Persistence and Type Abstraction

Luca Cardelli
David MacQueen

AT&T Bell Laboratories
Murray Hill, NJ 07974

Introduction

Abstract types are a well known and effective way of structuring programs. The basic ide of information hiding can however conflict with the need to store data for long periods of time, ar make it accessible to different activities. In particular a typechecker must be able to recognize tl occurrence of the *same* abstract type during different activations, and must enforce the privacy data representations.

To achieve this, the persistent storage of data must preserve type information, and mu respect type abstraction. The use of type abstractions in the presence of persistent storage requir that abstract types be made persistent as well. Under these conditions, we can preserve ty security across distinct activations of the typechecker.

The following is a brief account of how various models of abstraction and persistenc interact. We start by sketching a simple polymorphic language and its types and showing vario ways of modeling type abstraction in such a language. We then discuss some basic notior underlying persistent storage of typed objects, such as the intern and extern primitives and tl special type *dynamic*, and describe three persistence strategies. Finally we discuss the particul problem of persistent abstract types.

Values and Types

We will base our discussion on a simple polymorphic language in the tradition of ML [Miln 84] and Amber [Cardelli 84]. The simplified language we have in mind is closely related to th language SOL [Mitchell and Plotkin 85], variants of which are described in [Reynolds 85] an [Cardelli and Wegner 85].

The basis of this language is a slightly sugared applied lambda calculus that is adequate f expressing certain kinds of values. Type annotations are added to the basic expressions in such way that one can statically determine a type for each expression. This type is a structur characterization of the value denoted by the expression. Types are viewed as meta-level terms in a type language designed to express certain structural properties of values. Types are not values, but can be interpreted semantically as sets of values.

A value may have many types, and correspondingly a value expression can be typed in several ways. For example, the "type-free" expression $\lambda x.\ x$ is the basis for the following type-annotated versions, each of which has a different type:

$\lambda x{:}\text{int}.\ x$	$: \text{int} \rightarrow \text{int}$
$\lambda x{:}\text{bool}.\ x$	$: \text{bool} \rightarrow \text{bool}$
$\Lambda t.\ \lambda x{:}t.\ x$	$: \forall t.\ t \rightarrow t$

These type annotations do not affect the value computed by the expression (the identity function in this case), they only help to characterize the structure of that value. The last type is a polymorphic type, which expresses the fact that the basic expression $\lambda x.\ x$ can have many types, namely all instances of the type schema $t \rightarrow t$.

Language of types

The class of type expressions is defined by the following abstract syntax:

$$\sigma ::= \\ \text{int} \mid \text{bool} \mid \ldots \mid t \mid \sigma \times \sigma \mid \sigma + \sigma \mid \sigma \rightarrow \sigma \mid \forall t.\ \sigma\ (t)$$

Types are *structural*, i.e. types are equivalent if their term structure matches (modulo change of bound type variables). Certain types are *atomic* in that they have no internal structure and match only themselves. The atomic types include certain primitive types such as int and bool, and also the *abstract* types discussed below.

A closed value expression (i.e. one that contains no free occurrences of unbound or lambda-bound variables and therefore denotes a particular value) must have a type that is a closed type expression, i.e. its type may not contain free type variables.

Existential types and packages

Following SOL [Mitchell Plotkin 85] we will introduce type abstraction through the notion of existentially quantified types. We introduce a new class of existentially quantified type expressions:

$$\lambda x.\ x\,x$$

$$\sigma ::= \exists t.\ \sigma(t)$$

$$\lambda x : \forall t.(t \rightarrow t).\ x[\forall s.(s \rightarrow s)]\,x$$

$$: (\forall s. s \rightarrow s) \rightarrow (\forall s.\ s \rightarrow s)$$

Roughly speaking, a value has type $\exists\, t.\, \sigma(t)$ if it has type $\sigma(\tau)$ for some particular type τ. An expression having existential type $\exists\, t.\, \sigma(t)$ must specify a particular type τ as a witness for the existential type quantifier, and an expression having the type $\sigma(\tau)$. We use the following syntax:

pack [$t = \tau$; $e : \sigma(t)$] : $\exists\, t.\, \sigma(t)$ (1)

Such expressions (and their values) will be called *packages*. In order to establish the typing (1) it is necessary and sufficient to establish

$e : \sigma(\tau)$

The type τ is called the *representation type* and the expression e defines the *interpretation* of the type t. The expression e typically denotes a tuple of values and functions through which we are allowed to create and manipulate values of the representation type. The existential type $\exists\, t.\, \sigma(t)$ is called the *interface* or *signature* of the package, which in turn is called an *implementation* of the interface. Matching of existential types is also structural (modulo renaming of bound variables).

The only way to make use of a package is to *open* it in a limited scope consisting of an expression:

open A **as** t,v **in** e (where A: $\exists\, s.\, \sigma(s)$)

The treatment of the binding of t will differ according to the model of abstraction we adopt, as explained in the next section.

Packages and abstraction

There are several alternative ways of treating the type component of a package, and these alternative treatments lead to different styles of type abstraction.

The transparent witness model

One approach, adopted for modules in ML [MacQueen 85], is to view the representation type simply as an accessible component of the package. For instance, if P = **pack** [t = int × int; ...] then within (**open** P **as** t, v **in** ...), t would be equivalent to int × int. In other words, the type component of a package is not at all hidden or "abstract". In this model, type abstraction is achieved by lambda abstraction with respect to a formal (therefore abstract) package variable.

The hypothetical witness model

A second approach is that of SOL, also adopted in [Reynolds 85] and [Cardelli Wegner 85].

$\lambda x.\, x x$ $s \to t$

$\forall t.\, ((\forall s.\, s \to t) \to t)$ s

$\forall s (s \to s) \to \forall s (s \to s)$

In SOL, the type component of a package (called a *data algebra*) is treated as purely "hypothetical", despite the fact that it is quite explicitly defined in the package expression.

In this model, (**open** P **as** t, v **in** ...) declares the names t and v to represent the (hypothetical) type component and interpretation of the package A during evaluation of the body expression. Within this scope, the witness variable t is treated as a new atomic type, and this type is not allowed to appear in the type of the open expression since the binding of t has no significance outside of it.

This means that the type component of the package has no meaningful permanent identity; it can never be related to any other type except within the local scope of an open expression. Thus we refer to the type component as being hypothetical, indicating an even stronger constraint on its use than that implied by the conventional meaning of type abstraction, where the type retains its identity even though its structure is hidden.

One advantage of this very restrictive approach is that an existential type is just an ordinary type, and correspondingly, packages are just ordinary values that can be manipulated in all the usual ways, such as being defined by conditional expressions and serving as arguments and results of ordinary functions.

The abstract witness model

The third approach is a compromise between the previous two and we could characterize it by saying that the type component of the package is real but "abstract". Given P = **pack** [t = τ; ...] we can refer to the witness type of P, but it is treated as an atomic type unique to the package P, and not as an abbreviation for the representation type τ. Under this interpretation, we will refer to a package as an *abstraction*, and to its type component as an abstract type. Abstract types can appear (as atomic elements) in other type expressions, including the type of the body of an open expression. If we want to continue to view packages as values and existential types as ordinary types in this model, the distinction between types and values becomes blurred and we have to impose some rather *ad hoc* constraints to preserve static type checking. For instance, if A,B: $\exists$ t. σ(t) and we define

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

then we will probably require that the witness type of C does not match either the witness of A or of B.

The problem of persistent abstractions in this model is more interesting, because the use of one abstraction in defining another can give rise to dependencies that need to be preserved when values and abstractions are made persistent.

Intern and Extern

A value in main memory can be made persistent [Atkinson et al. 83] by an extern operation which writes it to persistent memory. Such a value can then be recovered by a symmetrical intern operation on it. Extern and intern preserve the structure of data, including sharing and circularities. Given a value A, intern(extern(A)) should return a value B which is at least isomorphic, modulo relocation, to A, and depending on the implementation of intern-extern can be A itself.

The precise way in which external values and internal values are related can change, according to the persistence model one adopts; a few alternatives will be sketched later. Independently of these different models, the basic idea is to be able to transfer arbitrary objects from main to persistent storage and back.

We assume we can intern-extern any value in memory, including programs and types. This allows us to make both types and values persistent. This ability to intern-extern values and types uniformly is useful in situations where values and types are mixed, as is the case for dynamic values, described in the next section.

Intern and extern can be implemented to work on basic storage formats, like strings and arrays. Values and types can then be built out of the same storage formats, so that intern-extern are not aware of what they are manipulating, and can work uniformly on values and types.

Dynamics

What is the type of an object returned by intern? This may be difficult to determine statically, as intern can read objects of any kind. Intern and extern primitives can be safely embedded in a strongly typed system by the use of *dynamic* objects [Cardelli 84]. The intuition is that an object of type dynamic is dynamically typechecked in the context of an otherwise statically typed language.

An object of type dynamic is a pair consisting of a type and an object of that type. Hence a dynamic object is self-describing, and can be manipulated, stored and retrieved without the usual restrictions imposed by static typechecking. A dynamic object is created by the syntax:

```
dynamic(τ,e)    : dynamic
```

A dynamic object can be coerced to a given, statically known, type by:

```
coerce d to τ    : τ
```

If the specified type τ matches the internal type of the dynamic, then the corresponding value is returned, stripped of the type. Otherwise a run-time type error is generated.

As dynamic objects are self-describing, they are well suited to be exported to persistent storage. They also allow us to preserve strong typing in situations where static typing is impossible, as when data has a longer life span than activations of the compiler performing typechecking.

Identifications and Handles

Identifications provide an unambiguous way of telling whether two objects are the *same* object. Identifications are made unique across all systems and users which may refer to them, usually by encoding their time and place of creation within them.

An identified object is an object permanently associated with an identification. At the time of creation of the identified object, a new identification is also created for it. The identification part of an identified object can be read and compared with other identifications, but cannot be modified.

Distinct identifications will be used (a) to mark external forms of abstract data types, and (b) to mark all objects, in those models of persistent store which rely on persistent identifiers (PIDs).

Handles are names interpreted through an external persistent environment that maps them to external forms of persistent objects. For example, if a persistent object is stored in a file, then the file name may be used as its handle.

Persistence Strategies

We are going to sketch here three different persistence models, which correspond to three different semantics for intern-extern. In the simplest model, intern-extern work on individual values. In a more elaborate model, they work on the database as a whole. Finally, they synchronize access to a shared database.

The fetch-store model

In the first scenario, persistent memory is just backup storage for ephemeral structures. The association between internal and external objects is mediated by a handles, e.g. explicit file names.

Extern makes a copy of an ephemeral object in persistent storage, associating it with a handle. Many calls to extern on the same object and different handles will make many independent copies. Calls of extern on two objects which share a substructure will duplicate the substructure.

Intern, given a handle, makes a copy of the persistent object in ephemeral storage. Many calls to intern on the same handle will make independent copies. Sharing is only preserved within persistent objects, not across them.

The load-dump model

In the second scenario, we assume the user has exclusive access to the persistent storage. Ephemeral memory is used as a cache for persistent storage. A one-to-one association between internal and external objects is maintained though the use of PIDs. Each object is identified by a PID, both in persistent and in ephemeral storage.

Extern makes a copy of an ephemeral object in persistent storage. All calls to extern on the same object will produce the same persistent object. Calls of extern on two objects which share a subobject will preserve the sharing of the subobject, as the common subobject has its own PID. Intern makes a copy of a persistent object in ephemeral storage. All calls to intern on the same PID will give the same object. Sharing of subobjects is preserved.

If intern-extern are automatically called when needed, this achieves the semantics of loading the whole database at the beginning of execution, and dumping the whole database at the end. Intern-extern are only used to load and dump the database incrementally.

This strategy is analogous to virtual memory, where virtual addresses play the role of PIDs.

The lock-commit model

In the third scenario, ephemeral memory is a cache for shared persistent storage with concurrent access. Intern-extern can no longer be fully automatic, because other users or processes may be affected.

Intern-extern work as in the previous case, through PIDs. Intern-extern must be explicit again, as in the first scenario, to control the synchronization aspects. Intern may be made to correspond to *lock* and extern to *commit*.

The load-dump model of persistence is the simplest one, conceptually, as intern-extern are automatically performed. Unfortunately it does not scale up to concurrent access (lock-commit), for which we need explicit intern-extern operations. This seems to be a point in favor of fetch-store with respect to load-dump, because the former is compatible with lock-commit.

Persistent Abstractions

Under the hypothetical witness model, packages are not distinguished from other values and the witness type has no permanent identity, so the usual treatment of intern and extern for dynamic values applies to packages as well. Hence we will deal hereafter with the abstract witness model; it appears that the transparent model can be treated in a similar fashion.

The load-dump model of persistent storage behaves exactly like a single programming session, and nothing special has to be done to provide persistent abstractions or values. Use of dynamics and the need for persistently identifying abstract types rises primarily in the fetch-store model, and in the more general lock-commit model. Hence we shall concentrate here on the latter models.

An abstraction consists of:

(1) an identified object representing the abstract type,
(2) a tuple of operations (the interpretation), and

(3) the interface specification.

Optionally the representation type itself may be included for debugging purposes.

(1) and (2) constitute the abstraction proper, which implements the interface (3). The interface is expressed as an existential type, i.e. it is a structural type not involving the abstract type. These components are all that is needed for manipulating objects and typechecking expressions of that type.

There is an analogy between an abstraction and its interface and dynamic objects. A dynamic object can be considered as having the interface: $\exists t.\ t$. However the type component of a dynamic object is not abstract, since it can be inspected by the coerce statement and matched against a contextual type.

An extern operation on an abstraction moves all the above pieces of information to persistent storage. Extern could be an explicit operation, or it could be automatically performed every time a new abstraction is declared.

One should make sure that an abstraction is made persistent before making objects of its type persistent. To guarantee this, abstractions might automatically become persistent at the time of their creation. This policy requires that a handle be automatically generated for each abstraction.

When externing an object of an abstract type we create a dependency between (a) the object and (b) the abstraction which carries the abstract type and the operations which are supposed to work on that object. When we intern such an object, the corresponding abstraction must also be fetched, to provide a context supporting the use of the abstract object.

For example,

```
signature S = ∃t. σ(t)
abstraction A = coerce (intern "abstraction") to S
value a = open A as t,p in coerce (intern "object") to t
```

As a more complex example, we can parameterize with respect to an abstraction and a handle:

```
λAbs:Sig. λx:handle.
    open Abs as t,p in
        pack [t'=t; (coerce (intern x) to t × (t→int)) : t' × (t'→int)]

: Sig → handle → ∃t'. t' × (t'→int)
```

Here the target type of the coercion is determined dynamically, as it must be extracted from the abstraction parameter. The resulting type of the function does not depend on it, because pack abstracts it away by existential quantification. The resulting package is a new abstraction, which is independent of the original parameter abstraction and therefore must be self-sufficient.

We may want to extern an object whose type involves an abstract type, e.g. a function having abstract types as parameters and results. Such an object is necessarily associated with the abstraction because the abstract type occurs in its type, and its full use may require the presence of the abstraction. Hence the abstraction itself should be persistent and should be interned along with the object that depends on it.

Conclusions

We have discussed several models of persistence and abstraction. The main idea is that of preserving the identity of abstract types by giving them persistent identifications. This can be achieved in slightly different ways in the different schemes of persistence and abstraction. We have mainly dealt with the combination of the abstract witness model of packages with the load-fetch model of persistence.

References

[Atkinson et al. 83] M.P.Atkinson, P.J.Bailey, K.J.Chisholm, W.P. Cockshott and R.Morrison: *An Approach to Persistent Programming,* The Computer Journal 26, 4, 360-365, 1983

[Cardelli 84] L.Cardelli: *Amber.* AT&T Bell Labs Technical Report, 1984.

[Cardelli Wegner 85] L.Cardelli, P.Wegner: *On understanding Types, Data Abstraction and Polymorphism.* To appear.

[MacQueen 85] D.B.MacQueen: *Modules for Standard ML.* In *Polymorphism* II.2 (to appear).

[Milner 84] R.Milner: *A proposal for Standard ML.* Proc. of the 1984 ACM Symposium on Lisp and Functional Programming.

[Mitchell Plotkin 85] J.C.Mitchell, G.D.Plotkin: *Abstract Types have Existential Type,* Proc. POPL '85.

[Reynolds 85] J.C.Reynolds: *Three Approaches to Type Structure.* TAPSOFT Advanced Seminar on the Role of Semantics in Software Development, Berlin, March 25-29, 1985.