Journal of Universal Computer Science, vol. 23, no. 1 (2017), 5-20
submitted: 1/8/16, accepted: 1/12/16, appeared: 28/1/17 © J.UCS

Abstract Data Types without the Types

Philip Wadler
(Edinburgh University, Scotland
wadler@inf.ed.ac.uk)

Dedicated to David Turner on the occasion of his 70'th birthday

Abstract: The data abstraction mechanism of Miranda may be adapted to a dynamically typed programming language by applying ideas from gradual typing.
Key Words: abstract data type, information hiding, gradual typing, Miranda, Haskell
Category: D.1.1, D.3.1, D.3.3, F.3.2

1 Introduction

Oh, what a joy to read a paper by David Turner! The prose flows, the ideas leap. Stay tuned, I will quote him at length shortly.

Turner's Miranda offers a style of data abstraction different from that found in other languages, such as ML, HOPE, and Haskell. Miranda uses type *signatures* where the others use information hiding to enforce *seals*; I will refer to the two approaches respectively as *signing* and *sealing*.

Here is how [Turner(1985)] describes the difference:

> Although the idea of an abstract data type is now standard, the reader will see from the example that the way in which they are presented in Miranda (compared with say ML or HOPE) involves some innovations. The first, and minor deviation is that we have separated the declaration of the signature of an abstract data type from the statement of how it is implemented, treating these as distinct syntactic acts.
>
> More significant is that the abstract data type mechanism used in Miranda does not require the division of the program into two regions (the inside and outside of a capsule, say) such that in one region the programmer has access to conversion functions (called '`abs_theorem`' and '`rep_theorem`', say) permitting him to move at will between the abstract type and its representation, while in the other region these are hidden from him. The mechanism used in Miranda is *transparent*, in that all the identifiers involved are visible throughout the script. The security of the abstract data type here depends, not on the hiding of declared identifiers, but on the fact that explicit acts of conversion between the abstract type and its representation are *nowhere* permitted.

> ...
>
> The advantages (in terms of security and convenience) of having the conversion functions installed by the compiler rather than the user, should be clear. It might be argued however, that this is pushing just too much onto the compiler and will lead to difficulties (perhaps that users will not understand the implications of what they are doing). The method of defining abstract data types must be regarded as one of the more experimental features of Miranda, and only after a period of experience with the language will we be in a position to say where the balance of advantage lies.

The above and all subsequent quotes are from [Turner(1985)]. Citing ML and HOPE while failing to cite Haskell was no slight to the latter, since it was not conceived until two years later. Also note that Standard ML would later offer users a choice of abstracting by either signing or sealing.

Turner goes on to observe:

> Note that the mechanism for data type abstraction which is presented here is inextricably bound up with strong (i.e., compile time) typing. There would seem to be no equivalent mechanism available in a language which delays its type checking until run time. By contrast, the traditional account of data type abstraction as an act of encapsulation would appear to be equally applicable to both strongly and weakly typed languages.

Here by 'weakly typed languages', Turner is referring to what are now called 'dynamically typed' or 'uni-typed languages', which I will refer to as *untyped*; such languages include Racket, Python, JavaScript, and Miranda's predecessor KRC [Turner(1981)]. Any support for data abstraction offered by such language, both then and today, is based on the traditional encapsulation account, usually in conjunction with object-orientation.

The purpose of this paper is to argue that, in one respect, Turner missed the mark: it is in fact possible to apply his mechanism for data type abstraction to an untyped language. The variant of Turner's mechanism for untyped languages that I will describe blends the two techniques, using signing to infer sealing, and is inspired by previous work on gradual typing [Guha et al.(2007), Matthews and Ahmed(2008), Ahmed et al.(2011), Siek and Wadler(2017)].

Two caveats. First, the method I describe works only for abstract data types that do not accept type parameters; extending it to abstract data types with parameters is left to future work. Second, because type constraints are enforced dynamically rather than checked statically, there is additional cost at runtime.

I admit it: I am a fan of Miranda's signing, and find Haskell's sealing to be far clumsier. Why did Haskell adopt sealing rather than signing? Largely

```
wff ::= Var [char] | wff $Implies wff | Not wff

abstype theorem
with axiom1, axiom3 :: wff->wff->theorem
     axiom2 :: wff->wff->wff->theorem
     modus_ponens :: theorem->theorem->theorem
     contents :: theorem->wff

theorem == wff
axiom1 a b = a $Implies (b $Implies a)
axiom2 a b c = (a $Implies (b $Implies c)) $Implies
                 ((a $Implies b) $Implies (a $Implies c))
axiom3 a b = Not (a $Implies b) $Implies (Not b $Implies Not a)
modus_ponens a (a $Implies b) = b
contents x = x
```

Figure 1: Theorem as an ADT in Miranda

because myself and the rest of the Haskell committee were unclear on how signing would interact with type classes, the most innovative feature in Haskell [Wadler and Blott(1989), Hudak et al.(2007)]. Later, my student Jeremy Yallop showed programs using signing can be translated into ones using sealing, and vice versa [Wadler and Yallop(2008), Yallop(2010)]. This means my initial conservatism was unnecessary. If I had it to do over again, I would push for Haskell to adopt a mechanism for abstraction like that found in Miranda.

The paper is organised as follows. Section 2 reviews Turner's mechanism for Miranda, and contrasts it with the mechanism found in Haskell and other languages. Section 3 gives an informal description of how to adapt Turner's mechanism to untyped languages. Section 4 fleshes this out by giving a formal description for a tiny core calculus. Section 5 concludes.

2 Two mechanisms for type abstraction

[Turner(1985)] presents two examples of abstract data types, which I reprise here.

The first is an ingenious implementation of a theorem prover, inspired by [Gordon et al.(1979)]. The theorem example in Miranda is given in Figure 1, and in Haskell in Figure 2. It is best introduced by quoting Turner.

> Suppose we are interested in writing programs to derive theorems in a formal system of inference. Such a system would typically be organised

```
module Theorem(Wff(..), Theorem, axiom1, axiom2, axiom3,
               modus_ponens, contents) where
import Prelude

data Wff = Var String | Wff `Implies` Wff | Not Wff
           deriving (Eq, Show)
newtype Theorem = Thm Wff

axiom1 :: Wff -> Wff -> Theorem
axiom1 a b = Thm (a `Implies` (b `Implies` a))

axiom2 :: Wff -> Wff -> Wff -> Theorem
axiom2 a b c = Thm ((a `Implies` (b `Implies` c)) `Implies`
                     ((a `Implies` b) `Implies` (a `Implies` c)))

axiom3 :: Wff -> Wff -> Theorem
axiom3 a b = Thm (Not (a `Implies` b) `Implies`
                   (Not b `Implies` Not a))

modus_ponens :: Theorem -> Theorem -> Theorem
modus_ponens (Thm a) (Thm (a' `Implies` b)) | a == a'  =  Thm b

contents :: Theorem -> Wff
contents (Thm x) = x
```

Figure 2: Theorem as an ADT in Haskell

as follows. There is a class of *wffs* (well formed formulae), which are correctly formed propositions of the theory. These can be defined by giving a grammar, say. *Theorems* are a distinguished subset of wffs, which are generated inductively from axioms by using rules of inference. For example in the standard formulation of propositional logic, there is an axiom which says that for any wffs A B, it is a theorem that: A implies (B implies A). There are two more axioms, and a single rule of inference (modus ponens) which enables us to derive new theorems from existing ones.

We would like to use the type system to guarantee that a well typed program cannot, even accidentally, make an invalid inference.

The trick is to introduce `theorem` as an abstract data type. Each `theorem` is represented as a `wff`. However, the only ways to introduced a `theorem` is via three axioms, each of which combines two or three `wff`s to yield a `theorem`, or via the inference rule `modus_ponens`, which combines two `theorem`s to yield a `theorem`. Finally, `contents` allows us to extract the underlying `wff` from a `theorem`.

Turner describes abstraction vividly:

> `Theorem` is an abstract data type based on `wff`. A theorem looks like a wff, but has been lifted to a higher world (think of it as being dyed blue). The entrances to this higher world are closely guarded (as in general are the exits, although that is not relevant in this example.) The only way to create a blue object is either by using an axiom, or by applying a rule of inference to objects that are already blue.

In both Miranda and Haskell, the functions making up the abstract data type have identical type signatures. However, in Miranda the signatures are required, and enforce the distinction between `wff` and `theorem`, while in Haskell the signatures may be inferred, and the key distinction is enforced by using the constructor `Thm`, which when used in a term converts a `Wff` to a `Theorem`, and in a pattern converts a `Theorem` to a `Wff`. The secret to abstraction in Haskell is that the concrete type `Wff` is exported from the module along with its three constructors (indicated by writing '`Wff(..)`' in the declaration in the first line) while the abstract type `Theorem` is exported without its constructor (indicated by writing '`Theorem`' with no following '`(..)`', a rather subtle distinction to convey such a crucial difference). The Miranda code is arguably easier to read, precisely because it keeps separate information that is interleaved in Haskell; in particular, the Haskell code is spotted with appearances of the constructor `Thm`. Incidental differences between the two are that Miranda implicitly defines equality over `wff` while Haskell requires an explicit declaration for `Wff` (via `deriving Eq`), and Miranda implicitly invokes equality via pattern matching (two appearances of `a` on the left hand side of the equation defining modus ponens) while Haskell requires explicit invocation (appearances of `a` and `a'` together with `a == a'`).

The second example is the 'hello world' of abstract data types, the stack. Again, the stack example in Miranda is given in Figure 3, and in Haskell in Figure 4. Turner's original example used polymorphic stacks, but here we restrict to stacks of numbers. Stacks are given the obvious implementation in terms of lists. Again, Miranda is arguably a little clearer than Haskell.

Turner outlined how to implement his mechanism (quoted verbatim).

> How to typecheck a script containing an `abstype` declaration (sketch):- First we use the binding of the abstract type to its representation type to compute the concrete signature from the abstract signature. The binding of the abstract type to the representation type is then suppressed

```
abstype stack
with empty :: stack
     isempty :: stack->bool
     push :: num->stack->stack
     pop :: stack->stack
     top :: stack->num

stack == [num]
empty = []
isempty x = (x=[])
push a x = a:x
pop(a:x) = x
top(a:x) = a
```

Figure 3: Stack as an ADT in Miranda

```
module Stack(Stack, empty, isempty, push, pop, top) where
import Prelude

newtype Stack = MkStk [Int]

empty :: Stack
empty = MkStk []

isempty :: Stack -> Bool
isempty (MkStk x) = null x

push :: Int -> Stack -> Stack
push a (MkStk x) = MkStk (a:x)

pop :: Stack -> Stack
pop (MkStk (a:x)) = MkStk x

top :: Stack -> Int
top (MkStk (a:x)) = a
```

Figure 4: Stack as an ADT in Haskell

> — from now on 'theorem' and 'wff' (or 'stack' and 'list', or whatever) are treated as two distinct and unrelated types throughout the script. Each identifier in the signature now has two types, a concrete type and an abstract type. When typechecking the implementation equations each such identifier is regarded as having been declared with its concrete type; when typechecking the rest of the script (i.e., outside the implementation equations) it is regarded as having been declared with its abstract type. All other identifiers in the script (i.e., those not listed in the signature) are treated as having the same type everywhere. (end of sketch)

In Miranda, the compiler can not only convert between `theorem` and `wff` at no cost, but also between structures involving the two, say `[theorem]` and `[wff]`. In Haskell, careful design of the compiler ensures that the former conversion also has no cost, but until recently the latter required mapping a function over the list, which does have a cost. To eliminate that cost, [Breitner et al.(2014)] introduce the `Coercible` class to Haskell, but this introduces a significant complication that is absent in Miranda.

Turner's mechanism depends crucially upon static typechecking to enforce data type abstraction. So it may come as a surprise that similar ideas may apply in an untyped language, as described in the next section.

3 Turner's mechanism in an untyped language, informally

I will first give an informal description of how to adapt Turner's mechanism to an untyped language, followed by a formal description for a tiny core language.

I will describe the mechanism in an untyped variant of Miranda, similar to KRC but also adapting algebraic data types from Miranda. For example, the definition of `wff` from Figure 1 is still permitted, and introduces the constructors `Var`, `Implies`, and `Not` as before. The type declaration now specifies constraints which are to be enforced dynamically rather than statically. Thus, whenever `Var` is applied to an argument, it is checked at runtime that the argument is a list of characters. (This dynamic check is, of course, more expensive than performing the check at compile time, since it takes time proportional to the length of the list.) Similarly, when `Implies` is applied it checks that both its arguments belong to type `wff`, and when `Not` is applied it checks that its argument belongs to type `wff`, where a value belongs to type `wff` when it is built using one of the constructors `Var`, `Implies`, or `Not`.

In this variant of Miranda, the abstract type `theorem` may be defined almost exactly as before. Our new programme is identical to Figure 1, save that the line

```
theorem == wff
```

is omitted. Untyped languages are sometimes called *uni-typed* because we may consider all values as having the same type. The type binding of `theorem` is not required because the representation type of `theorem`, just like of everything else, is this one type (which includes values of type `wff` along with all other possible values).

The signatures provided for the functions that implement the `theorem` abstract type are dynamically checked at runtime. We have already seen how to dynamically check that an argument or result is of type `wff`. The type `theorem`, being declared as abstract, is treated differently. A new dynamic constructor corresponding to the type `theorem` is introduced; let's call it `Thm`, since it plays the same role as the constructor of that name in the Haskell program in Figure 2. The type signature provided for each function guides sealing and unsealing with this constructor. Whenever type `theorem` appears in a positive position in the signature (as the final result, or to the left of an even number of arrows) the corresponding value is sealed using `Thm`. Wherever type `theorem` appears in a negative position in the signature (to the left of an odd number of arrows) it checks the corresponding argument was sealed using `Thm` and unseals the argument; an attempt to unseal with `Thm` a value that was not sealed using `Thm` raises an error at run time.

Hence, a call to one of the three axioms dynamically checks that each argument belongs to `wff`, applies the function body, and dynamically seals the result with constructor `Thm`. A call to modus ponens dynamically checks that both arguments are sealed with `Thm`, unseals the arguments, applies the function body, and then seals the result with `Thm`. A call to contents dynamically checks that its argument is sealed with Thm, unseals the argument, applies the function body, and dynamically checks that the result belongs to wff. (Again, all this dynamic checking, sealing, and unsealing is more expensive than performing the checks at compile time.)

Similarly, the abstract type `stack` may be defined almost exactly as in Figure 3, save that the line

```
stack == [num]
```

is omitted. A new constructor corresponding to `stack` is introduced; let's call it `Stk`. A call to `empty` seals its result with the constructor Stk. A call to `push` or `pop` checks that the relevant argument is sealed with `Stk`, applies the function body, and seals the result with `Stk`. A call to `isempty` or `top` checks that the argument is sealed with `Stk`, applies the function body, and checks that the result is `bool` or `num`, respectively.

Additional checking may correspond to polymorphism in type signatures. Checking for polymorphism is dual to checking for abstract data types. Each polymorphic type variable in a signature corresponds to a fresh constructor, allocated each time the signature is checked. Whenever a polymorphic type variable appears in a negative position in the signature the corresponding argument is sealed with the corresponding constructor, and whenever a polymorphic type variable appears in a positive position in the signature it means the corresponding value must have been sealed using the corresponding constructor and is now unsealed. Note the duality: abstraction seals at even positions and unseals at odd positions, whereas polymorphism seals at odd positions and unseals at even positions.

As an example, say that a signature is supplied for the constant function in our untyped language.

$$\texttt{const} :: \texttt{*} \to \texttt{**} \to \texttt{*}$$
$$\texttt{const}\;\texttt{x}\;\texttt{y} = \texttt{x}$$

The type signature for `const` corresponds to the universally quantified type $\forall X. \forall Y. X \to Y \to X$. Each dynamic call to const allocates two new constructors, lets call them α and β. The first argument is sealed with α and the second argument is sealed with β, the function body is applied, and then result is unsealed with α.

A fundamental semantic property of polymorphic types is *relational parametricity*, introduced by [Reynolds(1983)] and popularised by [Wadler(1989)] under the slogan "Theorems for free"; see also [Wadler(2007)]. For instance, any function of type $\forall X. \forall Y. X \to Y \to X$ must be either the constant function that returns its first argument and ignores its second, or the undefined function that ignores both arguments and always loops. Our system has the remarkable property that it guarantees relational parametricity holds for any term given a polymorphic type, even though the type is enforced by dynamic checking rather than static checking. Our technique for ensuring this property builds upon related work by [Guha et al.(2007)], [Matthews and Ahmed(2008)], [Ahmed et al.(2011)], and [Siek and Wadler(2017)].

Relational parametricity is a strong property and imposes strong constraints on the programming language. Miranda does not quite satisfy it because it defines equality at every type, whereas Haskell does satisfy it because it constrains equality using type classes [Wadler(1989)]. So our claims of relational parametricity hold only if equality (and other ad-hoc polymorphic functions) are suitably constrained, such as by use of type classes.

4 Turner's mechanism in an untyped language, formally

Let's now turn to a formal development that captures the essence of the informal exposition given above, based on a tiny core language.

To model algebraic data types, such as wff, I introduce types of the form

$$\Sigma_i \kappa_i(\vec{A}_i)$$

where κ_i ranges over value constructors (such as Var, Implies, and Not) and $\vec{A}_i$ ranges over sequences of types. Fixpoints model recursive types. For example, the wff type is modelled by

$$\texttt{wff} = \mu X.\, \texttt{Var}(\texttt{[char]}) + \texttt{Implies}(X, X) + \texttt{Not}(X)$$

or, to precisely correspond to the notation above,

$$\texttt{wff} = \mu X.\, \Sigma_i \kappa_i(\vec{A}_i)$$

where i ranges from 1 to 3 with $\kappa_1 = \texttt{Var}$, $A_1 = [\texttt{[char]}]$, $\kappa_2 = \texttt{Implies}$, $A_2 = [X, X]$, $\kappa_3 = \texttt{Not}$, $A_3 = [X]$.

Abstract data types are modelled by existential quantification while polymorphic data types are modelled by universal quantification. For example, the stack type in Figure 3 is modelled by

$$\exists X.\, (X \times (X \to \texttt{bool}) \times (\texttt{num} \to X \to X) \times (X \to X) \times (X \to \texttt{num})$$

and the type of const is modelled by

$$\forall X.\, \forall Y.\, X \to Y \to X.$$

Whereas X, Y range over type variables bound by quantifiers, we let α, β range over seals, which freshly allocated as part of dynamic checking of quantifiers. Type $\{-\alpha\}$ corresponds to sealing and type $\{+\alpha\}$ corresponds to unsealing; our choice of signs aligns with that in [Siek and Wadler(2017)].

We include `any` as a general type that matches any value. The formalism does not include separate constructs for products (or tuples), but these are easily modelled using suitable constructors. We only model the base type `num`, but it is easy to include others.

The syntax and reduction rules of the core are summarised in Figure 5.

Let A, B range over types. A type is either the general type `any`, a base type such as `num`, a function type $A \to B$, a sum of constructors $\Sigma_i \kappa_i(\vec{A}_i)$, a type variable X, a recursive type $\mu X.\, A$, a universal type $\forall X.\, A$, an existential type $\exists X.\, A$, a seal $-\{\alpha\}$, or an unseal $+\{\alpha\}$.

Let L, M, N range over terms. Terms are as usual, including numerical constants n, variables x, function abstraction $\lambda x.\, N$, function application $L\, M$, the form $\kappa(\vec{V})$ for construction, and a case expression

$$\texttt{case}\ L\ \texttt{of}\ \{\kappa_i(\vec{x}) \to N_i\}_i$$

for deconstruction. Most importantly, we add the form

$$M\ \texttt{@}\ A$$

Syntax

$$
\begin{aligned}
A, B &::= \texttt{any} \mid \texttt{num} \mid A \to B \mid \Sigma_i \kappa_i(\vec{A}) \mid X \mid \mu X.\, A \\
&\qquad \mid \exists X.\, A \mid \forall X.\, A \mid \{+\alpha\} \mid \{-\alpha\} \\
L, M, N &::= n \mid x \mid \lambda x.\, N \mid L\, M \mid C(\vec{M}) \mid \texttt{case}\ L\ \texttt{of}\ \{\kappa_i(\vec{x}) \to N_i\}_i \mid M \mathbin{\texttt{@}} A \\
V, W &::= n \mid x \mid \lambda x.\, N \mid \kappa(\vec{V}) \mid V \mathbin{\texttt{@}} A \to B \mid V \mathbin{\texttt{@}} \{-\alpha\} \\
\mathcal{E} &::= \Box \mid \mathcal{E}\, N \mid V\, \mathcal{E} \mid C(\vec{V}, \mathcal{E}, \vec{M}) \mid \mathcal{E} \mathbin{\texttt{@}} A
\end{aligned}
$$

Reduction

$$
\begin{aligned}
(\lambda x.\, N)\, V &\longrightarrow N[x := V] \\
\texttt{case}\ \kappa_k(\vec{V})\ \texttt{of}\ \{\kappa_i(\vec{x}_i) \to N_i\}_i &\longrightarrow N_k[\vec{x}_k := \vec{V}] \\
M \mathbin{\texttt{@}} \texttt{any} &\longrightarrow M \\
n \mathbin{\texttt{@}} \texttt{num} &\longrightarrow n \\
(V \mathbin{\texttt{@}} A \to B)\, W &\longrightarrow V\, (W \mathbin{\texttt{@}} -A) \mathbin{\texttt{@}} B \\
\kappa_j(\vec{V}) \mathbin{\texttt{@}} \Sigma_i \kappa_i(\vec{A}_i) &\longrightarrow \kappa_j(\vec{V} \mathbin{\texttt{@}} \vec{A}) \\
V \mathbin{\texttt{@}} \mu X.\, A &\longrightarrow V \mathbin{\texttt{@}} A[X := \mu X.\, A] \\
V \mathbin{\texttt{@}} \exists X.\, A &\longrightarrow \Delta,\ V \mathbin{\texttt{@}} A[Y := \{-\alpha\}], \qquad \text{fresh } \alpha \\
V \mathbin{\texttt{@}} \forall X.\, A &\longrightarrow \Delta,\ V \mathbin{\texttt{@}} A[Y := \{+\alpha\}], \qquad \text{fresh } \alpha \\
(V \mathbin{\texttt{@}} \{-\alpha\}) \mathbin{\texttt{@}} \{+\alpha\} &\longrightarrow V
\end{aligned}
$$

$$
\frac{M \longrightarrow N}{\mathcal{E}[M] \longrightarrow \mathcal{E}[N]}
$$

Seal negation

$$
\begin{aligned}
-\texttt{any} &= \texttt{any} & -\texttt{num} &= \texttt{num} \\
-(A \to B) &= (-A) \to (-B) & -(\Sigma_i \kappa_i(\vec{A}_i)) &= \Sigma_i \kappa_i(-\vec{A}_i) \\
-X &= X & -(\mu X.\, A) &= \mu X.\, (-A) \\
-(\exists X.\, A) &= \exists X.\, (-A) & -(\forall X.\, A) &= \forall X.\, (-A) \\
-\{+\alpha\} &= \{-\alpha\} & -\{-\alpha\} &= \{+\alpha\}
\end{aligned}
$$

Figure 5: Untyped lambda calculus with contracts

to indicate that term M is dynamically checked to conform to type A. Following terminology of the Racket community, we refer to the form $M \mathbin{\texttt{@}} A$ as a *contract*.

Although Miranda and Haskell are both call-by-need, I give a call-by-value formulation of the core as it is more straightforward. It is easy to adapt a call-by-value calculus to be call-by-need [Ariola et al.(1995), Maraist et al.(1998)].

Let V, W range over values. As usual values include numerical constants n, function abstraction $\lambda x. N$, and constructors over values $\kappa(\vec{V})$. We also take as values function contracts $V \mathbin{\texttt{@}} A \to B$, and seal contracts $V \mathbin{\texttt{@}} \{-\alpha\}$.

Let $\mathcal{E}$ range over evaluation contexts, which are standard. The operational semantics presented is small-step, based on reductions of the form $M \longrightarrow N$. Reductions are closed under evaluation contexts.

The reduction rules are as follows. Reduction of function applications and case expressions is standard.

A term with contract `any` reduces to itself.

A numerical constant with contract `num` reduces to itself.

A function V with contract $A \to B$ when applied to a value W reduces to a term that applies contract $-A$ to the argument, applies the function, and then applies contract B to the result. The domain contract is negated while the range contract is not, corresponding to the fact that functions are contravariant in their domain and covariant in their range. Similar rules are found in many works on contracts and gradual typing, going back to the seminal paper of [Findler and Felleisen(2002)]. The negation of a type changes seals to unseals and vice versa. It leaves all other forms of type unchanged.

A recursive type in a contract is unfolded in the usual way. An existential causes the bound type variable of the quantifier to be instantiated to $-\alpha$, where α is a fresh seal. Dually, a universal causes the bound type variable of the quantifier to be instantiated to $+\alpha$, where α is a fresh seal. Finally, unsealing a sealed value reduces to the original value.

All other reductions become stuck, and in practice would signal an error. For instance, this would happen if attempting to apply a value that is not a function, or if contract `num` acts of a value that is not a numerical constant.

Two examples are shown in Figure 6.

The first demonstrates existential quantification, and is based on a simplification of the stack example from Figures 3 and 4. Take `empty`, `push`, and `top` to be as defined in those figures, and let `empty′`, `push′` and `top′` be the same functions wrapped in an appropriate type signature (that is, corresponding to the functions `empty`, `push`, and `top` as they appear anywhere in the script outside of the definition of the abstract data type). It shows how sealing and unsealing of the abstract stack type proceeds in computing the term `top′ (push′ 2 empty′)`. Observe that `top` behaves identically to the function `head` that extracts the first element of a list. But if `top′` is replaced by `head` the argument stack will not be

Existential quantification

$$\begin{aligned}
&\texttt{let}\ (\texttt{top}', \texttt{push}', \texttt{empty}') = \\
&\quad (\texttt{top}, \texttt{push}, \texttt{empty})\ @\ \exists X.\ ((X \to \texttt{num}) \times (\texttt{num} \to X \to X) \times X) \\
&\texttt{in}\ \texttt{top}'\ (\texttt{push}'\ 2\ \texttt{empty}')
\end{aligned}$$

$$\begin{aligned}
&\longrightarrow (\texttt{top}\ @\ \{-\alpha\} \to \texttt{num})\ ((\texttt{push}\ @\ \texttt{num} \to \{-\alpha\} \to \{-\alpha\})\ 2\ (\texttt{empty}\ @\ \{-\alpha\})) \\
&\longrightarrow (\texttt{top}\ @\ \{-\alpha\} \to \texttt{num})\ ((\texttt{push}\ @\ \texttt{num} \to \{-\alpha\} \to \{-\alpha\})\ 2\ (\texttt{[]}\ @\ \{-\alpha\})) \\
&\longrightarrow (\texttt{top}\ @\ \{-\alpha\} \to \texttt{num})\ (\texttt{push}\ (2\ @\ \texttt{num})\ @\ \{-\alpha\} \to \{-\alpha\})\ (\texttt{[]}\ @\ \{-\alpha\})) \\
&\longrightarrow (\texttt{top}\ @\ \{-\alpha\} \to \texttt{num})\ (\texttt{push}\ 2\ @\ \{-\alpha\} \to \{-\alpha\})\ (\texttt{[]}\ @\ \{-\alpha\})) \\
&\longrightarrow (\texttt{top}\ @\ \{-\alpha\} \to \texttt{num})\ (\texttt{push}\ 2\ ((\texttt{[]}\ @\ \{-\alpha\})\ @\ \{+\alpha\})\ @\ \{-\alpha\}) \\
&\longrightarrow (\texttt{top}\ @\ \{-\alpha\} \to \texttt{num})\ (\texttt{push}\ 2\ \texttt{[]}\ @\ \{-\alpha\}) \\
&\longrightarrow (\texttt{top}\ @\ \{-\alpha\} \to \texttt{num})\ (\texttt{[2]}\ @\ \{-\alpha\}) \\
&\longrightarrow \texttt{top}\ ((\texttt{[2]}\ @\ \{-\alpha\})\ @\ \{+\alpha\})\ @\ \texttt{num} \\
&\longrightarrow \texttt{top}\ \texttt{[2]}\ @\ \texttt{num} \\
&\longrightarrow 2\ @\ \texttt{num} \\
&\longrightarrow 2
\end{aligned}$$

Universal quantification

$$\begin{aligned}
&((\lambda x.\, \lambda y.\, x)\ @\ \forall X.\, \forall Y.\, X \to Y \to X)\ 2\ 3 \\
\longrightarrow &((\lambda x.\, \lambda y.\, x)\ @\ \forall Y.\, \{+\alpha\} \to Y \to \{+\alpha\})\ 2\ 3 \\
\longrightarrow &((\lambda x.\, \lambda y.\, x)\ @\ \{+\alpha\} \to \{+\beta\} \to \{+\alpha\})\ 2\ 3 \\
\longrightarrow &((\lambda x.\, \lambda y.\, x)\ (2\ @\ \{-\alpha\})\ @\ \{+\beta\} \to \{+\alpha\})\ 3 \\
\longrightarrow &(\lambda x.\, \lambda y.\, x)\ (2\ @\ \{-\alpha\})\ (3\ @\ -\beta)\ @\ \{+\alpha\} \\
\longrightarrow &(2\ @\ \{-\alpha\})\ @\ \{+\alpha\} \\
\longrightarrow &2
\end{aligned}$$

Figure 6: Example reductions

unsealed, and the computation will get stuck, as we should expect.

The second demonstrates universal quantification, and is based on the constant function as described in the previous section. It shows how sealing and unsealing proceeds in computing $\lambda x.\, \lambda y.\, x$ contracted at type $\forall X.\, \forall Y.\, X \to Y \to X$ and applied to 2 and 3. Recall that relational parametricity guarantees that any function of type $\forall X.\, \forall Y.\, X \to Y \to X$ must either be the constant function that returns its first argument and ignores its second, or the undefined function that ignores both arguments and always loops. Observe that if $\lambda x.\,\lambda y.\,x$ is replaced by $\lambda x.\,\lambda y.\,y$ then the final result will be $(3\,@\,\{-\beta\})\,@\,\{+\alpha\}$. Since the sealing and unsealing variables do not match the computation will get stuck, as we should expect.

5 Conclusion

The novel data abstraction mechanism of Miranda arguably leads to code that is easier to read than that written using the more common data abstraction mechanism of Haskell and other languages. Turner's mechanism depends crucially on type signatures, so it is surprising that it can be applied in an untyped language using dynamic rather than static checking. I presented a design based on ideas from gradual typing, and formalised it with a tiny core calculus. The design does not include abstract data types with parameters, which remains an area for future work. Just as Turner argued that 'a period of experience' would be required to judge his new mechanism for abstraction, I look forward to experiments to evaluate the ideas described here.

I leave you with a final quote from Turner, and a last surprise regarding data abstraction.

> It is interesting to note that if you take the complete Miranda script containing an abstract date type declaration like that of [Figure 1] and remove from it just the '`abstype` ... `with` `<signature>`' part, leaving everything else intact, including the implementation equations, the resulting script is still well-typed and describes exactly the same computations as before, but now has a coarser type structure — '`theorem`' has collapsed back into '`wff`'.
>
> This observation seems to throw light on the real purpose of introducing type abstractions into our program. It is to provide the compiler with more information about what we are doing, so that it can impose a finer type structure on the program. (So we see that here, data type abstraction should not be thought of as a matter of hiding information — quite the reverse.)

Acknowledgements

I thank Simon Peyton Jones, Jeremy Siek, Jack Williams, Jeremy Yallop, and the referees for their comments. This research was supported by EPSRC Programme Grant EP/K034413/1.

References

[Ahmed et al.(2011)Ahmed, Findler, Siek and Wadler] Ahmed, A., Findler, R. B., Siek, J. G., Wadler, P.: "Blame for all"; Principles of Programming Languages (POPL); 201–214; 2011.

[Ariola et al.(1995)Ariola, Maraist, Odersky, Felleisen and Wadler] Ariola, Z. M., Maraist, J., Odersky, M., Felleisen, M., Wadler, P.: "A call-by-need lambda calculus"; Principles of Programming Languages (POPL); 233–246; 1995.

[Breitner et al.(2014)Breitner, Eisenberg, Jones and Weirich] Breitner, J., Eisenberg, R. A., Jones, S. P., Weirich, S.: "Safe zero-cost coercions for haskell"; International Conference on Functional Programming (ICFP); 189–202; 2014.

[Findler and Felleisen(2002)] Findler, R. B., Felleisen, M.: "Contracts for higher-order functions"; International Conference on Functional Programming (ICFP); 48–59; 2002.

[Gordon et al.(1979)Gordon, Milner and Wadsworth] Gordon, M., Milner, R., Wadsworth, C.: Edinburgh LCF: a mechanized logic of computation; volume 78 of Lecture Notes in Computer Science; 1979.

[Guha et al.(2007)Guha, Matthews, Findler and Krishnamurthi] Guha, A., Matthews, J., Findler, R. B., Krishnamurthi, S.: "Relationally-parametric polymorphic contracts"; Dynamic Languages Symposium (DLS); 29–40; 2007.

[Hudak et al.(2007)Hudak, Hughes, Peyton Jones and Wadler] Hudak, P., Hughes, J., Peyton Jones, S., Wadler, P.: "A history of haskell: being lazy with class"; History of Programming Languages (HOPL); 12–1–12–55; ACM, 2007.

[Maraist et al.(1998)Maraist, Odersky and Wadler] Maraist, J., Odersky, M., Wadler, P.: "The call-by-need lambda calculus"; Journal of Functional Programming; 8 (1998), 3, 275–317.

[Matthews and Ahmed(2008)] Matthews, J., Ahmed, A.: "Parametric polymorphism through run-time sealing"; European Symposium on Programming (ESOP); 16–31; 2008.

[Reynolds(1983)] Reynolds, J. C.: "Types, abstraction and parametric polymorphism"; IFIP Congress; 513–523; North-Holland, 1983.

[Siek and Wadler(2017)] Siek, J., Wadler, P.: "Conversions and casts: compare and contrast"; (2017); unpublished draft.

[Turner(1981)] Turner, D. A.: "The semantic elegance of applicative languages"; Functional Programming Languages and Computer Architecture (FPCA); 85–92; 1981.

[Turner(1985)] Turner, D. A.: "Miranda: A non-strict functional language with polymorphic types"; Functional Programming Languages and Computer Architecture (FPCA); volume 201 of Lecture Notes in Computer Science; 1–16; Springer, 1985.

[Wadler(1989)] Wadler, P.: "Theorems for free"; Functional Programming Languages and Computer Architecture (FPCA); 1989.

[Wadler(2007)] Wadler, P.: "The girard-reynolds isomorphism (second edition)"; Theoretical Computer Science; 375 (2007), 1–3, 201–226.

[Wadler and Blott(1989)] Wadler, P., Blott, S.: "How to make ad-hoc polymorphism less ad hoc"; Principles of Programming Languages (POPL); 60–76; 1989.

[Wadler and Yallop(2008)] Wadler, P., Yallop, J.: "Signed and sealed"; (2008); unpublished draft.

[Yallop(2010)] Yallop, J.: Abstractions for web programming; Ph.D. thesis; University of Edinburgh (2010).

References

[Ahmed et al.(2011)] Ahmed, A., Findler, R. B., Siek, J. G., Wadler, P.: "Blame for all"; Principles of Programming Languages (POPL); 201–214; 2011.

[Ariola et al.(1995)] Ariola, Z. M., Maraist, J., Odersky, M., Felleisen, M., Wadler, P.: "A call-by-need lambda calculus"; Principles of Programming Languages (POPL); 233–246; 1995.

[Breitner et al.(2014)] Breitner, J., Eisenberg, R. A., Jones, S. P., Weirich, S.: "Safe zero-cost coercions for haskell"; International Conference on Functional Programming (ICFP); 189–202; 2014.

[Findler and Felleisen(2002)] Findler, R. B., Felleisen, M.: "Contracts for higher-order functions"; International Conference on Functional Programming (ICFP); 48–59; 2002.

[Gordon et al.(1979)] Gordon, M., Milner, R., Wadsworth, C.: Edinburgh LCF: a mechanized logic of computation; volume 78 of Lecture Notes in Computer Science; 1979.

[Guha et al.(2007)] Guha, A., Matthews, J., Findler, R. B., Krishnamurthi, S.: "Relationally-parametric polymorphic contracts"; Dynamic Languages Symposium (DLS); 29–40; 2007.

[Hudak et al.(2007)] Hudak, P., Hughes, J., Peyton Jones, S., Wadler, P.: "A history of haskell: being lazy with class"; History of Programming Languages (HOPL); 12–1–12–55; ACM, 2007.

[Maraist et al.(1998)] Maraist, J., Odersky, M., Wadler, P.: "The call-by-need lambda calculus"; Journal of Functional Programming; 8 (1998), 3, 275–317.

[Matthews and Ahmed(2008)] Matthews, J., Ahmed, A.: "Parametric polymorphism through run-time sealing"; European Symposium on Programming (ESOP); 16–31; 2008.

[Reynolds(1983)] Reynolds, J. C.: "Types, abstraction and parametric polymorphism"; IFIP Congress; 513–523; North-Holland, 1983.

[Siek and Wadler(2017)] Siek, J., Wadler, P.: "Conversions and casts: compare and contrast"; (2017); unpublished draft.

[Turner(1981)] Turner, D. A.: "The semantic elegance of applicative languages"; Functional Programming Languages and Computer Architecture (FPCA); 85–92; 1981.

[Turner(1985)] Turner, D. A.: "Miranda: A non-strict functional language with polymorphic types"; Functional Programming Languages and Computer Architecture (FPCA); volume 201 of Lecture Notes in Computer Science; 1–16; Springer, 1985.

[Wadler(1989)] Wadler, P.: "Theorems for free"; Functional Programming Languages and Computer Architecture (FPCA); 1989.

[Wadler(2007)] Wadler, P.: "The girard-reynolds isomorphism (second edition)"; Theoretical Computer Science; 375 (2007), 1–3, 201–226.

[Wadler and Blott(1989)] Wadler, P., Blott, S.: "How to make ad-hoc polymorphism less ad hoc"; Principles of Programming Languages (POPL); 60–76; 1989.

[Wadler and Yallop(2008)] Wadler, P., Yallop, J.: "Signed and sealed"; (2008); unpublished draft.

[Yallop(2010)] Yallop, J.: Abstractions for web programming; Ph.D. thesis; University of Edinburgh (2010).