Universal and Existential Types

Eduardo Bonelli

"There may, indeed, be other applications of the system other than its use as a logic"

Alonzo Church, 1932

April 25, 2019

Types of (Parametric) Polymorphism

$$\lambda X.\lambda y : X.y :: \forall X.X \to X$$

What is the set of types that we can apply it to?

1. Predicative Polymorphism
2. Impredicative Polymorphism
3. Type:Type Design

Predicative Polymorphism

$$\lambda X.\lambda y : X.y :: \forall X.X \to X$$

What is the set of types that we can apply it to?

- Types of the simply typed lambda calculus (eg. $\mathbb{N}$ or $\mathbb{B} \to \mathbb{N}$), hence non-polymorphic types
- OCaml's let-polymorphism is this type of polymorphism (extended with a special rule for let).
- We will study let-polymorphism in detail today

Impredicative Polymorphism

$$\lambda X.\lambda y : X.y :: \forall X.X \to X$$

What is the set of types that we can apply it to?

- Any type, including those constructed using Π (such as $\forall X.X \to X$ itself)
- Developed independently by Jean-Yves Girard (1971-72) and John Reynolds (1974)
- Called System F or $\lambda 2$
- We will study it in further detail later

Type:Type Design

$$\lambda X.\lambda y : X.y :: \forall X.X \to X$$

What is the set of types that we can apply it to?

- We introduce a type of all the types $\star$
- We can apply the above function to any type, including those constructed using Π and the type $\star$
- This introduces much more that polymorphism:
  - Functions over types. Eg.

    $$(\lambda X.\lambda y : X.x)\star \to \lambda y : \star.y$$

  - Dependent types. Eg.

    $$vec : \star \to \mathbb{N} \to \star$$

- We will study it when we introduce dependent types

System F

1. Type expressions
2. Terms
3. Operational semantics
4. Type system

Type Expressions

$$
\begin{array}{rcll}
\sigma & ::= & \mathbb{N} & \text{Base type} \\
& | & X & \text{Type Variable} \\
& | & \sigma \to \sigma & \\
& | & \forall X.\sigma & \text{universal type}
\end{array}
$$

Examples

1. $\forall X.X \to X$
2. $(\forall X.X \to X) \to Y$
3. $(\forall X.X \to X) \to (\forall X.X \to X)$
4. $\forall X.X$
5. $\forall X.\mathbb{N}$

Terms

$$
\begin{array}{rcll}
M & ::= & x & \text{variable} \\
 & | & \lambda x : \sigma . M & \text{abstraction} \\
 & | & M\,M & \text{application} \\
 & | & \lambda X . M & \text{type abstraction} \\
 & | & M\ [\sigma] & \text{type application}
\end{array}
$$

Examples

1. $\lambda X.\lambda x : X.x$
2. $\lambda X.\lambda f : X \to X.\lambda x : X.f\ f(x)$
3. $(\lambda X.\lambda x : X.x)\,[Y \to Y]$

Note: constructors and observers for $\mathbb{N}$ omitted

Operational Semantics (1/2)

- Values

$$
\begin{array}{rcll}
V & ::= & \underline{n} & \text{numerals} \\
 & | & \lambda x.M & \text{abstraction} \\
 & | & \lambda X.M & \text{type abstraction}
\end{array}
$$

- Rules (1/2)

$$
\frac{M_1 \to M_1'}{M_1\ M_2 \to M_1'\ M_2}\ (\textsc{E-App1}) \qquad \frac{M_2 \to M_2'}{V\ M_2 \to V\ M_2'}\ (\textsc{E-App2})
$$

$$
\frac{}{(\lambda x : \sigma.M)\ V \to M\{x := V\}}\ (\textsc{E-AppAbs})
$$

Operational Semantics (2/2)

- New Rules (2/2)

$$\frac{M_1 \to M_1'}{M_1\,[\sigma] \to M_1'\,[\sigma]}\;(\textsc{E-TApp})$$

$$\frac{}{(\lambda X.M)\,[\sigma] \to M\{X := \sigma\}}\;(\textsc{E-TAppTAbs})$$

Example

$$
\begin{array}{rl}
& (\lambda X.\lambda f : X \to X.\lambda x : X.f\,(f\,x))\ [\mathbb{N}]\ \mathit{succ}\ 0 \\
\to & (\lambda f.\mathbb{N} \to \mathbb{N}.\lambda x : \mathbb{N}.f\,(f\,x))\ \mathit{succ}\ 0 \\
\to & (\lambda x : \mathbb{N}.\mathit{succ}\,(\mathit{succ}\,x))\ 0 \\
\to & \mathit{succ}\ (\mathit{succ}\ 0)
\end{array}
$$

$$
\begin{array}{rl}
& (\lambda X.\lambda f : X \to X.\lambda x : X.f\,(f\,x))\ [\mathbb{B}]\ \mathit{not}\ \ \mathit{True} \\
\to & (\lambda f.\mathbb{B} \to \mathbb{N}.\lambda x : \mathbb{B}.f\,(f\,x))\ \mathit{not}\ \ \mathit{True} \\
\to & (\lambda x : \mathbb{B}.\mathit{not}\,(\mathit{not}\,x))\ \ \mathit{True} \\
\to & \mathit{not}\ (\mathit{not}\ \ \mathit{True}) \\
\to & \mathit{not}\ \ \mathit{False} \\
\to & \mathit{True}
\end{array}
$$

Type System - Typing Contexts

- (Typing) Contexts

$$\begin{array}{rcll} \Gamma & ::= & \emptyset & \text{empty context} \\ & | & \Gamma, x : \sigma & \text{term variable binding} \\ & | & \Gamma, X & \text{type variable binding} \end{array}$$

- Well-formed Typing Contexts
  - A context of the form $\Gamma, x : \sigma, \Gamma'$ is well-formed if
    - $FTV(\sigma) \subseteq Dom(\Gamma)$ and
    - $x \notin Dom(\Gamma)$
  - Example:
    - $X, x : X$ is wff
    - $x : Y$ and $X, x : Y, Y, x : Y$ are not wff
- In the sequel we assume contexts to be wff

Type System - Typing Rules

$$\frac{x : \sigma \in \Gamma}{\Gamma \rhd x : \sigma}\ (\text{T-Var}) \qquad \frac{\Gamma, x : \sigma \rhd M : \tau}{\Gamma \rhd \lambda x : \sigma . M : \sigma \to \tau}\ (\text{T-Abs})$$

$$\frac{\Gamma \rhd M : \sigma \to \tau \quad \Gamma \rhd N : \sigma}{\Gamma \rhd M\, N : \tau}\ (\text{T-App})$$

$$\frac{\Gamma, X \rhd M : \sigma}{\Gamma \rhd \lambda X . M : \forall X . \sigma}\ (\text{T-TAbs})$$

$$\frac{\Gamma \rhd M : \forall X . \sigma}{\Gamma \rhd M\, [\tau] : \sigma\{X := \tau\}}\ (\text{T-TApp})$$

Examples of Typing Derivations

1. $\triangleright\ \lambda X.\lambda x : X.x : \forall X.X \to X$
2. $\triangleright\ \lambda X.\lambda f : X \to X.\lambda x : X.f\,(f\,x) : \forall X.(X \to X) \to X \to X$

Another Example – Self-Application

$$\lambda x : \forall X.X \to X.\ x\,[\forall X.X \to X]\,x$$

- Has type

$$(\forall X.X \to X) \to (\forall X.X \to X)$$

Properties of System F

Type Preservation

If $\Gamma \rhd M : \sigma$ and $M \rightarrow M'$, then $\Gamma \rhd M' : \sigma$.

Progress

If M is closed an typable, then either M is a value or there exists M' s.t. $M \rightarrow M'$.

Termination

If $\Gamma \rhd M : \sigma$, then M terminates.

- Non-trivial proof

Type Inference – Undecidable

- Consider the following type erasure function:

$$
\begin{array}{rcl}
Erase(x) & = & x \\
Erase(\lambda x : \sigma.M) & = & \lambda x.Erase(M) \\
Erase(M\,N) & = & Erase(M)\,Erase(N) \\
Erase(\lambda X.M) & = & Erase(M) \\
Erase(M\,[\sigma]) & = & Erase(M)
\end{array}
$$

Wells, 1994

It is undecidable whether, given a closed term U in the untyped LC, there exists typed M in System F s.t. $Erase(M) = U$

- Open problem since beginning of the 70s
- Decidable variants
  - Given Γ, M and σ, decide if $\Gamma \triangleright M : \sigma$ is derivable
  - Given Γ and M, compute σ s.t. $\Gamma \triangleright M : \sigma$ is derivable, or fail if not.

Universal Types

Encoding Data in System F

- Church Encoding
- Scott Encoding

Existential Types

Data Structures in System F

Two approaches (we use Lists as example)

1. External:
   1.1 Applicative: Extend language with
      - New type constructor $\mathit{List}\,\sigma$
      - New constants: $\mathit{nil}, \mathit{cons}, \mathit{head}, \mathit{tail}, \mathit{isNil}$, etc.
      - New rules for operational semantics
      - Associate type to each constant

   1.2 Functional: Extend language with
      - New type constructor $\mathit{List}\,\sigma$
      - New terms: $\mathit{nil}, \mathit{cons}(M, N), \mathit{head}(M), \mathit{tail}(M), \mathit{isNil}(M)$, etc.
      - New typing rules for these terms
      - New rules for operational semantics

2. Internal: Encode lists in terms of terms in System F

$$\mathit{List}\,\sigma \stackrel{\text{def}}{=} \tau$$

   Here τ is a type expression in System F

Polymorphic Lists – External (Applicative)

- We assume given a type constructor $\mathit{List} : * \to *$ and operations

$$
\begin{aligned}
\mathit{nil} &: \forall X.\mathit{List}\, X \\
\mathit{cons} &: \forall X. X \to \mathit{List}\, X \to \mathit{List}\, X \\
\mathit{isNil} &: \forall X.\mathit{List}\, X \to \mathbb{B} \\
\mathit{head} &: \forall X.\mathit{List}\, X \to X \\
\mathit{tail} &: \forall X.\mathit{List}\, X \to \mathit{List}\, X
\end{aligned}
$$

Polymorphic Lists – External

- Polymorphic *map*

$$
\begin{array}{lcl}
map & : & \forall X.\forall Y.(X \to Y) \to List\ X \to List\ Y \\
map & = & \lambda X.\lambda Y. \\
 & & \quad \lambda f : X \to Y. \\
 & & \quad\ (fix\ [List\ X]\ [List\ Y]\ (\lambda m : List\ X \to List\ Y. \\
 & & \qquad \lambda l : List\ X. \\
 & & \qquad\quad if\ \ isnil\ [X]\ l \\
 & & \qquad\quad then\ nil\ [Y] \\
 & & \qquad\quad else\ cons\ [Y]\ (f\ (head\ [X]\ l)) \\
 & & \qquad\qquad (m\ (tail\ [X]\ l))))
\end{array}
$$

Note:

- Typing: $fix : \forall X.\forall Y.((X \to Y) \to (X \to Y)) \to X \to Y$
- Reduction: $fix\, \sigma\, \tau\, V_1\, V_2 \ \to\ V_1\, (fix\, \sigma\, \tau\, V_1\, V_2)\, V_2$

Encoding Lists – Internal

- Before addressing lists we first address some simpler examples
- The aim is to encode data structures in terms of expressions in System F
- There are two well-known internal encodings
  - Church
  - Scott
  - Also: Church-Scott or Parigot numerals (Parigot 1988, 1992). Less well-known, we will not cover them
- We will focus on Church encodings and mention Scott encodings later

Encoding the Natural Numbers

- Church numerals in the untyped LC

$$
\begin{array}{rcl}
\underline{0} & = & \lambda s.\lambda z.z \\
\underline{1} & = & \lambda s.\lambda z.s\,z \\
\underline{2} & = & \lambda s.\lambda z.s(s\,z) \\
\underline{3} & = & \lambda s.\lambda z.s(s(s\,z))
\end{array}
$$

- Church numerals are based on iteration
- No notion of pattern matching built-in
  - Some operations are hard to define (eg. pred)

Iteration in Church Numerals

- Church numerals in the untyped LC

$$
\begin{array}{rcl}
\underline{0} & \stackrel{\text{def}}{=} & \lambda s.\lambda z.z \\
\underline{1} & \stackrel{\text{def}}{=} & \lambda s.\lambda z.s\, z \\
\underline{2} & \stackrel{\text{def}}{=} & \lambda s.\lambda z.s(s\, z) \\
\underline{3} & \stackrel{\text{def}}{=} & \lambda s.\lambda z.s(s(s\, z))
\end{array}
$$

- That Church numerals support iteration means that there should exist an operation *It* that behaves as follows:

$$
\begin{array}{lcl}
It\; d\; f\; \underline{0} & \rightarrow^* & d \\
It\; d\; f\; (\underline{Succ}\; x) & \rightarrow^* & f\; (It\; d\; f\; x)
\end{array}
$$

  where $\underline{Succ} \stackrel{\text{def}}{=} \lambda n.\lambda s.\lambda z.s\,(n\, s\, z)$

- Take *It* to be defined as follows: $It\ d\ f\ M \stackrel{\text{def}}{=} M\, f\, d$

Typing the Natural Numbers in System F

- Church numerals in the untyped lambda calculus

$$\begin{array}{lcl} \underline{0} & \stackrel{\text{def}}{=} & \lambda s.\lambda z.z \\ \underline{1} & \stackrel{\text{def}}{=} & \lambda s.\lambda z.s\,z \\ \underline{2} & \stackrel{\text{def}}{=} & \lambda s.\lambda z.s(s\,z) \\ \underline{3} & \stackrel{\text{def}}{=} & \lambda s.\lambda z.s(s(s\,z)) \end{array}$$

- Most general type we can assign to them:

$$CNat \stackrel{\text{def}}{=} \forall X.(X \to X) \to X \to X$$

- Numerals in System F:

$$\begin{array}{lcl} \underline{0} & \stackrel{\text{def}}{=} & \lambda X.\lambda s : X \to X.\lambda z : X.z \\ \underline{1} & \stackrel{\text{def}}{=} & \lambda X.\lambda s : X \to X.\lambda z : X.s\,z \\ \underline{2} & \stackrel{\text{def}}{=} & \lambda X.\lambda s : X \to X.\lambda z : X.s(s\,z) \\ \underline{3} & \stackrel{\text{def}}{=} & \lambda X.\lambda s : X \to X.\lambda z : X.s(s(s\,z)) \end{array}$$

Operations on Numerals

- Successor

$$\begin{array}{rcl} csucc & : & CNat \to CNat \\ csucc & = & \lambda n : CNat.\lambda X.\lambda s : X \to X.\lambda z : X.n\ [X]\ s\ (s\, z) \end{array}$$

- Sum

$$\begin{array}{rcl} csum & : & CNat \to CNat \to CNat \\ csum & = & \lambda n : CNat.\lambda m : CNat.m\ [CNat]\ csucc\ \ n \end{array}$$

- Sum (alternative)

$$\begin{array}{rcl} csum & : & CNat \to CNat \to CNat \\ csum & = & \lambda n : CNat.\lambda m : CNat.\lambda X.\lambda s : X \to X.\lambda z : X. \\ & & \quad m\ [X]\ s\ (n\ [X]\ s\ \ z) \end{array}$$

Operations on Numerals (cont.)

- Product

$$\begin{array}{rcl} ctimes & : & CNat \to CNat \to CNat \\ ctimes & = & \lambda m : CNat.\lambda n : CNat.m\ [CNat]\ (cplus\ n)\ c_0 \end{array}$$

- Product (alternative)

$$\begin{array}{rcl} ctimes & : & CNat \to CNat \to CNat \\ ctimes & = & \lambda m : CNat.\lambda n : CNat.\lambda X.\lambda s : X \to X.\lambda z : X. \\ & & \quad m\ [X]\ (n\ [X]\ s) \end{array}$$

- Exponentiation

$$\begin{array}{rcl} cexp & : & CNat \to CNat \to CNat \\ cexp & = & \lambda n : CNat.\lambda m : CNat.n\ [X \to X]\ (m\ [X]) \end{array}$$

- Exercise: Implement the function $isZero : CNat \to CBool$

Encoding Booleans

- In untyped LC

$$\begin{aligned} tru &= \lambda t.\lambda f.t \\ fls &= \lambda t.\lambda f.f \end{aligned}$$

- Most general type we can assign to these terms?

$$CBool \stackrel{\text{def}}{=} \forall X.X \to X \to X$$

- Hence we encode them in System F as:

$$\begin{aligned} tru &= \lambda X.\lambda t : X.\lambda f : X.t \\ fls &= \lambda X.\lambda t : X.\lambda f : X.f \end{aligned}$$

Encoding Booleans (cont.)

- An example of an operation over booleans: negation

$$not : CBool \to CBool$$
$$not = \lambda b : CBool.\lambda X.\lambda t : X.\lambda f : X.\ b\,[X]\ f\ t$$

- Exercises:
  - Evaluate $not\,True$.
    - Do you need to evaluate under lambdas to obtain the resulting boolean?
  - Encode conjunction
  - Encode *ifThenElse* and use it with an example

Encoding Lists

$$\mathit{List}\ A \stackrel{\text{def}}{=} \forall R.(A \to R \to R) \to R \to R$$

- The type of lists is the type of its recursion scheme (foldr)
- Constructors

$$\begin{array}{rcl} \mathit{nil} & : & \forall A.\mathit{List}\ A \\ \mathit{nil} & = & \lambda A.\lambda R.\lambda c : (A \to R \to R).\lambda z : R.z \\ \\ \mathit{cons} & : & \forall A.A \to \mathit{List}\ A \to \mathit{List}\ A \\ \mathit{cons} & = & \lambda A.\lambda hd : A.\lambda tl : \mathit{List}\ A. \\ & & \quad (\lambda R.\lambda c : (A \to R \to R).\lambda z : R. \\ & & \qquad c\ hd\ (tl\ [R]\ c\ z)) \end{array}$$

- Exercise: Type and evaluate $\mathit{cons}\ [\mathbb{N}]\ 1\ (\mathit{cons}\ [\mathbb{N}]\ 2\ \mathit{nil})$

Observers – isNil

$$
\begin{array}{rcl}
isNil & : & \forall A. List\ A \to CBool \\
isNil & = & \lambda A. \lambda l : List\ A. l\ [CBool]\ (\lambda hd : A. \lambda tl : CBool. False)\ True
\end{array}
$$

Exercises:

- Evaluate: $isNil\ nil$
- Evaluate: $isNil\ (cons\ [\mathbb{N}]\ 1\ (cons\ [\mathbb{N}]\ 2\ nil))$

Observers – Head

Problem: must yield "error" if list is empty

- First we model the error (using non-termination)

$$\begin{array}{rcl} \mathit{diverge} & : & \forall X.\mathbb{U} \to X \\ \mathit{diverge} & = & \lambda X.\lambda\_ : \mathbb{U}.\mathit{fix}(\lambda x : X.x) \end{array}$$

  - We add *fix* to System F (it is not definable)
  - *diverge* diverges when applied to *unit*

Lets try again,

$$\begin{array}{rcl} \mathit{head} & : & \forall A.\mathit{List}\ A \to A \\ \mathit{head} & = & \lambda A.\lambda l : \mathit{List}\ A. \\ & & \quad l\ [A]\ (\lambda hd : A.\lambda tl : A.hd)\ (\mathit{diverge}\ [A]\ \mathit{unit}) \end{array}$$

Observers – Head (cont.)

- Problem: still diverges with non-empty lists

$$\begin{array}{lcl} head & : & \forall A. List\ A \to A \\ head & = & \lambda A.\lambda l : List\ A. \\ & & \quad l\ [A]\ (\lambda hd : A.\lambda tl : A.hd)\ (diverge\ [A]\ unit) \end{array}$$

- Exercise: Evaluate *head* $[\mathbb{N}]$ $(cons\ [\mathbb{N}]\ 1\ nil)$
- Must reorganize so that *unit* is passed to *diverge* only if *head* is applied to the empty list

$$\begin{array}{l} \lambda A.\lambda l : List\ A. \\ \quad (l\ [\mathbb{U} \to A]\ (\lambda hd : A.\lambda tl : \mathbb{U} \to A.\lambda\_ : \mathbb{U}.hd)\ (diverge\ [A])) \\ \quad unit \end{array}$$

- Exercise: Evaluate *head* $[\mathbb{N}]$ $(cons\ [\mathbb{N}]\ 1\ nil)$

Observers – Tail

- In order to define *tail* we need pairs
- Pairs in the untyped LC

$$pair\ x\ y \ = \ \lambda p.p\ x\ y$$

- Suppose X is the type of x and Y of y, what is the type of pairs?

$$Pair\ X\ Y \ = \forall R.(X \to Y \to R) \to R$$

- Constructors and observers

$$\begin{array}{rcl} pair & : & \forall X.\forall Y.X \to Y \to Pair\ X\ Y \\ fst & : & \forall X.\forall Y.Pair\ X\ Y \to X \\ snd & : & \forall X.\forall Y.Pair\ X\ Y \to Y \end{array}$$

- Exercise: define *fst* and *snd*.

Observers - tail (cont.)

$$
\begin{array}{lcl}
tail & : & \forall A. List\ A \to List\ A \\
tail & = & \lambda A.\lambda l : List\ A. \\
& & \quad (fst\ [List\ A]\ [List\ A]\ (\\
& & \quad\ l\ [Pair\ (List\ A)(ListA)] \\
& & \quad\ (\lambda hd : A.\lambda tl : Pair\ [List\ A]\ [List\ A]. \\
& & \quad\quad pair\ [List\ A]\ [List\ A] \\
& & \quad\quad\quad (snd\ [List\ A]\ [List\ A]\ tl) \\
& & \quad\quad\quad (cons\ [A]\ hs\ (snd\ [List\ A]\ [List\ A]\ tl))) \\
& & \quad\quad (pair\ [List\ A]\ [List\ A]\ (nil\ [A])\ (nil\ [A]))))
\end{array}
$$

Arbitrary ADTs – Church Encoding

- Church's encoding was for the untyped lambda calculus
  - The calculi of lambda conversion, Alonzo Church, Princeton University Press, Princeton, NJ, 1941.
- Shown to be typable in System F
  - Corrado Bohm and Alessandro Berarducci: Automatic Synthesis of Typed Lambda-Programs on Term Algebras Theoretical Computer Science, 1985, v39, pp. 135–154.
- Each constructor C_i of arity $a(i)$ is represented with

$$\lambda x_1 \ldots x_{a(i)} . \lambda c_1 \ldots c_n . c_i \, (x_1 \, \vec{c}) \, \ldots (x_{a(i)} \, \vec{c})$$

Arbitrary ADTs – Church Encoding

- Each constructor C_i of arity $a(i)$ is represented with
$$\lambda x_1 \dots x_{a(i)}.\lambda c_1 \dots c_n.c_i\,(x_1\,\vec{c})\,\dots\,(x_{a(i)}\,\vec{c})$$
- Example

```
type nat = Z | S of nat
```

$$\begin{aligned} S &\stackrel{\text{def}}{=} \lambda x_1.\lambda s.\lambda z.s\,(x_1\,s\,z) \\ Z &\stackrel{\text{def}}{=} \lambda s.\lambda z.z \end{aligned}$$

Arbitrary ADTs – Scott Encoding

- Example

```
type nat = Z | S of nat
```

$$S \stackrel{\text{def}}{=} \lambda x_1.\lambda s.\lambda z.s\,x_1$$
$$Z \stackrel{\text{def}}{=} \lambda s.\lambda z.z$$

- Each constructor C_i of arity $a(i)$ is represented with

$$\lambda x_1 \ldots x_{a(i)}.\lambda c_1 \ldots c_n.c_i\,x_1\ \ldots x_{a(i)}$$

- Typable in System F
  - Requires recursive types too (but (positive) recursive types may be encoded in System F) `http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt`
  - Types for the Scott numerals, M. Abadi, L, Cardelli and G. Plotkin `http://lucacardelli.name/Papers/Notes/scott2.ps`

Arbitrary ADTs – Scott Encoding

- Example

```
type nat = Z | S of nat
```

$$\begin{aligned} S &\stackrel{\text{def}}{=} \lambda x_1.\lambda s.\lambda z.s\, x_1 \\ Z &\stackrel{\text{def}}{=} \lambda s.\lambda z.z \end{aligned}$$

- Scott numerals

$$\begin{aligned} \underline{0} &\stackrel{\text{def}}{=} \lambda s.\lambda z.z \\ \underline{1} &\stackrel{\text{def}}{=} \lambda s.\lambda z.s\, \underline{0} \\ \underline{2} &\stackrel{\text{def}}{=} \lambda s.\lambda z.s\, \underline{1} \end{aligned}$$

- Problem: no recursion built-in; must add recursion

Arbitrary ADTs – Scott Encoding

- Scott numerals

$$\begin{aligned}
\underline{0} &\stackrel{\text{def}}{=} \lambda s.\lambda z.z \\
\underline{1} &\stackrel{\text{def}}{=} \lambda s.\lambda z.s\,\underline{0} \\
\underline{2} &\stackrel{\text{def}}{=} \lambda s.\lambda z.s\,\underline{1}
\end{aligned}$$

- Type of Scott numerals?

Arbitrary ADTs – Scott Encoding

- Scott numerals

$$\begin{aligned} \underline{0} &\stackrel{\text{def}}{=} \lambda s.\lambda z.z \\ \underline{1} &\stackrel{\text{def}}{=} \lambda s.\lambda z.s\,\underline{0} \\ \underline{2} &\stackrel{\text{def}}{=} \lambda s.\lambda z.s\,\underline{1} \end{aligned}$$

- Type of Scott numerals?

$$\mu X.\forall R.(X \to R) \to R \to R$$

Existential Types

- Operational reading of M if $M : \forall X.\sigma$
  - M behaves as a function that given a type τ returns a term that behaves according to the more specialized type $\sigma\{X := \tau\}$
- Operational reading of M if $M : \exists X.\sigma$?
  - M behaves like a module (or package)
  - A module is an expression of the form $\langle\tau, N\rangle$ where
    - τ is the type of the internal representation and
    - N is a term of type $\sigma\{X := \tau\}$
  - In general, M is a record of functions that "operate" over τ

Module Construction – Example

$$\exists X.\{a : X, f : X \to \mathbb{N}\}$$

- What would a value of this type be?
  - A module

    $$\langle \tau, N \rangle$$

    where N is a record of type $\{a : \tau, f : \tau \to \mathbb{N}\}$
- Two examples:

  $$\langle \mathbb{N}, \{a = 0, f = \lambda x : \mathbb{N}.succ(x)\} \rangle$$

  $$\langle \mathbb{B}, \{a = True, f = \lambda x : \mathbb{B}.0\} \rangle$$

- The type represents a measurable abstract entity

Another Example – ADTs

$$\begin{aligned} \exists Counter.\quad & \{new : Counter, \\ & get : \mathbb{N} \to Counter, \\ & inc : Counter \to Counter\} \end{aligned}$$

Two examples of values of this type

$$\begin{aligned} counterADT \;=\; & \langle Nat, \\ & \{new = 1, \\ & get = \lambda i : Nat.i, \\ & inc = \lambda i : Nat.succ(i)\}\rangle \end{aligned}$$

$$\begin{aligned} counterADTBis \;=\; & \langle \{x : Nat\}, \\ & \{new = \{x = 1\}, \\ & get = \lambda i : \{x : Nat\}.i.x, \\ & inc = \lambda i : \{x : Nat\}.\{x = succ(i.x)\}\}\rangle \end{aligned}$$

- Exercise: Define the ADT of stack of natural numbers

Type Expressions and Terms

- Type expressions

$$
\begin{array}{rcll}
\sigma & ::= & \mathbb{N} & \text{Base type} \\
& \mid & X & \text{Type variables} \\
& \mid & \sigma \to \sigma & \\
& \mid & \exists X.\sigma & \text{Existential abstraction}
\end{array}
$$

  Examples

  1. $\exists X.X \to X$
  2. $\exists X.\{x : \mathbb{N}, y : X\}$

- Terms

$$
\begin{array}{rcll}
\sigma & ::= & \ldots & \\
& \mid & \langle \sigma, M \rangle & \text{Module/package} \\
& \mid & \mathit{let}\ \{X, x\} = M\ \mathit{in}\ M & \text{Unpack}
\end{array}
$$

Typing Modules

$$\frac{\Gamma \triangleright M : \sigma\{X := \tau\}}{\Gamma \triangleright \langle \tau, M \rangle : \exists X.\sigma} \text{(T-PACK)}$$

- Exercise: type
  - $m1 = \langle \mathbb{N}, \{a = 0, f = \lambda x : \mathbb{N}.succ(x)\}\rangle$
  - $m2 = \langle \mathbb{B}, \{a = True, f = \lambda x : \mathbb{B}.0\}\rangle$
- Note how the type of $m1$ hides the internal representation (ie. $\mathbb{N}$)

Using Modules – let

$$let\ \{X, x\} = M\ in\ N$$

- If M is a module, then we can associate
  - X with its type component and
  - x with its term component,

  and use them when computing with N
- Example:

  $$let\ \{X, x\} = m1\ in\ (x.f\ x.a):$$

  - Opens the module $m1$
  - Associates X with $\mathbb{N}$ and x with $\{a = 0, f = \lambda x : \mathbb{N}.succ(x)\}$
  - Uses the fields a and f of the module to compute the numeric result ($\mathbb{N}$)

Typing let

$$\frac{\Gamma \rhd M : \exists X.\sigma \quad \Gamma, X, x : \sigma \rhd N : \tau}{\Gamma \rhd let\ \{X, x\} = M\ in\ N : \tau}\ (\textsc{T-UnPack})$$

- Exercise: Type $let\ \{X, x\} = m1\ in\ (x.f\ \ x.a)$
- Exercise: Type $let\ \{X, x\} = m2\ in\ (x.f\ \ x.a)$

Typing let (cont.)

$$\frac{\Gamma \rhd M : \exists X.\sigma \quad \Gamma, X, x : \sigma \rhd N : \tau}{\Gamma \rhd let\ \{X, x\} = M\ in\ N : \tau}\ (\text{T-UnPack})$$

- Incorrect! (type error)

$$let\ \{X, x\} = m1\ in\ succ(x.a)$$

- Makes sense: the type of the module is considered abstract
- Also, there are modules of type $\exists X.$ whose type component is not $\mathbb{N}$ (as already seen):
  - $m1 = \langle \mathbb{N}, \{a = 0, f = \lambda x : \mathbb{N}.succ(x)\} \rangle$
  - $m2 = \langle \mathbb{B}, \{a = True, f = \lambda x : \mathbb{B}.0\} \rangle$

Typing let (cont.)

$$\frac{\Gamma \triangleright M : \exists X.\sigma \quad \Gamma, X, x : \sigma \triangleright N : \tau}{\Gamma \triangleright let\ \{X, x\} = M\ in\ N : \tau}\ (\textsc{T-UnPack})$$

- Another, more subtle, example of incorrect usage

$$let\ \{X, x\} = m\ in\ x.a$$

- Error in scope
  - This expression has type X
  - Problem: X does not make sense outside its scope
    - Note that X and x are removed from the typing context in the conclusion of $(\textsc{T-UnPack})$
  - Consequently, variables X and x may be used in N but not in the type of N

Operational Semantics

- Values

$$
\begin{array}{rcll}
V & ::= & \underline{n} & \text{Numeral} \\
  & | & \lambda x.M & \text{Abstraction} \\
  & | & \langle \sigma, V \rangle & \text{Module}
\end{array}
$$

- Reduction rules

$$\frac{M \to M'}{\langle \sigma, M \rangle \to \langle \sigma, M' \rangle} \ (\textsc{E-Pack})$$

$$\frac{M \to M'}{\mathit{let}\ \{X, x\} = M\ \mathit{in}\ N \to \mathit{let}\ \{X, x\} = M'\ \mathit{in}\ N} \ (\textsc{E-UnPack})$$

$$\frac{}{\mathit{let}\ \{X, x\} = \langle \sigma, V \rangle\ \mathit{in}\ N \to N\{X := \sigma\}\{x := V\}} \ (\textsc{E-UnPackPack})$$

- Note: (E-UnPackPack) can be interpreted as a linking step

Data Abstraction with Existentials

- The following program evaluates to 2

$$\begin{gathered} let\ \{Counter, counter\} = \\ counterADT\ \ in\ counter.get\ (counter.inc\ \ counter.new) \end{gathered}$$

- In

$$let\ \{Counter, counter\} = counterADT\ \ in\ restOfProg$$

  the "*restOfProg*" may make use of *Counter* as if it were any base type (eg. $\mathbb{N}$)

- Example,

$$\begin{array}{l} let\ \{Counter, counter\}\ =\ counterADT\ \ in \\ \ let\ add2\ =\ \lambda c : Counter.counter.inc\ (counter.inc\ \ c)\ \ in \\ \ \ counter.get\ (add2\ counter.new) \end{array}$$

Existential Types in OCaml

Two approaches:

- Use modules which implicitly use existential types
- Use explicitly typed constructors
    - Should have functional type
    - Type variables not present in return type are existentially quantified
    - Example:

```
type m = C : ('a -> int) * 'a -> m
```

Existential Types in OCaml

Printable objects

```
type p = D : ('a -> string) * 'a -> p
```

Measurable objects

```
type m = C : ('a -> int) * 'a -> m
```

- Type of `C`:

  `C`: ∀ `'a. ('a -> int)* 'a -> m`

- Logical identities

$$
\begin{array}{rcl}
\forall x.A(x) \supset B & \equiv & \exists x.(A(x) \supset B) \\
\exists x.A(x) \supset B & \equiv & \forall x.(A(x) \supset B)
\end{array}
$$

Existential Types in OCaml

$$\langle \mathbb{N}, \{a = 0, f = \lambda x : \mathbb{N}.succ(x)\}\rangle$$

```
# type m = C : ('a -> int) * 'a -> m;;
type m = C : ('a -> int) * 'a -> m
# let m1 = C ((fun x -> List.length x), [1;2;3]);;
val m1 : m = C (<fun>, <poly>)
# let m2 = C ((fun x -> x), 7);;
val m2 : m = C (<fun>, <poly>)
# let size (C(f,a)) =  f a;;
val size : m -> int = <fun>
# size m1;;
- : int = 3
# size m2;;
- : int = 7
```

Existential Types in OCaml

$$\frac{\Gamma \triangleright M : \exists X.\sigma \quad \Gamma, X, x : \sigma \triangleright N : \tau}{\Gamma \triangleright let\ \{X, x\} = M\ in\ N : \tau}\ (\textsc{T-UnPack})$$

```
# let m1 = C ((fun x -> List.length x), [1;2;3]);;
val m1 : m = C (<fun>, <poly>)
# match m1 with
| C(f,a) -> 3::a;;
Error: This expression has type $C_'a but an
expression was expected of type int list
```

- First, types whose name starts with a \$ are existentials
- `$Constr_'a` denotes an existential type introduced for the type variable `'a` of the constructor `Constr`

Existential Types through Modules

```
module type Measurable_type = sig
  type t
  val data:t
  val size:t -> int
end
module Measurable_Int : Measurable_type = struct
  type t=int
  let data = 4
  let size = fun x -> x
end
```

```
# #use "et.ml";;
module type Measurable_type = sig type t val data : t val size : t -> int end
module Measurable_Int : Measurable_type
# MModule.data;;
- : MModule.t = <abstr>
# MModule.size;;
- : MModule.t -> int = <fun>
# MModule.size MModule.data;;
```

Encoding Existentials in System F

$$\exists X.\sigma \stackrel{\text{def}}{=} \forall Y.(\forall X.\sigma \to Y) \to Y$$

- An existential can be thought of as an expression that
  - Takes the type of the result; and
  - A continuation; and
  - Applies the continuation to a type and a term parameterized over the type
- In other words:

$$\begin{aligned} \langle \tau, M^{\sigma} \rangle \;&=\; \lambda Y.\lambda f : (\forall X.\sigma \to Y).f\ [\tau]\ M \\ \mathit{let}\ \{X, x\} = M^{\exists X.\sigma}\ \mathit{in}\ N^{\tau} \;&=\; M\ [\tau]\ (\lambda X.\lambda x : \sigma.N) \end{aligned}$$

- Exercise: Show that if $\langle \tau, M^{\sigma} \rangle$ is typable, then so is its encoding (same for $\mathit{let}\ \{X, x\} = M^{\exists X.\sigma}\ \mathit{in}\ N^{\tau}$).