $\gamma(B) = t_{ij}$,

we define the rewriting of the tree t by:

$$\begin{aligned} t \to \text{Univ\_seq}(&\text{Univ\_assign}(\gamma(\text{can}(B)) \\ &, \delta(\text{dst}(\text{temporary\_am}(\langle \text{temporary } \theta \rangle)))) \\ &, t/[t_{ij} = \delta(\text{src}(\text{temporary\_am}(\langle \text{temporary } \theta \rangle)))]) \end{aligned}$$

where δ is the substitution

$$\delta = \{\langle \theta, \text{tmp}\alpha \rangle\}$$

where tmpα is a new temporary location.

Rule R3. *When matching a tree t with an instruction class pattern, if there is a subtree t_i of t which is not an instance of any access mode pattern, then starting from the leaves of t_i we look for a subtree t_{ij} of t_i which is an instance of an access mode after replacement of a temporary access mode by one of the access modes defined in the rewriting rules declared in the interface and the replacement is done.*

Let t be the tree transform. Suppose there exist $T \in \text{IC}$ and a substitution σ_{IC} such that

$$\sigma_{IC} = \{\langle \Delta_i, t_i \rangle \mid \Delta_i \in \text{Var}(T), 1 \leq i \leq n) \quad \text{with } \sigma_{IC}(T) = t.$$

Also, suppose there exists t_i a subtree of t such that $\langle \Delta_i, t_i \rangle \in \sigma_{IC}$, and, if the context of Δ_i in T is the position pos, and for each $A \in \mathrm{AM}_{pos}$ there exists no substitution μ such that $\mu(A) = t_i$, then, when the following conditions hold:

- there exists a subtree t_{ik} of t_i and a substitution τ, such that

$$\tau(\mathrm{src}(\mathrm{temporary\_am}(\langle \mathrm{temporary}\ \theta \rangle))) = t_{ik},$$

- there exists $R \in \mathrm{TR}$ and $M_R \in \mathrm{AM}_{source}$ such that $\mathrm{src}(R) = \mathrm{can}(M_R)$,
- there exists a biggest subtree t_{ij}, starting from the leaves of t_i, strictly larger than t_{ik} such that:
  - there exists $M \in \mathrm{AM}$ ($\mathrm{AM} = \mathrm{AM}_{pos}$ when $t_{ij} = t_i$ or $\mathrm{AM} = \mathrm{AM}_{source}$ when $t_{ij} < t_i$),
  - a substitution δ such that $\delta(M) = t_{ij}/[t_{ik} = \tau(M_R)]$,

we define the rewriting of the tree t by:

$$t \rightarrow t/[t_{ik} = \tau(M_R)].$$

Rule R4. *The universal sequence operator has the associativity property*:

$$\mathrm{Univ\_seq}(U_1, \ldots, U_p, \mathrm{Univ\_seq}(V_1, \ldots, V_n)) \rightarrow \mathrm{Univ\_seq}(U_1, \ldots, U_p, V_1, \ldots, V_n).$$

Rule R5. *If neither Rule* R1, *nor Rule* R2, *nor Rule* R3 *can be applied and a subtree of t on the destination position context is not reduced to the canonical form of an access mode in the destination position, then the subtree is matched with access mode patterns in source position. If it succeeds, a universal assignment tree is built whose source operand is the selected access mode in its canonical form. The subtree is replaced in t by the temporary access mode in source position applied to a new temporary location. The universal assignment tree and the rewritten tree are rooted by a universal sequence operator.*

Let t be the tree to transform. Suppose there exist $T \in \mathrm{IC}$ and a substitution σ_{IC} such that

$$\sigma_{IC} = \{\langle \Delta_i, t_i \rangle \mid \Delta_i \in \mathrm{Var}(T), 1 \leq i \leq n\} \quad \text{with } \sigma_{IC}(T) = t$$

and neither Rules R1, R2, R3 can be applied. Then, there exists a subtree t_i of t such that $\langle \Delta_i, t_i \rangle \in \sigma_{IC}$ and, if the context of Δ_i in T is destination and there exist $A \in \mathrm{AM}_{source}$ and a substitution σ_{AM} such that $\sigma_{AM}(A) = t_i$, then we define the rewriting of the tree t by:

$$t \rightarrow \mathrm{Univ\_seq}(\mathrm{Univ\_assign}(\sigma_{AM}(\mathrm{can}(A)), \delta(\mathrm{dst}(\mathrm{temporary\_am}(\langle \mathrm{temporary}\ \theta \rangle))), t/[t_i = \delta(\mathrm{src}(\mathrm{temporary\_am}(\langle \mathrm{temporary}\ \theta \rangle)))]).$$

Rule R6. *If neither Rule* R1, *nor Rule* R2, *nor Rule* R3 *can be applied and a subtree of t_i of t on the destination position context is not reduced to the canonical form of an access mode in the destination position, then the subtree t_i is matched with access mode patterns in source position. If it fails and if there exists an inner subtree which is a temporary access mode, such a tree is rewritten according to the rewriting rules of the interface in the source position context for the subtree t_i.*

Let t be the tree to transform. Suppose there exist $T \in \mathrm{IC}$ and a substitution σ_{IC} such that

$$\sigma_{\mathrm{IC}} = \{\langle \Delta_i, t_i \rangle | \Delta_i \in \mathrm{Var}(T), 1 \leq i \leq n\} \quad \text{with } \sigma_{\mathrm{IC}}(T) = t$$

and neither Rules R1, R2, R3 can be applied. Then, there exists a subtree t_i of t such that $\langle \Delta_i, t_i \rangle \in \sigma_{\mathrm{IC}}$ and, if the context of Δ_i in T is destination, and there exist a subtree t_{ik} of t_i and a substitution τ such that

$$\tau(\mathrm{src}(\mathrm{temporary\_am}(\langle \mathrm{temporary}\ \theta \rangle))) = t_{ik}$$

and there exists $R \in \mathrm{TR}$ and $M_R \in \mathrm{AM}_{\mathrm{source}}$ such that

$$\mathrm{src}(R) = \mathrm{can}(M_R),$$

and there exists $M \in \mathrm{AM}_{\mathrm{source}}$ and a substitution δ such that

$$\delta(M) = t_i/[t_{ik} = \tau(M_R)],$$

then we define the rewriting of the tree t by:

$$t \to t/[t_{ik} = \tau(M_R)].$$

Rule R7. *If a subtree t_i of t in the destination position is an instance of a temporary access mode in source position, and there exists an access mode A of the interface whose source template is the destination template of an access mode B, then t_i is replaced by the instantiated access mode B in destination position.*

Let t be the tree to transform. Suppose there exist $T \in \mathrm{IC}$ and a substitution σ_{IC} such that

$$\sigma_{\mathrm{IC}} = \{\langle \Delta_i, t_i \rangle | \Delta_i \in \mathrm{Var}(T), 1 \leq i \leq n\} \quad \text{with } \sigma_{\mathrm{IC}}(T) = t.$$

Then, there exists a subtree t_i of t such that $\langle \Delta_i, t_i \rangle \in \sigma_{\mathrm{IC}}$ and if the context of Δ_i in T is destination and there exists a substitution τ such that

$$\tau(\mathrm{src}(\mathrm{temporary\_am}(\langle \mathrm{temporary}\ \theta \rangle))) = t,$$

and there exists $R \in \mathrm{TR}$ and $M_R \in \mathrm{AM}_{\mathrm{source}}$ such that $R = \mathrm{can}(M_R)$, and there exist $B \in \mathrm{AM}_{\mathrm{destination}}$ and a substitution δ such that $\delta(B) = \tau(M_R)$, then we define the rewriting of the tree t by:

$$t \to t/[t_i = \delta(B)].$$

Rule R8. *If the matching of a tree t with an instruction pattern I fails because a subtree of t which corresponds to a source operand of I does not belong to the right access class and if there exists a rewriting of a temporary access mode which belongs to the right class, then a universal assignment tree is built and, in the tree t, the subtree corresponding to the source operand is replaced by the selected rewriting of the temporary access mode.*

Let t be the tree to transform. Suppose there exist $T \in \mathrm{IC}$ and a substitution σ_{IC} such that

$$\sigma_{\mathrm{IC}} = \{\langle \Delta_i, t_i \rangle \mid \Delta_i \in \mathrm{Var}(T),\ t_i \in \mathrm{INST(AM)},\ 1 \leq i \leq n\}$$

$$\text{with } \sigma_{\mathrm{IC}}(T) = t, \quad \mathrm{AM} = \mathrm{AM}_{\mathrm{source}} \cup \mathrm{AM}_{\mathrm{destination}}.$$

Let I be the instruction pattern of the instruction class T which is the closest to t, and let γ be the substitution such that $\gamma(T) = I$:

$$\gamma = \{\langle \Delta_i, a_i \rangle \mid \Delta_i \in \mathrm{Var}(T),\ a_i \in \mathrm{CA} \cup \mathrm{INST(AM)}\}.$$

Then, there exists t_i in source position in T such that for all $A \in a_i$, $(\langle \Delta_i, a_i \rangle)$ and $a_i \in \mathrm{CA}$, t_i has the following property: $t_i \notin \mathrm{INST}(A)$ and, if there exist $R \in \mathrm{TR}$ and $R \in a_i$, then we define the rewriting of the tree t by:

$$t \rightarrow \mathrm{Univ\_seq}(\mathrm{Univ\_assign}(t_i, \mathrm{dst}(\delta(R))), t/[t_i = \mathrm{src}(\delta(R))]),$$

where δ is the substitution

$$\delta = \{\langle \theta, \mathrm{tmp}\alpha \rangle\},$$

where $\mathrm{tmp}\alpha$ is a new temporary location.

Rule R9. *The goal is to match the tree t by an instruction pattern where a variable A occurs twice (once in source position, once in destination position). The matching of t fails because the subtrees of t corresponding to the two occurrences of A are not identical or they are identical but they do not belong to the right access class, a rewriting must be done. First, the subtree of t corresponding to the source operand is saved in a temporary; second, an instance of the instruction pattern is generated using this temporary to instantiate the variable A; and third, a universal assignment tree is created whose destination is the subtree of t corresponding to the destination operand.*

Let t be the tree to transform. Suppose there exist $T \in \mathrm{IC}$ and a substitution σ_{IC} such that

$$\sigma_{\mathrm{IC}} = \{\langle \Delta_i, t_i \rangle \mid \Delta_i \in \mathrm{Var}(T),\ t_i \in \mathrm{INST(AM)},\ 1 \leq i \leq n\}$$

$$\text{with } \sigma_{\mathrm{IC}}(T) = t.$$

Let I be the instruction pattern of the instruction class T which is the closest to t, and let γ be the substitution such that $\gamma(T) = I$:

$$\gamma = \{\langle \Delta_i, a_i \rangle \mid \Delta_i \in \mathrm{Var}(T),\ a_i \in \mathrm{CA} \cup \mathrm{INST(AM)}\}.$$

Then, there exist $a_i, a_j \in CA$, $\langle \Delta_i, a_i \rangle \in \gamma$ and $\langle \Delta_j, a_j \rangle \in \gamma$, $i \neq j$, such that $\text{Var}(a_i) = \text{Var}(a_j)$. (By definition of an access class, $\text{Var}(a_i)$ is reduced to a single variable.) Let α_1 and α_2 be the substitutions such that

$$t_i = \alpha_1(\text{src}(A)), \qquad t_j = \alpha_2(\text{dst}(A)), \qquad a_1 = \{\langle A, M \rangle\}).$$

If $\alpha_1 \neq \alpha_2$ or if $\alpha_1 = \alpha_2$ and for all $P \in a_i$, $M \notin \text{INST}(P)$, and a_i is a source operand and a_j is a destination operand in I, then we define the rewriting of the tree t by:

$$t \rightarrow \text{Univ\_seq}(\text{Univ\_assign}(t_i, \delta(\text{dst}(\text{temporary\_am}(\langle \text{temporary } \theta \rangle))) , \delta_2 \circ \delta_1(I)) , \text{Univ\_assign}(\delta(\text{src}(\text{temporary\_am}(\langle \text{temporary } \theta \rangle))), t_j)),$$

where δ is the substitution

$$\delta = \{\langle \theta, \text{tmp}\alpha \rangle\},$$

where $\text{tmp}\alpha$ is a new temporary location, δ_1 is the substitution

$$\delta_1 = \{\langle \text{Var}(a_i), \delta(\text{src}(\text{temporary\_am}(\langle \text{temporary } \theta \rangle))) , \langle \text{Var}(a_j), \delta(\text{dst}(\text{temporary\_am}(\langle \text{temporary } \theta \rangle)))\rangle\},$$

and δ_2 is the substitution

$$\delta_2 = \{\langle \text{Var}(a_k), t_k \rangle \mid 1 \leq k \leq n, k \neq j, k \neq i, \langle \Delta_k, t_k \rangle \in \sigma_{IC} \text{ and } a_k \in CA\}.$$

Rule R10. *If the matching of a tree t with an instruction pattern I fails, because a subtree of t which corresponds to a destination operand of I does not belong to the right access class. If there exists a rewriting of a temporary access mode which belongs to the right class, the subtree of t corresponding to the destination operand is replaced by the selected rewriting of the temporary access mode, and a universal assignment tree is built whose destination operand is the previous destination operand of t.*

Let t be the tree to transform. Suppose there exist $T \in IC$ and a substitution σ_{IC} such that

$$\sigma_{IC} = \{\langle \Delta_i, t_i \rangle \mid \Delta_i \in \text{Var}(T), t_i \in \text{INST(AM)}, 1 \leq i \leq n\}$$

$$\text{with } \sigma_{IC}(T) = t, \quad \text{AM} = \text{AM}_{\text{source}} \cup \text{AM}_{\text{destination}}.$$

Let I be the instruction pattern of the instruction class T which is the closest to t, and let γ be the substitution such that $\gamma(T) = I$:

$$\gamma = \{\langle \Delta_i, a_i \rangle \mid \Delta_i \in \text{Var}(T), a_i \in CA \cup \text{INST(AM)}\}.$$

Then, there exists t_i in destination position in T such that for all $A \in a_i$, $(\langle \Delta_i, a_i \rangle)$ and $a_i \in CA$, t_i has the property: $t_i \notin \text{INST}(A)$ and if there exist $R \in TR$ and $R \in a_i$, then we define the rewriting of the tree t by:

$$t \rightarrow \text{Univ\_seq}(t/[t_i = \text{dst}(\delta(R))], \text{Univ\_assign}(\text{src}(\delta(R)), t_i)),$$

where δ is the substitution

$$\delta = \{\langle \theta, \text{tmp}\alpha \rangle\},$$

where tmpα is a new temporary location.

2.4.3. The code generation algorithm

2.4.3.1. The instruction selection algorithm Let t be the IR tree of a modification:

Step 1. Apply either Rule R1 or Rule R2 or Rule R3 whenever possible.
Step 2. Apply either Rule R5 or Rule R6 or Rule R7 whenever possible.
Step 3. Apply Rule R9.
Step 4. Apply Rule R8.
Step 5. Apply Rule R10.
Step 6. Apply Rule R4.

Every application of a rewriting rule that searches for an access mode must have as result the access mode corresponding to the largest subtree starting from the leaves (it leads to locally optimal code).

In practice, when a rewriting rule uses the rewriting rules of the interface, it is necessary to keep the result of all successful applications of these rules.

2.4.3.2. Binding and register assignment Locally, for each tree of this sequence using a temporary, the selection instruction process builds a list of allowed access modes for this temporary access mode. For each temporary, such information is processed over the whole tree in order to get the list of the access modes available all over the sequence.

Then, according to the available storages, the system first picks for each temporary the storage class related to the cheapest access mode of the preceding set and then, it chooses the name of an available location of this storage class for the temporary.

2.4.4. Example

Let us consider in the C language, the IR subtree t corresponding to the assignment statement:

$$*J = \text{cpt} + *I$$

where I and J are formal parameters of a procedure, declared as pointers on integer; cpt is a local variable of type integer.

A tree which designates a variable in a call can be written by indexing the offset of the variable to the base content of the procedure to which this variable is local. We assume that (see Fig. 4):

- the address register a_6 points to the beginning of the activation record of the current call,
- the offset of I from the base of the procedure is 8,
- the offset of J from the base of the procedure is 12,
- the offset of cpt from the base of the procedure is -4.

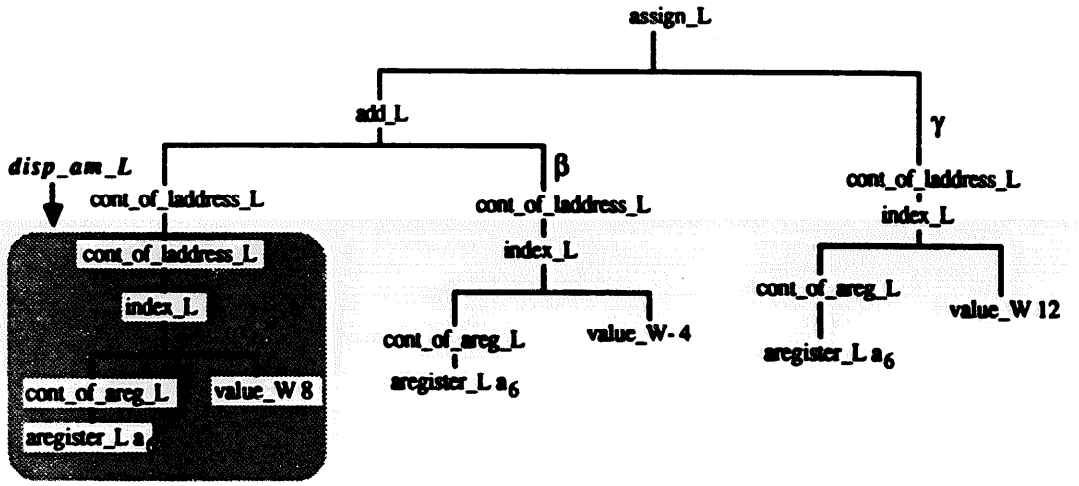

Fig. 4

Figures 5-15 show successive rewritings. The access modes are designated by dotted squares. The instruction pattern candidate to a matching of the modification of root assign_L may be as in Fig. 5.

Step a. The disp_am access mode in source position is recognized. As its father is not a node of an instruction template, we apply Rule R2 using the temporary tmp1 as temporary location. We generate a universal store tree such that (Fig. 6):

- its right child is an access mode temporary_am in destination position,
- its left child is the instantiated access mode disp_am.

Step b. Using the equivalence declaration of the interface, Rule R3 can be applied. Since

areg_am_L(⟨aregister_L tmp1⟩)

is a rewriting of the temporary access mode and since

src(areg_am_L(⟨aregister_L tmp1⟩))

is the canonical form of cont_of_areg_L(⟨aregister_L tmp1⟩),

src(temporary_am(⟨temporarary tmp1⟩))

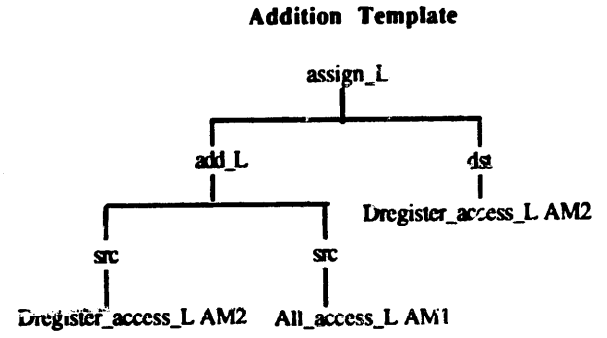

Fig. 5

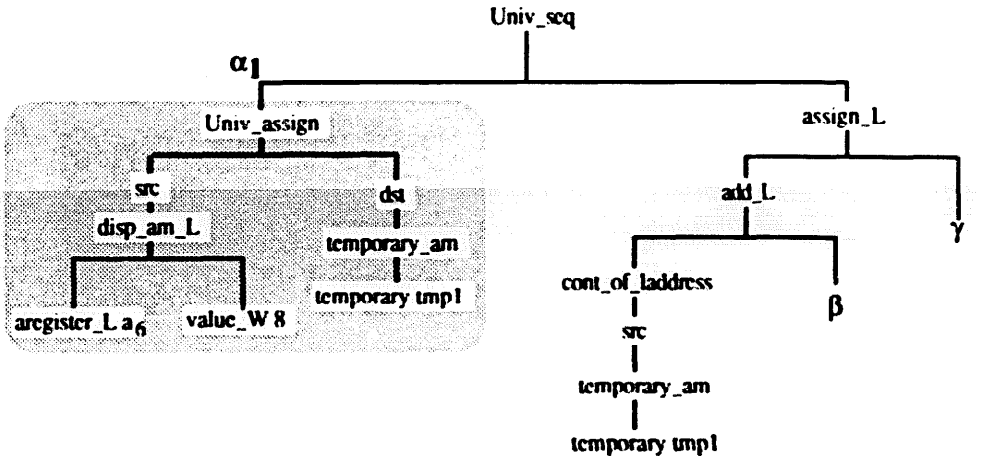

Fig. 6

is rewritten in cont_of_areg_L(⟨aregister_L tmp1⟩) in order to get an instance of the areg_ind_am_L access mode (Fig. 7).

Step c. Rule R1 may be applied because an instance of the address register indirect access mode is recognized in source position. It is replaced by the instantiated canonical form of the access mode in the tree (Fig. 8).

Step d. In the same way, Rule R1 may be applied because the tree β is recognized as an instance of the disp_am access mode in source position. β is replaced by the instantiated canonical form of the disp_am access mode (Fig. 9).

Step e. Now let us consider the γ tree; it is the destination position. But γ is not an instance of an access mode in destination position. Below the root of γ, there is no instance of access mode in source position. It is necessary to use a new strategy since γ is an instance of the disp_am access mode in source position, we apply Rule R5 (see Fig. 10).

Step f. The tree γ has a root labelled by src in a destination position. It is an inconsistency that we need to delete. This is the purpose of Rule R7. Since the temporary access mode is equivalent to the aregister access mode via the interface,

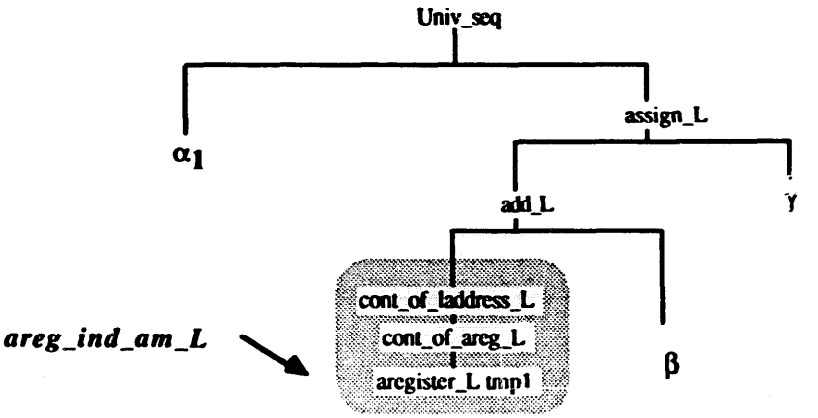

Fig. 7

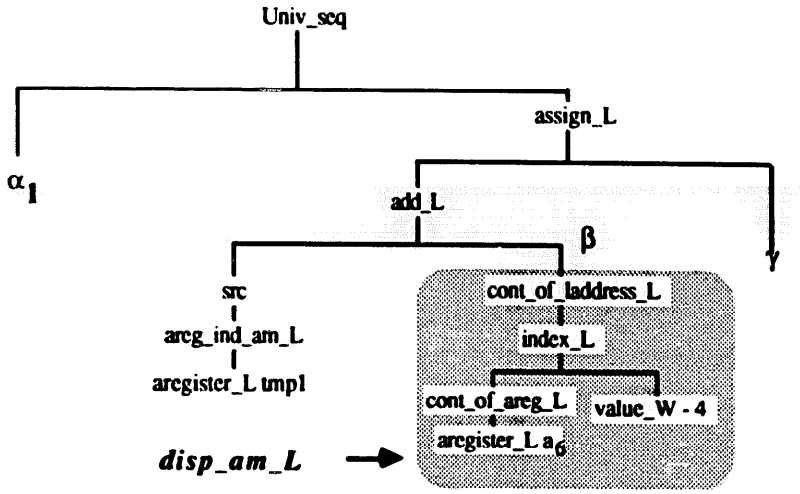

Fig. 8

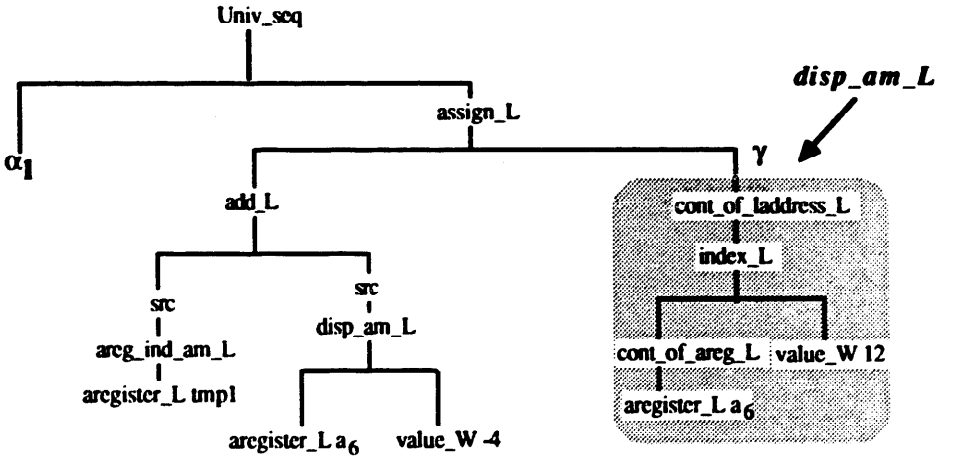

Fig. 9

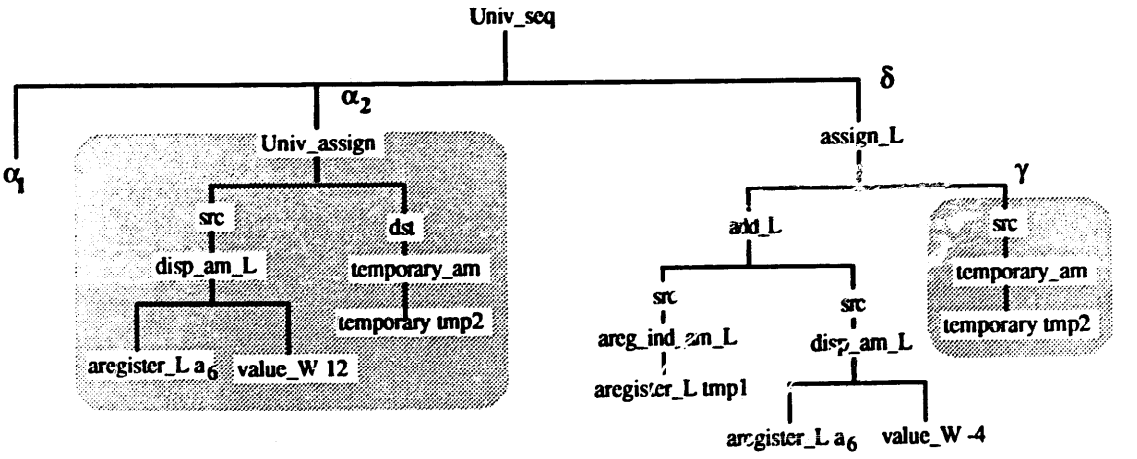

Fig. 10

let us replace the temporary access mode by the full template definition of the aregister access mode because it corresponds to the template of the areg_ind_am access mode in destination position. See Fig. 11.

Step g. As Rules R1, R2, R3, R5, R6, R7 are no more applicable, we compare t with the instruction template where a variable occurs twice (once in source position, once in destination position). The matching of t fails because the subtrees S_1 and S_3 are not identical. That is an application of Rule R9 (see Fig. 12).

The Rule R4 describing the associativity of the Univ_seq operator can be applied twice. The sequence is shown in Fig. 13.

Now, each subtree of the sequence is an instance of instructions templates.

It is now necessary to bind each temporary with an actual resource. In the sequence obtained from the instruction selection step, all temporaries have already been bound to storage classes. That is the case of tmp1 bound to an address register because Step b used the equivalence rule between the temporary access mode and the address access mode. In the same way tmp2 is bound to an address register (by Step f). The data register access mode is the only allowed rewriting of the temporary access mode on tmp3 according to the addition template. Thus tmp3 is bound to a data register.

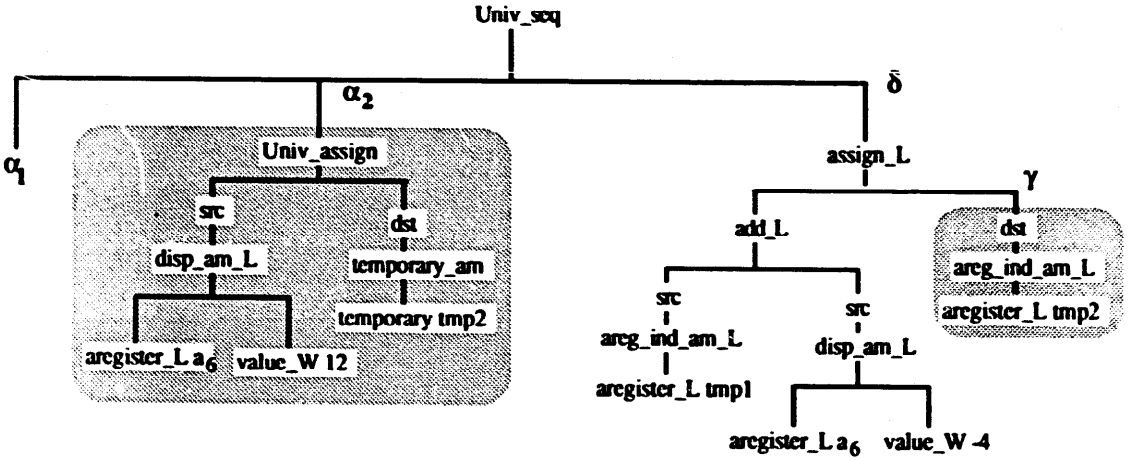

Fig. 11

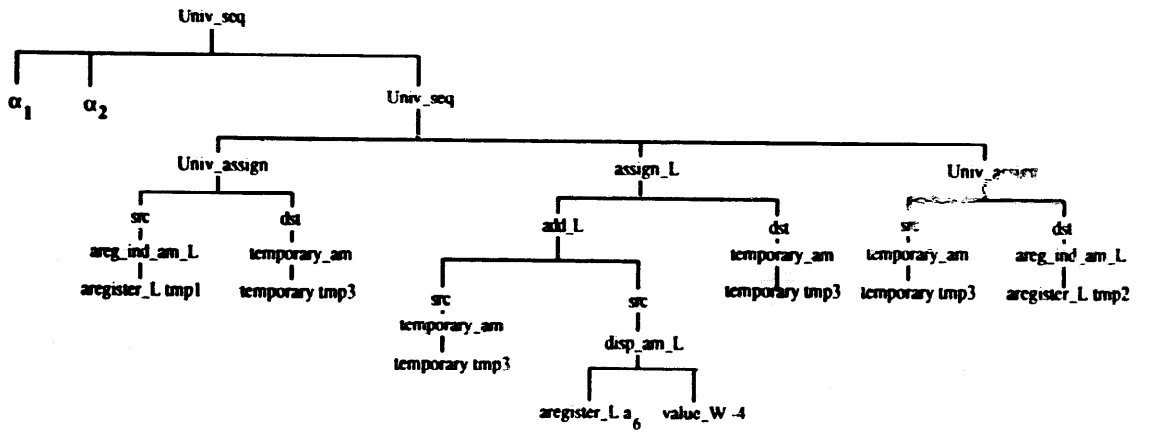

Fig. 12

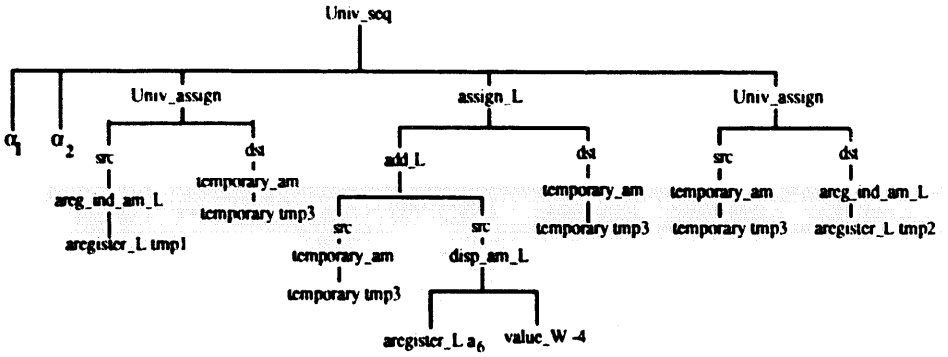

Fig. 13

Finally, the universal operators are rewritten in order to get instances of instructions templates. The result of the binding step is the sequence in Fig. 14.

The register assignment step can now take place. Each temporary name tmp_i is replaced by a name of an actual resource (a_0, is assigned to tmp1, a_1 to tmp2, d_0 to tmp3). The final sequence is shown in Fig. 15.

As each instruction template corresponds exactly to an assembly instruction, the code emission can be performed by a simple left-to-right tree traversal producing the following sequence of assembly code:

```
MOVE.L  8(a6), a0
MOVE.L  12(a6), a1
MOVE.L  (a0), d0
ADD.L   -4(a6), d0
MOVE.L  d0, (a1)
```

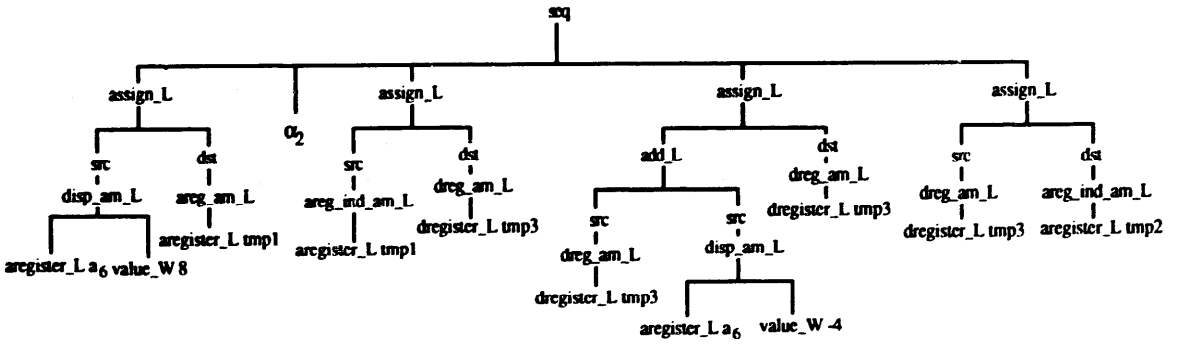

Fig. 14

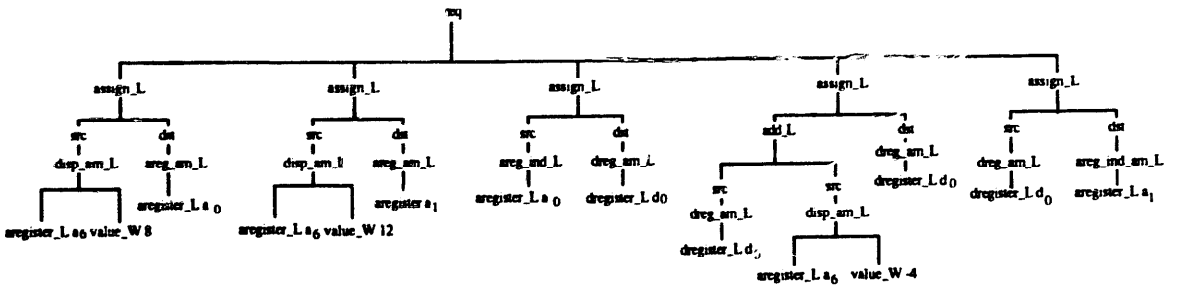

Fig. 15

3. Correctness of the code generator produced

3.1. Introduction

We intend to prove that the rewriting of a term A into a term B preserves the semantics of the term A.

A term of the IR is a modification corresponding to an instruction of the source program. Let us recall briefly from the theory developed by Gaudel in her thesis that the semantics of a modification can be written by means of a primitive modification (substitution) applied to a state S. A state is set of formulas $t = t'$ (where t and t' are ground terms of the abstract data type).

Let m be a term. We note

$$S' = appl(m, S),$$

with S' the resulting state when applying the modification denoted by m to the state S.

The action of a primitive modification $subst(f(\lambda), \mu)$ on a state S is a new state S' where the formulas of S corresponding to the old value of f at the point λ will be replaced by formulas taking into account the new value μ. The new state S' is the smallest consistent state built from the formulas of S and the substitution.

Thus the definition of the assign operation is:

$$\text{assign}(v, i) = subst(\text{value\_of}(v), i).$$

As the assignment instruction template which is dealt with by the generator process is of the form:

$$\text{assign}(\text{scr}(\text{can}_1), \text{dst}(\text{can}_2)),$$

the assign modification has the following definition:

$$appl(\text{assign}(\text{scr}(\text{can}_1), \text{dst}(\text{can}_2)), S) = appl(subst(\text{contents}(\text{dst}(\text{can}_2), \text{src}(\text{can}_1)), S)$$

or

$$\text{assign}(\text{scr}(\text{can}_1), \text{dst}(\text{can}_2)) = subst(\text{contents}(\text{dst}(\text{can}_2)), \text{src}(\text{can}_1))$$

A convenient abbreviation used in the sequel will be:

$$Sem[\text{assign}(\text{scr}(\text{can}_1), \text{dst}(\text{can}_2)) = \{\text{contents}(\text{dst}(\text{can}_2) = \text{src}(\text{can}_1)\}.$$

The compiler writer has to specify this semantic function in the axioms in a way like:

$$\text{instruction\_context}(\text{assign}(\text{src}(\text{can}_1), \text{dst}(\text{can}_2))) \text{ is true} \Rightarrow \text{CONTENTS}[\text{dst}(\text{can}_2)] = \text{src}(\text{can}_1).$$

The compiler writer must define the axioms describing the semantics of the modification "modif_name" appearing in the instruction. For that purpose, he uses a built-in function, CONTENTS. When applied to a storage designated by dst(X), this function returns the term denoting the dereference operation applied to this storage.

For instance, for the temporary_am access mode, we define:

$$\text{CONTENTS}[\text{dst}(\text{temporary\_am}(\langle\text{temporary temp}\rangle))] = \text{contents\_of}(\langle\text{temporary temp}\rangle).$$

The axiomatic definition of the modification modif_name must be written as follows:

$$\text{instruction\_context}(\text{modif\_name}(\text{op}(\text{src}(\text{can}_1), \text{src}(\text{can}_2), \text{dst}(\text{can}_3)))) \text{ is true}$$
$$\Rightarrow \text{CONTENTS}[\text{dst}(\text{can}_3)] = \text{SEM\_VALUE}[\text{modif\_name}, \text{op}(\text{src}(\text{can}_1), \text{src}(\text{can}_2))]$$

where SEM_VALUE is a user-defined semantic function.

It is now necessary to enrich our definition for a sequence of modifications. The semantics of a sequence is defined as a composition of applications *appl*:

$$\textit{appl}(\text{seq}(m_1, m_2), S) = = \textit{appl}(m_2, \textit{appl}(m_1, S)).$$

We shall apply such a composition under a simplification hypothesis. As each rewriting generates always a new name for a temporary, the temporaries generated have a local scope. As a consequence, the states we are interested in are the initial and the final states. The intermediate states generated by the composition of *appl* inside a sequence cannot generate inconsistency with the initial state. We shall use the notation:

$$\textit{Sem}[\text{seq}(m_1, m_2)] = \textit{Sem}[m_1] \oplus \textit{Sem}[m_2]$$

where $\oplus$ is the adjunction of formulas; i.e. the smallest consistent state built from the composition of the formulas of S, $\textit{Sem}(m_1)$ and $\textit{Sem}(m_2)$.

Since the code generator works by successive rewritings of a term into another term, the correctness of the code-generator process can be achieved by proving each elementary rewriting.

3.2. Correctness proof of the rewriting steps

In order to prove the correctness of the rewriting step, it would be necessary to prove that each rewriting rule preserves the semantics. The full proof of all rewriting rules is not given here. Most of them are alike:

- Rule R1 replaces an instance of an access mode template by its instantiated canonical form. Such a transformation is the straightforward application of the axiomatic definition of the canonical form of the access mode.

- Rules R3, R6, R7 are compositions of the axiomatic definitions of the access modes and of the equivalence rules of the interface. Their correctness proof is also straightforward.
- Rule R2 consists in the store of an instance of an access mode in a temporary followed by the replacement of this instance in the input term by the access to a temporary. The correctness proofs of Rules R5 and R8 are analogous.
- Rule R9 deals with terms whose leaves are in the canonical form but do not satisfy the binding of the variables according to the instruction template.

We choose to present the scheme of the correctness proof for Rules R2 and R9.

3.2.1. Correctness proof of Rule R2

The IR term I to which Rule R2 is applied is of the form shown in Fig. 16, which is matched by the instruction class:

$$\text{modif\_name}(\text{src}(X), \text{dst}(Y)).$$

In the term I, let us consider an inner subterm of root h which is an instance of an access mode can_1. It is convenient to denote it by $\text{src}(\text{can}_1)$ without loss of generality. In the same way, the subtree of root d will be denoted by $\text{dst}(\text{can}_2)$ in the sequel. Thus a convenient abbreviation of the term I could be:

$$\text{modif\_name}(f(g(\text{src}(\text{can}_1))), \text{dst}(\text{can}_2)).$$

The semantics related to I is described by the state:

$$S_I = \{Sem[\text{modif\_name}(f(g(\text{src}(\text{can}_1))), \text{dst}(\text{can}_2))]\}.$$

The initial term I is rewritten into the term F:

$$\begin{aligned}&\text{Univ\_seq}(\text{Univ\_assign}(\text{src}(\text{can}_1), \text{dst}(\text{temporary\_am}(\langle\text{temporary temp}_1\rangle))))\\ &\quad, \text{modif\_name}(f(g(\text{src}(\text{temporary\_am}(\langle\text{temporary temp}_1\rangle)))))\\ &\quad\quad, \text{dst}(\text{can}_2)))\end{aligned}$$

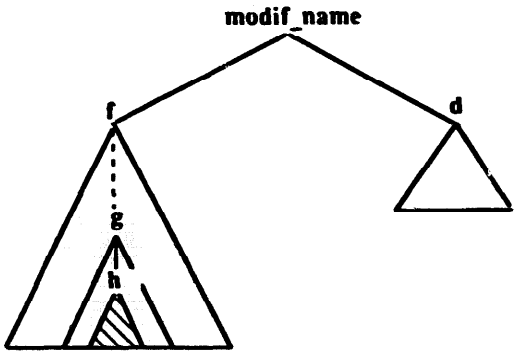

Fig. 16

which will be abbreviated into Univ_seq(A_1, A_2). The semantics related to F is:

$$S_F = \{Sem[\text{Univ\_seq}(A_1, A_2)]\} = \{Sem[A_1] \oplus Sem[A_2]\},$$

$$Sem[A_1] = Sem[\text{Univ\_assign}(\text{src}(\text{can}_1), \text{dst}(\text{temporary\_am}(\langle\text{temporary temp}_1\rangle)))],$$

$$Sem[A_2] = Sem[\text{modif\_name}(f(g(\text{src}(\text{temporary\_am}(\langle\text{temporary temp}_1\rangle))), \text{dst}(\text{can}_2))]$$

According to the value of the predicate "instruction_context", the condition of the last axiom in Section 2.3.2 is true. Thus, we get the following formula:

$$\text{contents\_of}(\langle\text{temporary temp}_1\rangle) = \text{src}(\text{can}_1)$$

and the state:

$$S_F = \{\text{contents\_of}(\langle\text{temporary temp}_1\rangle) = \text{src}(\text{can}_1) \oplus Sem[A_2]\},$$

$$Sem[A_2] = Sem[\text{modif\_name}(f(g(\text{src}(\text{temporary\_am}(\langle\text{temporary temp}_1\rangle))), \text{dst}(\text{can}_2))].$$

As the predicate "source_operand_context" of the access mode temporary_am is true, we get:

$$\text{src}(\text{temporary\_am}(\langle\text{temporary temp}_1\rangle)) = \text{contents\_of}(\langle\text{temporary temp}_1\rangle)$$

and $Sem[A_2]$ becomes:

$$Sem[A_2] = Sem[\text{modif\_name}(f(g(\text{contents\_of}(\langle\text{temporary temp}_1\rangle)), \text{dst}(\text{can}_2))]$$

and the state:

$$S_F = \{\text{contents\_of}(\langle\text{temporary temp}_1\rangle) = \text{src}(\text{can}_1) \oplus Sem[\text{modif\_name}(f(g(\text{contents\_of}(\langle\text{temporary temp}_1\rangle)), \text{dst}(\text{can}_2))]\}.$$

Taking into account the first formula of the state S_F, the second formula becomes:

$$Sem[\text{modif\_name}(f(g(\text{src}(\text{can}_1))), \text{dst}(\text{can}_2))]$$

and the state

$$S_F = \{\text{contents\_of}(\langle\text{temporary temp}_1\rangle) = \text{src}(\text{can}_1) \oplus Sem[\text{modif\_name}(f(g(\text{src}(\text{can}_1))), \text{dst}(\text{can}_2))]\}.$$

Let us compare the states S_I and S_F, we can notice that S_F is an enrichment of the state S_I by the formula:

$$\text{contents\_of}(\langle\text{temporary temp}_1\rangle) = \text{src}(\text{can}_1).$$

The adjunction of this formula cannot lead to removal of an inconsistent formula from the state S_I because we are sure that there is no formula containing temp_1 in S_I. The reason is that we use a new name each time we need to store an intermediate result in a temporary location.

3.2.2. *Correctness proof of Rule R9*

Rule R9 is often applied on terms denoting arithmetic or logical operations. The instruction template to be matched is of the following form:

$$\text{modif\_name}(\text{op}(\text{src}(X), \text{src}(Y)), \text{dst}(X))$$

and the IR term is of the form:

$$\text{modif\_name}(\text{op}(\text{src}(\text{can}_1), \text{src}(\text{can}_2)), \text{dst}(\text{can}_3)).$$

The semantics related to the IR term is the following state:

$$S_I = \{Sem[\text{modif\_name}(\text{op}(\text{src}(\text{can}_1), \text{src}(\text{can}_2)), \text{dst}(\text{can}_3))]\}.$$

Since the conditional part of the axiom above is true,

$$S_I = \{\text{CONTENTS}[\text{dst}(\text{can}_3)] = \text{SEM\_VALUE}[\text{modif\_name}, \text{op}(\text{src}(\text{can}_1), \text{src}(\text{can}_2))]\}.$$

The term I is rewritten into the term F:

$$\begin{aligned}&\text{Univ\_seq}(\text{Univ\_assign}(\text{src}(\text{can}_1)\\ &\qquad,\text{dst}(\text{temporary\_am}(\langle\text{temporary temp}_1\rangle)))\\ &\quad,\text{modif\_name}(\text{op}(\text{src}(\text{temporary\_am}(\langle\text{temporary temp}_1\rangle))\\ &\qquad,\text{src}(\text{can}_2))\\ &\qquad,\text{dst}(\text{temporary\_am}(\langle\text{temporary temp}_1\rangle)))\\ &\quad,\text{Univ\_assign}(\text{src}(\text{temporary\_am}(\langle\text{temporary temp}_1\rangle))\\ &\qquad,\text{dst}(\text{can}_3)))\end{aligned}$$

which will be abbreviated into $\text{Univ\_seq}(A_1, A_2, A_3)$. The semantics related to F is described by the state:

$$S_F = \{Sem[\text{Univ\_seq}(A_1, A_2, A_3)]\} = \{Sem[A_1] \oplus Sem[A_2] \oplus Sem[A_3]\}$$

$$\begin{aligned}Sem[A_1] = Sem[&\text{Univ\_assign}(\text{src}(\text{can}_1)\\ &,\text{dst}(\text{temporary\_am}\\ &\qquad(\langle\text{temporary temp}_1\rangle)))].\end{aligned}$$

Let us apply the axiomatic definition of the universal operator "Univ_assign" to $Sem[A_1]$, we get the following formula:

$$\text{contents\_of}(\langle\text{temporary temp}_1\rangle) = \text{src}(\text{can}_1).$$

$$S_F = \{\text{contents\_of}(\langle\text{temporary temp}_1\rangle) = \text{src}(\text{can}_1) \oplus Sem[A_2] \oplus Sem[A_3]\},$$

$$Sem[A_2] = Sem[\text{modif\_name}\ (\text{op}(\text{src}(\text{temporary\_am}(\langle\text{temporary temp}_1\rangle)), \text{src}(\text{can}_2)), \text{dst}(\text{temporary\_am}(\langle\text{temporary temp}_1\rangle)))].$$

According to the canonical definition of temporary_am in source position, we get:

$$\text{src}(\text{temporary\_am}(\langle\text{temporary temp}_1\rangle)) = \text{contents\_of}(\langle\text{temporary temp}_1\rangle).$$

Thus,

$$Sem[A_2] = Sem[\text{modif\_name}\ (\text{op}(\text{contents\_of}(\langle\text{temporary temp}_1\rangle), \text{src}(\text{can}_2)), \text{dst}(\text{temporary\_am}(\langle\text{temporary temp}_1\rangle)))].$$

Let us sum up:

$$S_F = \{\text{contents\_of}(\langle\text{temporary temp}_1\rangle) = \text{src}(\text{can}_1) \oplus Sem[\text{modif\_name}(\text{op}(\text{contents\_of}(\langle\text{temporary temp}_1\rangle), \text{src}(\text{can}_2)), \text{dst}(\text{temporary\_am}(\langle\text{temporary temp}_1\rangle)))] \oplus Sem[A_3]\}.$$

Taking into account the first formula of S_F for the second formula, we get:

$$S_F = \{\text{contents\_of}(\langle\text{temporary temp}_1\rangle) = \text{src}(\text{can}_1) \oplus Sem[\text{modif\_name}(\text{op}(\text{src}(\text{can}_1), \text{src}(\text{can}_2)), \text{dst}(\text{temporary\_am}(\langle\text{temporary temp}_1\rangle)))] \oplus Sem[A_3]\}.$$

The second formula of the state S_F computes the semantics of the modification modif_name. According to its related axiom S_F becomes:

$$S_F = \{\text{contents\_of}(\langle\text{temporary temp}_1\rangle) = \text{src}(\text{can}_1) \oplus \text{CONTENTS}[\text{dst}(\text{temporary\_am}(\langle\text{temporary temp}_1\rangle))] = \text{SEM\_VALUE}[\text{modif\_name}, \text{op}(\text{src}(\text{can}_1), \text{src}(\text{can}_2))] \oplus Sem[A_3]\}.$$

Using the definition of the function CONTENTS applied to

$$\text{dst}(\text{temporary\_am}(\langle\text{temporary temp}_1\rangle)),$$

we get:

$$S_F = \{\text{contents\_of}(\langle\text{temporary temp}_1\rangle) = \text{src}(\text{can}_1) \oplus \text{contents\_of}(\langle\text{temporary temp}_1\rangle) = \text{SEM\_VALUE}[\text{modif\_name}, \text{op}(\text{src}(\text{can}_1), \text{src}(\text{can}_2))] \oplus Sem[A_3]\}.$$

The first formula becomes inconsistent since contents_of(⟨temporary $temp_1$⟩) is redefined. So the state S_F becomes:

$$S_F = \{\text{contents\_of}(\langle\text{temporary } temp_1\rangle) = \text{SEM\_VALUE}[\text{modif\_name}, \text{op}(\text{src}(can_1), \text{src}(can_2))] \oplus Sem[A_3]\},$$

$$Sem[A_3] = Sem[\text{Univ\_assign}(\text{src}(\text{temporary\_am}(\langle\text{temporary } temp_1\rangle)), \text{dst}(can_3))].$$

As the following axiom for Univ_assign exists:

$$\text{instruction\_context}(\text{Univ\_assign}(\text{src}(\text{temporary\_am}(\langle\text{temporary temp}\rangle)), \text{dst}(\text{can}))) \text{ is true}$$
$$\Rightarrow \text{CONTENTS}[\text{dst}(\text{can})] = \text{src}(\text{temporary\_am}(\langle\text{temporary temp}\rangle)).$$

Thus,

$$Sem[A_3] = \{\text{CONTENTS}[\text{dst}(can_3)] = \text{src}(\text{temporary\_am}(\langle\text{temporary temp}\rangle))\}.$$

Let us apply the canonical definition of src(temporary_am ...)

$$Sem[A_3] = \{\text{CONTENTS}[\text{dst}(can_3)] = \text{contents\_of}(\langle\text{temporary temp}\rangle)\}.$$

Taking into account the first formula of S_F for the second formula, we get:

$$S_F = \{\text{contents\_of}(\langle\text{temporary } temp_1\rangle) = \text{SEM\_VALUE}[\text{modif\_name}, \text{op}(\text{src}(can_1), \text{src}(can_2))] \oplus \text{CONTENTS}[\text{dst}(can_3)] = \text{SEM\_VALUE}[\text{modif\_name}, \text{op}(\text{src}(can_1), \text{src}(can_2))]\}.$$

We can notice that the state S_I is an enrichment of the state S_F by the formula:

$$\text{contents\_of}(\langle\text{temporary } temp_1\rangle) = \text{SEM\_VALUE}[\text{modif\_name}, \text{op}(\text{src}(can_1), \text{src}(can_2))].$$

3.2.3. Comments

Such correctness proof frameworks can be developed for each rewriting rule. They can be developed once and for all because they are independent of the target machine.

As the two first steps of the compiling process can be proved by techniques like [19], the correctness proof of the whole compiling process until the code selection process may be achieved.

4. Conclusion

We have provided ways to specify target machines in the framework of a compiler rewriting system using abstract data type specifications. The kernel of the code generator is a set of rewriting rules that preserve the semantics of the IR. Such a proof can be done by means of axioms on universal operators of the target abstract data type and, thus, it is machine-independent. The termination of the code selection algorithm is straightforward because the replacement of a temporary access mode is only done when it allows to match an access mode that strictly contains the rewriting of the temporary access mode. Thus, the depth of the IR term decreases until there is a blocking situation or until the level of the operand of the instruction is reached.

Generally speaking, the code selection algorithm is not confluent. The more global optimal code may be obtained by computing the cumulated costs for each solution.

Prolog has allowed us to develop quickly the experimental system PAGODE in order to achieve the production of code generators from a target machine specification based on term templates. The code produced is locally optimal.

A work currently in progress aims at restricting the large number of temporaries before the binding step takes place. For that purpose, we use a concept of alive temporaries in the sequence. More generally, it is necessary now to integrate optimizing tasks in this framework.

References

[1] A.V. Aho and M. Ganapathi, Efficient tree pattern matching: An aid to code generation, in: *Conference Record Twelfth Annual ACM Symposium on Principles of Programming Languages* (1985) 334-340.

[2] M. Bidoit, M.C. Gaudel and A. Mauboussin, How to make algebraic specifications more understandable: An experiment with the PLUSS specification language, *Sci. Comput. Programming* **12** (1989) 1-38.

[3] R.G.G. Cattell, A survey and critique of some models of code generation, Tech. Rept. 78-115, Carnegie-Mellon University, Computer Sciences Department, Pittsburgh, PA (1977).

[4] R.G.G. Cattell, Automatic derivation of code generators from machine description, *ACM Trans. Programming Languages Syst.* **2** (1980) 173-199.

[5] P. Deschamp, PERLUETTE: A compiler producing system using abstract data types, in: *Proceedings International Symposium on Programming*, Turin (1982).

[6] A. Despland, M. Mazaud and R. Rakotozafy, Code generator generation based on template-driven target term rewriting, in: *Proceedings of Rewriting Techniques and Applications*, Bordeaux, France, Lecture Notes in Computer Science **256** (Springer, Berlin, 1987) 105-120.

[7] M. Ganapathi and C.N. Fischer, Description-driven code generation using attributed grammars, in: *Proceedings Ninth Annual ACM Symposium on Principles of Programming Languages* (1982).

[8] H. Ganzinger and R. Giegerich, A truly generative semantics-directed compiler generator, in: *Proceedings SIGPLAN 82 Symposium on Compiler Construction*, ACM SIGPLAN Notices **17** (6) (1982).

[9] M.-C. Gaudel, P. Deschamp and M. Mazaud, Compiler construction from high level specification, in: *Automatic Program Construction Techniques* (Macmillan, New York, 1984).

[10] R. Giegerich, Logic specification of code generation techniques, in: *Programs as Data Objects*, Lecture Notes in Computer Science **217** (Springer, Berlin, 1985) 96-111.

[11] R. Giegerich and K. Schmal, Code selection techniques: Pattern matching, tree parsing, and inversion of derivors, in: *ESOP' 88*, Lecture Notes in Computer Science **300** (Springer, Berlin 1988) 247-268.

[12] S.L. Graham and R.S. Glanville, A new method for compiler code generation, in: *Conference Record Fifth Annual ACM Symposium on Principles of Programming Languages*, Albuquerque, NM (1978) 108-119.

[13] S.L. Graham, R.R. Henry et al., Experiment with a Graham-Glanville style code generator, in: *Proceedings SIGPLAN 84 Symposium on Compiler Construction, ACM SIGPLAN Notices* **19** (6) (1984).

[14] G. Huet and D.C. Oppen, Equations and rewriting rules: A survey, in: R. Book, ed., *Formal Languages: Perspective and Open Problems* (Academic Press, New York, 1980).

[15] M. Jourdan and D. Parigot, The FNC-2 system user's guide and reference mannual, INRIA, Le Chesnay, France (1988).

[16] S. Kaplan, Positive/negative conditional rewriting, in: *Proceedings First International Workshop on Conditional Term Rewriting Systems*, Lecture Notes in Computer Science **308** (Springer, Berlin, 1987).

[17] B.W. Leverett, Register allocation in optimizing compilers, CMU-CS-81-103, Carnegie-Mellon University, Pittsburgh, PA (1981).

[18] B.W. Leverett, R.G.G. Cattell, S.O. Hobbs, J.M. Newcomer, A.H. Reiner, B.R. Schatz and W.A. Wulf, An overview of the production quality compiler-compiler project, CMU-CS-79-105, Carnegie-Mellon University, Pittsburgh, PA (1979).

[19] E. Madelaine, Système d'aide à la preuve de compilateurs, Thèse de 3ème cycle. Université de Paris VII (1983).

[20] M. Mazaud, R. Rakotozafy and A. Szumachowski-Despland, Code generator generation based on template-driven target term rewriting, Rapport de recherche INRIA RR-582, Le Chesnay, France (1986).

[21] S.W.K. Tjiang, Twig language manual, Computing Science Tech. Rept. 120, AT&T Bell Laboratories, Murray Hill, NJ (1986).