Proceedings of TAPSOFT'95, Sixth International Joint Conference on Theory and Practice of Software Development, Aarhus, Denmark, May 1995. LNCS 915, pages 741–755, 1995.

A Model Inference System for Generic Specification with Application to Code Sharing

Didier Bert, Catherine Oriat

LGI - IMAG, BP 53, 38041 Grenoble Cedex 9, France
e-mail: {Didier.Bert,Catherine.Oriat}@imag.fr

Abstract. This paper presents a model inference system to control instantiation of generic modules. Generic parameters are specified by properties which represent classes of modules sharing some common features. Just as type checking consists in verifying that an expression is well typed, *model checking* allows to detect whether a (possibly generic) instantiation of a generic module is valid, i.e. whether the instantiation module is a *model* of the parameterizing property. Equality of instances can be derived from a canonical representation of modules. At last, we show how the code of generic modules can be shared for all instances of modules.

1 Introduction

Genericity is a useful feature for specification languages, and for programming languages alike, because it allows to reuse already written packages by instantiating them on various ways, thus limits the risk of bugs and reduces software costs. When a generic module is instantiated and imported into another module, one has to check that the instantiation is valid, i.e. that the instantiation module is a *model* of the formal part. For that, one can either rely on the syntax, i.e. on the theory defined by the modules, or on the semantics of the modules in the given specification language. In the first case one has to prove that some formulae are theorems in the theory. This problem is semi-decidable if the semantics is purely loose, but is undecidable if we work in an initial semantics [14]. In the second case, one has to check properties on classes of algebras, which is hard to do automatically. Consequently, in almost all specification languages (e.g. PLUSS [11, 6, 7], ACT-TWO [9], OBJ [10, 12], ...), such verifications are left to the user.

In this paper, we show that such verifications can partly be done automatically. We describe the model inference system used by the specification language LPG to control instantiation of generic modules. LPG (Langage de Programmation Générique, i.e. language for generic programming) is a specification language developed at the IMAG Institute by Didier Bert and Rachid Echahed [4, 5]. LPG allows on the one hand to define and combine generic components of specifications, and on the other hand to make prototypes thanks to an evaluation tool. There is also a solver of goals associating functional and logic programming.

In LPG, generic *modules* are parameterized by *properties*. The semantics of LPG mixes loose and initial features: the semantics of a property is a class of algebras, while the semantics of a generic module is a free functor. An LPG module can be instantiated by another one only if the other module *is a model* of the required property. Properties and modules are related by *constraints*. These constraints are similar those given by Ehrig and Mahr [9] in that they put restrictions on classes of algebras. However, they differ in several points: firstly, Ehrig and Mahr only consider inclusion of specifications, whereas we consider any morphism in LPG. Secondly, for Ehrig and Mahr, the initial (or free) semantics of a specification is stated at the time of its importation. In other words a specification `boolean` can be imported once with a loose semantics, and once with an initial semantics. In LPG, the semantics of a unit is stated once for all at the time of its definition. Thirdly, and this is the original feature of LPG we want to stress in this paper, the language provides an inference system to generate new constraints from declared ones.

This inference system can be compared to type systems used for programming languages: just as types allow to control utilizations of variables, constraints allow to control instantiations of modules. There is one important difference though: constraints apply at the level of units, and are therefore category theoretic (i.e. formulated with morphisms) rather than set theoretic (i.e. formulated with membership or inclusion). In particular, there are various ways a module can be a model of a property.

Such verifications for modules already exist in some programming languages. For instance in Ada [1], homology rules are used to check the validity of instantiations; e.g. with *private* and *limited* types. In Ada, these rules apply to one type only. In contrast, M. V. Aponte proposed a type system for checking SML modules [2], based on unification and sharing, and which performs verifications w.r.t. the whole specification of the generic part of a module. In this approach, verifications are based on the names of types and functions, and therefore there are not various ways an SML-structure (i.e. a module) can match an SML-signature (i.e. a property).

Constraints allow us to reason locally about units. The semantics of algebraic specification languages often seems complicated because it is global, i.e. one has to know the semantics of all imported units to know the semantics of the current unit. Making constraints explicit does not change the semantics, but allows to make safe deductions without having to be aware of all importations at the same time. The inference system presented in this paper is sound with respect to the algebraic semantics of the language. Note that it is not complete, and *cannot* be complete with respect to this semantics. One reason is that we work in initial algebras, and therefore it is impossible to deduce all semantically true statements from any deductive system. All we can do is to rely on the user's declarations, and make safe deductions.

The paper is organized as follows: section 2 and 3 present LPG units and constraints. In section 4 we describe the inference system which allows to deduce new constraints, and thus checks the validity of instantiations. In section 5 we show the representation used for instances of generic modules. This representation allows to share imported modules consistently. Section 6 presents the compilation of modules. In contrast to languages such as Ada [1] or C++ [13], generic modules are compiled only once in LPG, all the instantiations sharing the same code. This is an interesting feature for a prototyping language, because it reduces compilation times a lot when developing highly generic programs.

2 The Language Constructions

The LPG language has two kinds of units, namely *properties* and *modules*. LPG modules allow to define abstract data types, and more generally to group together a set of types and operators logically related. LPG modules can be generic, i.e. parameterized by a set of types and operators. The generic part of a module is itself an LPG unit, and is called the *property required by the module*. LPG units are composed of a *signature* and a set of formulae, which are conditional equations. In modules, the equations may be oriented; in this case they can be compiled and executed by the evaluation tool.

```
module BOOLEAN
types boolean
constructors
   true, false : boolean
operators
   not : boolean -> boolean
   and : boolean, boolean -> boolean
   or  : boolean, boolean -> boolean
equations
   not(true) ==> false
   not(false) ==> true
   and(true,x) ==> x
   and(false,x) ==> false
   or(true,x) ==> true
   or(false,x) ==> x
```

```
property ANY
types any

property DISCRETE
types t
operators first, last : t
          next : t -> t

property MONOID
types t
operators e : t
          op : t,t -> t
equations
   op(e,x) == x
   op(x,e) == x
   op(x,op(y,z)) == op(op(x,y),z)
```

Fig. 1. Simple examples of modules and properties

Figure 1 shows some simple examples of LPG units: the module `BOOLEAN`, the properties `ANY` (specifying a single type `any`), `DISCRETE` and `MONOID`. Figure 2 defines the generic module of lists, parameterized by the property `ANY`.

```
module LIST requires ANY[elem]
types list
constructors nil : list ;  cons : elem,list -> list
operators head : list -> elem ;  tail : list -> list
equations head(cons(e,l)) ==> e ;  tail(cons(e,l)) ==> l
```

Fig. 2. Generic module of lists

Given a module M which requires a property P, there is an injective signature morphism from P to M, which is merely a renaming of the types and operators of the property P into the module M. These morphisms are noted $P \xhookrightarrow{r} M$. The module of lists is noted `ANY` $\xhookrightarrow{r}$ `LIST`, where r is the morphism $\{$`any` $\mapsto$ `elem`$\}$. r is given in the module `LIST` by the statement `requires ANY[elem]`.

A non generic module M (such as `BOOLEAN` for instance) can be considered as a generic module $\Phi \xhookrightarrow{\phi_M} M$ parameterized by the empty property Φ, which contains no types nor operators. ϕ_M is the only morphism from Φ to M. (Categorically speaking, Φ is the initial object of the category of signatures.)

We suppose the reader is familiar with basic concepts of algebraic specification (see e.g. [8]). The class of algebras which satisfy a unit U together with homomorphisms is a category noted $Alg\,(U)$. If $m : U_1 \to U_2$ is a signature morphism, then there is a forgetful functor $\mathcal{U}_m : Alg\,(U_2) \to Alg\,(U_1)$, and a free functor left adjoint to $\mathcal{U}_m$, $\mathcal{F}_m : Alg\,(U_1) \to Alg\,(U_2)$. We do not define the whole semantics of LPG here, but only present the features which are relevant for this paper. For a complete description of the semantics of LPG, see [15].

Definition 1. (semantics of a property) The semantics of a property P is a class of algebras $mod\,(P)$, which satisfy the specification P, i.e.: $mod\,(P) \subseteq Alg\,(P)$. The semantics of a property need not be the whole class $Alg\,(P)$ because some algebras may be left out to preserve imported modules.

Definition 2. (semantics of a module) The semantics of a module $P \xhookrightarrow{r} M$ is the free functor $\mathcal{F}_r : Alg\,(P) \to Alg\,(M)$. The free functor associates to each algebra of $Alg\,(P)$ the algebra freely generated on M. This functor must be strongly persistent on algebras of $mod\,(P)$, i.e.: for all algebras A of $mod\,(P)$, $\mathcal{U}_r(\mathcal{F}_r(A)) = A$.

This condition expresses that previously defined units must be preserved, i.e. that introducing a new module does not change the semantics of old units. Let 1 be the only algebra satisfying the empty property Φ. When a module is not generic, i.e. when P is the empty property, then $\mathcal{F}_r(1)$ is the initial algebra.

3 Constraints

There are five kinds of constraints relating LPG units, namely *model*, *satisfaction*, *combination*, *importation of a module into a property*, and *into a module.* A constraint is composed of a signature morphism and of a semantic condition, which states the validity of the constraint.

Definition 3. (model constraint) A module $P \xhookrightarrow{r} M$ is a *model* of a property P_1 if there is a signature morphism $P_1 \xrightarrow{m} M$ and if the formulae of P_1 hold (through the translation induced by m) in M.

$$\text{Model}\,(P_1 \xrightarrow{m} M) \overset{\text{def}}{\Leftrightarrow} \mathcal{U}_m(\mathcal{F}_r(mod\,(P))) \subseteq mod\,(P_1)$$

For instance, we can express that the module BOOLEAN is a model of the property ANY with the following declaration of model (written in the module BOOLEAN):

```
models ANY[boolean]
```

This declaration defines the signature morphism {any $\mapsto$ boolean} from ANY to BOOLEAN. As there is no equation in the property ANY, nothing else has to be checked. In the same way, we can define different models of DISCRETE with natural numbers, for instance the natural numbers from 1 to 10 with the successor operator; or natural numbers from 49 to 0, with the predecessor operator.

```
models DISCRETE[natural,1,10,succ], DISCRETE[natural,49,0,pred]
```

We can also express that the module BOOLEAN is a model of MONOID:

```
models MONOID[boolean,true,and], MONOID[boolean,false,or]
```

Definition 4. (satisfaction constraint) A property P_2 *satisfies* a property P_1 if there is a signature morphism $P_1 \xrightarrow{s} P_2$ and if any module which is a model of P_2 is (through the translation induced by s) a model of P_1.

$$\text{Sat}\,(P_1 \xrightarrow{s} P_2) \overset{\text{def}}{\Leftrightarrow} \mathcal{U}_s(mod\,(P_2)) \subseteq mod\,(P_1) \;\Leftrightarrow\; mod\,(P_2) \subseteq \mathcal{U}_s^{-1}(mod\,(P_1))$$

For instance, we can state that the property MONOID satisfies ANY, with the declaration `satisfies ANY[t]` in the unit MONOID. The declaration states that there is a morphism ANY $\xrightarrow{s}$ MONOID $=$ {any $\mapsto$ t}, such that any model of MONOID is a model of ANY.

Combination. Properties can be *combined*, i.e. put together to form a new property. Figure 3 shows a property specifying any type and a discrete type.

```
property ANY+DISCRETE
combines ANY[elem], DISCRETE[index,first,last,next]
```

Fig. 3. Property ANY+DISCRETE

The combination constraint states that any model of ANY+DISCRETE is a model of ANY and is a model of DISCRETE (i.e. ANY+DISCRETE satisfies ANY and DISCRETE). Conversely, any two models of ANY and DISCRETE allow to construct a model of ANY+DISCRETE.

In this example, the "union" of both properties happens to be disjoint, i.e. no symbol of type nor operator appears twice. We can for instance specify a property ANY_DISCRETE, where the type of ANY and the type of DISCRETE are shared. We thus specify a class of modules with *one* type which is a model of both ANY and DISCRETE. Then any two models of ANY and DISCRETE *which share this type* allow to construct a model of ANY_DISCRETE.

Definition 5. (combination constraint) A property P is a combination of the properties $P_1, P_2, \ldots P_k$ w.r.t. the morphisms $P_i \xrightarrow{c_i} P$, $\forall i \in \{1, \ldots k\}$ if models of $P_1, P_2, \ldots P_k$ which share the same types and operators as specified in P allow to construct a model of P.

$$\mathrm{Comb}\,(P_1, \ldots P_k \xrightarrow{c_1, \ldots c_k} P) \overset{\text{def}}{\Leftrightarrow} \mathit{mod}\,(P) = \bigcap_{i=1}^{k} \mathcal{U}_{c_i}^{-1}(\mathit{mod}\,(P_i))$$
$$\Leftrightarrow \begin{cases} \forall i \in \{1, \ldots k\},\, \mathcal{U}_{c_i}(\mathit{mod}\,(P)) \subseteq \mathit{mod}\,(P_i) \\ \forall A \in \mathit{Alg}\,(P),\ (\forall i \in \{1, \ldots k\},\, \mathcal{U}_{c_i}(A) \in \mathit{mod}\,(P_i)) \Rightarrow A \in \mathit{mod}\,(P) \end{cases}$$

Importation and Instantiation. Once a generic module has been defined, it is possible to use it in another unit. This is called importation into a module or into a property. When a module is imported, its formal part (i.e. the signature contained in its required property) must be instantiated, either with actual, or formal parameters, or both. This instantiation defines a signature morphism from the imported module to the currently defined unit.

We define on figure 4 a module called `VECTOR`, parameterized by the property `ANY+DISCRETE`. The property `ANY` gives the type of information stored in a *vector*, and the property `DISCRETE` defines the index. We are not concerned here with the actual representation of vectors, therefore we only specify two operations: `store` which assigns a new value to an index, and `get` which picks up the value associated to an index. From now on, the axiomatization of operators is omitted.

```
module VECTOR requires ANY+DISCRETE[t,index,first,last,next]
types vector
operators store : vector, index, t -> vector
          get : vector, index -> t
```

Fig. 4. Part of the module `VECTOR`

Then we may define vectors of integers with some new operations. For that, we have to import the module `INTEGER` containing integer values as well as usual operations on them. This module no longer requires a type for the information stored, so it is only parameterized by the property `DISCRETE`.

```
module INTEGER_VECTOR requires DISCRETE[index,first,last,next]
imports INTEGER, VECTOR[integer,index,first,last,next]
operators scalar_prod : vector, vector -> integer
```

Fig. 5. Module of vectors of integers

Another example: given a binary operator on the type `t`, we can define a binary operator on vectors. The module figure 6 defines a null vector and a sum of vectors, given a null element `e` and an associative binary operator `op`. Note that we have also stated that vectors with these two operators form a monoid.

```
module VECTOR_SUM requires MONOID+DISCRETE[t,e,op,index,first,last,next]
imports VECTOR[t,index,first,last,next]
operators null_vect : vector ;  sum_vect : vector, vector -> vector
models MONOID[vector,null_vect,sum_vect]
```

Fig. 6. Vectors with a binary operator

In example 5, the type `vector` refers to the type *vector of integers*, whereas in example 6 it refers to *vector of t*. There is no confusion because the module `VECTOR` is only imported once in each module. If we want to import a module several times (with different instantiations) in a module, we have to name the instantiated modules:

```
INTEGER_V = VECTOR[integer,index,first,last,next]
      T_V = VECTOR[t,index,first,last,next]
```

and then to refer to the types and operators as `INTEGER_V.vector`, `T_V.vector`, `INTEGER_V.store` and so on.

The originality of LPG is that not any importation is valid. For instance, the importation `imports LIST[integer]` is valid only if the module `INTEGER` is a model of the property `ANY` with the morphism $\{$`any` $\mapsto$ `integer`$\}$. This can be the case either if the user has defined such a model with the declaration `models ANY[integer]`, or if the system can deduce it from other declarations, using the inference system presented next section. For instance, if `INTEGER` is a model of `MONOID`, and if `MONOID` satisfies `ANY`, then `INTEGER` is a model of `ANY`.

The examples we have presented here are importations of a module into another module. It is also possible to import a module into a property.

Definition 6. (constraint of importation of a module into a module) Let $P_1 \xhookrightarrow{r_1} M_1$ and $P_2 \xhookrightarrow{r_2} M_2$ be two modules. $P_1 \xhookrightarrow{r_1} M_1$ is imported into $P_2 \xhookrightarrow{r_2} M_2$ with the morphism $M_1 \xrightarrow{i} M_2$ if:

$$\text{Import\_M}\,(M_1 \xrightarrow{i} M_2) \overset{\text{def}}{\Leftrightarrow} \mathcal{U}_i(\mathcal{F}_{r_2}(\mathit{mod}\,(P_2))) \subseteq \mathcal{F}_{r_1}(\mathit{mod}\,(P_1))$$

$$\Leftrightarrow \begin{cases} \text{Model}\,(P_1 \xrightarrow{i \circ r_1} M_2) \\ \forall A_2 \in \mathit{mod}\,(P_2),\ \mathcal{U}_i(\mathcal{F}_{r_2}(A_2)) = \mathcal{F}_{r_1}(\mathcal{U}_{r_1}(\mathcal{U}_i(\mathcal{F}_{r_2}(A_2)))) \qquad (\mathcal{H}_M) \end{cases}$$

The morphism i expresses the *instantiation* of the generic part of the module $P_1 \xhookrightarrow{r_1} M_1$ with a part of the module $P_2 \xhookrightarrow{r_2} M_2$, and the *inclusion* of the non generic part of $P_1 \xhookrightarrow{r_1} M_1$ into $P_2 \xhookrightarrow{r_2} M_2$.

Definition 7. (constraint of importation of a module into a property) Let $P_1 \xhookrightarrow{r_1} M_1$ be a module, P_2 be a property. $P_1 \xhookrightarrow{r_1} M_1$ is imported into P_2 with the morphism $M_1 \xrightarrow{i} P_2$ if:

$$\text{Import\_P}\,(M_1 \xrightarrow{i} P_2) \overset{\text{def}}{\Leftrightarrow} \mathcal{U}_i(\mathit{mod}\,(P_2)) \subseteq \mathcal{F}_{r_1}(\mathit{mod}\,(P_1))$$

$$\Leftrightarrow \begin{cases} \text{Sat}\,(P_1 \xrightarrow{i \circ r_1} P_2) \\ \forall A_2 \in \mathit{mod}\,(P_2),\ \mathcal{U}_i(A_2) = \mathcal{F}_{r_1}(\mathcal{U}_{r_1}(\mathcal{U}_i(A_2))) \qquad (\mathcal{H}_P) \end{cases}$$

4 Inference Rules for Model Checking

In this section, we describe the rules which allow to combine constraints to build new ones, and thus provide an inference system of constraints. Every declaration of a model, satisfaction or combination constraint gives a corresponding axiom. The user must check that these axioms are semantically correct, i.e. that the associated semantic condition is satisfied.

$$\{\mathcal{H}_M\}\frac{\text{Model}\,(P_1 \xrightarrow{i o r_1} M_2)}{\text{Import\_M}\,(M_1 \xrightarrow{i} M_2)} \quad \text{(IM)} \qquad \{\mathcal{H}_P\}\frac{\text{Sat}\,(P_1 \xrightarrow{i o r_1} P_2)}{\text{Import\_P}\,(M_1 \xrightarrow{i} P_2)} \quad \text{(IP)}$$

$$\frac{}{\text{Sat}\,(\Phi \xrightarrow{\phi_P} P)} \ (1) \qquad \frac{}{\text{Model}\,(\Phi \xrightarrow{\phi_M} M)} \ (2) \qquad \frac{}{\text{Model}\,(P \xrightarrow{r} M)} \ (3)$$

$$\frac{\text{Sat}\,(P_1 \xrightarrow{s_1} P_2)\ ;\ \text{Sat}\,(P_2 \xrightarrow{s_2} P_3)}{\text{Sat}\,(P_1 \xrightarrow{s_2 o s_1} P_3)} \quad (4)$$

$$\frac{\text{Sat}\,(P_1 \xrightarrow{s} P_2)\ ;\ \text{Model}\,(P_2 \xrightarrow{m} M)}{\text{Model}\,(P_1 \xrightarrow{m o s} M)} \quad (5)$$

$$\frac{\text{Model}\,(P \xrightarrow{m} M_1)\ ;\ \text{Import\_M}\,(M_1 \xrightarrow{i} M_2)}{\text{Model}\,(P \xrightarrow{i o m} M_2)} \quad (6)$$

$$\frac{\text{Model}\,(P_1 \xrightarrow{m} M_2)\ ;\ \text{Import\_P}\,(M_2 \xrightarrow{i} P_3)}{\text{Sat}\,(P_1 \xrightarrow{i o m} P_3)} \quad (7)$$

$$\frac{\text{Comb}\,(P_1, \ldots P_k \xrightarrow{c_1, \ldots c_k} P)}{\forall j \in \{1, \ldots k\},\ \text{Sat}\,(P_j \xrightarrow{c_j} P)} \quad (8)$$

$$\frac{\text{Comb}\,(P_1, \ldots P_k \xrightarrow{c_1, \ldots c_k} P)\ ;\ \forall j \in \{1, \ldots k\},\ \text{Sat}\,(P_j \xrightarrow{s o c_j} P')}{\text{Sat}\,(P \xrightarrow{s} P')} \quad (9)$$

$$\frac{\text{Comb}\,(P_1, \ldots P_k \xrightarrow{c_1, \ldots c_k} P)\ ;\ \forall j \in \{1, \ldots k\},\ \text{Model}\,(P_j \xrightarrow{m o c_j} M')}{\text{Model}\,(P \xrightarrow{m} M')} \quad (10)$$

Fig. 7. Main inference rules

Figure 7 shows the set of *main rules* used by the system. Properties are noted P, P_1, P_2, ... Modules such as $P \xhookrightarrow{r} M$, $P_1 \xhookrightarrow{r_1} M_1$, $P_2 \xhookrightarrow{r_2} M_2$, ... are just noted M, M_1, M_2, ... The rules (IM) and (IP) are associated to declarations of importations. Their application is conditioned by the hypothesis $\mathcal{H}_M$ or $\mathcal{H}_P$, which must be checked by the user. The other rules are not associated with any hypothesis, which means that their application is always possible. Axioms 1 and 2 state that any property P satisfies the empty property Φ, and that any module $P \xhookrightarrow{r} M$ is a model of Φ. As Φ is initial, the morphisms ϕ_P and ϕ_M are unique. Axiom 3 expresses that a module $P \xhookrightarrow{r} M$ is a model of its own property P, with the morphism r. In particular, if two modules are parameterized by the same property, then one can instantiate one module with the formal part of the other one. Rules 4 to 7 are composition rules. Rules 8 to 10 are related to the `combines` constraint.

$$\frac{\text{Import\_P}\,(M \xrightarrow{i} P_1)\ \ ;\ \ \text{Sat}\,(P_1 \xrightarrow{s} P_2)}{\text{Import\_P}\,(M \xrightarrow{s \circ i} P_2)} \tag{11}$$

$$\frac{\text{Import\_P}\,(M \xrightarrow{i} P_1)\ \ ;\ \ \text{Model}\,(P_1 \xrightarrow{m} M_2)}{\text{Import\_M}\,(M \xrightarrow{m \circ i} M_2)} \tag{12}$$

$$\frac{\text{Import\_M}\,(M_1 \xrightarrow{i_1} M_2)\ \ ;\ \ \text{Import\_P}\,(M_2 \xrightarrow{i_2} P_3)}{\text{Import\_P}\,(M_1 \xrightarrow{i_2 \circ i_1} P_3)} \tag{13}$$

$$\frac{\text{Import\_M}\,(M_1 \xrightarrow{i_1} M_2)\ \ ;\ \ \text{Import\_M}\,(M_2 \xrightarrow{i_2} M_3)}{\text{Import\_M}\,(M_1 \xrightarrow{i_2 \circ i_1} M_3)} \tag{14}$$

Fig. 8. Derived inference rules

These rules are actually used by the LPG system. One can note that we have not considered all possible compositions. The remaining compositions are described in figure 8. These *derived* rules are not used by the system, because we have the following result:

Theorem 8. *Any proof involving derived rules can be transformed into a proof only involving main rules.*

Proof. Any introduction of an Import_P constraint is preceded by a satisfaction constraint, and any Import_M constraint is preceded by a model constraint. This allows to get rid of all derived rules, from the axioms to the conclusion.

Theorem 9. *The inference system is sound with respect to the semantics.*

This result means that provided the conditions associated to declaration axioms and rules (IM, IP) are satisfied, the constraints deduced by the inference system are semantically correct.

Examples of Deductions. In this paragraph, we reconsider the examples of importations given in the previous section and prove their validity using the inference system.

The importation of a non generic module into a module or into a property is always valid in the system, provided that the corresponding condition $\mathcal{H}_M$ or $\mathcal{H}_P$ is satisfied. This can be shown by using rule (2) followed by rule (IM), or by using rule (1) followed by rule (IP). In particular, the importation of the module `INTEGER` into `INTEGER_VECTOR` (figure 5) is valid.

Let us now consider the importation of `VECTOR` into `INTEGER_VECTOR` (figure 5), as well as the importation of `VECTOR` into `VECTOR_SUM` (figure 6). We are going to take shorter notation, in order to be able to draw the proofs.

Properties	
`ANY`	A
`DISCRETE`	D
`MONOID`	M
`ANY+DISCRETE`	AD
`MONOID+DISCRETE`	MD

Modules	
`INTEGER`	I
`VECTOR`	$AD \xhookrightarrow{r_1} V$
`INTEGER_VECTOR`	$D \xhookrightarrow{r_2} IV$
`VECTOR_SUM`	$MD \xhookrightarrow{r_3} VS$

$$\dfrac{\dfrac{\text{Comb}\,(A, D \xrightarrow{c_1,c_2} AD) \quad \dfrac{\dfrac{\text{Model}\,(A \xrightarrow{m} I) \quad \text{Import\_M}\,(I \xrightarrow{i_1} IV)}{\text{Model}\,(A \xrightarrow{i_1 \circ m} IV)}\,(6)}{\text{Model}\,(A \xrightarrow{i \circ r_1 \circ c_1} IV)}\,(=) \quad \dfrac{\dfrac{}{\text{Model}\,(D \xrightarrow{r_2} IV)}\,(3)}{\text{Model}\,(D \xrightarrow{i \circ r_1 \circ c_2} IV)}\,(=)}{\text{Model}\,(AD \xrightarrow{i \circ r_1} IV)}\,(10)}{\text{Import\_M}\,(V \xrightarrow{i} IV)}\,(\text{IM})$$

$$\dfrac{\dfrac{\text{Sat}\,(A \xrightarrow{s} M) \quad \dfrac{\dfrac{\text{Comb}\,(M, D \xrightarrow{c'_1,c'_2} MD)}{\text{Sat}\,(M \xrightarrow{c'_1} MD)}\,(8) \quad \dfrac{}{\text{Model}\,(MD \xrightarrow{r_3} VS)}\,(3)}{\text{Model}\,(M \xrightarrow{r_3 \circ c'_1} VS)}\,(5)}{\text{Model}\,(A \xrightarrow{r_3 \circ c'_1 \circ s} VS)}\,(5)}{\text{Model}\,(A \xrightarrow{i' \circ r_1 \circ c_1} VS)}\,(=)$$

$$\dfrac{\dfrac{\dfrac{\text{Comb}\,(M, D \xrightarrow{c'_1,c'_2} MD)}{\text{Sat}\,(D \xrightarrow{c'_2} MD)}\,(8) \quad \dfrac{}{\text{Model}\,(MD \xrightarrow{r_3} VS)}\,(3)}{\text{Model}\,(D \xrightarrow{r_3 \circ c'_2} VS)}\,(5)}{\text{Model}\,(D \xrightarrow{i' \circ r_1 \circ c_2} VS)}\,(=)$$

$$\dfrac{\dfrac{\text{Comb}\,(A, D \xrightarrow{c_1,c_2} AD) \quad \text{Model}\,(A \xrightarrow{i' \circ r_1 \circ c_1} VS) \quad \text{Model}\,(D \xrightarrow{i' \circ r_1 \circ c_2} VS)}{\text{Model}\,(AD \xrightarrow{i' \circ r_1} VS)}\,(10)}{\text{Import\_M}\,(V \xrightarrow{i'} VS)}\,(\text{IM})$$

Fig. 9. Proofs of importations

We suppose the user has declared the following constraints:

`INTEGER` is a model of `ANY`: Model $(A \xrightarrow{m} I)$. `MONOID` satisfies `ANY`: Sat $(A \xrightarrow{s} I)$.

`ANY+DISCRETE` is a combination of `ANY` and `DISCRETE`: Comb $(A, D \xrightarrow{c_1,c_2} AD)$;

and `MONOID+DISCRETE` of `MONOID` and `DISCRETE`: Comb $(M, D \xrightarrow{c'_1,c'_2} MD)$.

The importation of `INTEGER` into `INTEGER_VECTOR` is noted Import_M $(I \xrightarrow{i_1} IV)$.

The proofs that Import_M $(V \xrightarrow{i} IV)$ and Import_M $(V \xrightarrow{i'} VS)$ are valid are shown figure 9. Note that we use a rule called $(=)$ which means that we use an equality between morphisms. Indeed we have $i_1 \circ m = i \circ r_1 \circ c_1$, $r_2 = i \circ r_1 \circ c_2$, $r_3 \circ c'_1 \circ s = i' \circ r_1 \circ c_1$, and $r_3 \circ c'_2 = i' \circ r_1 \circ c_2$. This rule appears here mainly to clarify the proofs. It is not used as such by the system which works with an internal representation of morphisms as a set of pairs, and not with a symbolic notation.

5 Representation of Imported Modules

When a module is instantiated and imported, there is no creation of a new module. For instance when we write

```
imports T_V = VECTOR[t,index,i1,in,s]
        T_V2 = VECTOR[t,index,i1,in,s]
```

`T_V` and `T_V2` represent the same module, and in particular, `T_V.vector` and `T_V2.vector` refer to the same type.

This implies that instantiations can be done in various orders, as shown figure 10: the names `INT_MAT1` and `INT_MAT2` refer to the same module. We thus have an equality of modules which is stronger than equality of names, in the sense that two modules with different names may be equal. The equality is of course extended to types and operators. This allows to make multiple enrichments: we may for instance make an enrichment of `VECTOR` by importing two different enrichments `ENRICH_VECTOR1` and `ENRICH_VECTOR2`. The common part of both modules (i.e. the module `VECTOR`) will be shared correctly.

```
module ENRICH_VECTOR requires ANY+DISCRETE[t,ind,i1,in,s]
imports INTEGER
        T_V = VECTOR[t,ind,i1,in,s]
        T_MAT = VECTOR[T_V.vector,ind,i1,in,s]
        INT_V = VECTOR[integer,ind,i1,in,s]
        INT_MAT1 = T_MAT[integer,ind,i1,in,s]
        INT_MAT2 = VECTOR[INT_V.vector,ind,i1,in,s]
```

Fig. 10. Example of instantiations

To achieve this, modules are encoded with two pieces of information: first the origin module (i.e. the module we want to import), and secondly the morphism from the required property of the origin module to the current module. That way, named intermediary modules used for clarification are never stored in the system. Similarly, types and operators are encoded with three pieces of information: their name, the module they come from and the morphism from the required property of the origin module to the current module. For instance, addition on integers is coded as $\texttt{+} = \langle \texttt{+}, \texttt{INTEGER}, \{\}\rangle$, where $\{\}$ is of course the initial morphism $\Phi \xrightarrow{\phi_I} \texttt{I}$. Let now m be the morphism
$m = \{\ \texttt{elem} \mapsto \texttt{integer}, \texttt{index} \mapsto \texttt{ind}, \texttt{first} \mapsto \texttt{i1}, \texttt{last} \mapsto \texttt{in}, \texttt{next} \mapsto \texttt{s}\}$:

$\texttt{INT\_V.vector} = \langle \texttt{vector}, \texttt{VECTOR}, m\rangle$
$\texttt{INT\_MAT1.store} = \texttt{INT\_MAT2.store} =$
$\langle\ \texttt{store}, \texttt{VECTOR}, \{\ \texttt{elem} \mapsto \langle\ \texttt{vector}, \texttt{VECTOR}, m\ \rangle,$
$\qquad \texttt{index} \mapsto \texttt{ind}, \texttt{first} \mapsto \texttt{i1}, \texttt{last} \mapsto \texttt{in}, \texttt{next} \mapsto \texttt{s}\ \}\ \rangle$

6 Compilation of Modules

The representation of imported modules allows to perform fast code generation for operators of generic modules. The point is to share the code of generic operators with all their instantiations. So, code generation is modular and avoids multiple copies of the common parts. Notice that copying the code of generic modules can be an option for run-time optimization, as for on-line generation of the code of procedure bodies. In this section, we give insights on principles of code generation without too many details about the generated code.

The execution abstract machine for generic operators is constituted of the usual components of such machines, i.e. return-address stack (`r-st`), parameters and memory stack (`m-st`) and evaluation stack (`v-st`). It can also deal with exception recovery mechanism and handling contexts. The technique presented here is independent from these features.

Compiling generic operators requires the introduction of a new stack to perform generic parameter bindings. At run-time, only generic parameters with dynamic behaviour have to be stored in this stack, which is called generic parameter stack (or `g-st`). For example, we assume that types have only a static scope and do not need to be represented in the g-stack. The dynamic part of a property P is noted $dyn(P)$. In the framework presented in the paper, $dyn(P)$ is the list of operators of P. So, for a generic module $P \xhookrightarrow{r} M$ the g-stack is intended to represent the morphism which binds formal operators of $dyn(P)$ to effective functions. The morphism i restricted to the operators of $dyn(P)$ is noted $dyn(i)$. Its cardinality is written $\#(dyn(P))$.

The compilation procedure *compile* is presented for an expression in a generic module $P \xhookrightarrow{r} M$. It takes an expression and produces the corresponding code for a machine with a state ⟨`v-st`,`m-st`,`r-st`,`g-st`⟩. Operations on stacks and other macro-commands for the abstract machine are defined figure 11. At run-time, the result of the evaluation of an expression is always at the top of the evaluation stack: `v-st`. Moreover, the elements at the top of `g-st` are the addresses of the effective functions bound to the formal operators of P. The convention adopted here is that identifiers of the compiling procedure are in italic type style whereas generated macrocode is in type-written type style. The procedure *gen* generate a macro-instruction for the abstract machine. All generating procedure can be used as functions. In this case, the value returned is the address of (or a reference to) the beginning of the generated code. In this text, address variables are "logical" variables, because they can be used before having been assigned to. If this happens, these variables are flagged by "$^\flat$" (for "before").

```
compile (f(e_1, ..., e_n)) =
    compile (e_1);                    -- code for evaluating e_1
    gen ("m-st.push(v-st.rtop)");     -- the value of e_1 will be kept in m-st
    ...;
    compile (e_n);                    -- code for evaluating e_n
    gen ("m-st.push(v-st.rtop)");     -- the value of e_n will be at the top of m-st
    compile_op (f, 0);                -- code for the call of f
    gen ("m-st.pop(n)");              -- code to remove the arguments of f
```

For a stack object `st` the following operations are available in the abstract machine:

`st.push(x)`	push `x` at the top of `st`
`st.top`	return the value of the top of `st`
`st.pop`	remove the top element of `st`
`st.pop(n)`	remove n elements at the top of `st`
`st.rtop`	return `top(st)` and also perform `pop(st)`
`st.elem(n)`	return the element of `st` at offset n from the top. `st.elem(0)` is the same as `st.top`.

The following macro instructions are used in the paper:

`call(f)`	push the address of the next statement onto `r-st` and jump to the address of the beginning of code of `f`.
`fcall(i)`	push the address of the next statement onto `r-st` and jump to the address of `g-st.elem(i)`. Notice that `i` is a constant.
`return`	perform `r-st.rtop` and jump to this address.
`jump_to(a)`	go to the given address `a`.

Fig. 11. Operations of the abstract machine

For example, if f is a function with n parameters the code generation corresponding to the use of the variable x_i in the body of f is:

$compile\,(x_i) = gen\,(\texttt{"v-st.push(m-st.elem(}n-i\texttt{))"});$

Now, let us consider the procedure $compile\_op\,(f,k)$. k is an integer giving the depth in the g-stack where the binding morphism of the current module $P \xhookrightarrow{r} M$ is. For the call above, the depth is clearly 0. Three cases have to be considered.
1. f is declared in M and is not a formal operator of the required property:

$compile\_op\,(f,k) = gen\,(\texttt{"call(}f\texttt{)"});$

2. f is declared in $P \xhookrightarrow{r} M$ and is a formal operator, at rank j in the list of operators. The address of the effective operator is in the g-stack:

$compile\_op\,(f,k) = gen\,(\texttt{"fcall(}\#(dyn(P)) - j\texttt{)"});$

3. f is declared in the module $P_1 \xhookrightarrow{r_1} M_1$, imported in M: Import_M $(M_1 \xrightarrow{i} M)$.

```
$compile\_op\,(f,k) =$
    $install\_generic\_context\,(dyn(i \circ r_1), k);$
    $gen\,(\texttt{"call(}f\texttt{)"});$
    $gen\,(\texttt{"g-st.pop(}\#(dyn(P_1))\texttt{)"});$            -- restore the generic context
```

The installation of the generic context consists in pushing onto the g-stack the addresses of pieces of code performing calls to each effective parameter. So, the first step is to develop the installation for each operator:

```
$install\_generic\_context\,(\{o_1 \mapsto f_1, \ldots, o_l \mapsto f_l\}, k) =$
    $compile\_par\,(\{o_1 \mapsto f_1\}, k);$
    ...
    $compile\_par\,(\{o_l \mapsto f_l\}, k);$
```

Now, for each binding, two cases occur: either the target operator f_j is an effective operator (in the module M) and then, we have to generate the call to f_j as a thunk and to push the address of this thunk on the g-stack, or the target is a

formal operator in M, say o'_i and then, at run-time, the address of the effective operator bound to o'_i will be already in the g-stack at a given offset from the top. In the first case, the depth of the morphism of the current required property P must be increased by the cardinality of the binding morphism of P_1. These two cases are presented below:
1. The target f_j is effective:

```
$compile\_par(\{o_j \mapsto f_j\}, k) =$
    $gen($"jump_to($link\_address^\flat$)"$)$;
    $thunk\_address := compile\_op(f_j, k + \#(dyn(P_1)))$;
    $gen($"return"$)$;
    $link\_address := gen($"g-st.push($thunk\_address$)"$)$;
```

2. The target o'_i is formal in M. The address to be fetched is at $\#(dyn(P)) - i$ from the top of `g-st` of the evaluation context of M. This top is at depth k, and has been modified by the installation of the generic context of the new call. So, the right address depends on the values of k and j, and is computed by the function *fetch*:

```
$compile\_par(\{o_j \mapsto o'_i\}$,k) =
    $gen($"g-st.push(g-st.elem( $fetch(\#(dyn(P)) - i, k, j)$ ))"$)$;
```

In this paper, we do not develop the proof of correctness of code generation. That can be done by showing that for each ground expression e, the semantics of the evaluation of e is equivalent to the result of evaluation of the compiled code of e with the abstract machine presented here.

The main characteristics of this code generation is that installation of a generic context (parameters in the g-stack) is done only for context changes and not for each call of generic operators as in higher-order functional programming. For example, all the local calls inside a generic module (including recursive calls) have no overhead with respect to non generic calls. In the same way, optimization is possible if there exist several consecutive calls with the same effective generic context, or for calls of operators of imported modules if they have the same list of formal operators as the current module. Last point, this implementation has been carried out successfully for the LPG language. The compilation technique presented here can be applied to languages with generic units if the effective generic context of any module M_1 imported into a module M_2 can be related to the generic context of M_2 (by the morphism $i \circ r_1$).

7 Conclusion

We propose a model inference system to check the validity of instantiation of generic modules. This system is based upon constraints relating whole units. We think these relationships are suitable for modules, whereas other notions such as subtyping or type hierarchies are more adapted to single types. We have shown that the rules of the system are sound with respect to the algebraic semantics of the language. The LPG language allows to instantiate modules either with formal or actual parameters, or both, thus provides partial instantiation at the level of modules. It is possible to define multiple enrichments with consistent sharing of submodules, in spite of renaming possibilities, thanks to a canonical form for types and operators of imported modules. From a practical point of view, all instances of a generic module share the same code, which is interesting for prototyping, specially for highly generic programs with a lot of code reuse.

References

1. *Reference Manual of the Programming Language Ada.* ANSI/MIL-STD 1815A, 1983.
2. M. V. Aponte. Extending record typing to type parametric modules with sharing. In *20^{th} Symposium on Principles of Programming Languages*, 1993.
3. G. Bernot and M. Bidoit. Proving correctness of algebraically specified software: Modularity and observability issues. In *Proceedings of AMAST'91.* Springer-Verlag, 1991.
4. D. Bert and R. Echahed. Design and implementation of a generic, logic and functional programming language. In *Proceedings of ESOP'86*, number 213 in LNCS, pages 119–132. Springer-Verlag, 1986.
5. D. Bert et al. Reference manual of the specification language LPG. Technical Report 59, LIFIA, mars 1990. Anonymous ftp at `imag.fr`, in `/pub/SCOP/LPG/NewSun4/man_lpg.dvi`.
6. M. Bidoit. The stratified loose approach: A generalization of initial and loose semantics. Technical Report 402, Université d'Orsay, France, 1988.
7. C. Choppy. About the "correctness" and "adequacy" of PLUSS specifications. In *Recent Trends in Data Type Specifications*, number 785 in LNCS, pages 128–143. Springer-Verlag, 1992.
8. H. Ehrig and B. Mahr. *Fundamentals of algebraic specification 1. Equations and initial semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1985.
9. H. Ehrig and B. Mahr. *Fundamentals of algebraic specification 2. Module Specifications and Constraints*, volume 21 of *EATCS Monographs on Theoretical Computer Science.* Springer-Verlag, 1990.
10. K. Futatsugi, J.A. Goguen, J.-P. Jouannaud, and J. Meseguer. Principles of OBJ2. In *Proc. Principles of Programming Languages*, pages 52–66, 1985.
11. M.-C. Gaudel. A first introduction to PLUSS. Technical report, Université d'Orsay, France, 1984.
12. J.A. Goguen, C. Kirchner, H. Kirchner, A. Mégrelis, J. Meseguer, and T. Winkler. An introduction to OBJ3. In *Proceedings of the 1^{st} International Workshop on Conditional Term Rewriting Systems*, number 308 in LNCS, pages 258–263. Springer-Verlag, 1987.
13. S. B. Lippman. *C++ Primer.* Addison-Wesley, 1992.
14. F. Nourani. On induction for programming logic. *EATCS Bulletin*, 13:51–63, 1981.
15. J.C. Reynaud. Sémantique de LPG. Research Report 651 I IMAG, LIFIA, mars 1987.

This article was processed using the LaTeX macro package with LLNCS style