PAGODE : A back end generator using attribute abstract syntaxes and term rewritings

Annie Despland* Monique Mazaud**
Raymond Rakotozafy*
* L.I.F.O. Université d'Orléans
BP 6759, 45067 Orléans CEDEX2, France and I.N.R.I.A.
** I.N.R.I.A. Domaine de Voluceau
BP 105, 78153 Le Chesnay CEDEX, France

Abstract

A major problem in deriving a compiler from a formal definition is the production of correct and efficient object code. We propose a solution to this problem in the framework of a compiler writing system where the compilation process may be viewed as successive translations from an attributed abstract syntax to another abstract syntax. The code-generator generator needs two kinds of specifications :

- an attributed abstract syntax (**AAS**) of the target machine : it is the description of the **I.R.** given as input to the code-generator.
- a target machine description where the basic concepts are hierarchically described by tree-patterns. These tree patterns are terms of the target abstract syntax.

The code generation process is divided into two steps : the instructions selection process and the register allocation one. The instruction selection process applies a set of rewriting rules driven by tree templates derived from the target machine specification to the **I.R.** term. The register allocation process consists of several evaluation passes of an attributed grammar derived automatically from the target machine specification. The first one sets the constraints on temporaries according to the whole context, the second one does life-time analysis and packing on temporaries, the last one assigns effective resources to temporaries.

1 Introduction

A compiler, in order to produce code, needs full knowledge not only of 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. Considerable research effort has been invested into making compiler construction as modular and as automatic as possible.

Tools based on formalisms such as abstract data types, attributed grammars are widely known and used to produce front ends. In order to have a uniform approach of

the whole compilation process, it would be useful to use the same formalism for the front end and the back end. An approach using high level semantics has been developped in MESS [LP 87] but the emphasis had been put essentially on the front-end.

This paper is devoted to the back end of the compiler writing system.

Usually, code generation is converted into a syntactic process using tree structured patterns describing instructions of the target machine and a tree structured **I.R.** The instruction selection is done by covering the input **I.R** by instruction patterns. Various works differ by the way in which the instruction set of the target machine is described and by the pattern matching and transformations used to reduce the **I.R** tree.

LR(1)-like parsing is used in [GG 78], [GH 84]. Attributed grammars are used in [GF 82] to add semantic constraints on symbols. In [Cat 80] , [FW 88] subgoals are selected by use of heuristics to try patterns. In [ESL 89] a reduction algorithm that computes the best cover tree according to cost function is used.

All these code-generator generators not are generally embedded in a full compiler writing system using a uniform formal framework. The instruction set of the target machine, except in the works of Giegerich [Gie 90] is not described by a formal semantics. It is often obscured by informations related to the reduction algorithm and cannot be got straightforward from the handbook of the target machine. As a consequence of this lack of formal semantics, there is no means to prove the correctness of the generated code.

Our compiler writing system produces a compiler from a specification including three parts : a source language, a target language definition and the description of the implementation choices.

The fundamental background of the specifications is that the whole compilation process is viewed as successive mappings from an abstract data type (**ADT**) into an other abstract data type [GDM 84]. The axioms of the various **ADT** allow to prove the correctness of each step of the compilation process [DMR 89].

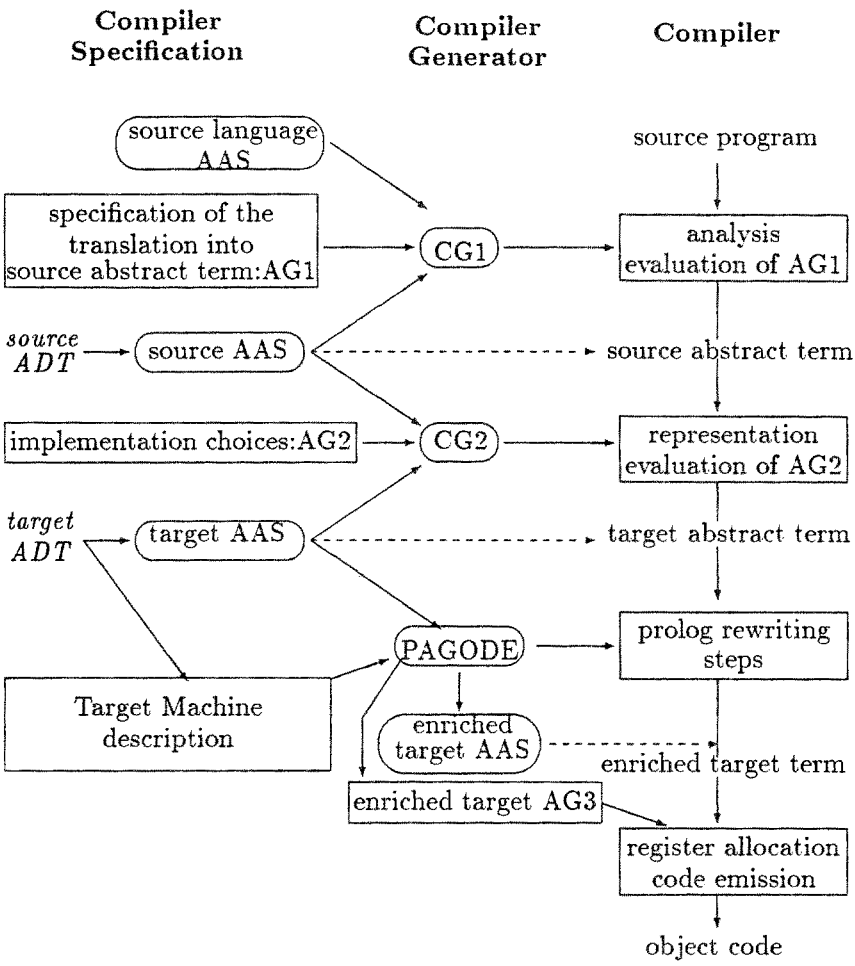
Since an abstract data type without axioms is hardly more than an abstract syntax, the tools used here to handle these specifications are attributed abstract syntaxes (**AAS** for abbreviation) and attribute grammars (**AG**) specifying the mappings between **AAS**es. An **AAS** is mainly an abstract syntax with declarations of attributes attached to phyla. Productions of the **AG** are operator definitions of the **AAS**. In the whole paper, we call semantic rules the attributes definitions related to the productions of the **AG**. The system FNC-2 has especially been designed for dealing with these tools [JP 90].

We focus on the back end of the compiler writing system. It uses the same formal background as the preceding steps.

The first section presents the code-generator specification for the instruction selection process. It needs two kinds of specifications :

- an **AAS** of the target machine : it is the description of the **I.R** given as input to the code-generator,

- a target machine description hierarchically structured in three levels [DMR 87]. It consists of a description of the target instruction set and a mapping of the instruction set into the target **AAS** (i.e the semantics of the instructions is given in terms of the target **AAS**).



The second section deals with the code generation mechanism which is divided into two steps : the instruction selection process and the register allocation process.

The compiler writer describes the semantics of each instruction of the target machine by a **I.R.** term. Thus the instruction selection process can be defined formally as a reverse translation as in [GS 88] [Gie 90], operationally the reverse translation is defined by a set of tree templates derived from the target machine specification and a set of rewriting rules that are machine independent. As the description language has a formal semantics, the correctness of the rewriting rules has been achieved [DMR 89]. They preserve the semantics of the **I.R** term.

This step produces a term of the canonical target **AAS**. It is the target **AAS** enriched by some universal operations on temporary ressources and restricted to the canonical form of instructions and addressing modes. This term is given as input to the register allocation step which is an evaluation pass of the **AG** based on the canonical target **AAS**, whose semantic rules are automatically generated by the system.

2 The code-generator specification

2.1 Abstract syntax of the target language

To define the abstract syntax of the target language, the compiler writer must describe phyla and operators, which are respectively similar to non-terminals and productions of a usual context-free grammar. The abstract syntax of the target machine mirrors virtual target machine with elementary instructions (called object modification following [GDM 84]) acting on cells via elementary operations denoting the access path to these cells. In such an approach, an instruction of the instruction set processor is represented by a modification or a sequence of modifications. Phyla represent basic concepts of the target machine such as modifications, addressing modes, operands and cells. Defining a phylum involves specifying which operators it includes. It follows according to the machine specification (see §3.1.1).

Object_modif	=	seq assign compare branch ...
Access_mode	=	dreg_am disp_am postinc_am ...
Operand	=	const_value cont_of_address index add ... src
Cell	=	designates_address designates_aregister ... dst

Phyla representing operands and cells are used to designate storages of the target machine such as registers and memory locations. Operators designate available operations of the target machine such as :

- the usual modifications :

assign	→	Operand Cell
compare	→	Operand Operand

- the usual arithmetic operations :

add	→	Operand Operand

- the dereference and cell constructor operations :

cont_of_address	→	address
cont_of_dreg	→	dregister
designates_aregister	→	aregister

- the operations describing access path to memory cells :

index	→	address immediate_value

- the built-in operations src and dst to introduce an operand in source or destination position :

src	→	Access_mode
dst	→	Access_mode

and addressing modes :

Access_mode	=	areg_am dreg_am predec_am ...
dreg_am	→	dregister
disp_am	→	aregister immediate_value
postinc_am	→	aregister

Notions such as the size of an operand, the type or the name of a storage, the value of an address are specified by attributes attached to phyla of the target **AAS**.

2.2 The target machine specification

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 **AAS** and takes into account its size. As the occurrences of a construct are related to the size of the addressable units, their semantic descriptions are nearly identical [DMR 87]. A solution proposed to deal with large algebraic specifications 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 asssociated to it. The instantiation mechanism is bound to a name generation mechanism. Throughout the paper the following notations will be used :

- If **n** is a name and **L** is a variable, when **L** is instantiated by **v** , **n!L** builds the name **n_v**.
- <S V> 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.

2.2.1 Storage classes

A component of the physical storage does not 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 longword operand. Thus, 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 storage 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 of a storage class,*
- *basic operations such as the dereference operation, i.e. the access operation to the contents of an element of the storage class and the cell constructor operation.*

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. As 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. The use of the parameterization and instantiation mechanisms allows 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 actual names.

```
Storage_class
    Denotation < dregister!size Dk>
    Attributes
        $Base = DREG
    Operations
        dereference is
            cont_of_dreg!size          : dregister → immediate_value
        cell_constructor is
            designates_dreg ! size     : dregister → dregister
    Symbolic_notation
        Dk 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 descriptions of actual 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.

2.2.2 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 operand 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.*
- *a template that describes the access path to the corresponding operand : the operand in source (resp. destination) position is defined by the term obtained by applying the dereference (resp. the cell constructor) operation to this template.*

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 indirect with displacement* addressing mode. This access mode has instances which depend on the size of the location indirectly accessed in source position. Thus the compiler writer defines a generic access mode pattern "disp_am!size" parameterized by the size.

```
Access_mode
    Canonical_form                  - - Indirect with displacement access modes
        disp_am!size ( <aregister_L reg> , <value_W val>)
    Attributes
        $length =   size            - - length of the addressable unit
        $fmt    =   ~val(reg)~      - - Assembly language format
    Template
        index ( cont_of_areg_L (<aregister reg>), const_value_W (<immediate_value val>))
    Instances size in {B, W, L}
End
```

As the template of the previous definition gives an address whose dereference operator is cont_of_address!size and the cell constructor is designates_address!size, the system derives the two following generic access modes, respectively in destination and source position :

```
designates_address!size (
    index (cont_of_areg_L (<aregister reg>), const_value_W (<immediate_value val>)))
cont_of_address!size (
    index (cont_of_areg_L (<aregister reg>), const_value_W (<immediate_value val>)))
```

This leads to three templates in source position (respectively in destination position) when the size is instantiated by {B, W, L}. This leads to three templates in source position (respectively in destination position) when the size is instantiated by {B, W, L}. The *indirect with index* access mode has instances which depend on the size of the location indirectly accessed in source position.

```
Access_mode
    Canonical_form              - - Indirect with displacement access modes
        dindex_am!index_size!size ( <aregister_L reg1 >, < dregister!index_size reg2 >
                                  , <value_B val >)
        Attributes
            $length =           size
            $fmt    =           ~val (reg1, reg2.index_size)~
                                when val = 0 ~0 (reg1, reg2.index_size)~
        Template
            index (cont_of_areg_L (< aregister reg1 >)
                , add_L (sign_extend_L (cont_of_dreg!index_size (<dregister reg2>))
                    , sign_extend_L (const_value_B ( <immediate_value val>))))
            when val = 0
                index (cont_of_areg_L (<aregister reg1>)
                      , cont_of_dreg!index_size ( <dregister reg2>))
        Instances size in {B, W, L}, index_size in { W, L}
End
```

Notice that an optimizing case is described in the specification of this access mode by the clause when, if the value of val is 0.

2.2.3 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
```

2.2.4 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 the 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 longword. We obtain the following specification :

```
Instruction
    Canonical_form
        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 *addi* instruction specifies the addition of an immediate operand to an appropriate operand of the alterable data access class :

```
Instruction
    Canonical_form
        addi!size ( <Immediate_access!size AM1>, <Altdata_access!size AM2>)
    Attributes
        $length =   size
        $fmt    =   ~ADDI.$length   $fmt (<Immediate_access!size AM1> )
                                   , $fmt (<Altdata_access!size AM2>)~
    Template
        assign!size (
            add!size (src ( <Altdata_access!size AM2>), src ( <Immediate_access!size AM1>))
          , dst (<Altdata_access!size AM2>))
    Instances size in {B, W, L}
End
```

The *asl* instruction specifies the shift of the content of a data register by a quick value :

```
Instruction
    Canonical_form
        asl!size ( <Quick_access AM1>, <Dregister_access!size AM2>)
    Attributes
        $length =     size
        $fmt    =     ~ASL.$length     $fmt(<Quick_access AM1> )
                                       , $fmt (<Dregister_access!size AM2>)~
    Template
        assign!size (
            shift_al!size  (src ( <Quick_access AM1>), src ( <Dregister_access!size AM2>))
            , dst (<Dregister_access!size AM2>))
    Instances size in {B, W, L}
End
```

2.3 The canonical target machine AAS

The bottom-up matching process of the **I.R** is carried out until each modification of the **I.R** is identified to an instruction template. For each modification, in the context of an instruction template, operand subterms are matched with access mode templates. If they are leaves of

the instruction template, they are replaced by their canonical form in the modification, else the location designated by the access mode is stored in a temporary and the modification is rewritten using this temporary. The process goes on in order to make the modification closer to an instruction template. Finally the modification is flattened in a sequence of universal store trees and an instance of the instruction template [DMR 89]. The universal store trees must be described. Thus it is necessary to enrich the target machine **AAS** with corresponding phyla and operators. The following specification is automatically added by the system.

```
Exec                  →  Object_modif                  - - root of the term
Object_modif          =  Univ_seq Univ_assign ...
Univ_seq              →  Object_modif+
Univ_assign           →  Operand Operand
Operand               =  temporary_am ...
temporary_am          →  Temporary
Temporary             =  Denotation_Temporary
Denotation_Temporary  →
```

2.4 The interface specification

The system needs to link the internal names such as temporary_am, Univ_assign, and the actual names of the machine specification that become possible synonyms during the rewriting process. For that purpose, the system uses an interface declaration module specified by the compiler writer. For the **MC68000** it follows :

```
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}
```

2.5 Target templates and attribute grammar derivation

Two modules are used to process the target machine specification and give two outputs, respectively a set of tree patterns and an attributed abstract syntax of the target machine. The first module builds three families of trees corresponding to access mode templates in source position and destination position and instruction templates. These families are written into Prolog clauses [DMR 88]. The properties of these trees derived from the specification are translated in Prolog clauses. These tree templates are used by the rewriting step to achieve the instruction selection. The second module builds an **AG** for FNC-2. The evaluation of the **AG** achieves the register allocation step.

3 The code-generator generator

3.1 Instruction selection

For each **I.R** 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. Two templates of an instruction class can be represented by a canonical representative. The instruction selection algorithm applies a set of rules as specified in [DMR 89]. 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 the search for access modes. The variables occur ones in a canonical representative and represent the operands, they are annotated by a property source or destination.

3.1.1 Rewriting rules

Notations 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 IC be the ordered set of instruction class patterns.

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. Some of the rules are more formally described in . 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 the search for access modes. The variables occur ones in a canonical representative and represent the operands, they are annotated by a property source or destination.

3.1.2 Rewriting rules

Notations 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 IC be the ordered set of instruction class patterns.

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. Some of the rules are more formally described in [DMR 89]. The first one describes the replacement of a subtree which is an instance of an access mode by the instantiated canonical form of this access mode : informally when matching a tree **t** with an instruction class pattern, the source and destination position contexts are set according to the position property in the instruction class pattern. If there is a subtree $\mathbf{t_i}$ of **t** which is an instance of an access mode pattern in the right position, then $\mathbf{t_i}$ 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. We recall here completely the second rule. It gives a good idea of the use of temporaries. It describes the tranformation to be done when a subtree supposed to be an operand in an instruction context is not an instance of an access mode but has inner subtrees that are instances of access modes.

Rule R2

When matching a tree **t** *with an instruction class pattern, if we find a subtree* $\mathbf{t_i}$ *of* **t** *which is not an instance of an access mode pattern, then starting from the leaves of* $\mathbf{t_i}$*, 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, supppose there exist $T \in IC$ and a substitution σ_{IC} such that $\sigma_{IC} = \{ < \Delta_i, t_i > \} \mid \Delta_i \in Var(T), 1 \leq i \leq n$ with $\sigma_{IC}(T) = t$

and suppose there exists t_i such that $< \Delta_i, t_i > \in \sigma_{IC}$, and if the context of Δ_i in T is the position pos and such that for all $A \in AM_{pos}$ there exists no substitution μ such that $\mu(A) = t_i$. Then if there exist :

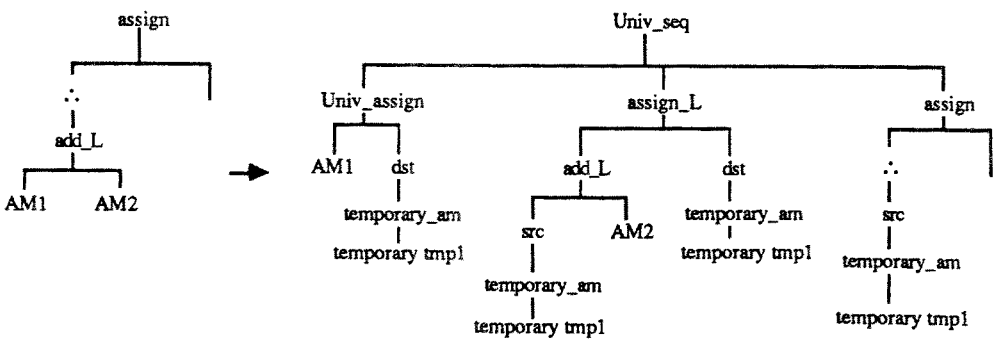
1. a largest subtree t_{ij} starting from the leaves of t_i,
2. and $B \in AM_{source}$
3. and a substitution γ such that $\gamma(B) = t_{ij}$.

we define the rewriting of the tree **t** by :

$$t \rightarrow \text{Univ\_seq}(\text{Univ\_assign}(\gamma(\text{can}(B)), \delta(\text{dst}(\text{temporary\_am}(< \text{temporary } \theta >)))) , t /[t_{ij} = \delta(\text{src}(\text{temporary\_am}(< \text{temporary } \theta >)))])$$

where δ is the substitution : $\delta = \{ < \theta, \text{tmp}\alpha > \}$ where tmpα is a new temporary location. The rules described in [DMR 89] only deals with terms whose subtrees considered as operands in an instruction context are nested access mode instances.

When the compiler writer describes the translation of an expression from the source **AAS** to the target **AAS**, the term produced contains embedded arithmetic and access path operators. Thus it is necessary to specify rules to deal with nested arithmetic operators. Informally, when in an instruction context, a subtree t_i cannot be reduced to an access mode instance using the rules of [DMR 89], then starting from the leaves of t_i, we look for the biggest subtree t_{ij} of t_i which match the left son of an arithmetic instruction A. First the subtree of t_{ij} corresponding to the source operand modified by the instruction is saved in a temporary, second the subtree of t_{ij} is replaced by the reference to the temporary in source position, third an instance of A is generated using t_{ij} and fourth, t_{ij} is replaced by the reference to the temporary in source position.



3.1.3 Example

Let us consider a short piece of program written in the Pascal-like language :

```
integer J , K ;
    array X(10) ;
    begin
        ...
        K := X(J) ; - - (1)
        ...
    end.
```

If the base of the main program is located in the address register a6 and local variables have negative offsets, the corresponding target term of statement (1) is the following :

```
assign_L (
    cont_of_address_L (
        index ( cont_of_areg_L (<aregister, a6>))
            , add_L (
                sub_L (
                    shift_al_L (const_quick_value (<quick_value, 2>))
                                , cont_of_address_L (
                                    index ( cont_of_areg_L (<aregister, a6>))
                                        , const_value_W( <immediate_value, -4>))))
                    , const_value_L( <immediate_value, 4>)))
                , const_value_L( <immediate_value, -48>)))))
    , designates_address_L (
        index ( cont_of_areg_L (<aregister, a6>))
            , const_value_W( <immediate_value, -8>)))))
```

The modification is matched with instructions templates. The second offspring of the shift_al_L node is an instance of a disp_am_L access mode. There are no access mode templates with shift_al_L as root. But the subtree rooted with shift_al_L is the left son of an arithmetic instruction (asl §1.1.2.d). Thus the last rule can be applied. It is repeated successively on the nodes sub_L and add_L. The tree resulting from the rewriting steps is the following :

```
Univ_seq (
    , Univ_assign ( src ( disp_am_L (<aregister, a6>, <immediate_value_W, -4>))
                , dst ( temporary_am (<temporary, tmp0>)))
    , assign_L (
        shift_al_L (src ( quick_am (<quick_value, 2>))
                , src (temporary_am (<temporary, tmp0>)))
        , dst ( temporary_am (<temporary, tmp0>)))
    , Univ_assign ( src ( temporary_am (<temporary, tmp0>))
                , dst ( temporary_am (<temporary, tmp1>)))
    , assign_L (
        sub_L (src ( temporary_am (<temporary, tmp1))
            , src (immediate_val_am (<immediate_value_L, 4>)))
        , dst ( temporary_am (<temporary, tmp1>)))
```

```
, Univ_assign ( src ( temporary_am (<temporary, tmp1>))
              , dst ( temporary_am (<temporary, tmp2>)))
, assign_L (
    add_L (src ( temporary_am (<temporary, tmp2))
        , src (immediate_val_am (<immediate_value_L, -48>)))
    , dst ( temporary_am (<temporary, tmp2>)))
, assign_L (src ( dindex_am_when_W_L (<aregister_L, a6>
                        , <dregister_W, tmp2>,
                        <immediate_value_B, 0>))
        , dst ( disp_am_L (<aregister_L, a6>,
                <immediate_value_W, -8>))))
```

The output of the rewriting step is a term of the canonical **AAS** where each subtree is an instance of the canonical form of an instruction. This term is given as input to the register allocation step. According to the handbook the compiler writer describes various instructions with the same name but with the different access classes operands. As the system needs to identify an instruction by single name, it renames all instructions. Thus the preceding sub instruction gets the internal name sub2. The final form of the rewriting tree is the following :

```
Univ_seq (
    Univ_assign (disp_am_L (<aregister_L, a6>, <immediate_value_W, -4>))
            , temporary_am (<temporary, tmp0>)))
    , asl_L (quick_am_L (<quick_value, 2>))
        , temporary_am (<temporary, tmp0>)))
    , Univ_assign (temporary_am (<temporary, tmp0>))
            , temporary_am (<temporary, tmp1>)))
    , sub2_L (immediate_val_am_L (<immediate_value_L, 4>))
            , temporary_am (<temporary, tmp1>))
    , Univ_assign (temporary_am (<temporary, tmp1>)
            , temporary_am (<temporary, tmp2>))
    , addi_L (immediate_val_am (<immediate_value_L, -48>)
            , temporary_am (<temporary, tmp2>))
    , move_L (dindex_am_when_W_L (<aregister_L, a6>, <dregister_W, tmp2>
            , disp_am_L(<aregister_L, a6>, <immediate_value_W, -8>)))
```

3.2 Register allocation

At the end of the instruction selection step, each subtree of the sequence is an instance of an instruction template. But the leaves of each instruction instance are either actual resources or temporary resources. Thus it is necessary to bind each temporary with an actual resource, i.e. a right storage class, an available element of this storage class. As the instruction selection process creates a very large number of temporaries, it is necessary to pack temporaries that are not live simultaneously into the same name. This needs contextual information. This information is evaluated and processed by means of the **AAS** generated from the target machine description.

3.2.1 Binding

During a first pass, the compound attribute ($h-cs and $s-cs) gathers the constraints on temporaries at the root. This value is assigned to the global attribute $symtab which is inherited into

each offspring of the sequence in order to replace each temporary access mode by the convenient access mode. These parts of semantic rules allow to build $s-cs at the root of the tree.

```
where Exec          →       Object_modif use
    $h-cs (Object_modif)    := empty-table ;
    $symtab (Object_modif):= $s-cs (Object_modif)
end where
where Univ_seq      →       Object_modif+ use
    $s-cs (Univ_seq)    :=  case arity is
                                0     :   $h-temp-cs (Univ_seq ) ;
                                other :   $s-temp-cs (Object_modif.last) ;
                            end case
    $h-cs               :=  case position is
                                first  $h-temp-cs (Univ_seq ) ;
                                other $s-temp-cs (Object_modif.left) ;
                            end case
end where ;
```

Each offspring of an instruction of the **AAS** is decorated by the $class attribute ; whose value is the intersection of the access class of the operand with the set of access modes of the interface (see §1.1.4).

```
where Univ_assign →     Operand Operand use
    $class (Operand.1) := Quick_access_interface ;
    $class (Operand.2) := Altdata_access_interface ;
end where ;
where asl         →     Operand Operand use
    $class (Operand.1) := Quick_access_interface ;
    $class (Operand.2) := Altdata_access_interface ;
end where ;
where sub2        →     Operand Operand use
    $class (Operand.1) := All_access_interface ;
    $class (Operand.2) := Dregister_access_interface ;
end where ;
where addi        →     Operand Operand use
    $class (Operand.1) := Immediate_access_interface ;
    $class (Operand.2) := Altdata_access_interface ;
end where ;
```

The intersection of the access classes Altmem, Dregister, Altdata are automatically computed and are respectively defined by :

```
All_access-interface        = [dreg_am_B, dreg_am_W,dreg_am_L,
                               areg_am_W, areg_am_L, relative_dlab_am_W];
Dregister_access-interface  = [dreg_am_B, dreg_am_W, dreg_am_L] ;
Altdata_access-interface    = [dreg_am_B, dreg_am_W, dreg_am_L] ;
```

The insertion of a constraint on a temporary is done in the null-ary rule describing a temporary. The lookup function which searches for the identifier related to the denotation of a temporary and returns the constant [ ] if the search fails.

```
where Denotation_temporary  →  use
    $s-cs := let info := lookup ($id (Denotation_temporary), $h-cs (Denotation_temporary))
    in if info = [ ] then
        if $univ_assign_son (Denotation_temporary) then
            - - if the temporary is the son of a universal assignment,
            - - the information binding to it is All_access-interface
            insert ($id (Denotation_temporary), All_access-interface
                  , $h-cs (Denotation_temporary))
            - elsif $univ_assign_son (Denotation_temporary )
            then $h-cs
            else replaceinfo ($class , $id (Denotation_temporary)
                        , $h-cs (Denotation_temporary))
        - - replace the information related to the temporary by the intersection of the
        - - constraints related to the left hand side instructions with that of current one
end where ;
```

For instance, in the asl tree, the constraint for tmp0 is Dregister_access-interface which binds the temporary tmp0 with a long data register as the size of the instruction is long. In the same way, the constraint for tmp1 in the sub tree is Altdata_access-interface which binds also tmp1 with a long data register. tmp2 is bound with a long data register because of the constraints on the second operand of an addi instruction.

The overloading of the denotation of a temporary is done by looking for the information coupled to the name of the temporary in the global attribute $symtab.

```
where Denotation_temporary  →  use
    $term   :=  let info : info-binding :=lookup ($id (Denotation_temporary)
                                                , $symtab (Denotation_temporary))
    in  if info = Aregister_access-interface
            then Temporary-union (Denotation_aregister ( )
                with   $type := aregister, $id := $id (Denotation_temporary)
                end with )
        elsif info = Dregister_access-interface
                then Temporary-union (Denotation_dregister ( )
                    with  $type := dregister, $id := $id (Denotation_temporary)
                    end with )
        else Temporary-union (Denotation_temporary ( )
            with   $type := temporary, $id := $id (Denotation_temporary)
            end with )
        end if ;
end where ;
```

The overloading of the temporary access mode is done using the union of types.

```
where Temporary_am  →  Temporary use
    $term  :=      case $term-Temporary(Temporary) is
                     Temporary-union(X : Temporary) : temporary_am(X) ;
                     Temporary-union(X : Dregister) : dreg_am(X) ;
                     Temporary-union(X : Aregister) : areg_am(X);
                   other : null-Operand( ) ;
                   end case ;
end where ;
```

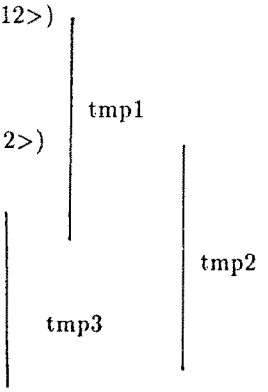
Thus after the binding step, the term becomes :

```
seq (
    move_L (disp_am_L (<aregister_L, a6>, <immediate_value_W, -4>)
        , dreg_am (<dregister_L, tmp0>))
    , asl_L (quick_am_L (<quick_value, 2>)
        , dreg_am (<dregister_L, tmp0>))
    , move_L (dreg_am (<dregister_L, tmp0>)
        , dreg_am (<dregister_L, tmp1>))
    , sub2_L (immediate_val_am_L (<immediate_value_L, 4>)
        , dreg_am (<dregister_L, tmp1>))
    , move_L (dreg_am (<dregister_L, tmp1>)
        , dreg_am (<dregister_L, tmp2>))
    , addi_L (immediate_val_am (<immediate_value_L, -48>)
        , dreg_am (<dregister_L, tmp2>))
    , move_L (dindex_am_when_L_L (<aregister_L, a6>
        , <dregister_L, tmp2>), <immediate_value_B, 0>)
        , disp_am_L(<aregister_L, a6>, <immediate_value_W, -8>)))
```

3.3 Life-time analysis of temporaries

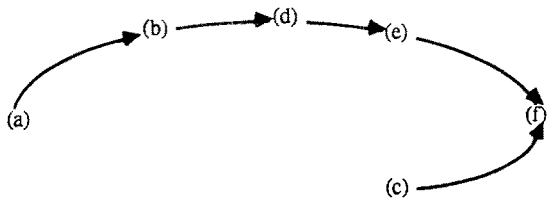
The rewriting process creates as many names of temporaries as required. All temporaries can not be held in registers and a great amount of space will have to be allocated. In order to decrease the number of temporaries, those that are not live simultaneously and whose constraint intersection is not empty can be packed into the same name. On the example, as the intervals definition of tmp0, tmp1, tmp2 are disjoint and all three of them are bounded to dregister_L, they can be packed into the same name tmp0. It follows that the third and the fifth move that now become move from tmp0 to tmp0 are no more useful and are deleted. The life-time analysis of the temporaries constitutes another pass of the **AG** related to the canonical **AAS**. The life-time analysis coupled with the data dependence graph of temporaries lead to rearrange the ordering of instructions in such a way that inside a block, physical adjacency of two instructions follows more closely data flow adjacently. This gives more efficient packing of temporaries. Let us consider the following term obtained after instruction selection.

```
Univ_seq (
(a) , Univ_assign (disp_am_L (<aregister_L, a6>, <immediate_value_W, -12>)
          , temporary_am (<temporary, tmp1>))
(b) , addi_L (immediate_val_am_L (<immediate_value_L, 24>)
          , temporary_am (<temporary, tmp1>))
(c) , Univ_assign (disp_am_L (<aregister_L, a6>, <immediate_value_W, 12>)
          , temporary_am (<temporary, tmp2>))
(d) , Univ_assign (temporary_am (<temporary, tmp1>)
          , temporary_am (<temporary, tmp3>))
(e) , mul_L (disp_am_L (<aregister_L, a6>, <immediate_value_W, -4>)
          , temporary_am (<temporary, tmp3>))
(f) , Univ_assign (temporary_am (<temporary, tmp3>)
          , areg_ind_am_L (<aregister_L, tmp2>))
```

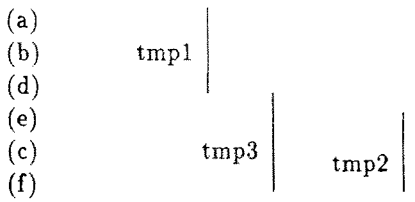



```
    binding (tmp1)  =  { aregister_L, relative_dlab_am_L }
    binding (tmp2)  =  { aregister_L }
    binding (tmp3)  =  { dregister_L }
```

No packing is possible here as tmp1 and tmp2 are live simultaneously and tmp1 and tmp3 have incompatible constraints. The data dependency graph is as follows :



The code is reordered : when a node depends from two branches, the longest branch is first written. The code is now the following after a new life-time analysis :



Now tmp1 and tmp2 can be packed into the same name with the constraint aregister_L. Note that the move (d) can not be removed and is used for the purpose of conversion between an address register and a data register. As the consequence of life-time analysis and packing of temporaries, some classical peephole optimization have no more to be apply inside a block, for instance a sequence of two moves from register to register transformed in a single move under hypothesis of live or dead register [DF 84].

3.3.1 Register assignment and code emission

- **register assignment.** According to the available storages, first for each temporary, we pick the storage class related to the cheapest access mode of the preceding set. Then we choose the name of an actual available location of this storage class for the temporary. In the first example, there is a strong constraint on tmp1 which must be a data register. If the first free data register is d0 , the register assignment is done for tmp0 this way.

- **code production.** The code emission is achieved by a decompilation process of the tree decorated by the format and the code operation attributes. Such a process is carried out using the PPAT (a Pretty Printer for Attributed Tree) module of FNC-2.

Thus, after the binding and the register assignment steps, the code emitted is :

```
MOVE.L   -4(a6), d0
ASL.L    #2, d0
SUBI.L   #1, d0
ADDI.L   #-48, d0
MOVE.L   0(a6,d0.L), -8(a6)
```

4 Conclusion

We have provided ways to specify compilers within the framework of a compiler writing system using **AAS** and **AG** with the background of abstract data types. The code-generator generator PAGODE is integrated using the same tools and formalism. Thus, each step of the compilation process is the mapping from a term of an **ADT** to a term of another **ADT**. In such a way, correctness proof of each step can be done [DMR 89].

The instruction selection process corresponds to a set of rewriting rules currently implemented in Prolog. We envisage more powerful tree pattern matching techniques such as in [DMR 89].

The instruction selection process corresponds to a set of rewriting rules currently implemented in Prolog. We envisage more powerful tree pattern matching techniques such as in [WW 88] to increase the efficiency of the instruction selection process.

A work currently in progress aims at infering peephole optimizing rules from the target machine specification.

Except in works of Fraser [FW 88] which aim at infering automatic peephole rules integrated in the code-generator system, peephole optimizations are done after code emission.

In our framework, each tree of the sequence is no more than one instance of an instruction after the instruction selection step. As temporaries stand for register or memory cell, it is possible to specify some general rules patterns for peephole especially for register tranfers. This rule can be used before register assignment or when instantiated by a coherent set of registers, access modes and instructions give transformation rules on the code itself.

Besides such global rule patterns, more machine specific peephole rules can be deduced from the machine description. One family of rules can be obtained using the semantics of the side effect operators involved in some access mode templates as predecrement. Another family of rules involved consecutive branch instructions.

References

[Cat 80] Cattell R. G. G. : Automatic Derivation of Code Generators from Machine Description. *ACM Transactions on Programming Languages and Systems*, Vol. 2, No.2, pp173-199, April 1980.

[DF 84] Davidson J.W., Fraser C.W. : Automatic Generation of Peephole Optimizations. *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*, ACM Vol. 19, 6, pp. 111-116, June 1984.

[DF 84] Davidson J.W., Fraser C.W. : Automatic Generation of Peephole Optimizations. *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*, ACM Vol. 19, 6, pp. 111-116, June 1984.

[DMR 87] Despland A., Mazaud M., Rakotozafy R. : Code generator generation based on template-driven target term rewriting. *Proceedings of Rewriting Techniques and Applications*, Bordeaux, France, May 1987 in LNCS no pp 105-120.

[DMR 88] Despland A., Mazaud M., Rakotozafy R. : An implementation of retargetable code generators in Prolog. *Proceedings of the International Workshop on Programming Language Implementation and Logic Programming*, Orleans, France, May 1988 in LNCS no 348 pp 81-104.

[DMR 89] Despland A., Mazaud M., Rakotozafy R. : *Using rewriting techniques to produce code-generators and proving them correct.* Rapport de Recherche INRIA RR 1046. 1989, to appear in Science of Computer Programming.

[Des 82] Deschamp Ph. : PERLUETTE : a compiler producing system using ADTs. *Proceedings of International Symposium on Programming*, Turin, April 1982.

[ESL 89] Emmelmann H., Schrörer F-W., Landwehr R. : BEG Generator for Efficient Back Ends. *Proceedings of the SIGPLAN 89, Symposium on Compiler Construction*, Portland, 21-23 June 1989.

[FW 88] Fraser C.W., Wendt A.L. : Automatic Generation of Fast Optimizing Code Generators. *Proceedings of the SIGPLAN 88 Symposium on Compiler Construction*, Sigplan Notices Vol. 23, 7, pp. 79-84, June 1988.

[Gan Gie 82] Ganzinger H, Giegerich R. : A truly Generative Semantics-Directed Compiler Generator. *Proceedings of the SIGPLAN 82, Symposium on Compiler Construction*, ACM SIGPLAN Vol. 17, 6, June 1982.

[GDM 84] Gaudel M. C., Deschamp Ph, Mazaud M. : *Compiler Construction From High Level Specification.* Automatic Program Construction Techniques, Macmillan Inc, 1984.

[GF 82] Ganapathi M., Fischer C.N. : Descriptive-Driven Code Generation Using attributed Grammars. *Conference of the Nineth Annual ACM Symposium on Programming Languages.*

[GG 78] Graham S.L, Glanville R. S. A New Method for Compiler Code Generation. Conference Record of the *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pp 231-240, January 1978.

[GH 84] Graham S. L., Henry R. R. & Al : Experiment with a Graham-Glanville Style Code generator. *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*, ACM Vol.. 19, 6, June 1984.

[Gie 90] Giegerich R. On the structure of Verifiable Code Generator Specifications *Proceedings of the SIGPLAN 90 Symposium on Compiler Construction*, ACM Vol.. 25, 6, June 1990.

[GS 88] Giegerich R., Schmal K. Code Selection Techniques : Pattern Matching, Tree Parsing, and Inversion of Derivors. *Proceedings of the ESOP'88* Nancy, France, March 88 in LNCS no 217 pp 247-268.

[JP 90] Jourdan M., Parigot D., Julie' C., Durin D., Le Bellec C : Design, Implementation and Evaluation of the FNC-2 Attribute Grammar System *Proceedings of the SIGPLAN 90 Symposium on Compiler Construction*, ACM Vol.. 25, 6, June 1990.

[LP 87] Lee P., Pleban U. : A Realistic Compiler Generator Based on High-Level Semantics. Conference *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, January 1987 pp 284-295.

[WW 88] Weisgerber B., Wilhelm R. Two tree pattern matchers for code selection in *Compilers Compilers and High Speed Compilation*, Berlin oct 88, D. Hammer, ed in LNCS no 371 pp 215-229.