Lecture Notes on Existential Types

15-814: Types and Programming Languages
Frank Pfenning

Lecture 11
Tue Sep 30, 2025

1 Introduction

Here is our language of types so far:

$$\begin{array}{rcll} \tau & ::= & \tau_1 \to \tau_2 \mid \forall\alpha.\,\tau \mid \alpha \mid \tau_1 \,\&\, \tau_2 \mid \top & \text{(behavioral types)} \\ & \mid & \tau_1 \times \tau_2 \mid 1 \mid \tau_1 + \tau_2 \mid 0 \mid \mu\alpha.\,\tau & \text{(observable types)} \end{array}$$

Here, we have thrown in $\top$ which is the nullary version the lazy pair $\tau_1 \,\&\, \tau_2$. Conspicuous by its absence perhaps is an existential type. By general principles, one might expect it to be an observable counterpart to the universal type. And it is. But despite its observable nature it will turn out to provide *data abstraction* [Mitchell and Plotkin, 1988], which is a fundamental concept in programming languages and software engineering. This complements the notion of *functional abstraction* we already discussed: we deem functions not to be observable, so one implementation can be replaced by an extensionally equivalent one without affecting the client. Data abstraction in addition allows us to replace one representation of data by another one without affecting the client. Of course, the two representations have to be related in a suitable way, which is what we will study in later lectures. For now, we'll study existential types (formally) and data abstraction (informally).

They are also at the core of studying module systems, because in many languages (including Standard ML) data abstraction is achieved at the module level instead of the core level. This is because data structures such as binary search trees, heaps, tries, double-ended queues, etc. are often provided as libraries.

2 Representing Integers

Our running example will be a representation of integers. So far, our data types only give us natural numbers $0, 1, 2, \ldots$. We could instantly think of one representation as a pair of a sign and a magnitude.

```
type nat = $nat. ('z : 1) + ('s : nat)
decl zero : nat
decl succ : nat
defn zero = fold 'z ()

type signed_nat = ('minus : nat) + ('plus : nat)
```

There is a slight issue here that 0 has two representations, `'plus zero` and `'minus zero`. We could arbitrarily pick one or allow both. In the live coded lecture, we used a slightly different and more awkward representation.

There is another representation whose basic idea is borrowed from fractions. We defined

```
type diff = nat * nat
```

where the pair (n, k) represents the integer $n - k$. Similar to fractions, there are many representation of the same number. For example, $(0, 0)$, $(1, 1)$, $(2, 2)$, etc. all represent the integer 0. Other examples would be $(1, 2)$ representing -1 and $(4, 2)$ representing $+2$.

In order to hide the representation from a client, we use existential quantification. For example, the integers with constant zero, operations successor, predecessor, and zerotest could be represented by the type

$$\exists\alpha.\,(\mathbf{zero} : \alpha)\ \&\ (\mathbf{succ} : \alpha \to \alpha)\ \&\ (\mathbf{pred} : \alpha \to \alpha)\ \&\ (\mathbf{iszero} : \alpha \to \mathsf{bool})$$

or, in LAMBDA syntax where $\exists\alpha.\,\tau$ is written `?a. t`:

```
type bool = ('true : 1) + ('false : 1)
decl true : bool
decl false : bool

defn true = 'true ()
defn false = 'false ()

type INT = ?a. ('zero : a)
             & ('succ : a -> a)
             & ('pred : a -> a)
             & ('iszero : a -> bool)
```

Instead of a lazy pair (with projections π_1 and π_2) we are using here a *lazy record* with projections **zero**, **succ**, **pred** and **iszero**. These projections are also called *fields*.

The type expresses that *there is type* τ (and we don't say which one!) and a record of type

$$(\mathbf{zero} : \tau)\ \&\ (\mathbf{succ} : \tau \to \tau)\ \&\ (\mathbf{pred} : \tau \to \tau)\ \&\ (\mathbf{iszero} : \tau \to \mathsf{bool})$$

It is important that we don't reveal what the type τ is, so that implementation with $\tau = \mathsf{diff} = \mathsf{nat} \times \mathsf{nat}$ and implementations with $\tau = \mathsf{signed\_nat} = \mathsf{sign} \times \mathsf{nat}$ are indistinguishable from the clients point of view.

An expression of type $\exists\alpha.\,\sigma$ should therefore be a pair whose first component is a type (the implementation type τ from the example above) and whose second component is an implementation of type $[\tau/\alpha]\sigma$. Formally:

$$\frac{\Gamma \vdash \tau\ \mathit{type} \quad \Gamma \vdash e : [\tau/\alpha]\sigma}{\Gamma \vdash \langle [\tau], e\rangle : \exists\alpha.\,\sigma}\ \mathsf{tp/paire}$$

We can use this to show the implementation of type `INT`.

```
decl Int1 : INT
defn Int1 = ([nat * nat], (| 'zero => (zero, zero)
                           | 'succ => \p. case p of ( (n, k) => (succ n, k) )
                           | 'pred => \p. case p of ( (n, k) => (n, succ k) )
                           | 'iszero => equal
                           |) )
```

In the first line, the field `'zero` which was declared as `('zero : a)` where `a` is the existentially quantified variable, must have type `nat * nat`, and it does. Similarly, the successor method takes a pair of natural numbers (n, k) to the pair of natural number $(n + 1, k)$. The equality function on natural numbers is used to check that the pair has the form (n, n), representing zero. Its straightforward definition is in lec11.cbv.

3 The Client's View

So far, we only have the constructor $\langle [\tau], e \rangle$. The guess that it would count in the column of observable types suggests that its destructor should be in the form of a case. And it is.

$$\frac{\Gamma \vdash e : \exists \alpha.\, \sigma \quad \Gamma, \alpha\ \mathit{type}, x : \sigma \vdash e' : \tau'}{\Gamma \vdash \mathbf{case}\ e\ (\langle \alpha, x \rangle \Rightarrow e') : \tau'}\ \mathsf{tp/casee}^*$$

(*) α not in Γ, e, or τ'

This rule is a somewhat tricky. Here e is the implementation of the data structure of existential type $\exists \alpha.\, \sigma$, and e' is the client of this data structure. The client can only see a type variable α, and the type σ (which may contain some occurrences of α). Even though this is often elided (like we have done), α must be *new*, that is, it cannot occur in Γ, nor can it occur in e or τ'. This ensures that the client e' cannot probe the implementation type τ that is used for α. Some languages provide a way to distinguish cases over types, a **typecase**, and in such languages this form of data abstraction would not work.

A sample client takes an implementation of type `INT` as an argument (representing a dependency upon a library), matches against it to get access to the type and implementation. Here, we take the predecessor of the successor of zero and check that the result is zero. We then give it two implementations `Int1 : INT` and `Int2 : INT` as arguments to run it.

```
type INT = ?a. ('zero : a)
             & ('succ : a -> a)
             & ('pred : a -> a)
             & ('iszero : a -> bool)

decl ex : INT -> bool
defn ex = \Int. case Int of (([a], I) => I.'iszero (I.'pred (I.'succ (I.'zero))))

decl Int1 : INT
defn Int1 = ... (* see above *)

decl Int2 : INT
defn Int2 = ... (* see lec11.cbv, using signed natural numbers *)

eval ex1v = ex Int1
eval ex2v = ex Int2
```

As anticipated, `ex1v` and `ex2v` both evaluate to true.

```
% lambda lec11.cbv
...
decl ex1v : bool
defn ex1v = 'true ()
```

```
decl ex2v : bool
defn ex2v = 'true ()
```

Looking back at the client, we see the last (outermost) call is to the `'iszero` method. That's import, because the type of all other methods contains α (written as `a`), so it would be a type error to smuggle such a value out of the scope of `(([a], I) => ...)`. We also observe that without this last method there would be no point to the whole interface because no value could ultimately be observed by a client.

4 Modules

In programming languages, modules play multiple roles. An important one is that of name-space management, which we will not address in this course. Another one is separate compilation, again something we will not discuss. One of the most important aspects is providing means for data abstraction. As we have seen, abstract types can modeled by existential types and this carries over to modules (see, for example, Harper et al. [1990]).

One issue with the sample client above

```
decl ex : INT -> bool
defn ex = \Int. case Int of (([a], I) => I.'iszero (I.'pred (I.'succ (I.'zero))))
```

is its limited scope. In more practical languages there is some concrete syntax associated with "opening" or importing a module. We might say something like

```
import Int1 : INT as [int], I
```

at the beginning of the file, binding `int` and `I` over the rest of current file or module.

This doesn't capture the ability to abstract over an implementation of a module. In ML terminology, such a module would be *functor*, that is, a function at the level of modules, which is a perfectly sensible construction.

The LAMBDA implementation, intended as a core language, does not have any special syntax for module-level constructs.

5 Preservation and Progress

Some of the restriction on existential quantification may seem arbitrary, even if they do appear to enforce data abstraction. In a later lecture, we will formally capture the idea of *representation independence* and will prove a theorem that existential types guarantee it.

In this lecture we go back to the fundamental properties of our core language, like preservation, progress, sequentiality, and canonical forms. Even these require the restrictions we have imposed. And without these properties, the language would be deeply flawed.

First, the rules of the dynamics.

$$\frac{e \ \mathit{value}}{\langle [\tau], e\rangle \ \mathit{value}}\ \mathsf{val/paire} \qquad \frac{e \mapsto e'}{\langle [\tau], e\rangle \mapsto \langle [\tau], e'\rangle}\ \mathsf{step/paire}$$

$$\frac{e_1 \mapsto e_1'}{\mathbf{case}\ e_1\ (([\alpha], x) \Rightarrow e_2 \mapsto \mathbf{case}\ e_1'\ (([\alpha], x) \Rightarrow e_2')}\ \mathsf{step/casee}_1$$

$$\frac{e_1 \ \mathit{value}}{\mathbf{case}\ \langle [\tau], e_1\rangle\ (([\alpha], x) \Rightarrow e_2) \mapsto [e_1/x][\tau/\alpha]e_2}\ \mathsf{step/casee}$$

Preservation. If $\cdot \vdash e : \tau'$ and $e \mapsto e'$ then $\cdot \vdash e' : \tau'$.

Recall that the proof is by rule induction on the derivation of $e \mapsto e'$. The congruence rules step/pair$e$ and step/casee_1 are straightforward as usual by inversion on typing, followed by an appeal to the induction hypothesis. The interesting case is step/case. Applying inversion twice we deduce

$\cdot \vdash \tau$ *type* and
$\cdot \vdash e_1 : [\tau/\alpha]\sigma$ and
α *type*, $x : \sigma \vdash e_2 : \tau'$

By the substitution property for types, we get

$x : [\tau/\alpha]\sigma \vdash [\tau/\alpha]e_2 : \tau'$

Here is the first subtlety: how come the type on the right is τ' and not $[\tau/\alpha]\tau'$? Here is where the restriction of tp/case is required: the result type τ' does not depend on the type variable α. Therefore $[\tau/\alpha]\tau' = \tau'$.

Now we can apply the regular substitution property for values and obtain

$\cdot \vdash [e_1/x][\tau/\alpha]e_2 : \tau'$

which is what we need to show since $e' = [e_1/x][\tau/\alpha]e_2$.

Progress. If $\cdot \vdash e : \tau'$ then either e *value* or $e \mapsto e'$ for some e'.

Recall that the proof is by rule induction on the typing derivation of e. The critical case is we have

$$\mathbf{case}\ e_1\ (([\alpha], x) \Rightarrow e_2)$$

where we know $\cdot \vdash e_1 : \exists\alpha.\,\sigma$ and e_1 *value*. By a suitable extension of the canonical forms theorem, we know that $e_1 = ([\tau], e_1')$ for some e_1' *value*. Therefore the given expression can be reduced by step/case.

References

Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Symposium on Principle of Programming Languages (POPL 1990)*, pages 341–354, San Francisco, California, January 1990. ACM Press.

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