Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE DELLA INFORMAZIONE

PISA

An "Executable" Impredicative Semantics for Ada Configuration

A. Bucci, P. Inverardi, S. Martini

Nota Interna IEI B4–48

Ottobre 1990

An "Executable" Impredicative Semantics for Ada Configuration

A. Bucci[𝄢] , *P. Inverardi*[𝄞] , *S. Martini*[𝄢]

[𝄞] Istituto di Elaborazione dell'Informazione-CNR, Pisa

[𝄢] Dipartimento di Informatica, Università di Pisa

Internal Report IEI-CNR B4-48 October 1990

" Unde aliud est modulari, aliud bene modulari" [1]

We present a translation of Ada configuration constructs, in a higher order, impredicatively typed, functional language (HOTFUL) with subtypes. Aim of this work is to provide an expressive executable semantics for Ada configuration constructs, and to verify the suitability of the chosen HOTFUL for such a task. We address, in particular, not only the description of single modular units, but the practicability of the approach when dealing with the development of a whole complex system. After giving the detailed rules for such a translation, we compare our approach with what could be obtained by choosing a different typed language as "target", namely the predicative type system of Standard ML.

Introduction

The use of Higher Order Typed Functional Languages (HOTFULs) for the description of configuration is well documented in the literature, one of the first attempts being the language Pebble [Lampson & Burstall '89] (but well circulated since '83). Subsequently, [Mitchell & Plotkin '85, Cardelli '85, Cardelli & Wegner '85] [MacQueen 85 86] and others clarified many concepts and notations, thus casting the foundations for more elaborated projects. Main purpose of these works, however, was more on putting in evidence the suitability of the various typing systems in modelling single configuration constructs rather than on addressing the description of real modular languages in all their complexity. Their results showed that the description of real configuration problems in term of HOTFULs was *a priori* possible, leaving room for the application of these formal tools to *existing* languages. In this respect, the approach taken in [Inverardi, Martini, Montangero '89] was a first attempt to verify the expressiveness of HOTFULs

[1] S. Augustinus, "De Musica", I, II, 2.

```
compunit A = pack
             let T be private type;
             let f be function (x : T) return int;
             end;
```

whose body could be:

```
compunit A1 = body of A is
              T = int
              f = function (x : T) return int is x;
              end;
```

Modules can import other modules (seeing only their interface) by using a **with** clause. The following is the interface of a generic module (parametric on an integer value) importing the module A:

```
compunit B = with A;
             generic
                x : int
             pack
             let S be private type;
             let g be function ( c : TC_i(x)) return A.T
             end;
```

(Adina has an enumerable number of dependent type constructors TC_i , that return a type and take an integer argument. They are the generalization of dynamic *array* costructors.)
An instantiation of B, giving rise to a third compilation unit, could be:

```
compunit C = with B; pack new B(5) end;
```

Main tool for the modelling of Adina is the formal system $\lambda\omega_{Tp:Tp}$, fully defined in the next section. As for now, it can be thought of as an extension of a typed lambda calculus where a very general form of abstraction on types is allowed, together with type constructors for *labelled tuples*. For instance, the type

(T:Tp, x:T, f:int—>T)

is the type of the labelled triples whose elements are a type A, an element of A, and a function that for an integer input, returns a value in A. For A≡bool, one such triple can be:

(T=bool, x=true, f=fun(n:int) false)

The starting point of our modelling is the use of tuple types (which are a generalization of existential types) to describe modules as abstract data types [Mitchell&Plotkin 85]. Shortly, and for fixing the notation, the interface of a module like:

```
compunit AT = pack
              let R be private type;
              let c : R;
              let makeR be function (x:int) return R;
```

```
    let op be function (t:R) return R
end;
```

is modelled by the type:

AT : Tp = (R:Tp, c:R, makeR : int —>R, op : R—>R)

(Tp stands for the collection of all types). Elements of this type are exactly the implementations of the abstract type. For instance, an implementation of AT that realizes R with pairs of integers and where op is just the "swap" of the two components, can be represented by the following term:

IntImp = <R = int×int, c = (0,0), makeR = fun(n:int)(n,n), op=fun(p:R) (snd(p), fst(p)) >

A different implementation is the following:

BoolImp = <R = bool×bool, c = (true,true), makeR = fun(n:bool)(n,n),
op=fun(p:R) (snd(p), fst(p))> .

For both, the type system will derive the type assignment IntImp : AT and BoolImp : AT.

Modules are then represented as abstract types, therefore adhering to some extent to the principle :

(•) An interface corresponds to a tuple type and any implementation is an object of the type corresponding to its interface.

The principle founds itself on the obvious ability to model basic modules as tuple types, and we use it only so far. To assume, further, that subordinate and generic modules should be thought as objects of the same kind has unpleasant consequences in some circumstances. Indeed, the language designer well knows that the latter notions hide much more complicated and dynamic concepts than the basic modules do. The main troubles come out when one tries to build complex hierachies of modules, where it could become difficult to express the sharing of some implementation, and, furthermore, a dependency of a module on another can be lost [MacQueen 86]. Subordinate and generic modules will then be described as more complex objects, in particular as objects that in their implemention or interface part (or both, as in case of parametricity) exhibit a functional behavior.

The translation in $\lambda^{\omega}_{Tp:Tp}$ of a (basic) *package declaration* (that is of a basic module) does not pose any problem and follows the paradigm (•) above.

Module A above, therefore, will be translated as (the notation [A] is used informally in this section for "the translation of A"):

[A] = (T : Tp , f : T→ int)

and its body A1 will result in an object of type [A]:

[A1] = (T=int, f=fun(x:int)x) : [A]

Let us now consider a *subordinate module*:

```
compunit  E = with A;
              pack
              let S be private type;
              let g be function (m : A.T) return S
              end;
```

Here a dependency of E on the (known) module A is expressed. Our choice is to express this dependency by translating this interface as a single tuple type whose first field has to be an object of type [A]. We have:

[E] = (y : [A] , S : Tp , g : y.T → S)

In this way we require that an implementation of a subordinate also contains an implementation of all the modules it depends on. However, a body for E can be defined before a body for A is available. For this reason, the translation of a body of E will be a function over implementations of A. For instance, the translation of the following body of E:

```
compunit  E1  = with A;
                body of E is
                S = bool;
                g = function (m : A.T) return S
                    is true
                end;
```

will be:

[E1] = fun(a:[A]) (y=a, S=bool, g=fun(m:y.T)true).

As for *generics*, they are translated as functions of their generic parameters. In this way it is possible to instantiate both the specification and the body. Furthermore it is possible to model the instantiation of a generic interface without any reference to the actual implementations.
The translation of the interface of the generic module:

```
compunit G = generic
             v : int
             pack
             let Q be private type;
             let f be function (u:TC_i(v)) return Q
             end;
```

will be

[G] = fun (v : int)(Q : Tp , f : [TC_i] (v) → Q) : int —> Tp

A body for G:

```
compunit G1 = generic
              v : int
              body of G is
              Q = bool;
              f = function (u:TC_i(v)) return Q
                  is true
              end;
```

will be translated as the following function over integer values:

[G1] = fun(v:int)(Q=bool, f=fun(u : [TC$_i$](v))true)

The type system will derive for this term

[G1] : All(v:int) [G]v

(All(x:A)B is the type of all the functions that for an argument a:A give back a result of type B[a/x]); note how the expression for the type of [G1] expresses the relation between a generic body and its specification.

So far so good. The problems come when we start *mixing generic and subordinate modules*. When this happens, indeed, these two forms of dependency interact in a subtle way, expecially when bodies are concerned. As an example of the issues involved here, let us consider the following case:

```
compunit K = with E;
             let proj be
                 function (u:E.S) return int
             end;

compunit J = with E;
             with K;
             generic
               x : E.S
             pack
             let foo be
             function (u:TC_j(K.proj(x))) return E.S
             end;
```

If we now suppose b to be a value in E.S, we can define a new module by instantiating J:

```
compunit M = with J;
             pack new J(b)
             end;
```

The translation of K, being a subordinate, is easy:

[K] = (e:[E], proj:e.S→int).

Translating J, instead, is not as trivial. Indeed, we must express the dependency of it on E and K, but still be able to instantiate it without having to explicitly give implementations for them. Our choice is to translate J as:

[J] = fun(e:[E], k:[K]) (y=e, z=k, F=fun(x:y.S)(foo : [TC$_j$](z.proj(x))))

and M as:

[M] = (u:[E], v:[K], ([J]uv).F (b))

Note how the translation expresses, in the fact that [M]:Tp, that the (full) instantiation of a subordinate generic is a plain subordinate module. It should be obvious that the more are the levels of dependency — and the more the interactions between subordination and parametricity — the

more the translation will become complicated. The full rules of section 4 will take care of the general case.

2 $\lambda^{\omega}_{Tp:Tp}$

The system $\lambda^{\omega}_{Tp:Tp}$ is an impredicative, higher order extension of the typed lambda-calculus with a type of all types, dependent universal ($\forall$) and existential ($\exists$) types (the latter viewed as labelled tuples), and a subtype relation between types. The literature is full of similar systems. In particular, $\lambda^{\omega}_{Tp:Tp}$ is derived from [Cardelli 86] with the addition of generalized tuples and subtypes à la Bounded Fun [Cardelli & Wegner 85].

The formal system consists of four sets of rules (context, typing, subtyping and conversion rules), given by multiple combined induction. Each set defines a specific judgement, namely, that "a context Γ is well formed" (Γ OK), "in context Γ term a has type A" (Γ |- a : A), "in context Γ type A is subtype of type B" (Γ |- B $\supseteq$A), "in context Γ, a and b are equal terms of type A" (Γ |- a =b : A).

It is convenient to start the description of the system with the grammar for the *raw* (or pre-) *terms*:

M ::= x | Tp | All(x:M) M | MM | fun(x:M) M | (x:M, ..., x:M) | (x=M, ..., x=M)| M.x

The rules below will pick up between the raw terms the *legal terms* (just terms from now on). The notion of free and bound variables are defined as usual (x is bound in fun(x:M) M, All(x:M) M, and all the x's are bound in (x:M, ..., x:M) and (x=M, ..., x=M)). However, that we do not allow alpha-conversion on tuples, consistently with our choice to adopt a selection by name mechanism for tuple elimination. For the same reason, we stipulate that the only occurrences of y in a.y are the occurrences of y in a (thus, for instance, y does not occur in x.y for x a variable).

Substitution is then defined in the usual way, taking care that no free variable is captured in the process. We use the notation M[N/x] for the result of substituting the term N for the free occurrences of x in M. We write $M[N_1/x_1] \ldots M[N_n/x_n]$ for $((M[N_1/x_1]) \ldots)[N_n/x_n]$.

Context rules Contexts, ranged over by Γ, are ordered, finite lists of pairs x:M built up accordingly to the following rules.

Notation: If $\Gamma \equiv x_1:M_1, \ldots, x_n:M_n$ we set

$Dom(\Gamma) = \{x_1, \ldots, x_n\}$

$\Gamma(x_i) = M_i$

formation $\qquad \varnothing$ OK

introduction $\qquad \dfrac{\Gamma \text{ OK} \qquad \Gamma \text{ |- } A : T}{\Gamma, x:A \text{ OK}} \quad x \notin Dom(\Gamma)$

elimination

$$\frac{\Gamma(x) = A \qquad \Gamma \text{ OK}}{\Gamma \vdash x : A}$$

In the following rules we always suppose the premise: Γ OK

Typing rules

Formation-Tp

$$\frac{\Gamma \text{ OK}}{\Gamma \vdash Tp:Tp}$$

Formation-All

$$\frac{\Gamma, x:A \vdash B : Tp}{\Gamma \vdash All(x:A)B : Tp}$$

Introduction-All

$$\frac{\Gamma, x:A \vdash b : B}{\Gamma \vdash fun(x:A)b : All(x:A)B}$$

$x \notin FV(\Gamma(y))$ for all $y \in FV(b)$

Elimination-All

$$\frac{\Gamma \vdash b : All(x:A)B \qquad \Gamma \vdash a : A}{\Gamma \vdash ba : B[a/x]}$$

Formation-$\exists$

$$\frac{\begin{array}{c} x_n \notin dom(\Gamma, x_1:A_1, \ldots, x_{n-1}:A_{n-1}) \\ \Gamma \vdash A_1 : Tp \quad \ldots \quad \ldots \quad \Gamma, x_1:A_1,\ldots,x_{n-1}:A_{n-1} \vdash A_n : Tp \end{array}}{\Gamma \vdash (x_1:A_1,\ldots,x_n:A_n) : Tp}$$

Introduction-$\exists$

$$\frac{\begin{array}{c} \Gamma \vdash (x_1: A_1,\ldots,x_n: A_n) : Tp \\ \Gamma \vdash a_1:A_1 \\ \ldots \\ \Gamma,x_1:A_1,\ldots, x_{n-1}:A_{n-1}[a_{n-2}/x_{n-2}]\ldots[a_1/x_1] \vdash a_n : A_n[a_{n-1}/x_{n-1}]\ldots[a_1/x_1] \\ x_n \notin FV(a_n) \end{array}}{\Gamma \vdash (x_1= a_1,\ldots\ldots\ldots,x_n= a_n) : (x_1:A_1,\ldots\ldots\ldots,x_n:A_n)}$$

Elimination-$\exists$

$$\frac{\Gamma \vdash a : (x_1:A_1,\ldots,x_n:A_n)}{\Gamma \vdash a.x_k : A_k[a.x_{k-1}/x_{k-1}]\ldots[a.x_1/x_1]}$$

Subtyping rules:

Reflexivity
$$\frac{\Gamma \vdash A : Tp}{\Gamma \vdash A \supseteq A}$$

Sub-All
$$\frac{\Gamma \vdash A \supseteq A' \qquad \Gamma, x{:}A \vdash B' \supseteq B}{\Gamma \vdash All(x{:}A')B' \supseteq All(x{:}A)B}$$

Sub-∃
$$\frac{\Gamma \vdash B_1 \supseteq A_1 \quad \Gamma, x_1{:}B_1 \vdash B_2 \supseteq A_2 \quad \ldots \quad \ldots \quad \Gamma, x_1{:}B_1,\ldots,x_{n-1}{:}B_{n-1} \vdash B_n \supseteq A_n \quad \Gamma,x_1{:}A_1,\ldots,x_i{:}A_i \vdash A : Tp \quad y \notin dom(\Gamma,x_1{:}A_1,\ldots,x_n{:}A_n) \quad 0\le i\le n}{\Gamma \vdash (x_1{:}B_1,\ldots,x_n{:}B_n) \supseteq (x_1{:}A_1,\ldots,x_i{:}A_i, y{:}A, x_{i+1}{:}A_{i+1},\ldots,x_n{:}A_n)}$$

Transitivity
$$\frac{\Gamma \vdash B \supseteq A \qquad \Gamma \vdash A \supseteq C}{\Gamma \vdash B \supseteq C}$$

Conversion
$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash A = B}{\Gamma \vdash a : B}$$

Subsumption
$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash B \supseteq A}{\Gamma \vdash a : B}$$

Conversion rules:

(Eq_β)
$$\frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (fun(x{:}A)b)a = b[a/x] : B[a/x]}$$

(Eq_π)
$$\frac{\Gamma \vdash a_1{:}A_1 \qquad 1\le k \le n \qquad \Gamma, x_1{:}A_1,\ldots, x_{n-1}{:}A_{n-1}[a_{n-2}/x_{n-2}]\ldots[a_1/x_1] \vdash a_n : A_n[a_{n-1}/x_{n-1}]\ldots[a_1/x_1]}{\Gamma \vdash (x_1 = a_1,\ldots,x_n = a_n).x_k = a_k[a_{k-1}/x_{k-1}]\ldots[a_1/x_1] : A_k[a_{k-1}/x_{k-1}]\ldots[a_1/x_1]}$$

Plus rules for making = a congruence.

Notation

(i) $fun(x_1{:}A_1,\ldots,x_n{:}A_n)b \equiv fun(x_1{:}A_1)\ldots fun(x_n{:}A_n)\, b$

(ii) $b(a_1,\ldots,a_n) \equiv ((ba_1)\ldots)a_n$

(iii) (*tupling*) In the following we will need to syntactically manipulate terms and types. It will be convenient to have an implicit notation for the "syntactical merging" of two (or more) tuple-types (and two or more tuples). We stipulate, for this purpose, that if $T \equiv (x_1{:}T_1, x_2{:}T_2, \ldots, x_n{:}T_n)$ and $S \equiv (y_1{:}S_1, \ldots, y_m{:}S_m)$ are two tuples types, then the expression $(y_1{:}S_1, \ldots, y_i{:}S_i,\ T,\ y_{i+1}{:}S_{i+1}, \ldots, y_m{:}S_m)$ — which is not a legal term (actually it is not even a raw term!) — denotes the type

$(y_1{:}\ S_1, \ldots, y_i{:}\ S_i,\ x_1{:}\ T_1,\ x_2{:}\ T_2, \ldots,\ x_n{:}\ T_n,\ y_{i+1}{:}\ S_{i+1}, \ldots, y_m{:}\ S_m)$.

Note that this type is *not* the same as $(y_1{:}\ S_1, \ldots, y_i{:}\ S_i,\ z{:}\ T,\ y_{i+1}{:}\ S_{i+1}, \ldots, y_m{:}\ S_m)$ where an additional level of "structure" has been introduced and which is a perfectly legal term by itself.

A similar convention will be used for tuples. A different way of describing these conventions is to say that the fields of tuple type (of a tuple) can be freely grouped with parenthesis.

Discussion and properties

This is not the place for an account of the syntactical and semantical properties of $\lambda^{\omega}_{Tp:Tp}$. The interested reader can see [Amadio & Longo 86] for a discussion of a similar language. We limit ourselves to a few comments on the above rules and to some general remarks.

1. Tuple types are a generalization of the more usual existential types $\exists(x{:}A)B$ (which we would write $(x{:}A,\ y{:}B)$, for a fresh variable y). In particular, our tuples corresponds to *strong* existentials (or strong sums), that is to existentials for which also a first projection is definable in the system. The tuple notation (especially name selection, rule (Elimination-$\exists$)) is handier for the purpose we aim at, although the rules are perhaps more cumbersome. Rule (Introduction-$\exists$), in particular, appears of towering complexity. It basically expresses that in a tuple $(x_1 = a_1, \ldots, x_n = a_n)$, in a given component a_i we can use all the identifiers $x_1, \ldots, x_{i-1}$ as abbreviations for (and with the type of) $a_1, \ldots, a_{i-1}$.

2. The subtyping relation, defined in a structural way à la Bounded Fun [Cardelli & Wegner 85], is introduced essentially for dealing with bodies defining more components than their interfaces. This is the main role of rule (Sub-$\exists$).

3. The constant Tp is the type of all types (rule (Formation-Tp)). It is well known (Girard's paradox) that this entails the logical inconsistency of the system (that is we cannot soundly interpret types as propositions and elements as proofs) and the existence of non-terminating terms. The conversion relation "=" is thus undecidable.

4. Since the typing relation ":" is defined by induction on equality (rule (Conversion)), also the typing relation is undecidable.

5. The reader can wonder, in view of 3 and 4, why choosing Tp:Tp if this property is actually never used in the translation below. The reason is that without Tp:Tp we would have got only a heavier system without avoiding 3 and 4! In fact, strong existentials (our tuples) are enough, in the context of a suitably powerful system, to derive Girard's paradox [Coquand 86]. As a

consequence, without Tp:Tp the typing relation would still be undecidable, while on the other hand the presentation of the system would have to be given by stratifying the terms in "values", "types" and "kinds", duplicating most of the rules (e.g. [Amadio & Longo 86]).

3 Adina

Let us introduce the subset of the Ada language, Adina, we will deal with. Differently from Ada, Adina's compilation units are packages and generics only, subprograms are functions and they can occur only inside another compilation unit, i.e. subprograms are not compilation unit. The main purpose to have subprograms as compilation units in Ada is that they can play the role of *main*, during the configuration of a program. Such issue is not relevant for our purposes since we are mainly interested in modelling the *real* Ada modular constructs, namely the package and the generic unit. For the same reason only declarations of private types, constants and functions are allowed in a package specification. Furthermore, the private part of a private type declaration appears in the body of the package and not as in Ada in the specification part, thus obtaining a real hiding of the implementation.

As regards Adina scope and visibility rules, they are the same of Ada, except that use clauses have not been introduced.

We allow the existence of more bodies in correspondence of the same module specification. Like in Ada, they can have a declarative part which extends the one defined in their interface specification. Bodies are conventionally denoted by composing the name of the correspondent specification with an index denoting the order in which they have been produced.

In Adina it is possible to declare a package inside another package while it is forbidden to declare a compunit package inside another one. In fact, in order to model a module inside another module we should express it as a subunit. In this case, the translation would be quite cumbersone; the same argument applies on the restriction Adina imposes on generics, i.e. generic units can occurr only at top level. Thus we decided to postpone these problems to section 4.2, when we show how to go from Adina to Ada.

In order to simplify the translation, we assume, as a certain number of Ada compiler do, that a generic instantiation occur only after all its bodies have been defined. In section 4.2, we will see how it is possible to relax even this constraint. For the same purpose there is no possibility, for a body, to rename the with clause of its parent unit.

Let us now introduce the complete syntax of Adina.

Adina Syntax :

< library > ::=	**compunit** < ide > = < definiens > ; < library > |
< definiens > ::=	< with clause > ; < definiens > | < simple definiens >
< simple definiens > ::=	< package declaration > | < package body > | < generic declaration >
< with clause > ::=	**with** < ide >

< package declaration > ::= **pack** < basic declarative item > **end** |
pack new < ide > (< actual parameters list >) **end**

< generic declaration > ::= **generic** < parameter declaration >
pack < basic declarative item > **end**

< basic declarative item > ::= < basic declaration> ; <basic declarative item> | < basic declaration >

< basic declaration > ::= **let** < ide > **be** < type > |
< function specification > |
let < ide > **be private type** |
let < ide > **be pack** < basic declarative item > **end**

< package body > ::= **body of** < ide > **is** < definition list > **end**

< definition list > ::= < definition > ; < definition list > | < definition >

< definition > ::= < ide > = < type> | < ide > = < function body > | < ide > = < expr > |
< ide > = < package body >

< function specification > ::= **function** (< parameter specification >) **return** <type>

< function body > ::= < function specification > **is** < expr > **end**

< actual parameters list > ::= < expr > , < actual parameters list > |
< type > , < actual parameters list > |
< expr > | < type >

< expr > ::= < ide > | < path > | < expr > (< expr >) | <algebraic expr>

<algebraic expr> := --*usual algebraic operations*

< parameter declaration > ::= < parameter > ; < parameter declaration > | < parameter >

< parameter > ::= < ide > : < type > | < ide > **is private type** |
< ide > : < function specification >

< type > ::= int | bool | < path > | < ide > | < type constructor > (< number >)

< type constructor > ::= TC_1 | TC_2 | .....| TC_k

< path > ::= < ide > . < ide > | < ide > . < path >

< parameter specification >::= < ide > : < type > , < parameter specification > | < ide > : < type >

4 The translation

We assume that Adina programs are syntactically correct according to the above grammar and that names of compilation units and names inside a package are unique. Once this is guaranteed, a failure in the translation (no rule is applicable or a term resulting from the translation does not typecheck) means that the source Adina program was incorrect with respect to the configuration semantics. In this way, the configuration task is completely described in term of the (independently defined) typing relation, thus meeting the definitional goal we mentioned in section 2.
Given a library, for every compilation unit the corresponding translation rule is compositionally applied, following the SOS style. At each step the result of the translation is recorded in an environment until the whole library has been translated.

4.1 The translation for Adina

Let us introduce a number of auxiliary definitions and notations:

- L, M, N, R range on $\lambda^{\omega}_{Tp:Tp}$ expressions,
- `a, b, A, B, T` range on the Adina identifier set ,
- x, y, z, w, t, a, b, A, B, T range on the set of $\lambda^{\omega}_{Tp:Tp}$ variables,
- V, W, Z range on Adina constructs.
- *Constraint list*: are lists of couples (A, (x,N)) where x is a variable and N is a term s.t. |- N : Tp. These lists are used during the translation of subordinate modules, to record the related *with clauses*. The intended meaning is that in the current environment (see below) there should be an association binding A to the type N. Inside the module being translated, any reference to A will be translated as a reference to x, fresh variable of type N.

∅ = empty constraint list, C ranges on constraint lists; C[A≈(y,M)] denotes the extension of the constraint list C with the new association A≈(y,M). CC' is the concatenation of C and C'; obviously C∅ ≡ C.

- *Parameter lists*: are lists of couples, (a, N), where |- N : Tp. They are used when translating generics to record their formal parameters. P ranges on parameter lists.

∅ = empty parameter list, P[a≈M] denotes the extension of P with the new association a≈M.

- *Environments*: are lists of couples (A, (M, C, P)) , E ranges on environments.

∅ = empty environment, E[A≈(M,C,P)] denotes the extension of the environment E with the new association A≈(M,C,P).

We will make use of three translation functions. The first one,{ }, will be used in the translation of libraries, the second one,{{ }}, in the translation of definiens and the third one,[ ], will be used to translate the other Adina constructs; their use will be clear in the following. Let Term be the set of legal terms of $\lambda^{\omega}_{Tp:Tp}$; the "type" of the translation functions is:

{ } : library → Environment → Environment

{{ }} : definiens → Environment × ConstraintList → Term* × ConstraintList × ParameterList

[ ] : AdinaTerm → Environment × ConstraintList → Term

A key point in the translation is whether or not a module is subordinate. In the translation this information is given by the constraint list, C. During the translation of a definiens, indeed, all the **with** clauses are accumulated in C, and when the translation of a package is attempted, if C = ∅ then the module is not subordinate. A similar role is played by the list of parameters P, P = ∅ being the case when a module is not a generic one. This explains the duplication (and the complication) of rules 3, 5 and (especially) 6.

Library

1 $\{\ \}_E = E$

2 $$\frac{\{\{< \text{definiens} >\}\}_{E,\varnothing} = N,M_1,\cdots M_n,C,P \qquad n \geq 0}{\{\texttt{compunit}\ A\ =\ < \text{definiens} >;< \text{system} >\}_E = \{< \text{system} >\}_{E[A\approx(N,C,P)][A_1\approx(M_1,C,P)]\vdots[A_n\approx(M_n,C,P)]}}$$

We note that, in the above rule, n will always be 0 except when the definiens is a generic instantiation. In that case we instantiate both the generic interface and its bodies and in the environment we record all this information. (This will be clearer, of course, when we see the translation for generic units and their instantiations, rules 6.)

The following rules present the translation of with clauses and the translation of packages, both basic and subordinate. Note that, in 3 below, Z could be a package, a generic, a generic instantiation, or a body, possibly preceded by one or more with clauses.

With clauses

3 (basic) $$\frac{[\ A\approx(M,C',\varnothing)\ ] \in E}{\{\{\texttt{with}\ A;\ Z\ \}\}_{E,C} = \{\{Z\ \}\}_{E,C[A\approx(y,M)]}} \qquad y \text{ fresh}$$

3 (gen) $$\frac{[\ A\approx(M,C',P)\ ] \in E \qquad P \neq \varnothing}{\{\{\texttt{with}\ A;\ Z\ \}\}_{E,C} = \{\{Z\ \}\}_{E,C}}$$

Rules 3 (basic) and 3 (gen) are used in the translation of subordinate modules (package and generics) and of generic instantiation (we recall that, in Adina, generic instantiations can occur only at top level). Rules 3 (basic) applies when an imported module is a basic one (that is its parameter list P is empty). In this case the translation of the subordinate module Z takes place by using the constraint list C extended with the new association "name of module"≈(y,M). On the other hand, if the imported module A is a generic one (its parameter list P is not empty, rule 3 (gen)) then the translation of the subordinate module Z uses the same constraint list C of the translation of its with clauses. This is because, when a generic unit is imported, the only available operation is the

instantiation, and since an instantiated module is an ordinary one, we think that it is not necessary to model it as a subordinate module.

Module declaration

4 (comp-unit)

$$[\ V_1\ ]_{E,C} = M_1 \text{.....} [\ V_h\ ]_{E,C} = M_h \qquad C=[B_1 \approx (y_1,N_1)] \text{.....} [B_k \approx (y_k,N_k)]$$

$$\{\{\texttt{pack let } a_1 \texttt{ be } V_1;\ldots;\texttt{let } a_h \texttt{ be } V_h \texttt{ end}\}\}_{E,C} =$$
$$(y_1:N_1,\ldots,y_k:N_k,a_1:M_1,\ldots,a_h:M_h),C,\varnothing$$

4 (program-unit)

$$[V_1]_{E,C}=M_1 \text{.....} [V_h]_{E,C}=M_h$$

$$[\texttt{pack let } a_1 \texttt{ be } V_1;\ldots;\texttt{let } a_h \texttt{ be } V_h \texttt{ end}]_{E,C} = (a_1:M_1,\ldots,a_h:M_h)$$

Rule 4 (comp-unit) is applied to a package that is a compunit — so it can be either a basic module (k=0) or a subordinate one. When this rule is applied to a subordinate module it returns a tuple type in which all the dependencies from the imported modules are made explicit in the first bindings of the type.

Rule 4 (program-unit) is analogous to rule 4 (comp-unit) except that it is not the translation of a compunit, so it does return neither the constraint list nor the parameter list. It is necessary for the translation of a generic declaration and for dealing with packages defined inside other packages.

Generic declaration

5 (gen basic) $\qquad [V_1]_{E,\varnothing} = M_1 \text{.....} [V_k]_{E,\varnothing} = M_k \qquad [W]_{E,\varnothing} = N$

$$\{\{\texttt{generic } b_1 : V_1;\ldots;b_k : V_k;\ W\ \}\}_{E,\varnothing} = \text{fun}(b_1:M_1,\ldots,b_k:M_k)N,\varnothing,[b_1 \approx M_1]\ldots[b_k \approx M_k]$$

5 (gen sub) $\qquad [V_1]_{E,C} = M_1 \text{.....} [V_k]_{E,C} = M_k \qquad [W]_{E,C} = N$

$$C \equiv [A_1 \approx (y_1,N_1)] \ldots [A_k \approx (y_k,N_k)] \qquad k \geq 1$$

$$\{\{\texttt{generic } b_1 : V_1;\ldots;b_k : V_k;\ W\ \}\}_{E,C} = \text{fun}(x_1:N_1,\ldots,x_k:N_k)(y_1=x_1,\ldots,y_k=x_k,$$
$$F= \text{fun}(b_1:M_1,\ldots,b_k:M_k)N),\ C,\ [b_1 \approx M_1]\ldots[b_k \approx M_k]$$
$$x_1,\ldots,x_k \text{ fresh}$$

Note, first of all, that in any application of this rule W = **pack...end** and therefore rule 4 (program-unit) will be used for [W]. Rule 5 (gen basic) is the formalization of the intuition about generics we discussed in section 2. Some more comments are needed for rule 5 (gen sub), dealing with the translation of subordinate generics and that, at a first glance, seems unnecessarily complicated. The naive translation of a package like

```
with A;
generic   p:P;
W;
```

is, by analogy to rule 5 (gen basic) (like in section 2 we informally use the notation [A] for the translation of A) :

fun(p:[P])(a:[A], [W]).

The problem with this approach is that it does not permit the use of the imported module A in the *parameter part* of the generics. In other words, if P is A.T, for some type T declared in A (a situation perfectly legal in Ada and in Adina), there is no way in our attempted translation to maintain this dependency, since the binding a:[A] occurs only *after* the binding p:[P]. The only possible solution is to anticipate the binding for a in a "fun" context, thus giving (in a second "attempted translation") fun(a:[A])fun(p:[P])[W]. This is a sound translation; note, however, that "a" is not a fresh variable, but has to be recovered from the list C as the variable associated to the name A. This seemed somehow obscure, and the solution has been the one given in the rule, by which the example above become

fun(x:[A])(x=a, F=fun(p:[P])[W])

where x is fresh, a comes from C, and the "side condition" on the coincidence of the two bindings is explicitly stated in "x=a".

Generic instantiation

In order to give the rule for the instantiation of a generic, it is convenient to introduce some more notation. Let C and C' be two constraint lists. We will write C'\C for the list obtained from C' by removing all pairs whose first component appears in C. That is, since compilation units have unique identifiers, we remove [B≈(x,L)] from C' if there exists [A≈(y,M)] in C such that A≡B. Note that this implies also L≡M, while x and y, introduced as fresh variables during the translation of with clauses, will be different. We will need the following *selection function* :

$$\sigma_{C'\setminus C}(x) = \begin{cases} x & \text{if } [B\approx(x,L)]\in C'\setminus C \\ y & \text{if } [B\approx(x,L)]\in C' \text{ and } [B\approx(y,L)]\in C \end{cases}$$

We will use σ for $\sigma_{C'\setminus C}$ when the constraint lists are clear from the context. The following two rules deal with the instantiation of a simple generic unit and a subordinate generic unit, respectively.

6 (simple) $[A \approx (M,C',P)] \in E \quad [A_1 \approx (R_1,C^1,P^1)] \in E \ldots\ldots\ldots [A_n \approx (R_n, C^n,P^n)] \in E$

$[V_1]_{E,C} = N_1 \cdots\cdots\cdots [V_h]_{E,C} = N_h$

$C \equiv [A_1 \approx (y_1,M_1)] \ldots\ldots [A_k \approx (y_k,M_k)] \qquad k \geq 0$

$C' \equiv \varnothing$

$\{\{$**pack new** $A(V_1,\ldots,V_h)$ **end**$\}\}_{E,C} = (\, y_1:M_1,\ldots\ldots,y_k:M_k,\ MN_1\ldots\ldots N_h\,),$
inst-gen-body$(R_1,C,C^1,N_1,\ldots\ldots,N_h),$
$\ldots\ldots,$
inst-gen-body$(R_n,C,C^n,N_1,\ldots\ldots,N_h),$
$C\,,\varnothing$

6 (sub) $[A \approx (M,C',P)] \in E \quad [A_1 \approx (R_1,C^1,P^1)] \in E \ldots\ldots\ldots [A_n \approx (R_n, C^n,P^n)] \in E$

$[V_1]_{E,C} = N_1 \cdots\cdots\cdots [V_h]_{E,C} = N_h$

$C \equiv [A_1 \approx (y_1,M_1)] \ldots\ldots [A_k \approx (y_k,M_k)] \qquad k \geq 0$

$C' \equiv [B_1 \approx (x_1,L_1)] \ldots\ldots [B_s \approx (x_s,L_s)] \qquad s \geq 1$

$C' \backslash C \equiv [B_{i_1} \approx (x_{i_1},L_{i_1})] \ldots\ldots [B_{i_r} \approx (x_{i_r},L_{i_r})]$

$\{\{$**pack new** $A(V_1,\ldots,V_h)$ **end**$\}\}_{E,C} = (\, y_1:M_1,\ldots\ldots,y_k:M_k,$
$x_{i_1}:L_{i_1},\ldots\ldots,x_{i_r}:L_{i_r},$
$M\sigma(x_1)\ldots\sigma(x_s).FN_1\ldots\ldots N_h\,),$
inst-gen-body$(R_1,C,C^1,N_1,\ldots\ldots,N_h),$
$\ldots\ldots,$
inst-gen-body$(R_n,C,C^n,N_1,\ldots\ldots,N_h),$
$C, \varnothing$

Note how in "$M\sigma(x_{i_1})\ldots\sigma(x_{i_r}).FN_1\ldots\ldots N_h$" the notation introduced for tuples is exploited. The clauses for the function **inst-gen-body** will be given below, after the rule dealing with bodies. Rule 6 (simple) is self explanatory in light of the considerations made in section 2. We discuss the meaning of rule 6 (sub) and of function $\sigma_{C' \backslash C}$ by means of the following example:

```
compunit A = pack
     let T be private type;
     let makeT be function (n : int) return T
     end;
```

[A] = (T : Tp , makeT : int → T)
by rule 4 (basic);

```
compunit B = with A;
```

```
generic
        v : A.T
pack
        let S be private type;
        let g be function (w : A.T) return int;
        let f be function (u : TC_i(g(v))) return S
        end;
```

[B]= fun (x : [A])(y = x , F = fun (v : y.T)(S : Tp , g : y.T→ int, f : [TC_i](g(v)) → S))
by rule 5 (gen sub);

```
compunit D = with A;
             with B;
             pack new B(A.makeT(3))
             end;
```

[D] = (z : [A], S : Tp , g : z.T→ int, f : [TC_i](g(z.makeT(3))) → S)
by rule 6 (sub) and conversion.

The crux is that D depends from A in a double way, namely for the with clause in its definition and for the with clause in the definition of B. In [D] we have made explicit this dependency only once. In fact, if we take all the couples of the constraint lists C and C' (without using the difference of the two, C'\C, and the corresponding function $\sigma_{C'\setminus C}$) the translation would be:
[D] = (z : [A], y : [A], S : Tp , g : y.T→ int, f : [TC_i](g(z.makeT(3))) → S)
where there is no way of inferring — or imposing — that y and z have to be the same implementation of A.

Bodies

7(basic)	$[A\approx(M,C',P')]\in E \quad [V_1]_{E,CC'} = L_1 \cdots\cdots [V_j]_{E,CC'} = L_j$ $CC'\equiv \varnothing \qquad P'\equiv \varnothing$
{{body of A is $b_1 = V_1; \ldots ; b_j = V_j$ end}}$_{E,C}$	$= (b_1 = L_1,\ldots,b_j = L_j),\varnothing,\varnothing$
7(sub)	$[A\approx(M,C',P')]\in E \quad [V_1]_{E,CC'} = L_1 \cdots\cdots [V_j]_{E,CC'} = L_j$ $CC'\equiv[A_1\approx(y_1,M_1)]\ldots\ldots[A_k\approx(y_k,M_k)] \qquad P'\equiv \varnothing$
{{body of A is $b_1 = V_1; \ldots ; b_j = V_j$ end}}$_{E,C}$ =	$\text{fun}(x_1: M_1,\ldots,x_k: M_k)(y_1= x_1,\ldots,y_k= x_k,b_1 = L_1,\ldots,b_j = L_j),CC',\varnothing$ $x_1,\ldots,x_k$ fresh

7(gen-basic) $[A \approx (M,C',P')] \in E \quad [V_1]_{E,CC'} = L_1 \cdots\cdots [V_j]_{E,CC'} = L_j$

$CC' \equiv \varnothing \quad P' \equiv [a_1 \approx N_1].....[a_h \approx N_h]$

$\{\{$**body of** A **is** $b_1 = V_1; \ldots ; b_j = V_j$ **end**$\}\}_{E,C} =$

$\text{fun}(a_1: N_1,.....,a_h: N_h)(b_1 = L_1,....,b_j = L_j),\varnothing,\varnothing$

7(gen-sub) $[A \approx (M,C',P')] \in E \quad [V_1]_{E,CC'} = L_1 \cdots\cdots [V_j]_{E,CC'} = L_j$

$CC' \equiv [A_1 \approx (y_1,M_1)].....[A_k \approx (y_k,M_k)] \quad P' \equiv [a_1 \approx N_1].....[a_h \approx N_h]$

$\{\{$**body of** A **is** $b_1 = V_1; \ldots ; b_j = V_j$ **end**$\}\}_{E,C} =$

$\text{fun}(x_1:M_1,\ldots,x_k:M_k)(y_1=x_1,\ldots,y_k=x_k,\ F=\text{fun}(a_1:N_1,.....,a_h:N_h)(b_1=L_1,....,b_j=L_j)),$

$CC',\varnothing$

$x_1,....,x_k$ fresh

7(program-unit) $[V_1]_{E,C} = L_1 \cdots\cdots [V_j]_{E,C} = L_j$

$[$**body of** A **is** $b_1 = V_1; \ldots ; b_j = V_j$ **end**$]_{E,C} = (b_1 = L_1,....,b_j = L_j)$

Rule 7(basic) deals with a body of a basic module. We have formalized its interface as a type, and therefore its implementation will be an object of the type corresponding to its interface. Rule 7(sub) refers to a subordinate package. We have decided to formalize the implementation of such a subordinate as a function on the implementations of the imported modules. In this way, the construction of the implementation is forced to follow the same modular technique its specification has been built with.

Since bodies inherit all the modules imported by their interfaces and can import further modules, the definitions of a body are translated in a constraint list resulting from both the with clauses of the body (recorded in C), and the constraint list C' (corresponding to the with clauses of its interface).

The two cases 7(gen-basic) and 7(gen-sub) refer to the body of a generic unit. Since we have modelled a generic interface as a function, it is obvious to consider also its implementation as a polymorphic function. This function has to be parametric with respect to the generic parameters — in the case of a simple generic unit (case gen-basic) — and also to the implementations of the imported modules, in the case of a subordinate generic unit.

We can now give the clauses for the function **inst-gen-body**$(M,C,C',N_1,...,N_h)$, that we used in rule 6. The definition is by cases on C and C'. The instantiation of the body of a subordinate generic unit is translated as a polymorphic function parametric with respect to the implementations of the imported modules. The instantiation of a simple generic unit will be the application of the corresponding function to the results of the translations of the actual parameters.

inst-gen-body$(M,\varnothing,\varnothing,N_1,...,N_h) = MN_1.....N_h$

inst-gen-body$(M,C,\varnothing,N_1,...,N_h) = fun(x_1:M_1,...,x_k:M_k)(y_1=x_1,...,y_k=x_k,MN_1....N_h)$

if $C \equiv [A_1 \approx (y_1,M_1)]....[A_k \approx (y_k,M_k)]$ $\quad k\geq 1$; $x_1,....,x_k$ are fresh

inst-gen-body$(M,C,C',N_1,...,N_h) = fun(z_1:M_1,...,z_k:M_k,w_{i_1}:L_{i_1},...,w_{i_r}:L_{i_r})$
$(y_1=z_1,...,y_k=z_k,x_{i_1}=w_{i_1},...,x_{i_r}=w_{i_r},$
$M\sigma(x_1)...\sigma(x_s).FN_1... N_h)$

if $C \equiv [A_1 \approx (y_1,M_1)]....[A_k \approx (y_k,M_k)]$ $\quad k\geq 0$

$C' \equiv [B_1 \approx (x_1,L_1)]....[B_s \approx (x_s,L_s)]$ $\quad s\geq 1$

$C \backslash C \equiv [B_{i_1} \approx (x_{i_1},L_{i_1})].....[B_{i_r} \approx (x_{i_r},L_{i_r})]$

$x_1,...,x_k$ and $w_{i_1},...,w_{i_r}$ are fresh

The following rules deal with last, non problematic features of Adina.

Type

8.1 $\quad [\texttt{private type}]_{E,C} = Tp$

8.2
$$\frac{A \approx (x,M) \in C}{[A.B_1.B_2....B_k.T]_{E,C} = x.B_1.B_2....B_k.T} \qquad k\geq 0$$

8.3
$$\frac{A \approx (x,M) \notin C}{[A.B_1.B_2....B_k.T]_{E,C} = A.B_1.B_2....B_k.T} \qquad k\geq 0$$

8.4 $\quad [\texttt{int}]_{E,C} = int \qquad [\texttt{bool}]_{E,C} = bool$

$[TC_i] = TC_i$

In 8.4 we have assumed the existence in $\lambda^{\omega}_{Tp:Tp}$ of enumerable terms TC_i : int→Tp.

Function declaration

9
$$\frac{[V1]_{E,C} = M1 \ ... \ [Vk]_{E,C} = Mk \qquad [W]_{E,C} = M}{}$$

$[\texttt{function } (a_1:V_1, \ldots, a_k:V_k) \texttt{ return } W]_{E,C} = (a_1:M_1,\ldots,a_k:M_k)\rightarrow M$

Expr

10.1
$$\frac{[V]_{E,C} = M \qquad [W]_{E,C} = M'}{[VW]_{E,C} = MM'}$$

10.2 $[A]_{E,C} = A$ for an identifier A

Function body

11
$$\frac{[V_1]_{E,C} = M_1 \ldots\ldots [V_k]_{E,C} = M_k \qquad [W]_{E,C} = M' \qquad [Z]_{E,C} = N}{[\texttt{function } (a_1:V_1,\ldots,a_k:V_k) \texttt{ return } W \texttt{ is } Z \texttt{ end}]_{E,C} = \text{fun } (a_1:M_1,\ldots..,a_k:M_k)N}$$

4.2 Discussing how to remove some Adina restrictions

In this section, we present how to model some of the Ada constructs that we did not introduce in Adina because they would have made the translation given in section 4.1 too cumbersome. This, in essence, may be seen as a deficiency of the expressive power of $\lambda^{\omega}{}_{Tp:Tp}$. Although it is suitable to model the main configuration constructs of Ada it is not flexible enough when considering more sophisticated notions like subunits.

In the following, we deal with subunits, generics and generic instantiations as program units and not only library units. Finally, we show how to cope with generic interface instantiations before all the generic bodies are provided.

• Starting from subunits let us see how to introduce the notion of subunit in Adina: an interface A declares a package B whose body is defined as a *stub* in the body of A;

```
compunit A = pack
let T be private type;
let f be function (x:int) return T;
...
let B be pack
let Q be private type;
...
end;
...
end;

compunit A1 = body of A is
T = TCj(5);
f = function (x:int) is A : T; A(1)= x; return A end;
B1 = body of B is separate;
...
end;


compunit B1 = body of B separate from A is
Q = int;.
...
end;
```

A suitable translation of A is the usual existential type:
A = (T :Tp , f : int → T, ..., B : (Q : Tp,...), ...) where also the interface B has been translated as an existential type.
Let us now see how to traduce the body B1. From within B1 it is possible to refer to all the elements defined in A which preceed the definition of B. Thus in B1 there can be references to T or f etc., therefore the way we translate B1 is the usual one except that the translation is not a closed term but it can contain free references. This means that its typing will be correct only if performed in a context which contains the appropriate information, that is a context in which all the free variable become bounded. Taking this approach results in a revision of the translation clauses which deal with package specification. In fact, if a package specification contains another package specification it is necessary to record that part of the context which could be used when translating the separate bodies of the nested package.

Let us now consider the translation of A1. The natural way of translating it is like a function parametric with respect to subunit bodies. In the above example it would be something like
[A1] = fun (b: [B]) (T= [TCj(5)], f= fun ..., B1=b, ...) where the translation of B, [B], has to be extracted and recorded when translating A. Now this translation can be performed only if we do not allow in A1 to refer to objects of B. Otherwise, in general, it is not possible to perform the translation of A1 independently from the body B1. To make this point clearer we use the following example:

```
A = pack
…;
B is pack T is private type end;
f is function (v: T)  return T ;
…;
end;

body of A is
…;
body of B is separate;
f = function (v: T) is ??? return ??? ;
…;
end;
```

It is clear from the example that it is not possible to translate A1 unless it is not available the body of B which defines the actual implementation of the private type T. Note that this specific problem arises in Adina but not in Ada where the actual implementation of a private type is contained in the specification with the explicit purpose of facilitating separate compilation..
Finally we mention the fact that in Ada it is possible for a subunit to refer to the extra context defined in the enclosing body. Also in this case we could model this situation by explicitely recording the extra information and then use it when translating the subunit bodies to bound its free references.

Now we can turn to the problem of the restriction about defining generics inside other compilation units. The main problem here is related to the instantiation. In the example below we show that the way we have formalized generic modules and their instantiations does not permit to model a generic inside another module because we were not able to type, in the object language, the resulting expression.
Let us consider the following example:

```
pack A is
let T be private type;
let B be generic
n : int
pack B is
f : T Æ T;
g : TC(n) Æ T
end;

with A;
pack C is new A.B(3)
end;
```

The translation of A is obvious since A is a basic module, i.e A = (T : Tp, B : All(n:int)(f: T→ T, g: TC(n) → T)).

Note that we have modelled generics as functions thus, in order to formalize B as a type in A, we had to transform the function equivalent to the translation of B in a type, and precisely in the type of the expression corresponding to its implementation.

The translation of C, following the rule 6, will be: C = (y : A, f: T → T, g: TC(3) → T).

This expression is not typable because, following the formation rule for the existential types, we would obtain that in Γ[y : A], T → T has type Tp, in other words, that T has type Tp.

But in the environment Γ, the association of T to Tp is only present into A, hence we cannot derive that T has type Tp, so we cannot find a type for A. This fact happens because we cannot express in C references to B in terms of A, exactly as it happens for the subunit bodies. Instead, when the generic units appear at the top level, we can translate their instantiations like if a module that imports a generic would inherit its with clauses. For example:

```
pack A is
let type T is private;
mkt : int → T
end;

with A;
generic
n : int
package B is
f : A.T → A.T;
v : TC(n) Æ A.T
end;

with B;
package C is new B(3) end
```

Also in this case a reference to C is in effect a reference to A, as in the above example; but now it is easily modelled. In fact, the translation of C will be: C = (y : A, f: y.T → y.T, g: TC(3) → y.T) that is typable. We have only to derive from Γ[y : A] that y.T→ y.T has type Tp, and hence that y.T : Tp, that is obviously true.

If we take an approach like the one proposed for subunits, then we can model generic instantiations as non closed terms which will correctly type check only in an context extended with the information preceeding the generic declaration. Note that in this case the exact Ada semantics for generics is modelled since a generic body has to be evaluated in the declaration environment and not in the instantiation environment.

Now let us model the generic instantiations removing the constraint, imposed in section 3, which allows the instantiation of a module to be performed only after all its bodies are defined.
In order to remove such constraint, a "new" environment is introduced; in it we can record the information necessary to instantiate the generic bodies defined after an instantiation of a generic unit.
This information is:
1) the name of the generic unit ;
2) the name of the compilation unit corresponding to the instantiated module;
3) actual parameter list;
4) constraint list of the generic interface;
5) constraint list of the instantiated interface.
This information is recorded in the new environment, when the instantiation of the generic interface occurs; then the instantiations of the bodies can take place as presented in section 4.1. As regards the bodies that were not defined yet when their interface was instantiated, we must give a new translation rule to cope also with their instantiations. This means that the translation rule for the bodies becomes more complex. It must be verified if in the new environment there exists an instantiation of the generic package whose body is going to be translated, if this is the case, the instantiation of the body has to be provided by using the information recorded in the new environment; obviously this instantiation will be analogous to that we have already seen in section 4.1 thus we refer to the corresponding rule.

At this point, it is clear that the formalization in $\lambda^{\omega}_{Tp:Tp}$ of more complex constructs of Ada, can be provided at the expense of making the translation rules less direct and less natural.
It is worth noting that in our case we have to express any dependency in an explicit way, see e.g. the translation of subunits, another difficulty arises from the fact that we allow more bodies to correspond to the same interface, which is indeed quite a natural situation when dealing with real systems development.
In this respect the capability of guaranteeing that an implementation is shared among several module is a problem that cannot be easily dealt directly in our object language. This problem will be discussed in the next section when discussing the use of SML as object language; in fact this language can very naturally treat the sharing problem.

5. Comparison with Standard ML

Standard ML is a well known functional programming language [Harper et al. 86, Harper 86]. It exhibits a polymorphic, predicative type system in which a parametric type is assigned to values. In such a system, types are not values, although the user can define them and assign them a name.

An important part of the language is the module system (originally proposed in [MacQueen 85]) that allows the incremental construction of programs by using the crucial notion of *structure*.
A structure can be seen as an encapsulated environment, where values, types or structures are assigned to names. A structure is not a value; it is a second level object (like a non polymorphic type). What can be considered to be the "type" of a structure is its *signature*, listing the identifiers bound inside the structure together with their types. It should be clear that signatures play the role of interfaces for structures, that can be thought of as their implementations.
An example of structure is the following:

```
structure S = struct
                type T = int ;
                val v:T = 0
              end;
```

Its signature will be

```
signature SS = sig
                 type T;
                 val v:T
               end;
```

Composition of structures is achieved by using *functors*, that is maps from structures to structures. An example is:

```
functor F (S:SS):SS =
      struct
        type T = S.T*S.T;
        val v = (S.v,S.v)
      end;
```

There is no notion of signature for a functor.
A peculiar feature of the ML's structuring of programs is the possibility of explicitly modelling in the type system the sharing of the same piece of information among different modules.
The functor definition, indeed, allow to express that some components of their input structures have to be the same. Since structures can be components of other structures, it is possible to guarantee, when describing the system interconnections, that a certain module has to be shared between several structures.
The module system of Standard ML is a work of rare elegance. It is deeply consistent with the other features of the language; it imposes a natural way of structuring a program; it can be described with a clear type system and a formal semantics; it allows the configuration of a program essentially in the same way it allows the writing of a single piece of code.
Our aim in this section is not, therefore, to judge the language as a configuration tool, but, instead, to verify what can be described of the Ada (Adina) mechanisms, with its peculiarities and

idiosyncrasies. Once again, we take an experimental stand, trying to better understand Ada's features by describing them in a different language.

We will informally describe the translation of the various constructs; a formal definition of the translation could be obtained along the lines of the previous section.

5.1 Basic modules

Basic modules are easily and soundly modelled by using the notions of structure and signature. The following basic interface:

```
compunit  A = pack
              let T be private type;
              let f be function (x : T) return int;
              end;
```

will be translated in ML as:

```
signature A =    sig
                   type T;
                   val f:T->int
                 end;
```

If the body of A is the following:

```
compunit  A1 = body of A is
               T = int
               f = function (x : T) return int is x;
             end;
```

its translation in ML becomes :

```
structure A1 : A = struct
                      type T = int;
                      fun f x = x
                   end;
```

Like in $\lambda^{\omega}_{Tp:Tp}$, structures can have more components than their signatures (*non-coercitive matching*), thus allowing bodies with more components than their interfaces.

An important point of the SML module system is that signatures have to be closed ("no signature may contain free references to identifiers that are neither locally declared within the signature nor pervasive system primitives", [Harper 86]).

****

A signature, S, may refer only those signatures whose structures are substructures of the structure whose interface is S; thus it is not easy to model the situation in which a module is defined within

another module. In fact, a substructure X of a structure Z, in SML, is a structure whose interface has to be explicitly imported by Z.

****

5.2 Subordinate modules

The (essential) dependency of a structure S on the structure T is expressed in SML by making T a substructure of S (that is T has to be declared inside S or they have to be both declared in the same structure. This is mandatory, for the signature closure rule, when the signature of S refers to the signature of T. The translation of Adina subordinates then can goes along the same lines of the translation in $\lambda^{\omega}_{Tp:Tp}$. All the modules imported (by with clauses) by A will be expressed as substructures of the structure for A. The following interface:

```
compunit  E = with A;
              pack
              let S be private type;
              let g be function (m : A.T) return S
              end;
```

is translated as:

```
signature E = sig
                struct a:A;
                type S;
                val g:a.T->S
              end;
```

The translation of the body of a subordinate module, instead, has to be given with more care. The point is, once again similarly to $\lambda^{\omega}_{Tp:Tp}$, whether a body for the imported is available. The right choice seems to model subordinate bodies with functors, thus delaying the need for the bodies of the imported interfaces. In this assumption, the translation of:

```
compunit  E1 = with A;
               body of E is
               S = bool;
               g = function (m : A.T) return S
                   is true
               end;
```

is the functor:

```
functor E1 (x:A):E = struct
                        structure a=x;
                        type S=bool;
                        fun g (m:a.T) = true
                     end;
```

5.3 Generic modules

Dealing with Ada generics in the context of SML is not as simple as the previous cases. Two points here make the difference. The first one is that, in Ada, we can instantiate generics interfaces and refer to the resulting modules as if they were simple modules. The second is that a value of any type can be the parameter of a generic. Both this features are problematic in the context of SML. In this language, indeed, parametricity of a structure on another is expressed by means of functors, that, we recall, are maps from structures to structures. There is no proviso for maps from *values* to structures (as needed for the second problem above) nor for maps returning a signature (as required for the first one). The reason for this lies in the very core of the SML type system. Its *predicativity*, indeed, makes impossible for a functor to return a signature, and the *absence of dependent types* makes impossible for a functor to have values as arguments (in SML there is no even dependent type constructors like Adina's `TC`s). This is consistent with the SML modular philosophy, but imposes a cumbersome translation for Adina.

First of all, we have to express the interface of a generic as composed of two signatures; a first one (`SigPar`, say) containing only the generic parameters declarations and a second one (`Sig`) defining all the components declared in the package. Since `Sig` (and not only structures of signature `Sig`) has to refer to the components of `SigPar`, it has to declare a substructure whose signature is `SigPar`. In conclusion, a generic interface is translated as if it were a pair of modules, one of which is subordinate of the other. Note, however, that for absence of dependent type constructors, the dependence of `Sig` on `SigPar` that we are able to express are very limited, amounting only to type parameters, like in the following example:

```
compunit  G = generic
                 T is private type;
              pack
                 let f be function(u : T) return T
              end;
```

which is translated as:

```
signature G_SigPar = sig
                        type T;
                     end;
signature G_Sig = sig
                     structure G_Par : G_SigPar;
                     val f : G_Par.T -> G_Par.T
                  end;
```

Note, now, that we cannot instantiate this interface by providing a specific type. Instantiation is possible only at the level of structures, where we will be able to apply a functor. A generic body can then be translated as a functor taking a structure of signature `SigPar` and returning a structure of signature `Sig`. The following body of G:

```
compunit  G1 = generic
                  T is private type;
               body of G is
                  f = ...
               end;
```

gives rise to the functor:

```
functor G1 (x_Par : G_SigPar) : G_Sig =
               struct
                   structure G_Par : G_SigPar = x_Par;
                   val f = ...
               end;
```

The instantiation is modelled by applying the functor `G1` to a structure of signature `G_SigPar`. In this way it is possible to model only the body instantiation since the instantiation of an interface would have implied to have a signature as the result of the functor application. The problem of instantiating a generic before its body is defined cannot be even stated in this framework.
Furthermore, since functors obey a *coercitive matching* rule, the result of a functor application is a structure which contains only the components defined in its signature. There is then no way to preserve an extra component, with respect to the signature, possibly present in the body.

5.4 Subordinate Generics

Apparently, the translation of subordinate generics is only a matter of assembling; the translation of an interface like (`A` is like in 5.1):

```
compunit  C =  with A;
               generic
                  S is private type;
               pack
                  f = S->A.T
               end;
```

will be the two signatures:

```
signature C_SigPar = sig
                        type S;
                     end;
signature C_Sig = sig
                     structure C_Par : C_SigPar;
                     structure a:A;
                     val f : C_Par.S -> a.T
                  end;
```

where, in the second, the **`with`** clause is represented with the insertion of a substructure of signature `A`. An implementation of `C` will be a functor which takes as input a structure of signature `C_SigPar` (corresponding thus to its parameters) and a structure of signature `A`.

What is implicit in the limitations discussed in the previous section is that there is no way of expressing a dependency like the one discussed after rules (5) in section 3, where the imported module is used in the parameter part of the generic.

5.5 Discussion

The above sections should have made manifest that the translation of Adina into SML is possible only under additional, sometimes strongly limitative, constraints. The main motivation lies in the different modular philosophy the two languages are based on. The principal role in SML is played by the structures and the signatures are only their types; on the contrary, in Ada the main modular construct is the interface, the bodies being secondary elements. On the other hand, SML is suitable for modelling the fact that pieces of information are shared among structures. The sharing problem does not explicitely arise in the Ada language, but this is mainly a problem of its supporting environment. In Ada, indeed, there is no notion of more bodies for a certain interface, i.e. references to the same interface share always the same (unique) body. The possibility of choosing different bodies is retrieved at the environment level, where a notion of "version" of a body may exist. Thus the responsability of choosing — and therefore possibly sharing — the same body in a given configuration is demanded to the linking operation. As for Adina, the situation is slightly different, since we have allowed at the language level the existence of several bodies for an interface. This extension, however, do not provide for linguistic constructs dealing with them, to be used, for instance, to specify when a certain body has to be shared. This is a direction for further work, but we explicitly note here that, under the hypothesis of such linguistic constructs, the object language $\lambda^{\omega}_{Tp:Tp}$ is not enough for dealing with them, as it is. This is essentially due to two factors: i) there do not exist in $\lambda^{\omega}_{Tp:Tp}$ concepts like "type equalizers" ; ii) the way we have modelled generic instantiation (coherently with the Ada semantics) does not require the existence of the bodies of the imported modules.

6. Conclusions

In this work we have presented a description of the Ada configuration constructs in the HOTFUL $\lambda^{\omega}_{Tp:Tp}$. As anticipated in the introduction, our work was both definitional and experimental. In this section we discuss the result of our work in the above two perspectives.

From the point of view of the definitional work, our aim was to provide a formal description of the Ada configuration constructs, which has been completely done only for the (significant) subset language Adina. The motivation related to why it has not been possible to carry out the translation for the full Ada constructs becomes a matter which can be properly discussed only when dealing with the experimental counterpart of our work. Thus as regards the definitional aspect of our work we will consider only the Adina translation. In this respect, we can certainly say that the attempt

has been successful; the translation of the single configuration constructs is quite natural and conveniently reflects the different role each constructs play in the configuration process. For example, the distinction between dependent and generic (interface) modules is well reflected in their modelling, being the former an existential type statically dependent from the imported modules, and the latter a function, dinamically dependent on its parameters, whose result is a basic module. The translation of the whole system is somewhat heavier, but also this aspect reflects the fact that the management of a program library, is not difficult in itself but requires a wide manipulation of environments to make all the "stuff" properly working.

From the experimental point of view our work is interesting on both sides, that is it gave rise to a number of considerations on both the adequacy of the used formal system and on the design of the Ada configuration constructs. In this analyses takes a role also the attempt we have done, in section 5, by using a different formal system.

Let us first discuss the main limitation to Ada we have in Adina.

In Adina it is not possible to have a declarative part in a body definition like it happens in Ada. This is direct consequences of the formal system we use. The problem is related to the fact that the declarative part of a function, in Ada, can, in principle, be separated from its body, i.e. a declaration does not imply a definition at the same point. This happens in Ada with the obvious intention of allowing mutually recursive definitions. Let us now examine what could happen if we try to model such a situation inside a body.

```
pack A
let T be private type;
let f be function (n: int) return T
end;

A1= body of A is
        T=int;
let g be function (m:int) return int;
f = fun(n:int) g(n);
g = fun(n:int) n+1
end;
```

Now the problem is how to translate A1:

Let us examine some possibility in order to illustrate the point:

[A1] = (T=int, g=int int, f = fun(n:int). g(n)

[A1] = (T=int, tg=int int, dg: tg, f = fun(n:int). dg(n), dg= fun(n:int) n+1))

This obviously cannot be written since there is a type binding, dg: tg, in a tuple object.

[A1] = (T=int, g= fun(n:int) n+1, f = fun(n:int). g(n))

In this case it is possible to express A1 because in the translation we have rejoined declaration and definition and postponed it after the definition of f. If g were defined in term of f this would not have been possible.

Thus unless we do not forbid the introduction of mutually recursive definition it is not easy to deal with the translation of a declaration in a body separately from its definition. In fact, in Adina it is possible to introduce new objects but only by means of their definition.

Another interesting point to discuss is related to the shift of information, we made in Adina, of the private type implementation in the body and not, as it is in Ada, in the interface specification. Apparently, there should not be any consequence from the semantics point of view. On the contrary as we discussed in section 4.2, this choice has an impact on our capability of modelling bodies of units which contain a subunit. Again, something that in Ada seemed to exist only for efficiency purposes, has in effect, a deeper meaning from the semantics side.

The last remark introduces the discussion about the Ada configuration constructs. As it emerged from time to time, several restrictions and tortuousities in the translation are direct consequences of the intented Ada configuration semantics.

**********Le seguenti nelle conclusioni?? **********

The above considerations again stresses the difference between the modular philosophy the two languages pursue. In our opinion it is not convenient to attempt a comparison between their expressive "power" in itself but they have to be compared in light of the particular situation one has to model.

References

ADA [1983] *Ada Reference Manual,* ANSI-MIL-STD 1815 A, January.

Amadio R., Longo G. [1986] "Type-free compiling of parametric Types", IFIP Conference *Formal description of Programming Concepts - III,* Ebberup (DK), Wirsing M. (Ed), North Holland, pp. 377- 397.

Barendregt H. [1984], *The lambda calculus; its syntax and semantics,* Revised and expanded edition, North Holland.

Burstall R. M., Lampson B. [1984] "A Kernel language for abstract data types and modules", *Symposium on Semantics of Data Types* (Kahn, MacQueen, Plotkin eds.), *Lecture notes in Computer Science* 173(1984),Springer-Verlag pp.1-50.

Cardelli L. [1986] "A polymorphic lambda-calculus with Type: Type", Preprint, Syst. Res. Center, Dig. Equip. Corp.

Cardelli L., Wegner P. [1985] "On understanding types, data abstraction, and polymorphism", *Computing Surveys,* vol 17(4) (471-522).

Curien P. L., Ghelli G. [1990] "Coherence of subsumption" Proc. CAAP'90, Lecture Notes in Computer Science 431, Springer-Verlag 1990.

Downes V. A. and Goldsack S. J. [1982] "Programming embedded systems with Ada" by Prentice Hall International, INC.

Ghelli G. [1989] "Decidability of typechecking for the language Fun" Unpublished note.

Girard J. [1972] "Interpretation fonctionelle et elimination des coupure dans l'arithmetic d'ordre superieur", These de Doctorat d'Etat, Paris.

Harper R. [1986] "Introduction to Standard ML" Laboratory for fondations of Computer Science, Comp. sc. Department University of Edinburgh. Edinburgh EH9 3JZ Great Britain.

Harper R. [1986] "Modules and Persistence in Standard ML" Laboratory for fondations of Computer Science, Comp. sc. Department University of Edinburgh. Edinburgh EH9 3JZ Great Britain.

Harper R., MacQueen D., Milner R. [1986] "Standard ML", Tech. Rep. ECS-LFCS-86-2, Edinburgh, march 1986.

Harper R., Milner R., Tofte M. [1987] "A type discipline for Program Modules" TAPSOFT '87, Pisa, *Lecture notes in Computer Science* 250(1987), pp.308-319.

Hindley R., Seldin J. (eds.) [1980] *To H. B. Curry : Essays in Combinatory Logic, Lambda calculus and formalism,* Academic Press.

Inverardi P., Martini S., Montangero C. [1989] "Is typechecking practical for system configuration?" TAPSOFT '89 Barcellona *Lecture notes in Computer Science* 352.(1989).

Inverardi P., Mazzanti F., Montangero C. [1985] "The use of Ada in the Design of Distribuited Systems", in *Ada in Use,* proceedings of *Ada Int. Conf. 1985,* Cambridge Univ. Press.

Kanellakis P. C., Mitchell J. C. [1989] "Polymorphic unification and ML typing" in 16th ACM Symposium on *Principles of Programming Languages.*, pp. 105-115.

MacQueen D. B. [1985] "Modules for Standard ML", *polymorphism* vol. II, n. 2, October.

MacQueen D. B. [1986] "Using Dependent Types to Express Modular Structure", in 13th ACM Symposium on *Principles of Programming Languages.*

Martini S. [1988] "Modelli non estensionali del polimorfismo in programmazione funzionale" PhD. Tesi, Dottorato di ricerca in Informatica, Università di Pisa, Genova, Udine.

Meyer A. R., Reinhold M. B. [1986] "Type is not a Type" *Proceedings Principles of Programming Languages.* ACM, pp. 287-295.

Mitchell J. C., Harper R. [1988] "The essence of ML" presented at the 15th ACM *Symposium on Principles of Programming languages,* San Diego, California, January.

Mitchell J. C., Plotkin G. [1985] "Abstract types have exixtential types", *Proceedings Principles of Programming Languages.* ACM.

Plotkin G. [1981] "A Structural Approach to Operational Semantics" DAIMI FN-19,

Computer Science Department, Aarhus University, Denmark, September.