Lecture Notes on Data Abstraction

15-814: Types and Programming Languages
Frank Pfenning

Lecture 17
Thursday, October 31, 2019

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 a counter with

constant *zero* and functions *inc* and *dec* to increment and decrement the counter. 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 `CTR` as the *signature* that describes the interface of a module.

```
CTR = {
  type ctr
  zero : ctr
  inc : ctr -> ctr
  dec : ctr -> (None : 1) + (Some : ctr)
}
```

The function `dec` returns an optional counter with the new value, since we consider the predecessor of 0 to be undefined. 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, for which is it helpful to have a predecessor function on unary numbers.

```
pred Z = None
pred (S n) = Some n
```

and then

```
NCtr : CTR = {
  type ctr = nat
  zero = Z
  inc = S
  dec = pred
}
```

An interesting aspect of this definition is that, for example, `zero : nat` while the interface specifies `zero : ctr`. But this is okay because the type `ctr` is in fact implemented by `nat` in this version. Next, we show the implementation using numbers in binary representation. This time, we define some of the functions directly in the module.

```
BCtr : CTR = {
  type ctr = bin

  zero = E

  inc E = B1 E
  inc (B0 x) = B1 x
  inc (B1 x) = B0 (inc x)

  dec E = None
  dec (B1 x) = Some (B0 x)
  dec (B0 x) = case dec x ( None => None
                          | Some y => B1 y )
}
```

Now what does a client look like? Assume it has an implemention `C : CTR`. 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 C : CTR

isZero : ctr -> bool
isZero x = case dec x
              ( None => true
              | Some => false )
```

but **not**

```
open C : CTR

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

because the latter supposes that the library `C : CTR` implements the type `num` by `nat`, which it may not.

3 Formalizing Abstract Types

We will write a signature such as

```
CTR = {
  type ctr
  zero : ctr
  inc : ctr -> ctr
  dec : ctr -> (None : 1) + (Some : ctr)
}
```

in abstract form as

$$\exists\alpha.\ \underbrace{\alpha}_{\textit{zero}} \times \underbrace{(\alpha \to \alpha)}_{\textit{inc}} \times \underbrace{(\alpha \to (\texttt{None} : 1) + (\texttt{Some} : \alpha))}_{\textit{dec}}$$

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

Now what should an expression

$$e : \exists\alpha.\, \alpha \times (\alpha \to \alpha) \times (\alpha \to (\texttt{None} : 1) + (\texttt{Some} : \alpha))$$

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

$$\frac{\Delta \vdash \sigma\ \mathit{tp} \quad \Delta\,;\,\Gamma \vdash e : [\sigma/\alpha]\tau}{\Delta\,;\,\Gamma \vdash \langle \sigma, e\rangle : \exists\alpha.\,\tau}\ \textsf{exists}$$

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*, *inc*, and *dec* are well-typed we substitute the implementation type for `ctr` (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.

$$\frac{\Delta\ ;\ \Gamma \vdash e : \exists\alpha.\,\tau \qquad \begin{array}{l}\alpha \notin \Delta \cup \mathrm{FTV}(\Gamma) \cup \mathrm{FTV}(\tau') \\ \Delta, \alpha\ \mathit{tp}\ ;\ \Gamma, x : \tau \vdash e' : \tau'\end{array}}{\Delta\ ;\ \Gamma \vdash \mathbf{case}\ e\ (\langle\alpha, x\rangle \Rightarrow e') : \tau'}\ \mathsf{case/exists}$$

The fact that the type α must be *new* is explicit here in the condition that is does not already appear in Δ or the free type variables of Γ or τ'. Such a condition is often left implicit, relying on the well-formedness invariants of the judgments. For example, the presupposition that Δ may not contain any repeated variables means that 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. Among other things, α may not appear in τ'.

To be a little more explicit about this (because it is critical here), whenever we write $\Delta\ ;\ \Gamma \vdash e : \tau$ we make the following *presuppositions*:

1. All the type variables in Δ are distinct.
2. All the variables in Γ are distinct.
3. $\Delta \vdash \tau_i\ \mathit{tp}$ for all $x_i : \tau_i \in \Gamma$.
4. $\Delta \vdash \tau\ \mathit{tp}$.

With these presuppositions the condition on α in the rule is automatically satisfied. Whenever we write a rule we assume this presuppositions holds for the conclusion and we have to make sure they hold for all the premises. Let's look at case/exists again in this light.

1. We assume all variables in Δ are distinct, which also means they are distinct in the first premise. In the second premise they are distinct because that's how we interpret $\Delta, \alpha\ \mathit{tp}$, which may include an implicit renaming of the type variable α bound in the the expression $\langle\alpha, x\rangle \Rightarrow e'$.
2. Similarly for the context Γ, where the freshness of x might be achieved by renaming it before applying the rule.
3. By assumption (from the conclusion), every free type variable in Γ appears in Δ. But what about τ? Actually, it is okay (and in fact mostly needed) for α to appear in τ.

4. By assumption (from the conclusion), $\Delta \vdash \tau'\ tp$. This covers the second premise. Often, this rule is given with an explicit premise $\Delta \vdash \tau'\ tp$ to emphasize τ' must be independent of α. Indeed, the scope of α is the type of x and e'.

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.

$$\frac{v\ \mathit{val}}{\langle \sigma, v\rangle\ \mathit{val}}\ \mathsf{val/exists} \qquad \frac{e \mapsto e'}{\langle \sigma, e\rangle \mapsto \langle \sigma, e'\rangle}\ \mathsf{step/exists}_1$$

$$\frac{e_0 \mapsto e_0'}{\mathbf{case}\ e_0\ (\langle \alpha, x\rangle \Rightarrow e_1) \mapsto \mathbf{case}\ e_0'\ (\langle \alpha, x\rangle \Rightarrow e_1)}\ \mathsf{step/case/exists}_0$$

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

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 16:

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

(*R*) $v \sim v' \in [R]$ iff $v\ R\ v'$.

We add

(∃) $v \sim v' \in [\exists\alpha.\,\tau]$ iff $v = \langle \sigma, v_1\rangle$ and $v' = \langle \sigma', v_1'\rangle$ for some closed types σ, σ' and values v_1, v_1', and there is a relation $R : \sigma \leftrightarrow \sigma'$ such that $v_1 \sim v_1' \in [[R/\alpha]\tau]$.

In our example, we ask if

$$\texttt{NCtr} \sim \texttt{BCtr} \in [\texttt{CTR}]$$

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

$$\langle Z, \langle S, \mathit{pred}\rangle\rangle \sim \langle E, \langle \mathit{inc}, \mathit{dec}\rangle\rangle \in [R \times (R \to R) \times (R \to 1 + R)]$$

Here we have disambiguated the occurrences of the decrement functions as operating on type *nat* or *bin*.

Since logical equality at type $\tau_1 \times \tau_2$ 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.

We will define a relation R and verify the property above in the next lecture

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.