Is type checking practical for system configuration?

P. Inverardi[♥], *S. Martini*[♠] *and C. Montangero*[♠]

[♠] Dipartimento di Informatica
Università di Pisa
[♥] IEI-CNR, Pisa
56100 Pisa, Italy.

The paper describes ongoing work on a facet of software specification, namely system configuration, i.e. the specification of the structure of the system and of the operations needed to build it. We want to verify the adequacy of Higher Order Typed Functional Languages (HOTFULs), like Pebble [Lampson&Burstall 88], SOL [Mitchell&Plotkin 85] and others [Cardelli 85, Cardelli&Wegner 85], to model the configuration facilities of modern languages for system programming like Ada, Chill and Modula-2: no thorough study has been done in this direction, even if the literature is full of small scale sketches, which are used to claim that such languages are indeed adequate. We are using the new configuration concepts for distributed systems introduced on top of Ada in the Cnet project [Inverardi&Mazzanti&Montangero 85, Cnet 85] as a case study, since they provide a good test bed being an enhancement of Ada advanced configuration facilities.
The main result is that checking correctness of a Cnet configuration can be reduced to type checking in a suitable HOTFUL. However, the process is not straightforward enough, so that the question in the title is still open. As a side result, requirements have been assessed for a suitable HOTFUL: definability of (generally) recursive types, availability of the type of all types and of a peculiar inheritance mechanism.

1. Introduction

In Ada, library packages and subprogram declarations *without* context clauses, and the related secondary units provide the elementary units of programming-in-the-large. They can be collectively referred to as *modules*, and are split in two parts, the interface (specification) and the implementation (body). Interfaces and implementations have been investigated in the literature thoroughly. Module generalizations play a critical role in the configuration process: *genericity* and *subordination*, are provided in Ada by generics and context clauses respectively. In both cases, new modules can be defined that depend on previously defined types, operations or even modules: Genericity allows for explicit parameterization of the interface, while subordination makes the implementation of a module a function of the implementation of another one.
A further concept is crucial in software development, namely the notion of a collection of modules sharing some kind of property, together with the ability of establishing relations among collections. We will call

such groups of modules *bubbles*, following the proposal of Cnet [Inverardi&Mazzanti&Montangero 85, Cnet 85], where this concept was introduced in order to model, at the specification level, the notion of *subsystem*. We think it useful that the application language provides the user with concepts to structure a program following a design methodology. Indeed, this was the original purpose of bubbles: They impose a structure on the Ada program library, collecting library units into sets and constraining the units that can be imported by a module within a bubble.
Bubbles were motivated by the specific configuration problem we had to face in Cnet: the specification (and the implementation driven by this specification) in Ada of the distributed structure of the system consisting of a set of "virtual nodes", each one having no visibility of the others, all sharing a unique global purely functional (i.e. without store) environment. It turned out that the features of Ada for configuration were not powerful enough to develop every component of a distributed software system as a collection of modules satisfying given constraints on the visibility of the rest of the system[1].
Note that, although proposed to solve specific problems in the design of distributed systems, the concept of bubble is more general and, when abstracting from the specific properties required to collect modules together, responds to a common need of every configuration environment. The notion of subsystem or configuration to denote collection of modules is currently present in many configuration environments (cfr. Mesa, Adele, Rationale, ...).

The goal of the paper is a formalization of all these concepts — basic, generic, subordinate modules and bubbles — in the context of higher order typed functional languages. These languages are appealing, since they deal with all the concepts we need at the same level, as *first-class objects*. That is, modules and implementations are ordinary values, and hence they can be manipulated, computed, embedded in data structures. This is a "must" for a language that is used to express configurations or system models. Moreover, they use the same control structures for programming in the small and in the large, namely functional abstraction and application. Configuring a system, linking some modules, are no longer primitive concepts: They can be described as the application of explicit functions, whose static correctness (i.e. that implementations match their interfaces) is reduced to type checking.

2. The language

The language we use is an higher order language with explicit polymorphism, inheritance, recursion (for types also) and a type of all types. Its complete definition is given in tables 1 and 2; some preliminary acquaintance with higher order polymorphic languages (like Pebble [Lampson & Burstall 88], or the more stylized formal systems based on second order lambda calculus, see [Cardelli & Wegner 85] for an introduction) is needed. We will refer also to [Mitchell & Plotkin 85] for the description of abstract data types as existential types (dependent tuples). Here is a short discussion of the features of the language; some notational conventions are also introduced.
The essential starting point is higher order typing, that is a type structure where types can be given to programs manipulating types. This can be achieved, in the simplest case, by *abstracting* a program over types: fun(T:Tp)a is a function which takes a type and returns a value (perhaps a type again, depending on the structure of a). The type of this function is All(T:Tp)A, where A is the type we can assign to a under the assumption that variable T has type Tp (which is the constant representing the collection of types). In the general case, All(x:A)B is the type of *dependent functions* (Pebble's notation is x:A→>B); its elements are

[1] In fact, modules in a bubble may share also some constraints on their internal structure, for instance due to the physical support. This issue is not addressed in this paper.

Table 1

Type assignment

	formation	introduction	elimination
valid contexts	$\varnothing$ **ok**	$\dfrac{\Gamma\textbf{ok} \quad \Gamma \vdash A:Tp \quad x \notin dom(\Gamma)}{\Gamma[x:A]\textbf{ok}}$	$\dfrac{[x:A] \in \Gamma \quad \Gamma\textbf{ok}}{\Gamma \vdash x: A}$
Tp	$\vdash Tp:Tp$		
All	$\dfrac{\Gamma[x:A] \vdash B:Tp}{\Gamma \vdash All(x:A)B:Tp}$	$\dfrac{\Gamma[x:A] \vdash b:B}{\Gamma \vdash fun(x:A)b:All(x:A)B}$	$\dfrac{\Gamma \vdash b:All(x:A)B \quad \Gamma \vdash a:A}{\Gamma \vdash ba:B[a/x]}$
$\exists$	$x_n \notin dom(\Gamma[x_1:A_1].....[x_{n-1}:A_{n-1}])$ $\dfrac{\Gamma \vdash A_1:Tp \ \ldots \ \Gamma[x_1:A_1].....[x_{n-1}:A_{n-1}] \vdash A_n:Tp}{\Gamma \vdash (x_1:A_1,.....,x_n:A_n):Tp}$	$\dfrac{\Gamma \vdash a_1:A_1 \ ..... \ \Gamma[x_1:A_1].....[x_{n-1}:A_{n-1}] \vdash a_n:A_n}{\Gamma \vdash (x_1=a_1,.....,x_n=a_n):(x_1:A_1,.....,x_n:A_n)}$	$\dfrac{\Gamma \vdash a:(x_1:A_1,....,x_n:A_n) \quad \Gamma[x_1:A_1]...[x_n:A_n] \vdash b:C[(x_1=x_1,....,x_n=x_n)/z]}{\Gamma \vdash match\ (x_1,....,x_n)=a\ in\ b\ :\ C[a/z]}$
+	$\dfrac{\Gamma \vdash A_1:Tp \ \ldots \ \Gamma \vdash A_n:Tp}{\Gamma \vdash [x_1:A_1,....,x_n:A_n]:Tp}$	$\dfrac{\Gamma \vdash a_i:A_i}{\Gamma \vdash [x_i=a_i]:[x_1:A_1,....,x_n:A_n]}$	$\dfrac{\Gamma \vdash c:[x_1:A_1,....,x_n:A_n] \quad \Gamma[y_1:A_1] \vdash b_1: B[[x_1=y_1]/z] \quad \Gamma[y_n:A_n] \vdash b_n: B[[x_n=y_n]/z]}{\Gamma \vdash case\ z=c\ of\ (x_1.y_1 =>b_1)...(x_n.y_n =>b_n) : B[c/z]}$
rec	$\dfrac{\Gamma[x:A] \vdash a:A}{\Gamma \vdash rec(x:A)a\ :\ A}$		

the functions which, taken a term m of type A, give a result of type B[m/x] (where this last notation denotes the substitution of the term m for the free occurrences of x in B). When x is not free in B, All(x:A)B is no longer a true dependent type, and we write it as A→B.

The *product* of two types A and B (A×B) is the type of the ordered pairs of terms of type A and B, respectively. We can generalize this notion by allowing the *type* of the second component of the pair to depend on the *value* of the first component. The type one obtaines in this case (which Pebble writes x:A××B) is sometimes called an existential type, for its connection with higher order logic; we adopt its generalization to arbitrary depended tuples $(x_1:A_1,...x_n:A_n)$ where the labels $x_1,...x_n$, allow also a selection by name of the different components. The term match $(x_1,...x_n)$=a in b can then be seen as a generalized (and dependent) form of projection (variables $x_1,...x_n$ are bound in b). As in the case of dependent functions, we maintain the usual notation for products, writing A×B for (x:A, y:B) when x is not free in B; in this case we will write also <a,b> for (x=a,y=b), and we define fst≡fun(a:A×B)match (x_1,x_2)=a in x_1; the definition for snd is similar. A "dot notation" for the selection of a component of a tuple will be also used: if $p:(x_1:A_1,...x_n:A_n)$, then $p.x_i$ is shorthand for match $(x_1,...x_n)$=p in x_i. Following the key idea of [Mitchell & Plotkin 85], generalized tuples and dependent functions can be used to assign a type to an implementation, and relate this type to the module interface: checking that an implementation fits an interface is then reduced to type-checking.

The other type constructor of the language is the dependent sum constructor (denoted by square brackets), which generalizes the idea of disjoint sum. The elimination rule is a generalization of the case construct; in the expression case z=c of $(x_1.y_1=>b_1)...(x_n.y_n=>b_n)$, c is the term over which we intend to perform the case, z is a fresh variable allowing the type of the whole expression to depend on c, the x-labels match the labels of the corresponding type definition and the y's are fresh variables, bound in the b's, which allow to express the dependency of the type of the result.

Rule (Tp) asserts that the collection of types is itself a type; (rec) allows the construction of generic fixpoints, thus recursive types too. This two features are needed to deal with bubbles and contrast with the formalisms based on higher order logic (like the second order lambda calculus and its more recent extension note as Calculus of Constructions [Coquand & Huet 88]), where the lack of a fixed point operator requires the introduction of the appropriate induction scheme for any recursive definition.

Inheritance, that is the availability of a notion of subtyping and its relation with type assignment as formalized by the second (trans) rule, adds flexibility to the language. We are here interested in its role in describing configuration requirements as subtyping relations in the model of bubbles. We insist on a *structural* subtyping discipline [Wand 87, Cardelli 87, Cardelli 88], where subtyping is determined only by the structure of type definitions, as opposed to the different practice of those languages where one is allowed *to force* a subtype relation explicitely.

The language has a unique flat space for names, which are introduced by the **let** construct: the notation "**let** *Identifier* : *Type* = *term*" means the introduction of the name *Identifier* for the term *term* whose type is *Type*. As an example, let us introduce the name Singleton, for the type possessing a single non divergent element, unit:

```
let Singleton:Tp = All(T:Tp)All(t:T)T
let unit: Singleton = fun(T:Tp)fun(t:T)t
```

By using the type Singleton, we can define the (polymorphic) type constructor for lists:

```
let List : Tp→Tp = fun(T:Tp)rec(L:Tp)[nil:Singleton, const:T×L]
let nil : All(T:Tp)List(T) = fun(T:Tp) [nil=unit]
```

Table 2

Conversion

β	π	κ	μ
$(\mathrm{fun}(x{:}A)b)a = b[a/x]$	$\mathrm{match}\ (x_1,\ldots,x_n)=(x_1=a_1,\ldots,x_n=a_n)\ \mathrm{in}\ b = b[a_1/x_1, \ldots, a_n/x_n]$	$\mathrm{case}\ z=[x_i=a_i]\ \mathrm{of}\ (x_1.y_1 \Rightarrow b_1)\ \ldots\ (x_n.y_n \Rightarrow b_n) = b_i[a_i/y_i]$	$\mathrm{rec}(x{:}A)a = a[(\mathrm{rec}(x{:}A)a)/x]$

Subtypes

Reflect	$\dfrac{\Gamma \vdash A{:}\mathrm{Tp}}{\Gamma \vdash A \subseteq A}$
All	$\dfrac{\Gamma \vdash A' \subseteq A \quad \Gamma[x{:}A] \vdash B \subseteq B'}{\Gamma \vdash \mathrm{All}(x{:}A)B \subseteq \mathrm{All}(x{:}A')B'}$
∃	$\dfrac{\Gamma \vdash A_1 \subseteq B_1 \quad \Gamma[x_1{:}B_1] \vdash A_2 \subseteq B_2 \quad \ldots \quad \Gamma[x_1{:}B_1]\ldots[x_{n-1}{:}B_{n-1}] \vdash A_n \subseteq B_n \quad \Gamma[x_1{:}A_1]\ldots[x_i{:}A_i] \vdash A{:}\mathrm{Tp} \quad y \notin \mathrm{dom}(\Gamma[x_1{:}A_1]\ldots[x_n{:}A_n])}{\Gamma \vdash (x_1{:}A_1,\ldots,x_i{:}A_i, y{:}A, x_{i+1}{:}A_{i+1},\ldots,x_n{:}A_n) \subseteq (x_1{:}B_1,\ldots,x_n{:}B_n)} \quad 0 \le i \le n$
+	$\dfrac{\Gamma \vdash A_1 \subseteq B_1 \quad \ldots \quad \Gamma[x_1{:}B_1]\ldots[x_{n-1}{:}B_{n-1}] \vdash A_n \subseteq B_n \quad \Gamma[x_1{:}B_1]\ldots[x_n{:}B_n] \vdash B_{n+1}{:}\mathrm{Tp} \quad \ldots \quad \Gamma[x_1{:}B_1]\ldots[x_{m-1}{:}B_{m-1}] \vdash B_m{:}\mathrm{Tp} \quad x_m \notin \mathrm{dom}(\Gamma[x_1{:}B_1]\ldots[x_{m-1}{:}B_{m-1}])}{\Gamma \vdash [x_1{:}A_1,\ldots,x_n{:}A_n] \subseteq [x_1{:}B_1,\ldots,x_n{:}B_n, x_{n+1}{:}B_{n+1},\ldots,x_m{:}B_m]}$
rec	$\dfrac{\Gamma[x{:}\mathrm{Tp}] \vdash A' \subseteq A}{\Gamma \vdash \mathrm{rec}(x{:}\mathrm{Tp})A' \subseteq \mathrm{rec}(x{:}\mathrm{Tp})A}$

Transitivity

trans	$\dfrac{\Gamma \vdash a{:}A \quad \Gamma \vdash A = B}{\Gamma \vdash a{:}B}$	$\dfrac{\Gamma \vdash a{:}A \quad \Gamma \vdash A \subseteq B}{\Gamma \vdash a{:}B}$	

```
let cons : All(T:Tp)T×List(T)→List(T) = fun(T:Tp)fun((a,li):T×List(T))[const=<a,li>]
```

Finally, we will use $[a_1,..., a_n]_T$ as a shorthand for the list of n elements of type T.

The reader will have noticed a similar background with Pebble (but with a cleaner type structure and the addition of inheritance) and the borrowings from [Cardelli 85]. A different approach to manage configuration concepts in the context of *implicit* polymorphism is the one of Standard-ML [Milner 85], where the full first class status of modules and implementations is given up, by considering them one level higher than ordinary functions [MacQueen 88][2]. As a consequence also the configuration functions (that is functions manipulating implementations) differ in level from ordinary functions (which manipulate plain values). However, since it is our belief that one should seriously attempt to program in the large with the same basic tools of programming in the small, that is to write configuration constructs in the same style of and with the same freedom allowed to standard programming, it seems crucial to maintain the first class status of interfaces and implementations, thus allowing their free manipulation. This has been the main reason for choosing the explicitly parametric polymorphism, as formalized in the extensions to second order lambda calculus. In these systems, types can be computed during evaluation, and one can parametrize not only over types, but also over *type constructors*.

3. Modelling Ada-Cnet

The approach we take to model system configuration by type checking is the following: we define a function [ ] translating Ada-Cnet programs into terms of the language in section 2; the function is defined in such a way to express the configuration requirements over the original program as type constraints. We can thus put forward the following

Definition of *configuration correctness*: An Ada-Cnet program P is correct w.r.t. Ada-Cnet configuration constraints if [P] is a sequence of well typed expressions.

It should be stressed that the translated program itself is not a definition of the binding process, but only a set of expressions whose well-typedness asserts that the original program can be linked safely.
The translation has been worked out taking into account design methodologies also (e.g. a body must be developed according to the modular structure of its specification). Motivations and comparisons with similar work in the literature are reported in [Inverardi&Martini&Montangero 88, Martini 88] extensively.
The approach relies upon the basic idea [Mitchell&Plotkin 85] of modelling the *specification* part of the ADT with an existential type, and its implementations with different elements of this type (that is pairs, or, more generally, tuples).
Modules can then be represented as abstract types, therefore adhering to some extent to the principle:

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

The principle can be obviously justified for *basic* modules, considered as a particular form of abstract data types. As for generic and subordinate modules, it seems to suggest that they should be thought of as objects of the very same kind of basic modules. However, language designers well know that genericity

[2]Technically, the full impredicative structure of the second order lambda calculus is abandoned, in favor of a predicative cumulative hierarchy, like in Martin-Löf's Intuitionistic Type Theory [Martin-Löf 73, Martin-Löf 79]; cfr [MacQueen 86].

and subordination hide much more complex dynamic concepts than simple modules. In fact, [MacQueen 86] already objects to the simple view of (•), by showing the unpleasant consequences it has when one tries to build complex hierachies of modules. Therefore we will abandon the principle whenever convenient from a modelling point of view. We will describe in the following the model (the translation) we propose for the four concepts discussed in the introduction.

Basic modules

Consistently with the discussion above and with principle (•), a package specification (basic, i.e.with no context clauses) is mapped to an existential type which explicates the interface structure (names, functions, etc.). The body is any value whose type is its interface[3]:

```
[  package spname is
   T : Type;
   f : Int -> T;
end; ] = let spname : Tp = (T:Tp, f : Int –> T)
```

```
[  package body spname is
   T is … ;
   f is … ;
   end ] = let spname_imp : spname = (T = [...], f = [...])
```

In order to model bodies of basic modules as they are defined in real languages like Ada, we need to extend this approach allowing implementations whose structure might be more complex than it results from their interface: bodies have a local declarative part besides that exported from the interface. This case will be modelled by *inheritance,* introducing the notion of subtyping. Consider a stack: the spec will be mapped into

```
let Intstack: Tp = (St:Tp, empty:St, push:Int×St —> St, pop:St—>St, top:St—>Int)
```

and a list based implementation into (recall the definitions of the terms List, nil and cons from Section 2):

```
let IntStack_List_imp: Intstack = (St=List Int, empty=nil Int,push=cons Int,pop=cdr Int,top=car Int)
```

Another implementation could be a tuple containing more components, eg a new operation g used by push: subtyping must be such that

```
(St=List Int, empty=nil Int,g=Int —> Int,push=...g...,pop=cdr Int,top=car Int): Intstack
```

holds. With respect to the usual notion of subtyping there is the need to be able to insert new components among those of the supertype, not just at the end. This is exactly the role of the rule (∃) for subtypes.

---

[3]We are using a kind of pidgin Ada, to make the translation straightforward, apart from the crucial points related to modularity issues. A number of problems should be tackled in order to have a complete translation scheme of full Ada. For instance, in this paper we consider only a functional subset of Ada, and packages exporting private types only. Moreover, we assume that the private part is in the body, as it should be logically, and disregard separate (re)compilation issues. Finally, some ingenuity may be necessary to translate references to the names exported by the packages correctly.

Generic (or polymorphic) modules

Generic modules come with a mechanism that allows their specialization. Again, the typical example is the Stack module, to be instantiated on the type of the stacked objects. Following Pebble, we translate a generic specification as a *function* which takes the generic parameters and returns the type of the instantiated body, i.e. a generic specification is a polymorphic function over the generic parameters:

```
[ generic
    T : Type;
  package spname is
    f : Int -> T;
  end; ] = let spname : All(T:Tp)Tp = fun(T:Tp) (f:Int ---> T)
```

Analogously, a generic body is a function over the parameter, which, once applied, returns an element of the type expressing the instantiated specification.

```
[ package body spname is
    f is ...T... ;
  end ] = let spname_body : All(T:Tp)(spname (T)) = fun(T:Tp) (f= [...T...])
```

So, the dependencies between Stack and Item, the type of the stacked elements, are modelled by

```
let GenStack : All(Item:Tp)Tp =
        fun(Item:Tp) (St:Tp,empty:St,push:Item×St--->St,pop:St--->St,top:St--->Int))
```

and an implementation of GenStack is a polymorphic function over Item:

```
let ListStack : All(Item:Tp)(GenStack Item) =
        fun(Item:Tp)(St=List Item,empty=nil Item,push=cons Item,pop=cdr Item,top=car Item)
```

At first glance, this approach has the drawback that a "generic module interface" is no longer a type and we can no longer characterize its implementations as the objects of that type: However, there is a tidy relationship between *instantiated* implementations and modules, as in "ListStack Int : GenStack Int".
The link between the implementation and the specification is given by the identifier GenStack in the type expression All(Item:Tp)(GenStack Item). Note that GenStack and ListStack have the same type, since the term (Genstack Item) reduces to Tp under β-conversion.

Compared to other approaches, we found that trying to maintain (•) at the level of generics resulted in involved modelling [Inverardi&Martini&Montangero 88]. Indeed, basic modules are the actual units of *programming in the large*, generics being instead tools to produce new basic modules. Therefore they must be modelled differently. Subordinate modules lie somewhat in between, as we will see immediately.

Subordinate modules

Module A is subordinate to module B if (the interface or the body of) A uses identifiers defined in (the interface of) B. In Ada, this is the case of packages with context clauses: A is subordinate to B if it

"imports" it, by the clause **with** B. Since a subordinate module is produced out of known elements, its specification is naturally translated as an existential type expression explicating the dependencies from the imported modules. In particular this dependency is conceptually different from the one of generic modules, being static rather than dynamic: such an interface cannot be instantiated; only the body of the imported module can be different in distinct implementations of A. For these reasons we differ from Pebble, where no distinction is made between genericity and subordination.

```
[ with iname;
  package spname is
     T: type;
     f : iname. S -> T;
  end; ] = let spname : Tp = (iname_imp: iname, T:Tp, f : iname_imp.S -> T)

[ package body spname is
    T is ...;
    f is ...;
  end ] = let spname_body : iname -> spname =
        fun(B: iname) (iname_imp=B, T = [...], f = [...])
```

Like in the generic case, the correspondence (•) does not hold, at least in the sense that we are not translating the body as a value of the type of its interface. However, any actual *implementation* of the subordinate module (obtained as *spname_body* (some value of type *iname)*) has type *spname.*

Generic Instantiation

```
[with genname; package giname is new genname (parameters);] =
   let giname : Tp = (F:All(p)genname(p), P:genname(parameters))
   let giname_imp : (All(p)genname(p) -> giname =
          fun(f:All(p)genname(p)) (F = f, P = F(parameters))
```

where p is the list of formal parameters of genname. For instance, consider

```
with Genstack;
package Intstack is new Genstack(Int);
```

We obtain:

```
let Intstack: Tp = (F: All(Item:Tp) Genstack (Item), P: Genstack(Int))
let Intstack_Body : (All(Item:Tp) Genstack (Item)) -> Intstack =
                        fun (f: All(Item:Tp) Genstack (Item)) (F= f, P=F(Int))
```

Interlude

Let us recall that the main goal of the translation is not the definition of the binding process of an Ada compilation, nor of the linking process, but only the definition of a set of expressions whose well-

typedness asserts that the original program satisfies the intended configuration constraints.
However, the translation has been designed so that the resulting terms can be used to express binding and linking processes: in a sense, they express the most general functionality that may be attached to a module in the library. For instance, modelling the body of a subordinate as a function, naturally expresses the fact that bindings are delayed, whenever a module body is put in the library before the bodies of the modules it imports. On the other hand, bindings are also naturally expressed as function application, when the imported bodies are supplied. A similar argument applies to generics and their use.
At the same time, care has been taken to ease the description of binding and linking strategies that take into account methodological aspects (e.g. that an implementation must have the same modular structure as its own interface). For instance, the process yielding an actual implementation of Intstack is described by the term Intstack_Body(Genstack_Body), that describes the method that has been used to derive it. This is far different from what we would have obtained by modelling the situation as a simple environment extension, as in: Intstack' : Tp = Genstack (Int). In this case, indeed, implementations of Intstack' can be obtained in a non modular way, that is without reference to the fact that Instack' is actually built from a generic.
Another issue is the following: the translation yields components that are redundant. For instance, in the example of Intstack, the implementation contains also the tool used to build it. However, the relevant components can be retrieved in the same way from the terms describing the interface and the implementation:

snd (Intstack_Body(Genstack_Body)) : snd Intstack

Redundancy may be introduced also by duplication, whenever more than one module import the same module A: a term containing A only once, whose type is a subtype of all the the importing interfaces, can be written to describe an irreduntant implementation.

Bubbles

We recall that, for the purposes of this paper, a bubble is a collection of interfaces closed with respect to the operation of importing interfaces, that is forming import lists of elements in a bubble and building with them a new interface must give an object still belonging to the same bubble.
The actual implementation [Inverardi&Mazzanti&Montangero 85] provides three *pragmas* for creating bubbles (`makebubble`), inserting a module into a bubble (`intobubble`), extending a bubble (`extend`). Examples of use of the first two operations, to create two bubbles `FOO` and `FIE` and to insert modules `ADT`, `ADText` in the first one and `A` in the second one, are given in the following:

```
makebubble (Foo);

intobubble(Foo);
package ADT is
   T : Type;
   make : Int -> T;
end;

intobubble (Foo);
with ADT;
package ADText is
   f : ADT.T ->ADT.T
end;
```

```
makebubble (Fie);

intobubble (Fie);
package A is
   S : Type;
   g : Int -> Int;
end;
```

Attempting to insert the module

```
with ADT,A;
package WRONG is
   h:ADT.T ->A.S
end;
```

into either bubble will result in an error, since neither of them provides the needed visibility.

Bubbles can extend the visibility they offer: `extend (Fie, Foo);` will allow the insertion in `FIE` of modules importing modules in `FOO`. Thus the following is correct:

```
intobubble (Fie);
with ADT, A
package OK is
   h:ADT.T ->A.S
end;
```

In order to model a bubble we would have liked to express it directly as a type, namely the type of the interfaces belonging to it. This amounts to characterize *with a type* a collection of types; unfortunately HOTFULs do not allow such a characterization: Types have type Tp and there is no way to distinguish them according to their structure. Then bubbles are described as a class of data structures where interfaces are supposed to be inserted into after a consistency check, for which it is necessary to keep track of the *import list* of every library unit which is inserted.

The requirements contained in the discussion above are modelled by the following term Bubble, which can be seen as the type of bubbles:

```
let Bubble : Tp = (B:Tp, outof : B -> Tp, into : List(B)×Tp -> B)
```

An object of this type will represent a specific bubble. Note how a specific bubble (an object of type Bubble) comes *explicitly* with the functions needed to use the objects (the interfaces, in the intended interpretation) in it. Given a bubble Bbl : Bubble; the function outof allows access to the raw interface; into provides for the consistency (visibility) check.

Type B has to be recursive. Indeed, the importing interface has to be inserted into the same bubble Bbl of the imported interfaces: that is, given an interface S importing $A_1,\ldots, A_k$, we want to define a term built out of S, $A_1,\ldots, A_k$ whose type is *the same as that of* $A_1,\ldots, A_k$, only if $A_1,\ldots, A_k$ belong to Bbl. Our choice is the following:

$< [A_1,\ldots, A_k]_{\text{Bbl.B}}, [\text{tag} = S] > : \text{Bbl.B}$

where tag is built systematically from the bubble name. Therefore, the type of the interfaces in Bbl is

$$\text{rec(T:Tp) List(T)} \times \text{[Bblname:Tp]}$$

The second component is a union (+) type because it allows to model the extend operation easily. Here is a bubble:

```
let Bbl1 : Bubble = ( B = rec(T:Tp) List(T) × [Bbl1name:Tp],
                      outof = fun(b:B) case snd(b) of Bbl1name.x => x,
                      into=fun((imp,int):List(B)×Tp)<imp,[Bbl1name=int]>)
```

To insert a basic interface, say (T:Tp, new:Int –> T), into Bbl1, with name inter1:

```
let inter1 : Bbl1.B = Bbl1.into (< nil(Bbl1.B), (T:Tp, new:Int –> T)>)
```

We can use outof to obtain the interface contained in an object (recall that = is the syntactical conversion of terms):

```
Bbl1.outof(inter1) = (T:Tp, new:Int –> T)
```

The insertion in Bbl1 of an interface importing inter1:

```
let inter2 : Bbl1.B=Bbl1.into (< [inter1], (a:Bbl1.outof(inter1), f:a.T–>a.T)>
```

Again, we can access interface inter2 by using outof:

```
Bbl1.outof(inter2) = (a:Bbl1.outof(inter1), f:a.T–>a.T) = (a:(T:Tp, new:Int –> T), f:a.T–>a.T).
```

A new bubble can be obtained by using a different label, Bbl2name, in the variant type forming B:

```
let Bbl2 : Bubble = ( B = rec(T:Tp) List(T) × [Bbl2name:Tp],
                      outof = fun(b:B) case snd(b) of Bbl2name.x => x,
                      into=fun((imp,int): List(B)×Tp)<imp,[Bbl2name=int]>)
```

The *extension* of Bbl1 by Bbl2, Bbl3 can now be defined, such that inter1 : Bbl3.B and inter2 : Bbl3.B:

```
let Bbl3 : Bubble = ( B= rec(T:Tp) List(T) × [Bbl1name:Tp, Bbl2name:Tp],
                      outof = fun(b:B) case snd(b) of Bbl1name.x => x,
                                                      Bbl2name.x => x,
                      into = fun((imp,int) : List(B)×Tp)<imp,[Bbl1name =int]>)
```

The subtyping rules prove that Bbl3.B is a supertype of both Bbl1.B and Bbl2.B: in this sense Bbl3 can then be considered as the extension of Bbl1 by Bbl2: Extension is the basic operation for subsystem composition.

We are eventually ready to complete the translation of an Ada-Cnet program, considering the Ada-Cnet pragmas makebubble, intobubble and extend, which allow to control the bubble world wrt to basic and dependent interfaces.

```
[makebubble bname] = let bname : Bubble =
                       ( B = rec(T:Tp) List(T) × [bnamelabel:Tp],
                         outof = fun(b:B) case snd(b) of bnamelabel.x => x,
                         into= fun((imp,int) : List(B)×Tp)<imp,[bnamelabel =int]>)
```

```
[intobubble bname;
  with A1,..., Ak; package spname is S end ] =
        let spname : bname.B = bname.into (< [A1,..., Ak]bname.B, S>)
```

where s is the type defined in [with A_1,..., A_k; package *spname* is S end].

```
[extend (bname1, bname2)] = let bname1 : Bubble =
   (    B=rec(T:Tp) List(T) × [bname1label:Tp, bname2label:Tp],
        outof = fun(b:B) case snd(b) of      bname1label.x => x
                                             bname2label.x => x,
        into = fun((imp,int) : List(B)×Tp)<imp,[bname1label =int]>)
```

The translation of the previous example is given in figure 1: The violation of a constraint in the package WRONG is detected as a type mismatch.

4. Conclusions

This work is a first step towards modelling the structure of software applications for a real system, namely the Cnet system, using HOTFULs.

The first lesson we learned is that, when used in real contexts, HOTFULs, despite what is claimed in most literature, are not such an easy and flexible tool to specify configuration issues. If on one side it is very natural to talk of implementations in this framework and there are evident advantages when considering the specification of concepts like those we have discussed in section 3, on the other side it is more difficult, as in the case of bubbles, to characterize sets of specifications (modules). The solution we adopted, in fact, does not appear very natural, since in order to be able to reason about sets of specifications their "symbolic" manipulation is required. Indeed, a specification has to be turned into a more complex type, to be recorded in a bubble: Such a manipulation is the purpose of function into that can be reversed by outof which in turn serves the purpose of retrieving the relevant information out of the bubble. We think that some work has to be done in type theory, in order to introduce ways to characterize types according to their structure.

The second lesson regards the better comprehension we gained of the configuration model which had been proposed in the linguistic context of Ada for the Cnet system. From this point of view the formalization helped us to review part of the semantics of bubbles which were initially viewed as objects evolving through different states, due to an implementation bias of our understanding. Furthermore, it provided a clear semantics for the Ada concepts (generics, with-clauses) including their implementations, thus making available a firm base on which all the configuration operations available in a real programming environment (e.g. recompilation, linking, version managing) can be precisely defined.

The third lesson concerns the use of type checking as a tool to check the correctness of a system under development with respect to configuration constraints. From a foundational point of view, we believe our work shows the feasibilty, under certain assumptions over the type structure, of a reduction of configuration to type checking. From a practical point of view, the impact of such an approach on the design of languages for programming in the large might be limited, unless type checking can be performed effectively. In fact, the type checking problem for our language is undecidable, for the presence of recursive types. Nevertheless, Luca Cardelli[4] has conducted extended experiences with a language

[4]Personal communication.

Example

```
makebubble (Foo);                    let Foo: Bubble =
                                     ( B = rec(T:Tp) List(T) × [Foolabel:Tp],
                                     outof = fun(b:B) case snd(b) of Foolabel.x => x,
                                     into = fun((imp,int) : List(B)×Tp)
                                     <imp,[Foolabel =int]>)
intobubble(Foo);                     let ADT : Foo.B =
package ADT is                       Foo.into ( nil(Foo.B), (T:Tp, new:Int -> T))
  T : Type;
  make : Int -> T;
end;
intobubble (Foo);                    let ADText:Foo.B =
with ADT;                            Foo.into ([ADT],
  package ADText is                  (a:Foo.outof(ADT), f:a.T ->a.T))
       f : ADT.T ->ADT.T
  end;
makebubble (Fie);                    let Fie : Bubble =
                                     ( B = rec(T:Tp) List(T) × [Fielabel:Tp],
                                     outof = fun(b:B) case snd(b) of Fielabel.x => x,
                                     into = fun((imp,int) : List(B)×Tp)
                                     <imp,[Fielabel =int]>)
intobubble (Fie);                    let A : Fie.B =
package A is                         Fie.into ( nil(Fie.B), (S:Tp, g:Int -> Int))
  S : Type;
  g : Int -> Int;
end;
intobubble (Fie);                    let WRONG : Fie.B =
with ADT,A;                          Fie.into ( [ADT, A] ,
  package WRONG is                   (b:Fie.outof(ADT),
       h:ADT.T ->A.S                 a:Fie.outof(A), h:b.T -> a.S))
  end;
-- type checking fails, since ADT is invisible from inside bubble Fie, i.e has not type Fie.B .
extend (Fie, Foo);                   let Fie : Bubble =
                                     ( B = rec(T:Tp) List(T) × [Fielabel:Tp, Foolabel: Tp],
                                     outof = fun(b:B) case snd(b) of Fielabel.x => x
                                     Foolabel.x => x,
                                     into = fun((imp,int) : List(B)×Tp) <imp,[Fielabel =int]>)
intobubble (Fie);                    let OK : Fie.B =
with ADT, A                          Fie.into ([ADT, A] ,
  package OK is                       (b:Fie.outof(ADT), a:Fie.outof(A),
       h:ADT.T ->A.S                 h:b.T -> a.S))
  end;
```

Figure 1.

similar to ours, reporting efficient type checking in all practical situations. Additional study and experience, however, is in order to carefully assess the features of the language with respect to the performance of the type checker.

Acknowledgements Ugo Montanari motivated this work and provided continuous encouragement. Anonymous referees made helpful comments.

References

[Cardelli 85] Cardelli L. "A polymorphic lambda-calculus with Type:Type", Preprint, Syst.Res.Center, Dig. Equip. Corp. 1985.

[Cardelli 88] Cardelli L. "Structural Subtyping and the notion of Power Type", *15n ACM symposium on Principles of Programming Languages (POPL).*

[Cardelli&Wegner 85] Cardelli L.,Wegner P. "On understanding types, data abstraction, and polymorphism", *Computing Surveys*, vol 17(4) (471-522)

[Cnet 1985] N. Lijtmaer (ed.) "Distributed Systems on Local Networks", Final project report, Pisa, June 24-28, 1985.

[Coquand & Huet 88] "The Calculus of Constructions" , *Information and Computation,* **76**, 93 (1988).

[Inverardi&Martini&Montangero 88] Inverardi P., Martini S.,Montangero C. "An assessment of system configuration by type checking", Internal Report IEI-CNR , March 1988.

[Inverardi&Mazzanti&Montangero 85] Inverardi P., Mazzanti F.,Montangero C. "The Use of Ada in the Design of Distributed Systems". *Proc. Ada Int. Conf. 1985.* In Ada in Use Cambridge Univ. Press.

[Lampson&Burstall 88] Lampson B.W., Burstall R.M "Pebble, a Kernel Language for Modules and Abstract Data Types" *Information and Computation,* **76**, 93 (1988).

[MacQueen 86] MacQueen D. "Using dependent types to express modular structure", *13tn ACM symposium on Principles of Programming Languages (POPL).*

[MacQueen 88] MacQueen D. "The implementation of modules in Standard ML", *ACM Symposium on Lisp and Functional Programming*, Snowbird, July 88.

[Martini 88] Martini S. "Non extensional models of polymorphism in functional programming" PhD thesis, Dottorato di ricerca in Informatica, Università di Pisa, Genova, Udine (1988) (in Italian).

[Martin-Löf 73] Martin-Löf M. "An intuitionistic theory of types" *Logic Colloqium 73*, Rose Shepherdson (Eds.), North-Holland 1975 (73-118).

[Martin-Löf 79] Martin-Löf M. "Constructive mathematics and computer programming" *Logic Methodology and Philosophy of Science*, North-Holland 1980 (153-175).

[Milner 85] Milner R. "The Standard ML core language" *Polymorphism Newsletter*, vol 2(2), October 1985.

[Mitchell&Plotkin 85] Mitchell J.,C., Plotkin G.,D. "Abstract types have existential type", *12tn ACM symposium on Principles of Programming Languages (POPL), 1985.*

[Reynolds 85] Reynolds J.C. "Three approaches to Type structure", *Proc. Int. Joint Conf. Theory and Practice of Soft. Devel. (TAPSOFT)*, Berlin March 1985, LNCS 185.

[Wand 87] Wand M. "Complete type inference for simple objects", *Proc. Symposium on Logic and Computer Science* (LICS), IEEE Computer Society Press.