Lecture Notes 11
System F: Polymorphism and Abstraction

Carlo Angiuli

B522: PL Foundations
April 8, 2026

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 to our propositions, under the propositions-as-types correspondence). These quantifiers extend the simply-typed λ-calculus with two important type-based abstraction mechanisms, namely *parametric polymorphism* and *abstract types*. Then, in the next lecture, we will study the expressive power of these mechanisms, culminating in what I consider the first "serious" theorems one encounters in programming language theory.

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`.

In this class we will study the *call-by-name* version of System F; the call-by-name version is more standard in the literature and will simplify our proof of the parametricity theorem in the next lecture. The syntax of System F is as follows.

Types	$\tau ::=$	$\texttt{arr}(\tau_1, \tau_2)$	$\tau_1 \to \tau_2$	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

As with the STLC, it is straightforward to add more types to System F (products, sums, isorecursive types, etc.) but we will see soon that this minimal calculus is more expressive than it may seem. The syntax is fairly standard by this point, with one exception: as with isorecursive types, we have type variable binders in the type $\forall \alpha.\tau$ and its introduction form $\Lambda \alpha.e$.

Unlike with isorecursive types, our typing judgment will 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. For this reason, it may be helpful to expand the above syntax chart for types into an explicit inductive definition.

Definition 11.1 (Well-formed types)**.** A *type variable context* Δ is a list of the form $\alpha_1\ \mathsf{ty}, \ldots, \alpha_n\ \mathsf{ty}$. Given such a Δ, the judgment $\Delta \vdash \tau\ \mathsf{ty}$ states that τ is a *well-formed type in* Δ, and is defined inductively by the following rules:

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

Definition 11.2 (Type system)**.** Let Δ be a type variable context. A *term variable context* Γ *in* Δ is a list of the form $x_1 : \tau_1, \ldots, x_n : \tau_n$ where $\Delta \vdash \tau_i$ ty for all i. Given such a Δ and Γ, the typing judgment $\Delta; \Gamma \vdash e : \tau$ is defined inductively as follows:

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

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

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

Remark 11.3. Our conventions for judgments hide an implicit side condition of the $\forall$-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.

Exercise 11.4. Note that System F has no base types. In the lecture on STLC we remarked that the grammar of types would be empty if we did not include a base type such as `unit`. Why does the same problem not apply here?

The operational semantics for System F are quite simple. We continue to require that terms have no free term variables in order to evaluate them; we will moreover require that they have no free type variables.

The v val judgment is defined inductively by:

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

The (call-by-name!) $e \longmapsto e'$ judgment is defined inductively by:

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

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

The type system and operational semantics satisfy all the properties we are used to by this point, including type safety.

Example 11.5. Returning to the polymorphic identity function, we define:

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

Then, supposing we had `unit` and `bool` in System F, we would have $\mathbf{I}@\texttt{unit} : \texttt{unit} \to \texttt{unit}$ and $\mathbf{I}@\texttt{bool} : \texttt{bool} \to \texttt{bool}$ and even

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

Exercise 11.6. Write down a closed term of type $\forall\alpha.\alpha \to \alpha \to \alpha$.

Exercise 11.7. Write down another closed term of type $\forall\alpha.\alpha \to \alpha \to \alpha$. How many closed terms of this type are there? (There are multiple correct answers to this question depending on how you interpret that question.)

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 (or many types) 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. Although they may seem superficially similar, the distinction between parametric and ad hoc polymorphism is central to the theory of programming languages.

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

Remark 11.8. Programming languages with polymorphism generally do not require users to manually instantiate polymorphic terms with types as in $e@\tau$. (They often do not require users to manually *abstract* type variables in polymorphic terms either, as in $\Lambda\alpha.e$!) In practice, type instantiation is typically handled by the type-checker, but for our purposes it is more uniform to consider it explicitly.

Remark 11.9. Type instantiation $e@\tau$ is also strange from an operational perspective: its only purpose is is to replace type variables in type annotations with concrete types τs, as in $(\Lambda\alpha.e)@\tau \longmapsto e[\tau/\alpha]$ rule, but the operational semantics ignores all type annotations anyway! An alternate way of presenting the languages we have considered this semester is to define an operation which deletes or "erases" all type annotations (from λ, `inl`, `fix`, etc.) and then define operational semantics only over erased terms. In this case, type instantiation can be erased as well.

Remark 11.10. The notation $\forall$ 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 propositions-as-types correspondence to universal quantification

over propositions. In logic, to prove (introduce) a $\forall$ statement, one must construct a proof of that statement for an arbitrary (variable) proposition; to use (eliminate) a $\forall$ statement, one may instantiate the quantified variable with any proposition.

mention System F_ω and higher-kinded types?

1.1 Church encodings

As previously mentioned, one can extend System F with all the other features we have considered so far this semester; in fact, System F with named finite sums and products, PCF's `fix`, and isorecursive types is a usable approximation of (the pure fragment of) typed functional programming languages such as OCaml.

Surprisingly, and unlike in STLC, we can actually *encode* algebraic data types in System F rather than adding them explicitly, using what are called *Church encodings*. We will not dwell on this point, but one can get the feel of Church encodings by considering a few simple examples:

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

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

Exercise 11.11. Define $\texttt{not} : \texttt{bool} \to \texttt{bool}$.

Exercise 11.12. Define $\texttt{even?} : \texttt{nat} \to \texttt{bool}$.

Exercise 11.13. Suppose we swapped our encodings of `true` and `false`. What change would we need to make to the encoding of `if`?

Note that the same term encodings work for the untyped λ-calculus by simply deleting Λ and @. They do *not* work in STLC because such an encoding must fix a single type into which the encoded data type can eliminate.

1.2 Free theorems

The *parametric* nature of parametric polymorphism provides very strong guarantees on the behavior of programs of type $\forall\alpha.\tau$. These guarantees are typically

known as "free theorems," following Wadler [Wad89]. We will prove some of these in the next lecture. (All following statements involving types such as `bool`, `nat`, and $\texttt{list}(\alpha)$ hold for both Church-encoded and built-in versions of these types.)

- 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 function $\texttt{map} : \forall\alpha.\forall\beta.(\alpha \to \beta) \to \texttt{list}(\alpha) \to \texttt{list}(\beta)$ must relate to the "real" map function $\texttt{realMap} : \forall\alpha.\forall\beta.(\alpha \to \beta) \to \texttt{list}(\alpha) \to \texttt{list}(\beta)$ as follows: for any $\ell : \texttt{list}(\tau)$,

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

Exercise 11.14. Think about one of the above theorems. Can you form an intuition about why it should hold? Does the theorem still seem true if System F allowed for *ad hoc* polymorphism?

2 Abstract types have existential type

Another important use of type variables in programming languages arises when considering abstract interfaces, e.g. of data structures. Suppose we want to implement a *queue of numbers*. Any such implementation 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 it is desirable to formulate an *abstract interface* for queues of numbers—an abstract type τ_r equipped with three terms as above—to insulate queue implementations from their clients. Every queue implementation is required to export a type with three such operations, and every queue client is implemented with respect to a generic type τ_r with those operations. 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

We can model abstract interfaces in System F by adding *existential types* $\exists\alpha.\tau$. We extend the previous judgments as follows. To the $\Delta \vdash \tau$ ty judgment we add:

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

To the $\Delta; \Gamma \vdash e : \tau$ judgment we add:

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

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

To the v val judgment we add:

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

Finally, to the $e \longmapsto e'$ judgment we add:

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

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

Remark 11.15. It is possible to Church encode existential types:

$$\exists \alpha.\tau := \forall \beta.(\forall \alpha.\tau \to \beta) \to \beta$$
$$\mathsf{pack}\ \langle \tau_r, e \rangle\ \mathsf{as}\ \exists \alpha.\tau_i := \Lambda \beta.\lambda x : (\forall \alpha.\tau \to \beta).x@\tau_r\ e$$
$$\mathsf{unpack}\ \langle \alpha, x \rangle = e\ \mathtt{in}\ e' : \tau' := e@\tau'\ (\Lambda \alpha.\lambda x : \tau.e')$$

Thus everything we say about System F with existential types can be understood to be about ordinary System F. However, it will be much clearer for us to treat existential types as primitive rather than working with this encoding.

Remark 11.16. 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 concrete 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 propositions-as-types correspondence to existential quantification (over propositions). 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 a *generic* witness satisfying the relevant property.

Let us return now to our example. The abstract interface of queues (i.e., the type of queue implementations) can be defined as the following existential type:

$$\mathtt{QueueImpl} := \exists \alpha.\alpha \times (\mathsf{nat} \to \alpha \to \alpha) \times (\alpha \to \mathsf{unit} + \mathsf{nat} \times \alpha)$$

We can define two elements of `QueueImpl`, `ListQueue` and `BatchedQueue`, by packing their respective representations `list(nat)` and `list(nat)` × `list(nat)` along with appropriate implementations of `empty`, enqueue, and dequeue.

$$\mathtt{ListQueue} := \mathsf{pack}\ \langle \mathtt{list(nat)}, (\mathtt{nil}, \textit{[enq]}, \textit{[deq]}) \rangle\ \mathsf{as}\ \mathtt{QueueImpl}$$
$$\mathtt{BatchedQueue} := \mathsf{pack}\ \langle \mathtt{list(nat)} \times \mathtt{list(nat)}, \ldots \rangle\ \mathsf{as}\ \mathtt{QueueImpl}$$

Now suppose we want to write a program that uses a queue to compute a result of some type τ. We can unpack the `ListQueue` implementation to obtain a type α along with a triple $impl : \alpha \times (\mathsf{nat} \to \alpha \to \alpha) \times (\alpha \to \mathsf{unit} + \mathsf{nat} \times \alpha)$.

$$\mathsf{unpack}\ \langle \alpha, \textit{impl} \rangle = \mathtt{ListQueue}\ \mathtt{in}\ e$$

where, by the ∃-ELIM rule,

$$\alpha\ \mathsf{ty}; \mathit{impl} : \alpha \times (\mathsf{nat} \to \alpha \to \alpha) \times (\alpha \to \mathsf{unit} + \mathsf{nat} \times \alpha) \vdash e : \tau$$

and τ is not allowed to depend on α.

Because we are still in System F, note that such an e must be parametrically polymorphic in α and thus can only interact with the type of queues α via the operations of *impl*. This is how we ensure data abstraction, and why we are covering parametric polymorphism and abstract interfaces in the same lecture.

Remark 11.17. Note that the operational semantics will actually instantiate α and *impl* with `list(nat)` and (`nil`, *[enq]*, *[deq]*). That is, we enforce data abstraction only in the type system and not at runtime.

2.1 Representation independence

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

$$\mathsf{unpack}\ \langle \alpha, \mathit{impl} \rangle = \texttt{BatchedQueue}\ \texttt{in}\ e$$

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, a different queue implementation's `dequeue` function may always return `inl(())`, and that can certainly lead to different results. However, in the case of `ListQueues` and `BatchedQueues`, we might imagine that despite having different runtime representations of queues under the hood, these implementations "have the same extensional behavior," and therefore $\mathsf{unpack}\ \langle \alpha, \mathit{impl} \rangle =$ `ListQueue` `in` e and $\mathsf{unpack}\ \langle \alpha, \mathit{impl} \rangle =$ `BatchedQueue` `in` e should compute the same result—even though they run completely 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

$$\texttt{QueueImpl}_1 \coloneqq \mathsf{pack}\ \langle \tau_1, (\mathsf{emp}_1, \mathsf{enq}_1, \mathsf{deq}_1) \rangle\ \mathsf{as}\ \texttt{QueueImpl}$$
$$\texttt{QueueImpl}_2 \coloneqq \mathsf{pack}\ \langle \tau_2, (\mathsf{emp}_2, \mathsf{enq}_2, \mathsf{deq}_2) \rangle\ \mathsf{as}\ \texttt{QueueImpl}$$

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 $n : \mathsf{nat}$, $q_1 : \tau_1$, 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
  - $\texttt{deq}\ q_1 \cong \texttt{inl}(())$ and $\texttt{deq}\ q_2 \cong \texttt{inl}(())$, or
  - $\texttt{deq}\ q_1 \cong \texttt{inr}(n_1, q_1')$ and $\texttt{deq}\ q_2 \cong \texttt{inr}(n_2, q_2')$ 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 $\texttt{QueueImpl}$. 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 e other than that the unpack is well-typed: 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 1986. 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 1989. New York, NY, USA: ACM, 1989, pp. 347–359. ISBN: 0897913280. DOI: 10.1145/99370.99404.