Abstract

This paper contains an examination of the typings associated with the construction of persistent systems through the use of parametric abstract modules. Of particular interest are the class of dependent, potentially dynamic typings required as a consequence of diamond import. An example is introduced, including the diamond problem; solutions to it are shown in Fibonacci, using its module provision, and in Napier88, using more general existential types and dynamic typing. The dependent typings are highlighted in each language, and an abstract module algebra is introduced which makes such typings more explicit. The module algebra is shown to have the expressible ability of both systems, as well as the original SOL language, making it a candidate for a general language mechanism to express both module and abstract type programming paradigms.

1. Introduction

It is a matter of some debate as to whether an explicit module mechanism is required or desirable in an orthogonally persistent programming language. An original motivation for modules is to allow the separate compilation of program units, whereas some persistent systems achieve this orthogonally by the use of a general dynamic binding mechanism. Another major purpose of modules, that of providing abstract interfaces, can also be achieved through other general language mechanisms, notably that of existentially quantified data types [MP88].

In this paper the typing of module systems is examined. An example, including the “diamond import” problem, is described, and its typing shown in the Fibonacci [AGO95] and Napier88 [StA96] languages. The Fibonacci solution uses an explicit module mechanism, whereas the Napier88 solution uses general programming mechanisms: specifically those of existential types [MP88, CDM+90] and explicit dynamic typing [ACP+91, CBC+90]. The different typings are examined in detail through the introduction of a new abstract algebra designed to describe module typings. Module typing issues are essentially those of existential types, but with an extended (and sometimes necessarily dynamic) model of type equivalence.

The intention is not to provide an answer to the question of whether a module mechanism is a useful or required language mechanism, but simply to shed some light on the typing issues.

* This work should be cited as “Modules and Type Abstraction in Persistent Systems” in Persistent Object Systems: Principles and Practice, Connor R and Nettles S (Eds) Morgan Kaufmann 1997 pp 48 - 59
involved. We do however demonstrate that a single language mechanism may be used to capture the semantics of both modules and abstract data types.

2. Modules - background

It is well established that the development of large scale applications requires a mechanism to support the subdivision of the programming task into subtasks, whose implementations may be separately specified, and which may be mechanically linked together to form the application. Such mechanisms are ubiquitously referred to as module mechanisms, after early work by Parnas [Par72]. Many different mechanisms have been suggested, specified, and implemented; the common features are briefly presented.

2.1. Interface and implementation

A module is an entity containing data, metadata and programs, the use of which may be specified without reference to any realisation. This is achieved by the concepts of interface and implementation; the interface specifies what functionality is provided by a module, and the implementation gives a realisation of this. Thus the interface contains sufficient information for any use of the implementation to be defined, without dependence on the implementation itself.

The advantages of such dependency limitation are well-documented, and include the following:

- a matching of the normal refinement patterns of problem decomposition
- an understandable program structure
- the confinement of errors (both static and dynamic) to sub-units
- separate compilation of sub-units

Various relationships between interface and implementation are possible. The most flexible systems support both multiple interfaces and multiple implementations. The intuition supporting this derives from the correspondence of module instances to language values, and of module interfaces to types. Thus each interface can have any number of implementations; in a language with polymorphism, an implementation may belong to any number of interfaces.

2.2. Import and export

Applications composed of modules are structured as a set of controlled interactions:

- each module realises some functionality and exports its use to other modules
- each module, to realise its own functionality, can import from other modules

There are two kinds of importation, constant and parametric:

Constant importation occurs when the binding from imported identifiers to their instances occurs at the same time as compilation. As the linking is effectively static, the subsequent modification of a module requires re-linking (and therefore re-compilation) of all dependent modules.

Parametric importation occurs when the compilation and binding phases of module construction occur separately. A module instance is yielded as a result of a function which is applied to other instances. Module implementations maybe modified without a requirement to re-link the code of dependent modules.
2.3. Abstract modules

Importing modules may use an exporting module only through its interface. By appropriately hiding the implementation of an interface, it is possible for a module to export abstract data types: the type’s representation, as specified within the implementation, may not be seen outside the module. Thus code using the abstract module cannot refer to the concrete type, but only to its abstraction. Clearly there is an important duality between abstract modules and abstract data type packages as defined by Mitchell and Plotkin. The introduction of abstract modules gives rise to the diamond import problem.

2.4. Diamond import

The diamond problem occurs when a module depends on another abstract module through two different paths. An example is shown in Figure 1.

![Figure 1: The diamond import example](image)

ListInt is an abstract module which exports an abstract type representing a package of integer list functions. The module interface exports an abstract type identifier, hiding the list representation, and a set of functions defined over it. The other three modules are parametrically defined, and they act as follows:

Interval imports an implementation of the module ListInt and exports a function which, given two integers representing an interval, returns a list of integers corresponding to that interval.

Sum also imports an implementation of the module ListInt, and exports a function which, given a list of integers, yields the sum of its elements.

Main imports the two modules Interval and Sum and exports a function which, given two integers, returns the sum of the interval between them.

The problem is that Main imports the same abstracted type, that provided by ListInt, by two different paths. If the module system allows interfaces to have more than one implementation, it may happen that Main imports two different implementations of ListInt. These implementations, although sharing a common interface, may be defined over different representations of the abstract type. Therefore the mixing of list values deriving from the different interfaces is a potential source of unsoundness within the system and can not be allowed without further checks.

The requirement is to preserve soundness without disallowing the flexibility allowed by the combination of abstract and parametric modules. Note that if we impose any of the restrictions of static importation, single module instances or concrete typing the problem can not occur.
The typing problem, in terms of existential typings, is as follows: there exist values \( \text{interval} \) of type \( \exists i. \text{fun} \ (\text{int, int} \to i) \) and \( \text{sum} \) of type \( \exists i. \text{fun} \ (i \to \text{int}) \), and it may be possible to determine from knowledge of the system composition that \( \text{sum}(\text{interval}(x, y)) \) is a meaningful combination. In the SOL [MP88] model of existential typing there is insufficient flexibility to type this expression; in general dynamic type checking may be required at the time of the \( \text{main} \) module instantiation. We must therefore introduce sound, but potentially dynamic, type mechanisms to allow such combinations where they are appropriate. Importantly, the presence of dynamic typing at the time of system construction does not imply the possibility of dynamic type failure during system execution [DCC93].

We continue by showing codings of the above problem in the language Fibonacci, using a purpose-designed module mechanism, and in Napier88, using a general purpose abstract type mechanism. Although both achieve the same purpose, the typing of the intermediate levels is subtly different. We conclude by introducing an abstract module model which is able to subsume the semantics and typing of both models.

3. The example codings

In the module systems we describe, modules are first class values and the commands which link modules together are invocations of first class procedures. It should be noted that this model only makes sense in the context of persistent higher-order languages, and is quite different from mechanisms employed by other module systems in which the modules do not form a part of the language’s semantic domain.

System installation proceeds as follows:

1. Generator execution - linker installation phase. In this phase code which includes the definition of the generator functions is executed, making these functions available within the persistent context.

2. Linker execution - module installation phase. In this phase the generator procedures themselves are executed, resulting in module instances. These instances may in general be the input of subsequent linker procedures, modelling module import and export.

If subsequent changes to a module are required, only its generator is re-instantiated, hence producing a new linker for that module; for changes to be propagated throughout the system, all dependent linking phases must be re-executed. The basic requirement in such a system is that as many consistency checks as possible are carried out at installation time. It is therefore possible for “dynamic” checking to occur during linking phases without compromising static safety requirements of the resulting system.

Fully static module checking would mean that every consistency check takes place when the generators and their application are typechecked. Installation time checking means that every consistency check takes place by the end of the installation process. Installation only happens once (unlike general functions, which are typically compiled once and executed many times), hence, up to the point of the whole application being generated, static and dynamic type-checking are not very different.

3.1. Module ListInt

In all the examples of this paper, the existence of an orthogonal persistence mechanism is assumed. No account is taken of persistent binding and scoping issues, and the discussion concentrates solely on the typing of modules. The presence of orthogonal persistence is a requirement, as otherwise values of these types could not persist and be used as modules; however details of the persistence mechanism are by definition orthogonal to the discussion.

First the type (interface) of the \( \text{ListInt} \) module is given in the two languages:
The codings are almost identical, and serve mainly to point out the differences in the concrete syntax of the languages. The most important difference is that Napier88 adopts the SOL approach to existential types, while Fibonacci represents an existential type by a tuple type (or “module type”) where every field may either be a value with a given type or a type with a given kind (\(<:\) Any is the kind which contains any type). The two approaches have the same expressive power, as should be apparent from the example above[CL90]. Another difference is that the Fibonacci function types include parameter names, whereas Napier88 do not.

We now proceed to make an instance of the type:

```
let listInt: ListInt = 
<[
  T <: Any;
  empty : T;
  cons : Fun( v : Int; l : T ) : T;
  isEmpty : Fun( l : T ) : bool;
  first : Fun( l : T ) : Int;
  rest : Fun( l : T ) : T
]>;
```

Once again there are no technical difficulties with the coding. Napier88 automatically overloads the definition of structured types as constructor functions for those types, but type equivalence is nonetheless structural in both systems. The use of `ListInt` as a constructor is parameterised by its specialising type; here the identifier `T` is used to correspond with the Fibonacci example and is not significant except for its consistent use throughout this program fragment.

### 3.2. Module Interval

We now make a first attempt to define the module `Interval`. Although the semantics of the following code are appropriate, it will become evident that the typings, although correct, are not sufficiently flexible to support the diamond application. In both languages the result of the `createInterval` procedure is a general existential type; more information than this is required to allow its subsequent matching with the `Sum` module.
```
Let Interval =
<[
    T <: Any;
    create: Fun( left, right: Int ): T
]>;

let createInterval = fun( m: ListInt ): ( Interval ) is
<[
    Let T = m.T;
    rec let create = fun( left, right: Int ): T is
        if left>right then m.null
        else m.cons(left; create(left+1; right) )
]>;
```

The body of the Napier88 `createInterval` procedure uses an abstract `use` clause, similar in intent to the SOL `open` mechanism, to create the required constant binding from the identifier `M` to the package `m`; Fibonacci uses more sophisticated static analysis [CL90] to achieve the same effect. Both languages introduce a denotation `T` for the witness type of the imported module `m` and proceed to declare a new procedure whose implementation uses the imported interface. The result of the execution is a new procedure which generates an `Interval` module implementation, based on the input implementation, and typed as `∃i.( create : fun( int, int -> i ) )`.

This typing requires to be enhanced in some way before the module `Main` will be able to successfully combine the applications of `Interval` and `Sum`. The typings required involve dynamic dependencies, as the correct typing of `Main` depends on the actual value supplied to the `Interval` and `Sum` generator functions. Each language has a different mechanism which may be used to achieve this.

### 3.2.1. Fibonacci dependent typing

The Fibonacci system of dependent types allows the description of a function where the type of the result depends on the value of a parameter, provided that the parameter is a tuple with a type component. For example, a function which receives a tuple `m` with type

```
[ T <: Any, a: T ]
```

and extracts the `a` field from `m`, has type

```
Fun( m : [ T <: Any, a: T ] ) : m.T
```

This ability may be exploited to give a more precise type to the `createInterval` function, which statically specifies the fact that the witness of the abstract type of the created module is the same as the witness of the imported module:

```
let createInterval =
    fun( m: ListInt ): ( Interval with T = m.T ) is
    ...
```

The notation `Interval with T = U` means “the same type as the tuple type `Interval`, but substitute `EQ(U)` to the kind of the `T` field” It does not add expressive power to the system, but enhances

---

1It should be mentioned for the sake of completeness that the Napier88 syntax requires the body of a `use` clause to be void; in these examples they appear with non-void bodies for the sake of simplicity. The reason for this restriction is not connected with these examples, and each may actually be programmed by means of a variable declared in an outer scope being assigned to within the body.
readability. \( EQ(U) \) is a kind whose only element is \( U \), hence every type in this kind is equivalent to \( U \). It is important to notice that the information representing the relationship between the witness types is explicitly arranged to be present.

3.2.2. Napier88 dependent typing

The syntax of Napier88 allows the description of the required type dependency as follows:

\[
\text{use } m \text{ as } M[ T ] \text{ in begin type Interval is } \text{proc}( \text{left, right: int } \rightarrow \text{T} ) \text{ rec let create } = \text{proc}( \text{left, right: int } \rightarrow \text{T} ) \ldots
\]

The type \( \text{Interval} \) is actually a dependent type, as the meaning of the identifier \( T \) depends on the value supplied as \( m \). Unfortunately however it is not possible for the value \( \text{create} \), which is typed as \( \text{Interval} \), to be exported from the local scope with this type: it is impossible for a denotation of the type to exist outside the scope of the \text{use} clause. However Napier88 has an explicit dynamic typing mechanism, used to provide persistent binding capabilities, and this may be used to code the example. The dynamic typing respects the semantics of the value dependency\(^2\), the use of abstract types in persistent bindings being earlier noted as otherwise overly restrictive [CM88, OTC+90].

The final version of the Napier88 module generator is as follows. Although the return value is typed as \( \text{any} \), the required semantics have been achieved. In fact this dynamic typing will be factored out during the execution of the \text{Main} generator function; the typing of \( \text{Interval} \) thus represents a coarser-grain dynamic check than is required for maximum elegance, rather than an unnecessary dynamic check.

\[
\text{type Interval is any let createInterval } = \text{proc}( m : \text{ListInt } \rightarrow \text{Interval} ) \text{ use } m \text{ as } M[ T ] \text{ in begin rec let create } = \text{proc}( \text{left, right: int } \rightarrow \text{T} ) \text{ if } \text{left } > \text{right then } M( \text{Null} ) \text{ else } M( \text{cons})( \text{left, create( left+1; right ) } ) \text{ any( create } ) \text{ end end}
\]

3.3. Module Sum

The codings of the module \text{Sum} generators are included for completeness, and should not generate any issues other than those already covered in \text{Interval}:

\[
\begin{align*}
\text{type Interval is any let createInterval } = \text{proc}( m : \text{ListInt } \rightarrow \text{Interval} ) \\
\text{use } m \text{ as } M[ T ] \text{ in begin rec let create } = \text{proc}( \text{left, right: int } \rightarrow \text{T} ) \\
\text{if } \text{left } > \text{right then } M( \text{Null} ) \\
\text{else } M( \text{cons})( \text{left, create( left+1; right ) } ) \\
\text{any( create ) end end}
\end{align*}
\]

\[\text{This is true of Napier88 Releases 2.0 and afterwards; releases before this have the static model reflected in the dynamic typing semantics, and the Main generator function shown later would fail with a dynamic type error.}\]
3.4. Module Main

In this module the threads are tied together and we introduce a final generator function which takes as parameters instances of Interval and Sum modules, and applies them to each other. Sufficient value dependency has been indicated in each system to allow appropriate dynamic checking to occur at the start of the generator procedure; if the diamond dependency is incorrect, that is the implementations of IntList used in the generation of Interval and Sum are different, then the generation of the Main module will fail with an appropriate high-level type error. If the same module has been used then the generation will succeed, and the resulting module is statically safe.

The Napier88 program is shown partly as pseudo-code. The reason for this is that, as Napier88 programmers will be aware, the collecting of persistent components as above is normally achieved with the environment binding mechanism. As our intention is to discuss typing issues only, an invented form of project is used in the example to reflect the typing underlying environment projection. Napier88 programmers will be able to mentally recode all the above examples to use environments instead of type any; and the elegance of the solution is actually enhanced. Readers who are not familiar with Napier88 may read the above example as simple projection from a dynamic typing.

The diamond import problem is solved in different ways in the two systems. In Napier88 dynamic typechecking is required to resolve the value dependency between the module m0 and the dependent types of m1 and m2, and an instance of the module m0 is required as an operand to allow this check to be made. In typical use of the Napier88 system, the dynamic typechecks occur at the same time as dynamic binding checks, and the reliance on the m0 module may be made implicit. In Fibonacci everything is statically typed however the sharing constraints
must be explicitly stated in the intermediate level modules. Once again, this exactly suits the typical system construction methodologies used in the Fibonacci environment. Thus there is a tradeoff in the way that the same information is transmitted up to the time of the final module creation.

The most important point, however, is that both systems give astatically checked final module. The presence of dynamic type checking causes no problems so long as it occurs during the installation phase. The requirement to list indirectly imported modules or sharing constraints may be a serious burden on the programmer during the construction of large systems, and both Napier88 and Fibonacci methodologies may require to rely on automatic tools to insert this kind of information.

Finally we show the combination of all the module generators to produce the main program, making the second phase of module system construction as identified above:

```ocaml
let listPack = createListInt() let listPack = createListInt()
let interval = createInterval( listPack ) let interval = createInterval( listPack )
let sum = createSum( listPack ) let sum = createSum( listPack )
let main = createMain( interval, sum ) let main = createMain( listPack, interval, sum )
```

Notice again how both systems eventually generate a statically typed and sound module; the dependency checking essential to the process has been entirely factored out during the module generation process.

To finish, we examine the typings present in both languages with respect to a more abstract module algebra.

4. The abstract module algebra

In this section the fundamental typings of the different language constructs are examined in a single conceptual framework, and the different typings of the Fibonacci and Napier88 solutions are described. This is achieved by the description of abstract module concepts as a language extension.

4.1. Outline

Dependent typings are unavoidable in flexible module mechanisms where the diamond problem can occur; a major purpose of the algebra is to explicitly decouple the type dependencies from the other operations of the language. To achieve this, in outline, a new type `ModuleWitness` is introduced; values of this type can also be treated as types. This is a similar approach to the witness type mechanism proposed by Ohori et al in [OTC+90]. In this context however values of type `ModuleWitness` are used to explicitly type general values, and replace the need for the `open` mechanism. Instead the concept of a pseudo-abstract type is introduced, such as in the following example:

```ocaml
type Concrete is structure( x : int, y : proc( int -> string ) )

type FullyAbstract is Module( i )( x : i, y : proc( i -> string ) )

type PseudoAbstract is structure( x : w, y : proc( w -> string ) )
```

where `w` is a value of type `ModuleWitness`. The only operation defined on values of `Module` (fully abstract) types is their specialisation to pseudo-abstract types; values thus specialised may be used as concrete types so long as their `ModuleWitness` components may be statically deduced to be compatible.\(^3\) Thus one property of the algebra is to disseminate the burden of static dependency checking from the abstract package to its components.

---

\(^3\) Orthogonal mechanisms for achieving this are known, e.g., [CL90], and the issue is not discussed further here.
It is worth observing at this point that the intermediate modules *Sum* and *Interval* are typed as pseudo-abstract types in both languages in the above examples, and the difference in typechecking reflects a tradeoff between the time of checking and context of legal use.

4.2. **Formal Description**

For an arbitrary language the following constructs are added to the type and expression domains:

\[
\begin{align*}
T & ::= \ldots \mid \text{Module}[i](T) \mid \text{ModuleWitness} \mid E \\
E & ::= \ldots \mid \text{mkModule}(E,T) \mid \text{mWitness}(E) \mid \text{repModule}(E,T)
\end{align*}
\]

In the types domain, *Module*[i](T) is the type of an abstract module, where *i* stands for the abstract (witness) type which, in non-trivial cases, will appear free in the type signature *T*. *ModuleWitness* is the type of an explicit dependent type; values of this type also appear in the type domain, explaining the presence of *E*. Notice however that this does not necessarily give rise to undecidable static checking, relying on static analysis techniques as mentioned above.

In the expression domain, the *mkModule* function is essentially a type-widening operation: its parameters are a value and an abstract module type; it returns the same value wrapped as an abstract module. It is at this point that the value of the module witness type is calculated, and also stored as part of the abstract module value. This is the value returned by the *mWitness* operation. The *repModule* operation takes a module instance and a module witness type (described as *T*, although *E* would be equally valid) and returns the abstract value in a pseudo-abstract instantiation. This operation is only defined if the module witness type provided is compatible with that of the module instance; we introduce the \(\equiv\) operator to define this compatibility, and leave its definition until later.

The following type rules formalise the context of use of the new expressions and types. Within the type rules the syntax \(f(t)\) in place of a type variable is used to represent a type signature imposed over an arbitrary type, such that if \(T = f(x)\) then \(f(y) = \{ x \leftarrow y \}T\) (where *x* and *y* are themselves type expressions).

\[
\begin{align*}
\pi \vdash e : \text{ModuleWitness} \\
\pi \vdash e : \text{Type} \\
\pi \vdash e : f(t) \\
\pi \vdash \text{mkModule}(e, \text{Module}[i](f(i))) \quad 12 : \text{Module}[i](f(i)) \\
\pi \vdash e : \text{Module}[i](T) \\
\pi \vdash \text{mWitness}(e) : \text{ModuleWitness} \\
\pi \vdash e_1 : \text{Module}[i](f(i)) \land \pi \vdash e_2 : \text{ModuleWitness} \land \{ e_2 \} = \text{mWitness}(e_1) \\
\pi \vdash \text{repModule}(e_1, e_2) : f(e_2)
\end{align*}
\]

The last of these rules introduces two potential problems with the static typing of programs. The first is the re-typing of *e_1* as *f(e_2)*, which gives the possibility of arbitrary expressions subsequently being typed as *ModuleWitness* types as discussed above. The second problem is more serious: that is, in general, the dynamic evaluation of *e_2* is required before the rule can be applied, giving rise to dynamic checking on those occasions where the result of the \(\equiv\) operator on the meaning of *e_2* cannot be determined statically. This dynamic checking however is essential to provide the required flexibility seen in the earlier examples. In cases where dynamic checking is not required for flexibility it is possible to factor the checking out statically.

A semantics for the new constructs is defined as follows:

\[
[[\text{mkModule}(E : f(t), \text{Module}[i](f(i)))]] = \text{pair}([[E]], \text{formWitnessRep}(t))
\]
\[[m\text{Witness}(E)]\] = \text{snd}(\[[E]\])
\[[\text{repModule}(E_1, E_2)]\] = \text{fst}(\[[E_1]\])

In the semantic model, a module consists of a pair, the elements of which are a value and a witness type representation. The \text{mkModule} operation forms the pair with the value \[[E]\] and a witness type representation. The \text{mWitness} operation returns the witness type representation. The \text{repModule} operation dereferences the enclosed value from the \(E_1\) module pair and allows it to be typed over the \(E_2\) ModuleWitness type value. Notice that the type rule for this operation ensures that the value \(E_2\) is compatible (using the \(\equiv\) operator) with the witness type representation of \(E_1\).

The semantics hinges upon the meaning of the \text{formWitnessRep} and \(\equiv\) functions; in fact a family of different semantic models may be specified by using different definitions. In the current context of use, we make the following definitions, where \text{unique()} generates a unique value and \(\approx\) signifies equality over such values:

\[
\text{formWitnessRep}(t) = \begin{cases} t & \text{if } t : \text{ModuleWitness} \\ \text{unique()} & \text{otherwise} \end{cases}
\]

\[
t_1 \equiv t_2 = \begin{cases} t_1 \approx t_2 & \text{if } t_1 : \text{ModuleWitness} \end{cases}
\]

The observation that the new language mechanisms capture typeabstraction in SOL, without introducing dynamic typing, is straightforward and not elaborated further. We proceed to show how the examples of module construction given earlier above maybe described.

### 4.3. Modelling module mechanisms

There is only a single typing of the base module ListInt, which is an abstracted record type containing both the data and functional components of the module:

```ocaml
type ListInt is module[ T ]( structure( empty : T; ... as before ... ) )
```

```ocaml
let aListInt = struct
(    empty = ListInt( Empty : nil ),
       .. as before ...
)
```

```ocaml
let listInt = mkModule( aListInt, ListInt )
```

However there are two possible typings of the intermediate modules Interval and Sum. The first typing for Sum is:

```ocaml
Module[ Rep ]( proc( Rep -> int ) )
```

and the other possibility is:

```ocaml
proc( Rep -> int ), where Rep = mWitness( aListInt )
```

These different types correspond to the terms fully abstract and pseudo-abstract introduced above. The essence of the different solutions is in the binding of the local abstract type variable Rep, which may be either closed in the fully abstract definition, opened and resolved as some value dependency in the pseudo-abstract definition. In the fully abstract model a new type closure is created as follows:
Given the above definition of formWitnessRep the call to mkModule in the text gives the same model as (Sum with T = m.T) in the original Fibonacci definition, by ensuring the placement of sufficient dependent type information in the abstract module. However the dynamic semantics of this placement would allow the information to be (dynamically) recaptured in a different static environment, whereas the Fibonacci model is completely static.

The pseudo-abstract equivalent is as follows:

```plaintext
let createSum = proc( listImpl : ListInt -> proc( mWitness( listImpl ) -> int ) )
{
    let T = mWitness( listImpl )
    let m = repModule( listImpl, T )

    rec let sumList = proc( list : T -> int );
        if m.isEmpty( list )
            then 0
        else m.first( list ) + sumList( m.rest( list ) )

    mkModule( sumList, Sum )
}
```

Notice the explicit value dependency which appears in the procedure header. This general form of typing certainly removes the possibility of fully decidable static typing. Notice, however, that this particular procedure can be statically checked.

The definitions of Interval add no new typings and are not elaborated. The procedure createMain following in the fully abstract model is as follows:

```plaintext
let createMain = proc( m1: Interval, m2: Sum -> proc( int, int -> int ) );
{
    let W1 = mWitness( m1 )
    let X1 = repModule( m1, W1 )
    let X2 = repModule( m2, W1 )

    proc( x, y : int -> int );
        X2( X1( x, y ) )
}
```

If this code is compared with the Fibonacci solution above, it may be observed that the condition (m2: Sum with T = m1.T) in the original Fibonacci program is replaced by the call repModule( m2, W1 ), which will succeed only if the modules share the same witness representation. However although the test is the same, it occurs in the procedure body rather than as part of the typing, giving a more general procedure type (notice that this is of particular
importance in a system with first-class executable forms) at the expense of dynamic typechecking; however, as noted, this dynamic checking occurs before the end of the application construction phase.

With the pseudo-abstract model the definition is as follows:

```plaintext
let createMain = proc( mo: ListInt;
  m1: proc( int, int -> mWitness( m0 ) );
  m2: proc( mWitness( m0 ) -> int )
    -> proc( int, int -> int ) );

proc( x, y: int -> int ); m2( m1(x, y) )
```

Notice that it is possible to statically type this procedure. (We do not provide a general mechanism, only the observation that it is possible in this instance.) In fact it may be possible to statically type the whole creation of an instance of Main, as for example

```plaintext
let listPack = createListInt()
let interval = createInterval( listPack )
let sum = createSum( listPack )
let main = createMain( listPack , interval , sum )
```

Notice however that the static typing of the createMain procedure requires the base module m0 to be explicitly provided as a parameter, unlike the fullyabstract coding where it is not required. Given parametric polymorphism this can be avoided by defining createMain polymorphically and specialising the call appropriately from outside the context, as follows:

```plaintext
let createMain = proc[ t ]( m1: proc( int, int -> t );
  m2: proc( t -> int )
    -> proc( int, int -> int ) );

proc( x, y: int -> int ); m2( m1(x, y) )
```

```plaintext
let listPack = createListInt()
let interval = createInterval( listPack )
let sum = createSum( listPack )
let t = mWitness( listPack )
let main = createMain[ t ]( interval , sum )
```

The important difference here is that, instead of the specification of all module dependencies needing to be explicit, a single dependent type value may be used to check all diamond dependencies within a module system. Once again, it is possible to statically typecheck the above code, which may be interesting to compare with the Fibonacci system solution above.

The fact that these particular examples can be statically typed does not imply that pseudo-abstract forms are to be preferred in general; in fact their typechecking is much harder to automate and undisciplined use, if statically allowed, must generate implicit dynamic checks to preserve soundness. Fully abstract forms, on the other hand, while possibly requiring more dynamic checking in some cases, have a more understandable typechecking regime and generate only explicit dynamic checks.

5. Conclusions

We have investigated the typing of module mechanisms in a persistent environment. The significance of orthogonal persistence is that the aspects of longevity and binding often associated with such mechanisms may be assumed to be orthogonal to their typing. The general-purpose abstract type mechanism of Napier88 was shown to go some way in the provision of module requirements, although some general restrictions in the language syntax mean that precise type definitions are sometimes lacking compared to equivalent solutions given
in the Fibonacci module syntax. A more detailed examination of the typing issues involved has resulted in the description of an abstract module mechanism which gives more flexibility, at the cost of less certain static type checking. The new mechanism has been shown to capture both of the languages investigated, and may be a candidate for a combined abstract type / module mechanism.

6. Acknowledgements
The moduleWitness type is a development of a dependent typing invented by Atsushi Ohori, and ideas generated from discussion with him have probably found their way into this paper. The implementation of the dependent witness types in Napier88 was carried out by Quintin Cutts. Thanks are due also for helpful comments from the anonymous referees. The authors are grateful for financial assistance from the Esprit III FIDE2 project (BRA 6309), which allowed Paolo Manghi to visit Scotland for an extended period. Richard Connor is supported by EPSRC Advanced Fellowship B/91/AF/1921.

7. References
[StA96] The St Andrews Persistence Research Group, http://www-fide.dcs.st-and.ac.uk