Lecture Notes on Data Abstraction

15-814: Types and Programming Languages
Frank Pfenning

Lecture 14
October 23, 2018

1 Introduction

Since we have moved from the pure λ-calculus to functional programming languages we have added rich type constructs starting from functions, disjoint sums, eager and lazy pairs, recursive types, and parametric polymorphism. The primary reasons often quoted for such a rich static type system are discovery of errors before the program is ever executed and the efficiency of avoiding tagging of runtime values. There is also the value of the types as documentation and the programming discipline that follows the prescription of types. Perhaps more important than all of these is the strong guarantees of data abstraction that the type system affords that are sadly missing from many other languages. Indeed, this was one of the original motivation in the development of ML (which stands for MetaLanguage) by Milner and his collaborators [GMM+78]. They were interested in developing a theorem prover and wanted to reduce its overall correctness to the correctness of a trusted core. To this end they specified an *abstract type* of *theorem* on which the only allowed operations are inference rules of the underlying logic. The connection between abstract types and existential types was made made Mitchell and Plotkin [MP88].

2 Signatures and Structures

Data abstraction in today's programming languages is usually enforced at the level of modules (if it is enforced at all). As a running example we consider a simple module providing and implementation of numbers with

constant *zero* and functions *succ* and *pred*. We will consider two implementations and their relationship. One is using numbers in unary form (type *nat*) and numbers in binary form (type *bin*), and we will eventually prove that they are logically equivalent. We are making up some syntax (loosely based on ML), specify interfaces between a library and its client.

Below we name `NUM` as the *signature* that describes the interface of a module.

```
NUM = {
  type Num
  zero : Num
  succ : Num -> Num
  pred : Num -> Option Num
}
```

The function `pred` returns a `Option Num` since we consider the predecessor of `zero` to be undefined. Recall the option type

```
  data Option a = Null | Just a
```

For the implementations, we use the following types for numbers in unary and binary representation.

```
data Nat = Z | S Nat
data Bin = E | B0 Bin | B1 Bin
```

Then we define the first implementation

```
NAT : NUM = {
  type Num = Nat

  zero = Z

  succ n = S n

  pred Z = Null
  pred (S n) = Just n
}
```

An interesting aspect of this definition is that, for example, `zero : Nat` while the interface specifies `zero : Num`. But this is okay because the type `Num` is in fact implemented by `Nat` in this version. Next, we show the implementation using numbers in binary representation. It is helpful to have a function `map` operating on optional values.

```
map : (a -> b) -> Option a -> Option b
map f Null = Null
map f (Just x) = Just (f x)

BIN : NUM = {
  type Num = Bin

  zero = E

  succ E = B1 E
  succ (B0 x) = B1 x
  succ (B1 x) = B0 (succ x)

  pred E = Null
  pred (B1 x) = Just (B0 x)
  pred (B0 x) = map B1 (pred x)
}
```

Now what does a client look like? Assume it has an implemention `N : NUM`. It can then "open" or "import" this implementation to use its components, but it will not have any knowledge about the type of the implementation. For example, we can write

```
open N : NUM

isZero : Num -> Bool
isZero x = case pred x
              Null => True
              Just y => False
```

but **not**

```
open N : NUM

isZero : Num -> Bool
isZero Z = true          % type error here: Nat not equal Num
isZero (S n) = false     % and here
```

because the latter supposes that the library `N : NUM` implements the type `Num` by `Nat`, which it may not.

3 Formalizing Abstract Types

We will write a signature such as

```
NUM = {
  type Num
  zero : Num
  succ : Num -> Num
  pred : Num -> Option Num
}
```

in abstract form as

$$\exists\alpha.\ \underbrace{\alpha}_{\textit{zero}} \otimes \underbrace{(\alpha \to \alpha)}_{\textit{succ}} \otimes \underbrace{(\alpha \to \alpha\ \mathbf{option})}_{\textit{pred}}$$

where the name annotations are just explanatory and not part of the syntax. Note that α stands for *Num* which is bound here by the existential quantifier, just as we would expect the scope of `Num` in the signature to only include the three specified components.

Now what should an expression

$$e : \exists\alpha.\, \alpha \otimes (\alpha \to \alpha) \otimes (\alpha \to \alpha\ \mathbf{option})$$

look like? It should provide a concrete type (such as *nat* or *bin*) for α, as well as an implementation of the three functions. We obtain this with the following rule

$$\frac{\Delta \vdash \sigma\ \textit{type} \quad \Delta\ ;\ \Gamma \vdash e : [\sigma/\alpha]\tau}{\Delta\ ;\ \Gamma \vdash \langle \sigma, e\rangle : \exists\alpha.\,\tau}\ \text{(I-}\exists\text{)}$$

Besides checking that σ is indeed a type with respect to all the type variables declared in Δ, the crucial aspect of this rule is that the implementation e is at type $[\sigma/\alpha]\tau$.

For example, to check that *zero*, *succ*, and *pred* are well-typed we substitute the implementation type for `Num` (namely `Nat` in one case and `Bin` in the other case) before proceeding with checking the definitions.

The pair $\langle \sigma, e\rangle$ is sometimes referred to as a *package*, which is opened up by the destructor. This destructor is often called **open**, but for uniformity with all analogous cases we'll write is as a **case**.

Types	τ	$::=$	$\ldots \mid \exists\alpha.\,\tau$
Expressions	e	$::=$	$\ldots \mid \langle \sigma, e\rangle \mid \mathbf{case}\ e\ \{\langle \alpha, x\rangle \Rightarrow e'\}$

The elimination form provides a new name α for the implementation types and a new variable x for the (eager) pair making up the implementations.

$$\frac{\Delta\ ;\ \Gamma \vdash e : \exists\alpha.\,\tau \quad \Delta, \alpha\ \mathit{type}\ ;\ \Gamma, x : \tau \vdash e' : \tau'}{\Delta\ ;\ \Gamma \vdash \mathbf{case}\ e\ \{\langle \alpha, x\rangle \Rightarrow e'\} : \tau'}\ (\text{E-}\exists)$$

The fact that the type α must be *new* is implicit in the rule in the convention that Δ may not contain an repeated variables. If we happened to have used the name α before then we can just rename it and then apply the rule. It is crucial for data abstraction that this variable α is new because we cannot and should not be able to assume anything about what α might stand for, except the operations that might be exposed in τ and are accessible via the name x.

We also see that the client e' is *parametric* in α, which means that it cannot depend on what α might actually be at runtime. It is this parametricity that will allow us to swap one implementation out for another without affecting the client as long as the two implementations are equivalent in an appropriate sense.

The operational rules are straightforward and not very interesting.

$$\frac{v\ \mathit{val}}{\langle \sigma, v\rangle\ \mathit{val}}\ (\text{V-}\exists)$$

$$\frac{e \mapsto e'}{\langle \sigma, e\rangle \mapsto \langle \sigma, e'\rangle}\ (\text{CI-}\exists) \qquad \frac{e_0 \mapsto e_0'}{\mathbf{case}\ e_0\ \{\langle \alpha, x\rangle \Rightarrow e_1\} \mapsto \mathbf{case}\ e_0'\ \{\langle \alpha, x\rangle \Rightarrow e_1\}}\ (\text{CE-}\exists)$$

$$\frac{}{\mathbf{case}\ \langle \sigma, v\rangle\ \{\langle \alpha, x\rangle \Rightarrow e\} \mapsto [\sigma/\alpha, v/x]e}\ (\text{R-}\exists)$$

4 Logical Equality for Existential Types

We extend our definition of logical equivalence to handle the case of existential types. Following the previous pattern for parametric polymorphism, we cannot talk about arbitrary instances of the existential type, but we must instantiate it with a relation that is closed under Kleene equality.

Recall from Lecture 12:

(∀) $e \sim e' : \forall\alpha.\,\tau$ iff for all closed types σ and σ' and admissible relations $R : \sigma \leftrightarrow \sigma'$ we have $e \sim e' : [R/\alpha]\tau$

(R) $e \sim e' : R$ with $e : \tau$, $e' : \tau'$ and $R : \tau \leftrightarrow \tau'$ iff $e\ R\ e'$.

We add

(∃) $e \sim e' : \exists\alpha.\,\tau$ iff $e \simeq \langle \sigma, e_0 \rangle$ and $e' \simeq \langle \sigma', e_0' \rangle$ for some closed types σ, σ' and expressions e_0, e_0', and there is an admissible relation $R : \sigma \leftrightarrow \sigma'$ such that $e_0 \sim e_0' : [R/\alpha]\tau$.

In our example, we ask if

$$\texttt{NAT} \sim \texttt{BIN} : \texttt{NUM}$$

which unfolds into demonstrating that there is a relation $R : \mathit{nat} \leftrightarrow \mathit{bin}$ such that

$$\langle Z, \langle S, \mathit{pred}_n \rangle \rangle \sim \langle E, \langle \mathit{succ}_b, \mathit{pred}_b \rangle \rangle : R \otimes (R \to R) \otimes (R \to R\ \mathbf{option})$$

Here we have disambiguated the occurrences of the successor and predecessor function as operating on type *nat* or *bin*.

Since logical equality at type $\tau_1 \otimes \tau_2$ just decomposes into logical equality at the component types, this just decomposes into three properties we need to check. The key step is to define the correct relation R.

5 Defining a Relation Between Implementations

$R : \mathit{nat} \leftrightarrow \mathit{bin}$ needs to relate natural numbers in two different representations. It is convenient and general to define such relations by using inference rules.

Once we have made this decision, the relation could be based on the structure of $n : \mathit{nat}$ or on the structure of $x : \mathit{bin}$. The former may run into difficulties because each number actually corresponds to infinitely many numbers in binary form: just add leading zeros that do not contribute to its value. Therefore, we define it based on the binary representation. In order to define it, we use a function *dbl* on unary numbers.

```
dbl : Nat -> Nat
dbl Z = Z
dbl (S n) = S (S (dbl n))
```

$$\frac{}{Z\ R\ E}\ R_e \qquad \frac{n\ R\ x}{(\mathit{dbl}\ n)\ R\ (B_0\ x)}\ R_0 \qquad \frac{n\ R\ x}{S\ (\mathit{dbl}\ n)\ R\ (B_1\ x)}\ R_1$$

6 Verifying the Relation

Because our signature exposes three constants, we now have to check three properties.

Lemma 1 $Z \sim E : R$

Proof: By definition $Z \sim E : R$ is equivalent to $Z\ R\ E$, which follows immediately from rule R_e. □

Lemma 2 $S \sim \mathit{succ}_b : R \to R$.

Proof: By definition of logical equality, this is equivalent to showing

For all n : *nat*, x : *bin with* $n\ R\ x$ *we have* $(S\ n)\ R\ (\mathit{succ}_b\ x) : R$.

Since R is defined inductively by a collection of inference rules, the natural attempt is to prove this by rule induction on the given relation, namely $n\ R\ x$.

Case: Rule

$$\frac{}{Z\ R\ E}\ R_e$$

with $n = Z$ and $x = E$. We have to show that $(S\,n)\ R\ (\mathit{succ}\ x)$ (abbreviating now succ_b as *succ*).

$Z\ R\ E$	By rule R_e
$(S\,(\mathit{dbl}\,Z))\ R\ (B_1\,E)$	By rule R_1
$(S\,Z)\ R\ (B_1 E)$	Since $\mathit{dbl}\,Z \simeq Z$
$(S\,Z)\ R\ (\mathit{succ}\,E)$	Since $\mathit{succ}\,E \simeq B_1\,Z$
$(S\,n)\ R\ (\mathit{succ}\,x)$	Since $n = Z$ and $x = E$

This proof is most likely discovered and should perhaps be read starting with the last line and going upwards.

Case: Rule

$$\frac{n'\ R\ x'}{(\mathit{dbl}\ n')\ R\ (B_0\ x')}\ R_0$$

where $n = \mathit{dbl}\ n'$ and $x = B_0\ x'$. We have to show that $(S\,n)\ R\ (\mathit{succ}\ x)$. Again, you may want to read the proof below starting at the bottom.

$n'\ R\ x'$	Premise in this case
$(S\,(\mathit{dbl}\ n'))\ R\ (B_1\ x')$	By rule R_1
$(S\,(\mathit{dbl}\ n'))\ R\ (\mathit{succ}\ (B_0\ x'))$	Since $\mathit{succ}\ (B_0\ x') \simeq B_1\ x'$
$(S\,n)\ R\ (\mathit{succ}\ x)$	Since $n = \mathit{dbl}\ n'$ and $x = B_0\ x'$

Case: Rule

$$\frac{n'\ R\ x'}{S\ (\mathit{dbl}\ n')\ R\ (B_1\ x')}\ R_1$$

where $n = S\ (\mathit{dbl}\ n')$ and $x = B_1\ x'$. We have to show that $(S\,n)\ R\ (\mathit{succ}\ x)$. Again, you may want to read the proof below starting at the bottom.

$n'\ R\ x'$	Premise in this case
$(S\ n')\ R\ (\mathit{succ}\ x')$	By induction hypothesis
$(\mathit{dbl}\ (S\ n'))\ R\ (B_0\ (\mathit{succ}\ x'))$	By rule R_0
$(S\,(S\,(\mathit{dbl}\ n')))\ R\ (B_0\ (\mathit{succ}\ x'))$	Since $\mathit{dbl}\ (S\ n') \simeq S\ (S\ (\mathit{dbl}\ n'))$
$(S\,(S\,(\mathit{dbl}\ n')))\ R\ (\mathit{succ}\ (B_1\ x'))$	Since $\mathit{succ}\ (B_1\ x') \simeq B_0\ (\mathit{succ}\ x')$
$(S\,n)\ R\ (\mathit{succ}\ x)$	Since $n = S\,(\mathit{dbl}\ n')$ and $x = B_1\ x'$

□

In order to prove the relation between the implementation of the predecessor function we should explicitly write out the interpretation of τ **option**.

(τ option) $e \sim e' : \tau$ **option** iff either $e \simeq$ **null** and $e' \simeq$ **null** or $e \simeq$ **just** e_1 and $e' \simeq$ **just** e_1' and $e_1 \sim e_1' : \tau$.

Lemma 3 $\mathit{pred}_n \simeq \mathit{pred}_b : R \to R$ **option**

Proof: By definition of logical equality, this is equivalent to show

> *For all n : nat, x : bin with $n\ R\ x$ we have either (i) $\mathit{pred}_n\ n \simeq$* **null** *and $\mathit{pred}_b\ x \simeq$* **null** *or (ii) $\mathit{pred}_n\ n \simeq$* **just** *$n'$ and $\mathit{pred}_b\ x \simeq$* **just** *$x'$ and $n'\ R\ x'$.*

This can now be proven by rule induction on the given relation, with a slightly more complicated argument.

Case: Rule

$$\frac{}{Z\ R\ E}\ R_e$$

with $n = Z$ and $x = E$. Then $\mathit{pred}_n\ Z =$ **null** $= \mathit{pred}_b\ E$.

Case: Rule

$$\frac{n'\ R\ x'}{(\mathit{dbl}\ n')\ R\ (B_0\ x')}\ R_0$$

where $n = \mathit{dbl}\ n'$ and $x = B_0\ x'$.

$n'\ R\ x'$ Premise in this case
Either $\mathit{pred}_n\ n' = \mathbf{null} = \mathit{pred}_b\ x'$
or $\mathit{pred}_n\ n' = \mathbf{just}\ n''$ and $\mathit{pred}_b\ x' = \mathbf{just}\ x''$ with $n''\ R\ x''$
By induction hypothesis

- $\mathit{pred}_n\ n' = \mathbf{null} = \mathit{pred}_b\ x'$ First subcase
- $n' = Z$ By inversion on the defn. of pred_n
- $\mathit{pred}_n\ (\mathit{dbl}\ n') = \mathit{pred}_n\ Z = \mathbf{null}$ By definition of pred_n
- $\mathit{pred}_b\ x = \mathit{pred}_b\ (B_0\ x') = \mathit{map}\ B_1\ (\mathit{pred}_b\ x')$
  $= \mathit{map}\ B_1\ \mathbf{null} = \mathbf{null}$ By definition of pred_b

- $\mathit{pred}_n\ n' = \mathbf{just}\ n''$ and $\mathit{pred}_b\ x' = \mathbf{just}\ x''$ and $n''\ R\ x''$
  Second subcase
- $n' = S\ n''$ By inversion on the definition of pred_n
- $\mathit{pred}_n\ (\mathit{dbl}\ n') = \mathit{pred}_n\ (S\ (S\ (\mathit{dbl}\ n'')))$
  $= \mathbf{just}\ (S\ (\mathit{dbl}\ n''))$ By definition of pred_n
- $\mathit{pred}_b\ (B_0\ x') = \mathit{map}\ B_1\ (\mathit{pred}_b\ x')$
  $= \mathit{map}\ B_1\ (\mathbf{just}\ x'') = \mathbf{just}\ (B_1\ x'')$ By definition of pred_b
- $(S\ (\mathit{dbl}\ n''))\ R\ (B_1\ x'')$ By rule R_1

Case: Rule

$$\frac{n'\ R\ x'}{S\ (\mathit{dbl}\ n')\ R\ (B_1\ x')}\ R_1$$

where $n = S\ (\mathit{dbl}\ n')$ and $x = B_1\ x'$.

- $\mathit{pred}_n\ n = \mathit{pred}_n\ (S\ (\mathit{dbl}\ n')) = \mathbf{just}\ (\mathit{dbl}\ n')$ By defn. of pred_n
- $\mathit{pred}_b\ x = \mathit{pred}_b\ (B_1\ x') = \mathbf{just}\ (B_0\ x')$ By defn. of pred_b
- $(\mathit{dbl}\ n')\ R\ (B_0\ x')$ By rule R_0

□

7 The Upshot

Because the two implementations are logically equal we can replace one implementation by the other without changing any client's behavior. This is

because all clients are parametric, so their behavior does not depend on the library's implementation.

It may seem strange that this is possible because we have picked a particular relation to make this proof work. Let us reexamine the (E-$\exists$) rule:

$$\frac{\Delta\,;\,\Gamma \vdash e : \exists\alpha.\,\tau \quad \Delta, \alpha\ \mathit{type}\,;\,\Gamma, x : \tau \vdash e' : \tau'}{\Delta\,;\,\Gamma \vdash \mathbf{case}\ e\ \{\langle\alpha, x\rangle \Rightarrow e'\} : \tau'}\ \text{(E-}\exists\text{)}$$

In the second premise we see that the client e' is checked with a fresh type α and $x : \tau$ which may mention α. if we reify this into a function, we find

$$\lambda x.\, e' : \forall\alpha.\,\tau \to \tau'$$

where τ' does not depend on α.

By Reynolds's parametricity theorem we know that this function is parametric. This can now be applied for any σ and σ' and relation $R : \sigma \leftrightarrow \sigma'$ to conclude that if $v_0 \sim v_0' : [R/\alpha]\tau$ then $[v_0/x]e' \sim [v_0'/x]e' : [R/\alpha]\tau'$. But α does not occur in τ', so this is just saying that $[v_0/x]e' \sim [v_0'/x]e' : \tau'$. So the result of substituting the two different implementations is equivalent.

References

[GMM+78] Michael J.C. Gordon, Robin Milner, L. Morris, Malcolm C. Newey, and Christopher P. Wadsworth. A metalanguage for interactive proof in LCF. In A. Aho, S. Zillen, and T. Szymanski, editors, *Conference Record of the 5th Annual Symposium on Principles of Programming Languages (POPL'78)*, pages 119–130, Tucson, Arizona, January 1978. ACM Press.

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