Lecture Notes 11
System F: Polymorphism and Abstraction

Carlo Angiuli

B522: PL Foundations
April 14, 2025

these notes are a bit short on explanations...

In this lecture, we will discuss System F [Gir72; Rey74], also known as the *polymorphic λ-calculus* or the *second-order λ-calculus*, a famous core calculus that goes beyond "simple" types by introducing universal and existential quantification to our syntax of types (or propositions, via Curry–Howard). These quantifiers extend the simply-typed λ-calculus with two important type-based abstraction mechanisms, namely *parametric polymorphism* and *abstract types*.

System F and abstract types are covered in Chapters 16 and 17 of Harper [Har16] respectively. Students may also wish to consult Sections 4 and 5 of Lau Skorstengaard's notes from Amal Ahmed's OPLSS lectures on logical relations. For those who wish to dig deeper, the seminal research papers on these topics are surprisingly accessible:

- "Types, Abstraction and Parametric Polymorphism" by Reynolds [Rey83]
- "Theorems for Free!" by Wadler [Wad89]
- "Representation Independence and Data Abstraction" by Mitchell [Mit86]
- "Abstract types have existential type" by Mitchell and Plotkin [MP88]

1 System F

In our lecture on type isomorphisms we talked about "*the* identity function **I**," but the STLC actually has many different identity functions, one for every type:

$$\cdot \vdash \lambda x : \texttt{unit}.x : \texttt{unit} \to \texttt{unit}$$
$$\cdot \vdash \lambda x : \texttt{bool}.x : \texttt{bool} \to \texttt{bool}$$
$$\vdots$$

The STLC does not let us use a single function at all of these types; remember, it has uniqueness of types! System F also has uniqueness of types, but what it adds to the STLC is the ability to express that the identity function $\lambda x : \alpha.x$ has type $\alpha \to \alpha$ *generically for any type* α, by giving it the *polymorphic* type $\forall \alpha.\alpha \to \alpha$. This polymorphic type tells us that we can instantiate the α in the function's type with any concrete type of our choice, such as `unit` or `bool`.

This is call-by-name System F:

Syntax:

Types	τ ::=	$\texttt{arr}(\tau_1, \tau_2)$	$\tau_1 \to \tau_2$	function type
		$\texttt{all}(\alpha.\tau)$	$\forall \alpha.\tau$	polymorphic type
Terms	e ::=	$\texttt{lambda}(\tau, x.e)$	$\lambda x : \tau.e$	λ-abstraction
		$\texttt{app}(e_1, e_2)$	$e_1\ e_2$	application
		$\texttt{Lambda}(\alpha.e)$	$\Lambda \alpha.e$	type abstraction
		$\texttt{App}(e, \tau)$	$e@\tau$	type application

The $\Delta \vdash \tau$ ty judgment:

$$\frac{}{\Delta, \alpha\ \mathsf{ty} \vdash \alpha\ \mathsf{ty}} \qquad \frac{\Delta \vdash \tau_1\ \mathsf{ty} \qquad \Delta \vdash \tau_2\ \mathsf{ty}}{\Delta \vdash \tau_1 \to \tau_2\ \mathsf{ty}} \qquad \frac{\Delta, \alpha\ \mathsf{ty} \vdash \tau\ \mathsf{ty}}{\Delta \vdash \forall \alpha.\tau\ \mathsf{ty}}$$

Since terms can now contain type variables, the typing judgment must be indexed by both a type variable context Δ and a term variable context Γ, where the types in Γ must be well-formed with respect to the context Δ.

The $\Delta; \Gamma \vdash e : \tau$ judgment:

$$\frac{}{\Delta; \Gamma, x : \tau \vdash x : \tau}\ \textsc{var} \qquad \frac{\Delta \vdash \tau_1\ \mathsf{ty} \qquad \Delta; \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash \lambda x : \tau_1.e_2 : \tau_1 \to \tau_2}\ \to\text{-}\textsc{intro}$$

$$\frac{\Delta; \Gamma \vdash f : \tau_1 \to \tau_2 \qquad \Delta; \Gamma \vdash e_1 : \tau_1}{\Delta; \Gamma \vdash f\ e_1 : \tau_2}\ \to\text{-}\textsc{elim}$$

$$\frac{\Delta, \alpha\ \mathsf{ty}; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash \Lambda \alpha.e : \forall \alpha.\tau}\ \forall\text{-}\textsc{intro} \qquad \frac{\Delta; \Gamma \vdash e : \forall \alpha.\tau \qquad \Delta \vdash \tau'\ \mathsf{ty}}{\Delta; \Gamma \vdash e@\tau' : \tau[\tau'/\alpha]}\ \forall\text{-}\textsc{elim}$$

Remark 11.1. Our conventions for judgments hide an implicit side condition of the ∀-INTRO rule: it can only be applied when the context Γ is well-formed in Δ, i.e., when α does not occur free in Γ.

The v val judgment:

$$\frac{}{\lambda x : \tau . e \ \mathsf{val}} \qquad \frac{}{\Lambda \alpha . e \ \mathsf{val}}$$

The $e \longmapsto e'$ judgment:

$$\frac{f \longmapsto f'}{f\ e_1 \longmapsto f'\ e_1} \qquad \frac{}{(\lambda x : \tau_1 . e_2)\ e_1 \longmapsto e_2[e_1/x]}$$

$$\frac{e \longmapsto e'}{e@\tau \longmapsto e'@\tau} \qquad \frac{}{(\Lambda \alpha . e)@\tau \longmapsto e[\tau/\alpha]}$$

Example 11.2. Returning to the polymorphic identity function, we define:

$$\mathbf{I} : \forall \alpha . \alpha \to \alpha$$
$$\mathbf{I} \coloneqq \Lambda \alpha . \lambda x : \alpha . x$$

Then $\mathbf{I}@\texttt{unit} : \texttt{unit} \to \texttt{unit}$ and $\mathbf{I}@\texttt{bool} : \texttt{bool} \to \texttt{bool}$ and even

$$\mathbf{I}@(\forall \alpha . \alpha \to \alpha) : (\forall \alpha . \alpha \to \alpha) \to (\forall \alpha . \alpha \to \alpha)$$

The ability to use a single piece of code at multiple types is called *polymorphism*. System F supports a specific kind of polymorphism called *parametric polymorphism*, in which polymorphic terms must behave uniformly at every type. For example, a term of type $\forall \alpha . \alpha \to \alpha$ is not simply a term that happens to work at type $\tau \to \tau$ for any τ; it has a single definition that is uniform or generic in the type α.

In contrast, *ad hoc polymorphism* refers to terms that are available at every type but may be defined differently at each one. For example, many languages have a + function which can operate on any type, but given two numbers performs numerical addition, given two strings performs string concatenation, etc. Similarly, `equal?` compares numbers for numerical equality, lists for structural equality, functions for pointer equality, etc.

In object-oriented languages, mechanisms for parametric polymorphism are often called *generics* and ad hoc polymorphism is often called *operator overloading*.

Remark 11.3.

type application is weird, usually it's inferred

Remark 11.4. The notation ∀ makes good sense from the type system perspective: a term with type $\forall\alpha.\alpha \to \alpha$ has type $\alpha \to \alpha$ "for all" types α. In addition, the typing rules extend the Curry–Howard correspondence to universal quantification. In logic, to prove (introduce) a ∀ statement, one must construct a proof of that statement for an arbitrary (variable) term; to use (eliminate) a ∀ statement, one may instantiate the quantified variable with any term whatsoever.

1.1 Church encodings

Note that System F has no base types, but the type grammar is not empty because of type variables.

We can extend System F with all the other types we have considered so far (in fact, roughly speaking, we can think of typed functional programming languages like OCaml as some combination of System F and PCF with isorecursive types.) But it turns out that we do not need them. (An aside.)

$$\begin{aligned}
\texttt{bool} &\coloneqq \forall\alpha.\alpha \to \alpha \to \alpha \\
\texttt{true} &\coloneqq \Lambda\alpha.\lambda x : \alpha.\lambda y : \alpha.x \\
\texttt{false} &\coloneqq \Lambda\alpha.\lambda x : \alpha.\lambda y : \alpha.y \\
\texttt{if}(e_1, e_2, e_3) : \tau &\coloneqq e_1@\tau\ e_2\ e_3
\end{aligned}$$

$$\begin{aligned}
\texttt{nat} &\coloneqq \forall\alpha.\alpha \to (\alpha \to \alpha) \to \alpha \\
\texttt{zero} &\coloneqq \Lambda\alpha.\lambda z : \alpha.\lambda s : \alpha \to \alpha.z \\
\texttt{suc}(e) &\coloneqq \Lambda\alpha.\lambda z : \alpha.\lambda s : \alpha \to \alpha.s\ (e@\alpha\ z\ s) \\
\texttt{natrec}(e, e_z, e_s) : \tau &\coloneqq e@\tau\ e_z\ e_s
\end{aligned}$$

Exercise 11.5. Define $\texttt{not} : \texttt{bool} \to \texttt{bool}$.

These encodings also work in the untyped λ-calculus, but they don't work in STLC because we would have to fix the type that we map out into.

1.2 Free theorems

See Wadler [Wad89]. Parametric polymorphism is very strong:

- There are no closed terms of type $\forall\alpha.\alpha$.
- The polymorphic identity function is the only term (up to observational equivalence) of type $\forall\alpha.\alpha \to \alpha$.
- There is no closed term of the "fixpoint" type $\forall\alpha.(\alpha \to \alpha) \to \alpha$.

- Any $\texttt{equal?} : \forall\alpha.\alpha \to \alpha \to \texttt{bool}$ must be constant.

- Any $\texttt{map} : \forall\alpha.\forall\beta.(\alpha \to \beta) \to \texttt{list}(\alpha) \to \texttt{list}(\beta)$ must satisfy the equations

$$
\begin{aligned}
&\texttt{map}@\tau@\tau'\ f \\
\cong\ &\lambda\ell : \texttt{list}(\tau).\texttt{map}@\tau@\tau'\ f\ (\texttt{map}@\tau@\tau\ (\mathbf{I}@\tau)\ \ell) \\
\cong\ &\lambda\ell : \texttt{list}(\tau).\texttt{map}@\tau'@\tau'\ (\mathbf{I}@\tau')\ (\texttt{map}@\tau@\tau'\ f\ \ell)
\end{aligned}
$$

2 Abstract types

An even more important application of type genericity arises when considering abstract interfaces, e.g. in data structures.

An implementation of a *queue* (of numbers) must provide:

- a representation type τ_r for queues,

- a queue $\texttt{empty} : \tau_r$,

- a function $\texttt{enqueue} : \texttt{nat} \to \tau_r \to \tau_r$, and

- a function $\texttt{dequeue} : \tau_r \to (\texttt{unit} + (\texttt{nat} \times \tau_r))$.

There are various ways to implement queues, including the simple `ListQueue` implementation in which $\tau_r = \texttt{list}(\texttt{nat})$, and the more efficient `BatchedQueues` in which $\tau_r = \texttt{list}(\texttt{nat}) \times \texttt{list}(\texttt{nat})$ [Oka99, Section 5.2].

There are many reasons why a program may wish to use a queue as part of some other computation. We would like to arrange that such programs, which we will call *clients* of the queue library, not only do not need to understand how queues are implemented but in fact are *prohibited* from knowing the implementation.

To this end we will define an abstract interface (here, a type and three terms) that all queue implementations will implement, and with respect to which all clients will be implemented. Our type discipline will then enforce that clients may not take the `length` of a queue even if we happen to link them against the `ListQueue` implementation. (Indeed, that would prevent us from swapping the `ListQueues` for `BatchedQueues`.)

Maintaining a strict separation of concerns between libraries and clients is crucial to programming in the large. Object-oriented languages call this separation *encapsulation*; non-OO programming language theorists typically call it *data abstraction*, and the type of queues an *abstract data type*.

> "Type structure is a syntactic discipline for enforcing levels of abstraction." —John C. Reynolds

Extend System F with existential types:
The $\Delta \vdash \tau$ ty judgment:

$$\ldots \qquad \frac{\Delta, \alpha \text{ ty} \vdash \tau \text{ ty}}{\Delta \vdash \exists \alpha.\tau \text{ ty}}$$

The $\Delta; \Gamma \vdash e : \tau$ judgment:

$$\ldots \qquad \frac{\Delta \vdash \tau_r \text{ ty} \qquad \Delta, \alpha \text{ ty} \vdash \tau_i \text{ ty} \qquad \Delta; \Gamma \vdash e : \tau_i[\tau_r/\alpha]}{\Delta; \Gamma \vdash \mathsf{pack}\ \langle \tau_r, e \rangle\ \mathsf{as}\ \exists \alpha.\tau_i : \exists \alpha.\tau_i}\ \exists\text{-INTRO}$$

$$\frac{\Delta; \Gamma \vdash e : \exists \alpha.\tau_i \qquad \Delta \vdash \tau' \text{ ty} \qquad \Delta, \alpha \text{ ty}; \Gamma, x : \tau_i \vdash e' : \tau'}{\Delta; \Gamma \vdash \mathsf{unpack}\ \langle \alpha, x \rangle = e\ \mathsf{in}\ e' : \tau'}\ \exists\text{-ELIM}$$

The v val judgment:

$$\ldots \qquad \frac{}{\mathsf{pack}\ \langle \tau_r, e \rangle\ \mathsf{as}\ \exists \alpha.\tau_i\ \mathsf{val}}$$

The $e \longmapsto e'$ judgment:

$$\frac{e \longmapsto e''}{\mathsf{unpack}\ \langle \alpha, x \rangle = e\ \mathsf{in}\ e' \longmapsto \mathsf{unpack}\ \langle \alpha, x \rangle = e''\ \mathsf{in}\ e'}$$

$$\frac{}{\mathsf{unpack}\ \langle \alpha, x \rangle = (\mathsf{pack}\ \langle \tau_r, e \rangle\ \mathsf{as}\ \exists \alpha.\tau_i)\ \mathsf{in}\ e' \longmapsto e'[\tau_r/\alpha][e/x]}$$

Remark 11.6. In fact it is possible to Church encode existential types, so everything we say about System F with existential types actually applies directly to plain System F. However, it will be much clearer for us to work with existential types without going through an encoding.

Remark 11.7. The notation $\exists$ may seem somewhat odd at first, although it makes some sense: to construct a term of type $\exists \alpha.\alpha \times \ldots$ there must "exist" some representation type τ_r for which we can construct a term of type $\tau_r \times \ldots$. In fact this notation is chosen because the typing rules for abstract types extend the Curry–Howard correspondence to existential quantification. In logic, to prove (introduce) an $\exists$ statement, one must exhibit a particular witness along with a proof of the statement for that witness. To use (eliminate) an $\exists$ statement, one can assume that a *generic* witness satisfying the relevant property exists.

Returning to the queue example,

$$\texttt{QueueImpl} := \exists\alpha.\alpha \times (\texttt{nat} \to \alpha \to \alpha) \times (\alpha \to \texttt{unit} + \texttt{nat} \times \alpha)$$
$$\texttt{ListQueue} := \mathsf{pack}\ \langle \texttt{list(nat)}, (\texttt{nil}, \textit{[enq]}, \textit{[deq]})\rangle\ \mathsf{as}\ \exists\alpha.\alpha \times \dots$$
$$\texttt{BatchedQueue} := \mathsf{pack}\ \langle \texttt{list(nat)} \times \texttt{list(nat)}, \dots\rangle\ \mathsf{as}\ \exists\alpha.\alpha \times \dots$$

Client:

$$\mathsf{unpack}\ \langle \tau_q, \mathit{impl}\rangle = \texttt{ListQueue}\ \texttt{in}\ \dots \texttt{fst}(\mathit{impl}) \dots$$

The type of the unpack has to not contain τ_q, and the client code $\dots \texttt{fst}(\mathit{impl}) \dots$ has to be parametrically polymorphic in τ_q. This is how we ensure data abstraction. Note however that at runtime there is no longer any data abstraction.

2.1 Representation independence

What happens when we replace `ListQueue` with `BatchedQueue`?

$$\mathsf{unpack}\ \langle \tau_q, \mathit{impl}\rangle = \texttt{BatchedQueue}\ \texttt{in}\ \dots \texttt{fst}(\mathit{impl}) \dots$$

This program is still well-typed, so it does not go wrong. But does it compute the same thing as the previous program?

In general, two implementations of an interface might behave completely differently. For example, the new queue's dequeue function may always return $\texttt{inl}(())$. In this case, we might imagine that despite having different runtime representations of queues under the hood, `ListQueue`s and `BatchedQueue`s "have the same extensional behavior," and thus expect that the two programs should compute the same result—even though they run different code!

One way to formalize the notion of "having the same extensional behavior" is by exhibiting what is known as a *bisimulation* between the two queue implementations. Given two implementations

$$\texttt{QueueImpl}_1 := \mathsf{pack}\ \langle \tau_1, (\mathsf{emp}_1, \mathsf{enq}_1, \mathsf{deq}_1)\rangle\ \mathsf{as}\ \exists\alpha.\alpha \times \dots$$
$$\texttt{QueueImpl}_2 := \mathsf{pack}\ \langle \tau_2, (\mathsf{emp}_2, \mathsf{enq}_2, \mathsf{deq}_2)\rangle\ \mathsf{as}\ \exists\alpha.\alpha \times \dots$$

a *queue bisimulation* between $\texttt{QueueImpl}_1$ and $\texttt{QueueImpl}_2$ is a binary relation R between closed terms of type τ_1 and closed terms of type τ_2, such that

- $R(\mathsf{emp}_1, \mathsf{emp}_2)$ holds,
- for all $n : \texttt{nat}$, $q_1 : \tau_1$, and $q_2 : \tau_2$ satisfying $R(q_1, q_2)$, $R(\mathsf{enq}_1\ n\ q_1, \mathsf{enq}_2\ n\ q_2)$ holds, and

- for all $q_1 : \tau_1$ and $q_2 : \tau_2$ satisfying $R(q_1, q_2)$, either
  - deq $q_1 \cong$ `inl(())` and deq $q_2 \cong$ `inl(())`, or
  - deq $q_1 \cong$ $\texttt{inr}(n_1, q_1')$ and deq $q_2 \cong$ $\texttt{inr}(n_2, q_2')$ where $n_1 \cong n_2$ and $R(q_1', q_2')$.

The *representation independence* theorem [Mit86] states that if we have a queue bisimulation between $\texttt{QueueImpl}_1$ and $\texttt{QueueImpl}_2$, then $\texttt{QueueImpl}_1$ and $\texttt{QueueImpl}_2$ are observationally equivalent at type `QueueImpl`. That is, no program's result can be affected by swapping $\texttt{QueueImpl}_1$ for $\texttt{QueueImpl}_2$. In particular, every client computes the same answers with respect to both implementations.

We emphasize that this theorem holds *with no conditions whatsoever* on the client code: our type system guarantees that no program can tell apart bisimilar terms of existential type.

In the next lecture we will use logical relations to prove the *parametricity theorem* for System F, a powerful result from which we can obtain all of the free theorems and representation independence theorems discussed in this lecture.

References

[Gir72] Jean-Yves Girard. "Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur". PhD thesis. Université Paris VII, June 1972. URL: https://girard.perso.math.cnrs.fr/These.pdf.

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Second Edition. Cambridge University Press, 2016. ISBN: 9781107150300. DOI: 10.1017/CBO9781316576892.

[Mit86] John C. Mitchell. "Representation Independence and Data Abstraction". In: *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '86. New York, NY, USA: ACM, 1986, pp. 263–276. ISBN: 9781450373470. DOI: 10.1145/512644.512669.

[MP88] John C. Mitchell and Gordon D. Plotkin. "Abstract types have existential type". In: *ACM Transactions on Programming Languages and Systems* 10.3 (July 1988), pp. 470–502. ISSN: 0164-0925. DOI: 10.1145/44501.45065.

[Oka99] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1999. DOI: 10.1017/CBO9780511530104.

[Rey74] John C. Reynolds. "Towards a theory of type structure". In: *Programming Symposium*. Ed. by B. Robinet. Berlin, Heidelberg: Springer Berlin Heidelberg, 1974, pp. 408–425. ISBN: 978-3-540-37819-8.

[Rey83] John C. Reynolds. "Types, Abstraction and Parametric Polymorphism". In: *Information Processing '83: Proceedings of the IFIP 9th World Computer Congress*. Ed. by R. E. A. Mason. North-Holland, 1983, pp. 513–523. ISBN: 0444867295.

[Wad89] Philip Wadler. "Theorems for Free!" In: *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*. FPCA '89. New York, NY, USA: ACM, 1989, pp. 347–359. ISBN: 0897913280. DOI: 10.1145/99370.99404.