Persistence and Type Abstraction Revisited*

Atsushi Ohori† Ivan Tabkha‡ Richard Connor§ Paul Philbrow¶

Abstract

Existential data types as described by Mitchell and Plotkin are an appealing model for data abstraction. However, they are somewhat restrictive within a persistent type system, as values created by an existentially defined package may not be stored and retrieved in different static contexts. This is because values of an abstract type are type equivalent only in the scope of the same package opening. This allows packages to be used as first-class values, but precludes the kind of dynamic type check associated with the retrieval of a value from the persistent store.

This paper refines Cardelli and MacQueen's analysis of the interaction between persistence and abstract data types and proposes a new adaptation of these types. The resulting type system loses none of the desirable static properties of Mitchell and Plotkin's model, but is more flexible dynamically and allows a value of a witness type to be stored in one context and retrieved in another. An intuitive explanation of the new model is given, along with formal type checking rules and an implementation strategy.

1 Introduction

The usefulness of abstract data types is widely recognized. They serve as a powerful tool for information hiding and modular programming. One way of achieving data abstraction is to hide implementation details by "packing" the concrete structure of its representation and restricting its use within a special open statement, which creates a scope where a particular implementation of an abstract data type is available through a fixed set of explicitly specified interface functions. Mitchell and Plotkin gave [MP88] a formal model for this intuitively appealing idea and showed that this feature can be uniformly integrated in the Girard-Reynolds [Gir72, Rey74] polymorphic lambda calculus. They interpreted a "package" implementing an abstract data type as a *value* having an *existential type*. They called these values *data algebras*. For persistent programming systems, the most attractive feature of Mitchell and Plotkin's model is that such data algebras may be first-class values. This allows the programmer the advantages of orthogonal persistence [Coc83]. This contrasts with the other models of abstract types such as those implemented in Standard ML [HMT88], CLU [LAB+81] and Ada [Ada80]. Mitchell and Plotkin achieved this flexibility by treating the implementation type of a data algebra (sometimes called a *witness type*) as a type which exists only within the scope of the special open statement for the data algebra. This mechanism, however, conflicts with persistence – a mechanism to allow values to survive independently of any particular program activation.

To see the problem, let us examine a simple program in Mitchell and Plotkin's SOL language (with a slight change of its syntax). In the following examples, we assume that $PointADT(\sigma)$ is shorthand for the following record type:

$$\texttt{[create:(real*real)} \to \sigma\texttt{, x\_val:}\sigma \to \texttt{real, y\_val:}\sigma \to \texttt{real, move:}\sigma \to \texttt{(real*real)} \to \sigma\texttt{]}$$

*Appeared in **Implementing Persistent Object Bases, Principles and Practice, Proc. 4th International Workshop on Persistent Object Stores**, Morgan-Kaufmann Publishing, pages 141–153, 1990.

†Supported by a British Royal Society Research Fellowship. On leave from OKI Electric Industry, Co., Japan.

‡Supported by ESPRIT II Basic Research Action 3070 – FIDE, and by the Committee of Vice-Chancellors and Principals of the Universities of the United Kingdom.

§Supported by ESPRIT II Basic Research Action 3070 – FIDE, and by SERC GRF 02953.

¶Supported by ESPRIT II Basic Research Action 3070 – FIDE.

[1] Authors' addresses. Ohori, Tabkha and Philbrow: Department of Computing Science, University of Glasgow, Glasgow, G12 8QQ, Scotland. Connor: Department of Computational Science, University of St Andrews, St Andrews, KY16 9SS, Scotland.

To create a data algebra, say for manipulating "points", we use the primitive **pack** which "packs" the concrete implementation type:

```
cart_point = pack real*real in
                   [create = λx:(real*real).x,
                    x_val = λp:(real*real) → real.p.1,
                    y_val = λp:(real*real) → real.p.2,
                    move = λp:real*real.λd:(real*real).(p.1 + d.1,p.2 + d.2)]
             to ∃t.PointADT(t)
             end
```

where $(\exists t.PointADT(t))$ is an existential type, which intuitively says that there is some type called `t` whose actual structure is hidden. To use this, the programmer must explicitly open it as:

```
open cart_point as point with
   P:PointADT(point)
in
   ...
   P.create(0.0,0.0)
   ...
end
```

Intuitively, this statement creates a "new" type called `point` and binds the variable `P` of type $PointADT(\texttt{point})$ to the value of the data algebra `cart_point` (i.e. a record of four functions). Therefore, in the body of this statement, `P.create(0.0,0.0)` is a type correct expression yielding a value of type `point`. One crucial assumption of this model is that the abstract type created in this open statement has no "real existence" and has no connection to any other type in the system. In the type system, this is enforced by treating the type name (`point` in the example) as a free type variable which will never be instantiated. Without the context of an **open** statement, such free type variables have no independent meaning. This is the mechanism that enables them to treat a data algebra as a first-class value. However, this feature also causes difficulty when we want to make a value having a witness type persist across different program activations, since for a value to persist it must have its own type and meaning independent of any program session. In the example, it is therefore meaningless to export the point `P.create(0.0,0.0)`.

Perhaps Cardelli and MacQueen were the first to consider this problem and sketched a solution [CM88]. In their *abstract witness model* witness types have "real" existence associated with the package that implements them and values of those types are type compatible whenever they originate from the same package. Since packages can exist independently of any program session, this may be used to solve the problem we have just stated. However, as they pointed out, the solution seems to cause a difficulty in static type checking when we want to continue to treat packages as first-class values. They suggested the possibility of an *ad hoc* method to resolve this conflict. It is, however, not entirely obvious that one can develop a sound method without constructing a proper formalism. The goal of this paper is to analyze the problem in the context of a typed functional calculus and to propose one such method. Our proposal is based on Mitchell and Plotkin's model but it could also be adapted to Cardelli and MacQueen's model with a moderate amount of change.

The idea behind our approach is to allow programmers to treat witness types as values so that they may be stored and subsequently "reactivated". Intuitively, this allows the programmer to re-establish necessary bindings from a particular opening of a package. To develop a sound type discipline based on this idea, we are going to introduce a mechanism to treat witness types as values in a restricted way. The major technical contribution of this paper is to show that such a mechanism can be safely introduced in Mitchell and Plotkin's formal calculus.

The rest of this paper is organized as follows. Section 2 sets the basis of our discussion by defining a model of persistence in a paradigm of functional calculus through extending Mitchell and Plotkin's SOL language with persistence. We then identify the problem. In Section 3, we analyze Cardelli and MacQueen's proposal within the above framework. Our major contribution is presented in Section 4 where we present the mechanism to reconcile persistence and abstract data types, first by examples and then by giving a formal system of types. An implementation is described in Section 5.

2 Persistence and Mitchell and Plotkin's Abstract Data Types

2.1 A Model of Persistence in a Typed Functional Language

In a functional calculus, one way to model persistence is to introduce two primitives, denoted here by **get** and **put**, to retrieve and store a value from a store. Of course, a practical persistent programming language should provide more convenient ways to store and retrieve data. However, our main concern here is a type discipline for persistence and this simple model provides us with sufficient detail to analyze the interaction between the type system and persistence. We believe that, with moderate effort, our analysis of a type discipline for persistence can be applied to languages with more elaborate persistence mechanisms.

In order to integrate these primitives in a typed functional calculus, we need to assign them a type and introduce them as term constructors in the calculus. This requires us to type objects that are in a persistent store. As pointed out in [CM88, ACPP89], an *infinite union type* is appropriate. This is the type whose values are a pair consisting of a value and (the description of) its type. A type system with such an infinite union is developed in [ACPP89], where the type is called `dynamic`. Here we call it `any` to avoid confusion with the dynamic types we shall introduce later. `env` in Napier88 [MBCD89] and `pntr` in PS-algol [Per87] can be regarded as infinite union types, the former combined with other features for easy manipulation of persistent stores. Any values can be injected into `any` by:

inject$(M{:}\sigma)$: `any`

and subsequently exported into a persistent store. Possible operations on `any` are dynamic inspection of the actual types and projection to a type. In [ACPP89] these are combined in a single `typecase` statement which achieves flexible manipulation of values of type `any`. For our purpose, however, it is enough to assume the following simple projection function:

project(M,σ) : σ

which projects a value of type `any` onto the specified type σ if the actual type of the value is the same as σ, otherwise it causes a run time exception.

If the type component of a value of type `any` has its own meaning independent of any program session, then we can safely store the value and later retrieve and project back to the type. For this purpose, the type component of a value of `any` is restricted to be a *closed type*, i.e. a type that does not contain type variables. This condition corresponds to the intuition that storable values are those that have their own meaning and existence which is independent of any program session. The meaning of values whose types contain type variables in general depends on the context in which they are used. Under this restriction, we can give the following types to **get** and **put**:

get : `string` $\rightarrow$ `any`
put : `(string*any)` $\rightarrow$ `unit`

where `unit` is a trivial type and the `string` is used as a search key in the persistent store. In examples, we also use the following shorthand:

intern`(s,`σ`)` = **project**(**get**`(s)`,σ)
extern`(s,`$M{:}\sigma$`)` = **put**`(s,`**inject**$(M{:}\sigma)$`)`

In the next subsection we extend Mitchell and Plotkin's SOL language with type `any` and the above two primitives. The extended language provides an appropriate medium to study the interaction of persistence and abstract data types. We still call the extended language SOL.

2.2 Mitchell-Plotkin's SOL Language with Persistence

The set of types (ranged over by σ) of SOL is given by the following syntax:

$\sigma ::= b \mid t \mid [l{:}\sigma,\ldots,l{:}\sigma] \mid \sigma \rightarrow \sigma \mid \forall t.\sigma \mid \exists t.\sigma$

b stands for base types including `any` and `unit`. t stands for type variables. `[`l`:`σ`,...,`l`:`σ`]` stands for labeled record types where $l_1, \ldots, l_n$ are all distinct and there is no significance in the order of the components. We write σ_1 `*` $\cdots$ `*` σ_n as shorthand for `[1:`σ_1`,...,n:`σ_n`]`. In our discussion, the introduction of record types is not essential. They are here for convenience. As was done in [MP88] the desired feature of record structures is easily simulated by product types with appropriate syntactic sugaring. $\forall t.\sigma$ stands for Girard-Reynolds' polymorphic type [Gir72, Rey74] and $\exists t.\sigma$ for Mitchell-Plotkin's existential types. These are binding mechanisms of type variables for which the notion of *free variables* and *bound variables* are standard. We write $\sigma_1[\sigma_2/t]$ to denote the type obtained from σ_1 by replacing all the free occurrences of the type variable t by σ_2.

The set of expressions (or terms) is given by the following abstract syntax:

$$\begin{aligned} M ::= {} & c^\sigma \mid x \mid \lambda x{:}\sigma\,.M \mid M\texttt{(}M\texttt{)} \mid \texttt{[}l\texttt{=}M\texttt{,...,}l\texttt{=}M\texttt{]} \mid M\texttt{.}l \mid \Lambda t\,.M \mid M\texttt{[}\sigma\texttt{]} \mid \\ & \textbf{pack } \sigma \textbf{ in } M \textbf{ to } \sigma \textbf{ end} \mid \textbf{open } M \textbf{ as } t \textbf{ with } M{:}\sigma \textbf{ in } M \textbf{ end} \mid \\ & \textbf{inject}\texttt{(}M\texttt{:}\sigma\texttt{)} \mid \textbf{project}\texttt{(}M\texttt{,}\sigma\texttt{)} \end{aligned}$$

c^σ stands for type constants including the primitive $\mathbf{get}^{string \longrightarrow any}$ and $\mathbf{put}^{(string * any) \longrightarrow unit}$ as well as ordinary atomic constants. `[`l`=`M`,...,`l`=`M`]` is the syntax for record expressions. We write `(`M_1`,...,`M_n`)` as shorthand for `[1=`M_1`,...,n=`M_n`]`. We also write x_1`=`M_1`;`M_2 as shorthand for `(`$\lambda x_1{:}\sigma_1\,.M_2$`)(`$M_1$`)` where σ is the type of M_1 and write x_1`=`M_1`;`$\cdots$`;`x_n`=`M_n as shorthand for x_1`=`M_1`;(`x_2`=`M_2`;(`$\cdots$`;`x_n`=`M_n`)`$\cdots$`)`. Assuming that the evaluation is call-by-value, this provides value binding and sequential evaluation.

The expression **pack** σ **in** M **to** σ **end** is used to create a package or data algebra implementing an abstract data type by packing the type σ, representing the intended abstract type. σ is sometimes called a *witness type* [CM88]. A package is opened by the **open** statement: **open** M_1 **as** t **with** $x{:}\sigma(t)$ **in** M_2, which associates the witness type of the package M_1 with the type variable t, binds the variable x to the package value M_1, and executes the body M_2.

The type system of SOL is given as a proof system for *typings*. Since the type of an expression depends on the types of its free variables, the type system is represented as a proof system for *typings* of the form $\mathcal{A} \rhd M : \sigma$ where $\mathcal{A}$ is a function from a finite set of variables to types, called a *type assignment*. We write $\mathcal{A}\{x : \sigma\}$ for the function $\mathcal{A}'$ such that $dom(\mathcal{A}') = dom(\mathcal{A}) \cup \{x\}$, $\mathcal{A}'(x) = \sigma$ and $\mathcal{A}'(y) = \mathcal{A}(y)$ for all $y \in dom(\mathcal{A}), y \neq x$. Some of the typing rules are shown belows:

$$\text{(RECORD)} \quad \frac{\mathcal{A} \rhd M_i : \sigma_i \quad (1 \le i \le n)}{\mathcal{A} \rhd [l_1{=}M_1, \ldots, l_n{=}M_n] : [l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n]}$$

$$\text{(DOT)} \quad \frac{\mathcal{A} \rhd M : [l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n]}{\mathcal{A} \rhd M.l_i : \sigma_i \quad (1 \le i \le n)}$$

$$\text{(INJECT)} \quad \frac{\mathcal{A} \rhd M : \sigma}{\mathcal{A} \rhd \textbf{inject}(M{:}\sigma) : \text{any}}$$

$$\text{(PROJECT)} \quad \frac{\mathcal{A} \rhd M : \text{any}}{\mathcal{A} \rhd \textbf{project}(M,\sigma) : \sigma}$$

$$\text{(PACK)} \quad \frac{\mathcal{A} \rhd M : \sigma_1[\sigma_2/t]}{\mathcal{A} \rhd \textbf{pack } \sigma_2 \textbf{ in } M \textbf{ to } \exists t.\sigma_1 \textbf{ end} : \exists t.\sigma}$$

$$\text{(OPEN)} \quad \frac{\mathcal{A} \rhd M_1 : \exists t.\sigma_1 \qquad \mathcal{A}\{x{:}\sigma_1\} \rhd M_2 : \sigma_2}{\mathcal{A} \rhd \textbf{open } M_1 \textbf{ as } t \textbf{ with } x{:}\sigma_1 \textbf{ in } M_2 \textbf{ end} : \sigma_2} \quad t \text{ not free in } \sigma_2 \text{ or } \mathcal{A}$$

The rules for the other term constructors are the same as in the second-order lambda calculus [Gir72, Rey74]. The rules for records are standard. The rule (PACK) says that *any* term of type $\sigma_1(\sigma_2)$ can be turned into a data algebra of type $\exists t.\sigma$ by packing the type σ_2. As seen from the definition of the set of types, existential types are first-class types in the sense that they can be freely mixed with other type constructors. The typing rule (PACK) therefore implies that data algebras are first-class values and can be freely mixed

with any other expression constructors. Mitchell and Plotkin achieved this very flexible treatment of data algebras by placing proper restrictions on their usage. The rule (OPEN) says that in the body M_2 of the **open** statement, the abstract type implementing the package M_1 must be treated as a free type variable. The constraints associated with this rule prohibit the programmer from making any connection (possibly through type abstraction or other **open** statements) between this type variable and any other types. This condition is a simplified version of the slightly more general one in [MP88]. However, assuming an unbounded supply of type variables, we do not lose any generality. Since free type variables carry no structural information, the compiler can check the type correctness of the program that uses a data algebra without inspecting the structure of its implementation type. Because of this very restrictive treatment of witness types, this model is named the *hypothetical witness model* [CM88]. The type variable t in the **open** construct can be thought of as a name, in the body of the construct, of the hypothetical witness which is supposed to exist.

In order to ensure the conditions for values of type `any` we have defined earlier, we have to constrain the usage of **inject**. To do this in the most general way, we define the notion of *closed expressions*. The notions of free and bound variables are defined as in the second-order lambda calculus. The definition of *bound type variables* in an expression M is obtained from that of the second-order lambda calculus by adding the rule: t is a bound type variable in M if $M \equiv$ **open** M **as** t **with** $\cdots$ **end** and extend it to a general expression in a usual manner. An expression is closed iff it does not contain free variables and free type variables. The desired constraint is then given:

(*) If M is a closed expression then all type variables in its expressions of the form **inject**$(M : \sigma)$ must be bound by $\forall t$.

The rationale behind this condition is that we allow type variables in the specification σ of **inject**`(`M`,`σ`)` only if they will eventually be instantiated (substituted) by some closed types before it is evaluated.

2.3 The Problem of Persistence and Abstract Types

The language we have just defined serves as a formal model for a polymorphic programming language with persistence and abstract data types such as Napier88 [MBCD89]. However, it has one serious drawback in that it cannot express the injection of values having a witness type into type `any`. From a database perspective, this implies that the language cannot provide an abstraction mechanism for values stored in persistent data. This becomes a serious problem since most objects in database programming are persistent ones.

One possible solution, requiring no more material than that of Mitchell and Plotkin's framework, is to use the **pack** statement and create a new package. Suppose we want to create a few points having a witness type, store them, and later retrieve them for further processing. We can achieve this by creating a new package containing the desired points together with any necessary operations, and storing the package as shown in the following example:

```
open cart_point as point with
   P:PointADT(point)
in
   a = P.create(1.0,2.0);
   b = P.create(1.0,3.0);
   c = P.create(1.0,4.0);
   extern("my_point_pack",
          pack point in [a=a,b=b,c=c,create=P.create,
                         x_val=P.x_val,y_val=P.y_val,move=P.move]
          to ∃t.[a:t,b:t,c:t,
                 create:(real*real) → t,
                 x_val:t → real,
                 y_val:t → real,
                 move:t → (real*real) → t]
          end:∃t.[a:t,...,move:t → (real*real) → t])
end
```

Then later, we can retrieve the package, open it and continue the computation as in:

```
my_point_pack = intern("my_point_pack",∃t.⋯[a:t,...,move:t → (real*real) → t]);
open my_cart_point as point with
   P:[a:point, b:point, c:point,
      create:(real*real) → point,
      x_val:point → real,
      y_val:point → real,
      move:point → (real*real) → point]
in
      ...
      P.x_val(a)
      ...
end
```

Since the package has a closed type (i.e. an existential type), this approach immediately gives a sound type system for persistent abstract objects. This solution is effective but it requires the programmer to create one package containing all the related objects and operations. This seems to be problematic when we are dealing with a community of users with a large shared database, since this approach requires the user to have a complete knowledge of the values of the witness type and retrieve all the values at once.

Another way of allowing these values to persist is to embed the persistent store access within the package. For example

```
cart_point_ps = pack ⋯
                to ∃t.[create:real*real → t,
                       x_val:t → real,
                       y_val:t → real,
                       move:t → (real*real) → t,
                       store:(string*t) → unit,
                       retrieve:string → t]
                end;

open cart_point_ps as point with ⋯ end
```

thus allowing users of the package to store new values within a context from which they may later be accessed. Although this is at least useful, the persistence involved is no longer orthogonal [ABC+83], as the programmer must invent a different naming scheme for each different abstract package through access functions (such as `store`, `retrieve` in the above example). This is a serious restriction for general persistent programming.

The goal of this paper is to propose a mechanism which allows values of witness types to achieve the full range of persistence without losing the desirable properties of Mitchell and Plotkin's model of abstract data types. But before presenting our system, we review Cardelli and MacQueen's proposal.

3 Cardelli and MacQueen's Proposal

Cardelli and MacQueen proposed [CM88] a different view of abstract data types called the *abstract witness model* and discussed the relationship between their model and persistence. In this model, the witness type of a data algebra is "abstract" but is supposed to have a "real" existence. This means that when we create a data algebra, we also create an actual type associated with it. Opening a data algebra no longer creates a new hypothetical witness but only associates a name to the data algebra's witness type. For example, under this model,

```
open M as point1 with P:PointADT(point1) in
   open M as point2 with Q:PointADT(point2) in
      (Q.move(P.create(1.0,2.0)))(3.0,4.0)
   end
end
```

is presumably legal and returns a value of a type associated with the data algebra M.

In this model, an abstract type seems to have its own meaning and existence so long as the associated data algebra exists. In order for a value of a witness type to be persistent, it therefore seems sufficient to make the associated data algebra persistent. For example,

```
extern("point_package",cart_point:∃t.PointADT(t));

open cart_point as point with
   P:PointADT(point)
in
   extern("a",P.create(0.0,0.0):point)
end;

open intern("point_package",∃t.PointADT(t)) as point2 with
   Q:PointADT(point2)
in
   intern("a",point2)
end
```

should presumably be type correct since both `point` and `point2` are the names of the same abstract witness type associated with the data algebra `cart_point`.

This model provides an intuitively appealing solution to the problem we have just described in the previous section. There are, however, several issues which require careful scrutiny. The first problem is to understand their informal description of the model as a formal definition of a type system. Without such a proper formalism, it is not obvious that we can extract some usefulness, such as a type-checking algorithm, out of the model. One well established way of giving a formal description of a model is to formulate the model as a typed functional calculus similar to SOL. The authors found this a non trivial challenge. Under this model, they suggested that the set of types should be extended with the expressions of the form M`.type` which denotes the *abstract witness type* associated with the expression M having an existential type. The **open** statement should then be treated as a shorthand to introduce a local name to the type M`.type` where M is the data algebra being opened in the statement. In order to give a formal account for the model we therefore need to define a type system in which those expressions can be treated as types. In doing this we need to answer several important questions. Some of them are: which class of terms can appear in a type expression of the form M`.type`, and when are two types M_1`.type` and M_2`.type` equal? One consistent solution is to follow the proposal of [Mac86] and to introduce *levels* in the type system; the first level of types are types of values and the second level of types are types of data algebras. All second level objects are compiled before the compilation of the first level objects. The expressions of the form M`.type` where M is a second level object are treated as types only in the first level. Under this approach data algebras are compile time objects and the equality of types is simply identity. This is adopted in Standard ML to introduce modules [HMT88] in a higher-order functional calculus. Under this approach, however, data algebras cannot mix with ordinary expressions and therefore it sacrifices most of the flexibility of Mitchell and Plotkin's model of abstract types.

The other extreme alternative is to extend the type of SOL with M`.type` where M may be *any* term having an existential type. The equality of types is the congruence induced by the value equality. An obvious drawback to this approach is that it does not yield a static type checking algorithm. Cardelli and MacQueen suggested that there might be some *ad hoc* way to compromise between the static type checking and the flexibility of treating data algebras as values. They gave the following example of a difficulty with static type checking

`C =` **if** b **then** A **else** B

where A and B are terms having the same existential type. The existence of a conditional statement was also suggested as the cause of the difficulty of static type checking in [DT88]. However, the problem seems to be more fundamental. The least thing we should be able to do with values is to bind names to them. In a functional calculus this is done through lambda abstraction and function application. This mechanism seems to be needed in any attempt to treat objects as values. Note that this also is needed for objects to persist since persistence is a feature that is achieved by special functions. Under this view, any attempt to treat data algebras as values might cause the difficulty with static type checking. Worse yet, there seems to be no existing proposal that enables a desired *ad hoc* method in a consistent way. A detailed analysis of this problem is outside the scope of this paper and here we only suggest the difficulty by examples. Consider the following function application:

```
(λx:∃t.PointADT(t).λy:∃t.PointADT(t)
   open x as p1 with P:PointADT(p1) in
      open y as p2 with Q:PointADT(p2) in
         Q.move(P.create(0.0,0.0))(1.0,2.0)
      end
end)(M)(M)
```

Under the abstract witness model, at least semantically the application of `Q.move(P.create(0.0,0.0))` is type correct. However, the type consistency is known only at run time: since functions are first class values they should be type checked before each application. Therefore if we maintain static type checking, the type system should reject the above code. Now consider the following example involving persistent store:

```
(λx:∃t.PointADT(t).λy:∃t.PointADT(t)
   open x as p1 with P:PointADT(p1) in
      extern("mycreate",P.create:(real*real) → p1);
      open y as p2 with Q:PointADT(p2) in
         Q.move(intern("mycreate":(real*real) → p2)(0.0,0.0))(1.0,2.0)
      end
end)(M)(M)
```

This example is essentially the same example as the one Cardelli and MacQueen gave in [CM88] and therefore should be type-checked under their model. But according to the obvious model of persistence, the above two examples are essentially the same. These two examples suggest a difficulty in developing a safe and consistent type system with persistence under the abstract witness model. Developing such a type system seems to require a significant extension to the existing system of types for persistence and data abstraction. In the next section, we attempt to construct one such system based on Mitchell and Plotkin's model. This also sheds some light on the abstract witness model as well. We will take up this point again in the next section.

4 Reconciling Abstract Data Types and Persistence

We now present a new mechanism which we believe to provide a satisfactory treatment of persistence without sacrificing the desirable features of Mitchell and Plotkin's abstract data types. We first explain this mechanism through examples and then provide its formal account as an extension of Mitchell and Plotkin's SOL language.

The intuition is that we allow the programmer to store the "hypothetical witness" created in an **open** statement and later "reactivate" it. Since, under our model of persistence, the only objects that can be stored are values that have a closed type, this requires us to treat hypothetical witnesses as values in the language. For this purpose, we introduce the new type `Witness`. This is the type of all the possible hypothetical witness types. The values of this type are themselves types and therefore can be used as types. Different from other types in SOL, however, these types are run time objects and cannot be examined by the compiler. To emphasize this difference, we use the term *dynamic types* for those types that contain values of the type `Witness` and distinguish them from the other ordinary types, which we call *static types*. The reader is again warned that the type `any` (which is also named `dynamic` in some literature) is not a dynamic type. As

we noted earlier, however, the general introduction of dynamic types would cause conflicts with static type checking. We solve this problem by introducing a mechanism to localize the typing judgements that involve dynamic types. The major technical contribution of this paper is to establish that such a mechanism is possible in the framework of Mitchell and Plotkin's calculus.

The only construct that creates values of type `Witness` is the **open*** statement, whose syntax is

open* M **as** t **with** w:`Witness` **and** x:$\sigma(t)$ **in** N **end**

As in the **open** statement, x is bound to the value implementing the data algebra M and is used in the body N as an expression of type $\sigma(t)$. In addition to this binding, **open*** creates a value of type `Witness` which represents the hypothetical witness associated with this statement and binds the variable w to it in the body N. Since the type `Witness` is a closed atomic type, w can then be used to export the hypothetical witness type into a persistent store using the ordinary primitives. Note that **open*** creates a value of type `Witness`. This reflects the hypothetical witness model of Mitchell and Plotkin. Another possibility would be to extend the language so that the **pack** statement creates a value of type `Witness`. For this purpose, we would presumably need a primitive such as **witness** [Mac86] which takes a value having existential type and returns a value of type `Witness`. By this alteration, we think we could achieve a type system which is faithful to the abstract witness mode. However, we have not yet examined the details of this alternative.

Since w is assigned the same hypothetical witness type associated with the type variable t in the body N, an expression which has a type of the form $\sigma(t)$ can be coerced to the expression having the type $\sigma(w)$. This coercion is embedded in the primitive **inject***. Different from **inject**, σ in **inject***(M::σ) is in general a dynamic type. To emphasize this property, we use `::` instead of ordinary static typing `:`. This statement, when evaluated, creates a value of type `any` whose type component is the dynamic type denoted by σ. As seen below our type system guarantees type correctness by maintaining the relationship between t and w.

By these mechanisms, we can safely export values having abstract types. The following is an example of **open***:

```
open* cart_point as point with
   mywitness:Witness and
   P:PointADT(point)
in
   a = P.create(1.0,2.0);
   put("mypt_type",inject(mywitness:Witness));
   put("somepoint",inject*(a::mywitness));
   put("my_y_val",inject*(P.y_val::mywitness → real))
end
```

`a` and `y_val` are injected into `any` as values having dynamic types `mywitness` and `mywitness` → `real` where `mywitness` is the value denoting the hypothetical witness created by the entire statement.

The important difference between ordinary types and types containing values of type `Witness` is that the latter are not compile time objects. To preserve the static type-checking, special constructs are needed to use values having dynamic types. For this purpose, we provide **project*** and **reopen*** statements. **project***(`(`M`,`σ`)`) projects the value M of type `any` onto a dynamic type σ. Similar to **project**, the entire expression has the type σ. However, since σ is a dynamic type, the type information about the result of this expression is not available at compile time. To indicate this dynamic nature of the typing judgement, we write **project***(`(`M`,`σ`)`)`::`σ. This statement can only be verified at run time. The only operation that can use values having dynamic types is the **reopen*** statement, whose syntax is given as:

reopen* M_1 **as** t **with** w:`Witness` **and** x:$\sigma(t)$ **is** M_2`::`$\sigma(w)$ **in** M_3 **end**

This statement first binds the variable w to the value M_1 of type `Witness` and then dynamically verifies the typing M_2`::`$\sigma(w)$. If it is verified it then binds the variable x to the value M_2 and executes the body M_3. The important feature of this statement is that the variable x has the *static type* $\sigma(t)$ where t is a type variable associated with the hypothetical witness being reopened by this statement. By this mechanism we successfully localize the dynamic type check. This implies that we can still check statically the type correctness of the body of the **reopen*** statement just as we do that of the **open** statement. The following is an example of **project*** and **reopen***.

```
p = intern("mypt_type",Witness);
reopen* p as mypoint
with
   w:Witness
and
   a:mypoint is project*(get("somepoint"),w),
   y_val:mypoint → real is project*(get("my_y_val"),w → real)
in
   ...
   y_val(a)
   ...
end
```

This "reopens" the hypothetical witness created and stored under the name `"mypt_type"` in the previous example as the type variable `mypoint`. To do this, the statement first checks that the type components of the values retrieved by the names `"somepoint"` and `"my_y_val"` are respectively the dynamic types `p` and `p → real` and then binds the value components of them respectively to the variables `a` and `y_val`, and executes the body. As with the **open** statement, the type correctness of the body is established statically.

In the rest of this section, we provide a formal system where **reopen*** is possible by extending the SOL language. We call the extended language SOL^*.

The syntax of the set of dynamic types (ranged over by τ) of SOL^* is given as follows:

$$\tau ::= t \mid b \mid \tau \rightarrow \tau \mid [l:\tau,\ldots,l:\tau] \mid \forall t.\,\tau \mid Witness \mid w$$

where w stands for variables having the type `Witness`. The set of static types is the following subset of dynamic types:

$$\sigma ::= t \mid b \mid \sigma \rightarrow \sigma \mid [l:\sigma,\ldots,l:\sigma] \mid \forall t.\,\sigma \mid Witness$$

The set of pre-terms of SOL^* is given by the following syntax:

$$\begin{array}{rl} M ::= & c^\sigma \mid x \mid \lambda x{:}\sigma\,.M \mid M(M) \mid \texttt{[}l\texttt{=}M\texttt{,}\ldots\texttt{,}l\texttt{=}M\texttt{]} \mid M.l \mid \Lambda t.M \mid M\texttt{[}\sigma\texttt{]} \mid \\ & \textbf{pack}\ \sigma\ \textbf{in}\ M\ \textbf{to}\ \sigma \mid \textbf{open}\ M\ \textbf{as}\ t\ \textbf{with}\ M{:}\sigma\ \textbf{in}\ M\ \textbf{end} \mid \\ & \textbf{open*}\ M\ \textbf{as}\ t\ \textbf{with}\ x{:}\texttt{Witness}\ \textbf{and}\ x{:}\sigma(t)\ \textbf{in}\ M\ \textbf{end} \mid \\ & \textbf{reopen*}\ M\ \textbf{as}\ t\ \textbf{with}\ x{:}\texttt{Witness}\ \textbf{and}\ x{:}\sigma(x)\ \textbf{is}\ M{::}\tau\ \textbf{in}\ M\ \textbf{end} \mid \\ & \textbf{inject}(M{:}\sigma) \mid \textbf{inject*}(M{::}\tau) \mid \textbf{project}(M,\sigma) \mid \textbf{project*}(M,\tau) \end{array}$$

As is done for SOL, the type system of SOL^* is given as a proof system of typings. As in SOL, a typing is a triple consisting of a set of assertions, a pre-term and a type. For SOL^* we have two forms of assumptions. One is an assignment of a static type to a variable $x:\sigma$; the other is an association of a type variable to a witness variable $t \leftrightarrow x$. This is needed to type check **inject***. An *environment* $\mathcal{E}$ (which is an extension of type assignment in SOL) is a finite set of the above two forms of assertions. We write $\mathcal{A}\{x:\sigma\}$ and $\mathcal{A}\{t \leftrightarrow x\}$ for extensions of $\mathcal{A}$ with $x:\sigma$ and with $t \leftrightarrow x$ respectively. We distinguish static typings and dynamic typings. A static typing is a formula of the form $\mathcal{A}\triangleright M : \sigma$ and corresponds to a typing in SOL. A dynamic typing is a formula of the form $\mathcal{A}\triangleright M :: \tau$, which has no equivalence in SOL. Dynamic typings are typing judgements which are verified only at run time. The set of rules for SOL^* typings (both dynamic and static) are the ones obtained from that of SOL by adding the following rules for new constructs we introduced in SOL^*:

$$(\textsc{inject}^*)\ \frac{\mathcal{A}\{x{:}\text{Witness},t \leftrightarrow x\}\triangleright M:\sigma(t)}{\mathcal{A}\{x{:}\text{Witness},t \leftrightarrow x\}\triangleright \textbf{inject*}(M{::}\sigma(x)):\text{any}}$$

$$(\textsc{project}^*)\ \frac{\mathcal{A}\{x{:}\text{Witness}\}\triangleright M:\text{any}}{\mathcal{A}\{x{:}\text{Witness}\}\triangleright \textbf{project*}(M,\sigma(x))::\sigma(x)}$$

$$(\text{OPEN}^*)\quad \frac{\mathcal{A}\{x{:}\text{Witness}, t \leftrightarrow x, y : \sigma_1(t)\} \rhd M_1 : \sigma_2 \qquad \mathcal{A} \rhd M_2 : \exists t.\sigma(t)}{\mathcal{A} \rhd \textbf{open*}\ M_2\ \textbf{as}\ t\ \textbf{with}\ x{:}\text{Witness}\ \textbf{and}\ y{:}\sigma_1(t)\ \textbf{in}\ M_1\ \textbf{end} : \sigma_2}$$
(t not free in σ_2 or $\mathcal{A}$)

$$(\text{REOPEN}^*)\quad \frac{\mathcal{A} \rhd M_1 : \text{Witness} \qquad \mathcal{A}\{y{:}\text{Witness}\} \rhd M_2 :: \sigma_1(y) \qquad \mathcal{A}\{y{:}\text{Witness}, x{:}\sigma_1(t), t \leftrightarrow y\} \rhd M_3 : \sigma_2}{\mathcal{A} \rhd \textbf{reopen*}\ M_1\ \textbf{as}\ t\ \textbf{with}\ y{:}\text{Witness}\ \textbf{and}\ x{:}\sigma_1(t)\ \textbf{is}\ M_2{::}\sigma_1(y)\ \textbf{in}\ M_3\ \textbf{end} : \sigma_2}$$
(t not free in σ_2 or $\mathcal{A}$)

The reader is encouraged to trace the typing derivation of the above examples using these rules.

5 Implementation

The static type checking of these new existential types is, as required, exactly the same as that of standard Mitchell and Plotkin existential types, and has been fully discussed elsewhere. Here we will give a possible implementation of the dynamic checking associated with the injection and projection of a witness value.

When a value is stored in an infinite union, it must be stored along with a representation of its type. When a projection statement is compiled, the compiler must construct a representation of the type the union is being projected onto, and must cause an equivalence function to be executed over the type representation stored with the value and that constructed by the compiler for the projection statement. The dynamic projection may only succeed if the two type representations are compatible.

The simplest way to arrange this is in a persistent system where the compiler may access the persistent store. When an injection statement is encountered, the compiler may place the type representation in a known location within the store as the program is compiled, and the address of this single representation may be stored with any values which are injected dynamically. When a projection is encountered, again a suitable type representation may be placed once within the store as the program is compiled, and code planted to access this representation dynamically as required. The function which tests two type representations for equivalence must also be at a fixed location within the store.

When a value of a free witness type is injected as a special `Witness` type, the injection must be slightly different from that described above. In this case, the type representation to be stored with the value may not be created statically by the compiler but must be created dynamically for each invocation of the **open*** statement which performs the injection. Two such type representations may be considered as compatible if and only if they represent the same dynamic invocation of a particular **open*** statement.

In a persistent system with referential integrity this is not hard to arrange; each such new type may be represented by a new object appropriately marked as dynamic for the equivalence algorithm. The equivalence algorithm should allow two objects so marked to pass the equivalence test if and only if they share the same identity. As this identity must in any case be preserved for the lifetime of the system to preserve the referential integrity of the store, no further work need be done for persistent values. Such a scheme is already in operation for abstract data types and polymorphic procedures in the language Napier88 [MBCD89].

Whenever the compiler encounters a definition of value of type `Witness` at the start of an **open*** statement, it must therefore plant code to construct a new store object appropriately marked for the type equivalence algorithm. It should also make a note of the offset within the current frame at which this new value will be located. When it encounters a special injection operation which uses this value as a type, the code generated to perform the injection must create a new union value which contains the unbound witness value in the value field, and this type representation value in the type field. When the Witness value itself is injected into the infinite union, the type representation stored with it is a simple marker to indicate that the value stored is a witness type representation.

For the **reopen*** clause, this value is first retrieved from the union. The value is then used as the type representation for the projection of all witness types stated at the head of the statement. As long as these values have been injected within the same dynamic scope as the witness type itself, the value of the type representation will have the same identity, and the dynamic type checks will succeed.

As can be seen, this implementation mirrors exactly the intuitive description and justification of the **reopen*** clause, by creating an unforgeable value for each dynamic execution of an **open*** clause which must be presented in order to perform the **reopen*** and achieve access to the same witness type definitions.

6 Conclusion

We have examined the interaction between persistence and Mitchell and Plotkin's abstract data types and proposed an extension to their system which allows values of witness types to achieve the full range of persistence without losing the desirable properties of the original model. In order to do this we introduced a mechanism to treat witness types as values in a restricted way so that the typing judgements that depend on values are localized.

The language we have described is a variant of a functional calculus that can serve as a core of the type system of practical persistent programming languages. As we have discussed in section 5, an efficient type-checking algorithm can be extracted from our type system.

There are a number of ways to extend the work described here. One of them is to exploit the mechanism introduced that allows the programmer to treat types as values in a restricted way while preserving static type checking. This might be useful when we want to achieve some degree of dynamic manipulation of objects which are usually restricted to compile time, such as database *schemas* (the database system analogue of types).

Acknowledgement

We would like to thank Malcolm Atkinson for his useful comments on an earlier version of the paper and and stimulating discussion on abstract data types.

References

[ABC+83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4), November 1983.

[ACPP89] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, 1989.

[Ada80] Reference manual for the Ada programming language. G.P.O. 008-000-00354-8, 1980.

[CM88] L. Cardelli and D. MacQueen. Persistence and type abstraction. In M.P. Atkinson, O.P. Buneman, and R. Morrison, editors, *Data Types and Persistence*, Topics in Information Systems, chapter 3, pages 31–41. Springer Verlag, 1988.

[Coc83] W.P. Cockshott. *Orthogonal Persistence*. PhD thesis, University of Edinburgh, February 1983.

[DT88] S. Danforth and C. Tomlinson. Type theories and object-oriented programming. *ACM Computing Surveys*, 20(1), 1988.

[Gir72] J.-Y. Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et théorie des types. In *Second Scandinavian Logic Symposium*, pages 63–92. Springer-Verlag, 1972.

[HMT88] R. Harper, R. Milner, and M. Tofte. The definition of Standard ML (version 2). LFCS Report Series ECS-LFCS-88-62, Department of Computer Science, University of Edinburgh, August 1988.

[LAB+81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1981.

[Mac86] D. B. MacQueen. Using dependent types to express modular structure. In *Conf. Record Thirteenth Ann. Symp. Principles of Programming Languages*, pages 277–286. ACM, January 1986.

[MBCD89] R. Morrison, A.L. Brown, R.C.H. Connor, and A. Dearle. Napier88 reference manual. Technical report, Department of Computational Science, University of St Andrews, 1989.

[MP88] J.C. Mitchell and G.D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[Per87] Persistent Programming Research Group. PS-algol reference manual - fourth edition. Technical Report PPRR-12-87, Universities of Glasgow and St Andrews, 1987.

[Rey74] J.C. Reynolds. Towards a theory of type structure. In *Paris Colloq. on Programming*, pages 408–425. Springer-Verlag, 1974.