Programming Languages: Functional Programming
7. Types and Logic

Shin-Cheng Mu

Spring 2022

Anatomy of a (Programming) Language

- To define a (programming) language, we typically have to define
  - its syntax;
  - its type system;
  - and its semantics.
- Syntax is considered by some an issue that is done with. There are occasionally interesting new research results, though.
- We briefly talked about semantics before, and unfortunately won't have time to cover more.
- Type is a hot topic in the area of programming languages.

What are Types For?

- What does a type system do?
- Kris de Volder: "... making sure that no operations are performed on inappropriate arguments."
  - e.g. `"abc"` $\times$ 123.
- "A type system is a tractable syntactic method for proving the absence of certain program behaviours by classifying phrases according to the kinds of values they compute" Benjamine Pierce, *Types and Programming Languages* (MIT, 2002).
- A type system guarantees safety properties by *limiting the programs you are allowed to write*.
- Certain safety properties are not decidable. Type systems for them cannot be precise, and some safe programs might be ruled out too.
- A *static* type system verifies the program text before it is run.
- A *dynamic* type system verifies the actual expression during it is run.
- A practical type system could be a mixture of both. This course mainly concerns the former.

Motivations for a Type System

- Safety: early detection of certain kinds of errors.
  - e.g. trivial things like integer + string.
  - Types that guarantees that no communication error occurs, polynomal running time, etc.
- Efficiency: allowing certain optimisations.
  - e.g. if we are sure that array indexing never goes out of bound, we do not have to do run-time bound check.
  - Some type systems guarantee certain resource usage: "this variable is used only once."
- Specification: the type specifies part of what a program does.
  - As we have seen, programs are often structured around the datatype it is defined on.
  - Type guarantees certain behaviour. E.g. if f :: List $a \to a$ we must have $f \cdot map\ g = g \cdot f$.
  - "This function computes $sort$."

Nothing Comes for Free

What's the price?

- A type system rules out certain programs as illegal. However, a static type system must make a conservative guess.
  - The following program does not generate a run-time type error, but is not typable in Haskell.

$$\begin{aligned} f\ b &= \mathbf{if}\ b\ \mathbf{then}\ g\ b\ \mathbf{else}\ ord\ (g\ b) \\ g\ b &= \mathbf{if}\ b\ \mathbf{then}\ 65\ \mathbf{else}\ \text{'}A\text{'} \end{aligned}$$

- A more expressive type system makes a finer guess, and also allows more to be said in the type. However, you often need to provide a lot more information and put more efforts persuading the type checker that a program is correct.

1 Intuitionistic Propositional Logic

Propositional Logic

- For reasons that will be clear later, we introduce some logic before talking about types.
- Propositional logic: a simple form of logic having some very nice properties.
- Let P be the set of propositional symbols. The syntax of propositional logic is given by

$$\begin{aligned} PL = \ & \mathit{True} \mid \mathit{False} \mid P \\ & \mid PL \Rightarrow PL \mid PL \wedge PL \mid PL \vee PL \end{aligned}$$

- There are several formal systems to prove statements in propositional logic. We will present one of them.

Natural Deduction for Intuitionistic Propositional Logic

- Let Γ be a set of propositions that are assumed to be true.
- The *judgement* $\Gamma \vdash P$ means that "given the assumptions in Γ, P is provable".

$$\frac{P \in \Gamma}{\Gamma \vdash P}\ \mathsf{Hyp}$$

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q} \Rightarrow\!\mathsf{I} \qquad \frac{\Gamma \vdash P \Rightarrow Q \qquad \Gamma \vdash P}{\Gamma \vdash Q} \Rightarrow\!\mathsf{E}$$

$$\frac{}{\Gamma \vdash \mathit{True}}\ \mathit{True}\text{-}\mathsf{I} \qquad \frac{\Gamma \vdash \mathit{False}}{\Gamma \vdash P}\ \mathit{False}\text{-}\mathsf{E}$$

Natural Deduction for Constructive Propositional Logic

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q} \wedge\!\mathsf{I}$$

$$\frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P} \wedge E_1 \qquad \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q} \wedge E_2$$

$$\frac{\Gamma \vdash P}{\Gamma \vdash P \vee Q} \vee I_1 \qquad \frac{\Gamma \vdash Q}{\Gamma \vdash P \vee Q} \vee I_2$$

$$\frac{\Gamma \vdash P \vee Q \qquad \Gamma, P \vdash R \qquad \Gamma, Q \vdash R}{\Gamma \vdash R} \vee\!\mathsf{E}$$

Observations...

- Each logical symbol comes with some *introduction* rule and some *elimination* rule...
  - no introduction rule for False.
- To prove a proposition, we work upwards from the bottom. Ex. prove that $(P \to Q \to R) \to (P \to Q) \to P \to R$.
- Negation can be defined by $\neg P = P \to \mathit{False}$.

Excluded Middle

- Note that we do not have such a rule:

$$\frac{}{\Gamma \vdash P \vee \neg P}\ \text{Excluded-Middle}$$

- This rule is valid in *classical* logic, which talks about truth or falsehood — a proposition is either true or false.
- It is questioned by the *constructive* (intuitionistic) school of logicians. Constructive logic is about *provability*: it is not always the case that either P or $\neg P$ has a proof.
- Such different views led to many famous debates in history.
- Not having such a rule makes intuitionistic incomplete (see next slide).

Consistency, Soundness, and Completeness

- A deduction system suggests a way to construct proofs. But do we know whether the proofs are correct?
- Correctness is discussed with respect to a semantics:
  - Assign each free identifier a true/false value.
  - Each logical operator is a function, etc
- A deduction system is
  - consistent: if falsehood cannot be proved;
  - sound: if every provable proposition is indeed true in the semantics;
  - complete: if every true proposition in the semantics is provable.
- The deduction system for prositional logic (with the addition of the law of excluded middle) has all these nice properties. It is not so for more complex logic.

2 Untyped and Typed λ Calculus

2.1 Untyped λ Calculus

λ Calculus

- A very concise model of computation. Let X be the set of variables. The syntax for λ calculus is given by:

$$Term = \lambda X.Term \mid Term\ Term \mid X$$

- Operationally, $\lambda x.e$ defines an anonymous function with local variable x, while $e_1\, e_2$ is function application.
- Occurrences of x in $\lambda x.e$ is *bound*. A variable occurrence that is not bound is called *free*.
  - E.g. in $\lambda x.(z\,(\lambda y.x\,y))\,y$, x is bound and z is free. The first y is bound, the second is free.

λ Calculus: α Conversion and β Reduction

- $e_1[x \backslash e_2]$: substitute the free occurrences of x in e_1 for e_2. More on the next slide.
- α conversion: $\lambda x.e \equiv \lambda y.e[x \backslash y]$ for some y not occurring free in e.
  - Meaning that names of bound variables do not matter.
- β reduction: $(\lambda x.e_1)\ e_2 \xrightarrow{\beta} e_1[x \backslash e_2]$.
  - Mimicking function application.
- These already constitute a Turing-complete model of computation!
  - You can model numbers (search for "Church encoding"), addition, subtraction...
  - You may perform recursion, and even non-terminating computation! (Search for "Y combinator")

λ Calculus: Substitution

- $e_1[x \backslash e_2]$: substitute the free occurrences of x in e_1 for e_2 — and perform necessary changes of names.
  - A seemingly trivial operation whose formal definition is surprisingly tedious. For this course we might not need all the details, so I'll go with a "learn by examples" approach.
- E.g. $(\lambda x.y\,x)[y \backslash \lambda z.z\,w] = \lambda x.(\lambda z.z\ w)\,x$.
- $(\lambda x.y\,x)[x \backslash \lambda z.z] = (\lambda x.y\,x)$, since x is bound.
- $(x\,(\lambda y.z\,x\,y))[x \backslash y\,z] \neq (y\,z)\,(\lambda y.z\,(y\,z)\,y)$! The free occurrence of y in $y\,z$ is "captured".
  - Haskell analog: let $f\,x = \mathbf{let}\ y = ...\ \mathbf{in}\ ..x...$
  - $f\,(y+3) \neq \mathbf{let}\ y = ...\ \mathbf{in}\ ..(y+3)..$, since the y in $y+3$ refers to some global y, not the y in f.
- It ought to be $(y\,z)\,(\lambda w.z\,(y\,z)\,w)$. The term $(\lambda y.z\,x\,y)$ is α-converted to avoid name capture.

Summary of λ Calculus

- A simple syntax.
- Two rules: α and β.
- Yet it is Turing-complete — every computation possible on a Turing machine can be expressed in λ calculus.
- You can see it as a small fragment of Haskell (or, LISP/Scheme). In fact, λ calculus forms the theoretical basis of functional languages.

2.2 Simply Typed λ Calculus

Simply Typed λ Calculus

- One of the typed version of λ calculus.
- We postulate existence of certain basic types, e.g. Nat, $Char$, etc.
- Each λ bound variables is annotated with its type. (It's like in many programming languages where you have to specify the types of arguments to functions.)

$$Term = \lambda(X :: Type).Term \mid Term\ Term \mid X$$

- Remark: there is another formulation of simply typed λ calculus (the Curry style, as opposed to the Church style here) without type annotations. The two styles are equivalent, however.

Extension with Basic Types

- For illustrative purposes, it is often convenient to extend λ calculus with some basic types, e.g.

$$Term = \lambda(X :: Type).Term \mid Term\ Term \mid X \mid Nat \mid Term \oplus Term$$

  - where $\oplus \in \{+, -, \times\}$, etc.

- So you can write, e.g. $(\lambda(x :: Nat).\lambda(y :: Nat).(x + 1) \times y)\ ((\lambda(x :: Nat).x \times x)\ 2)\ z$

Typing Rules

- A *typing context*: a mapping from variable names to types.
  - Empty context: $\emptyset$, or sometimes just left blank.
  - $\Gamma, x :: \tau$ denotes Γ extended with the assumption that x has type τ ($(,)$ is like $(:)$ for lists, but appending the new element to the right-hand side).
- In the typing rules below we assume that all bound names in expressions have been α converted to unique names, which is possible because there is an infinite supply of names.
- A *typing relation*: $\Gamma \vdash e :: \tau$ says that "the expression e has type τ in the typing context Γ."
- *Typing rules*:

$$\frac{x :: \tau \in \Gamma}{\Gamma \vdash x :: \tau}\ \text{Var} \qquad \frac{\Gamma, x :: \sigma \vdash e :: \tau}{\Gamma \vdash \lambda(x : \sigma).e :: \sigma \to \tau} \to\text{I}$$

$$\frac{\Gamma \vdash e_1 :: \sigma \to \tau \qquad \Gamma \vdash e_2 :: \sigma}{\Gamma \vdash e_1\, e_2 :: \tau} \to\text{E}$$

- With extensions:

$$\frac{n \in Nat}{\Gamma \vdash n :: Nat}\ \text{Nat}$$

$$\frac{\Gamma \vdash e_1 :: Nat \qquad \Gamma \vdash e_2 :: Nat}{\Gamma \vdash e_1 \oplus e_2 :: Nat}\ \text{NatOp}$$

Several Ways to Use These Rules

- Type checking: given Γ, e, and τ, verify that $\Gamma \vdash e :: \tau$.
- Type inference: given Γ and e, find τ such that $\Gamma \vdash e :: \tau$.
- Type inhabitation: given Γ and τ, find e such that $\Gamma \vdash e :: \tau$.

Type Checking Example

See Figure 1

Several Things to Note

- In each step there is only one rule we could apply, guided by the syntax.
  - Typing tree of an expression is composed by the typing trees of its sub-expressions.
  - In the 90's there was a trend to make every static analysis a type system, since type systems are very structured.

Several Things to Note

- Can the same expression be typed by $(Bool \to Bool \to (Bool \to Nat)) \to Bool \to (Bool \to Nat)$, given suitable Γ, and extending the typing rules with those for $Bool$)?
- Yes. Typing is not unique. Which brings up the question whether there is a "most general" type we can give to an expression.

2.3 Type Safety

Type Safety

- Type systems try to guarantee certain safety properties.
- **Subject reduction** (or type preservation): if $\Gamma \vdash e_1 :: \tau$ and $e_1 \xrightarrow{\beta} e_2$, we have $\Gamma \vdash e_2 :: \tau$.
  - In words, typable terms are still typable by the same types after β reduction.
- **Progress**: if $\Gamma \vdash e_1 :: \tau$, either $e_1 \xrightarrow{\beta} e_2$ for some e_2, or e_1 is a value.
  - Definition of a "value" varies. E.g., a normal form.
  - In words, we never get stuck in a state where no further reductions are possible (counter example: $1 + \texttt{'}c\texttt{'}$).
- Slogan "well-typed programs don't go wrong."
- A language guarantees certain type safety, that typable programs don't go wrong, is called *strong typing*.
- But there is always a grey area: what about $1/0$?

Denote $x :: Nat \to Nat \to Nat, y :: Nat$ by Γ (we omit the type annotations in λ to save space):

$$\dfrac{\dfrac{\dfrac{\dfrac{x :: Nat \to Nat \to Nat \in \Gamma}{\Gamma \vdash x :: Nat \to Nat \to Nat}\,\mathsf{Var} \quad \dfrac{y :: Nat \in \Gamma}{\Gamma \vdash y :: Nat}\,\mathsf{Var}}{\Gamma \vdash x\,y :: Nat \to Nat}\,{\to}\mathsf{E} \quad \dfrac{y :: Nat \in \Gamma}{\Gamma \vdash y :: Nat}\,\mathsf{Var}}{\Gamma \vdash (x\,y)\,y :: Nat}\,{\to}\mathsf{E}}{x :: Nat \to Nat \to Nat \vdash \lambda y.(x\,y)\,y :: Nat \to Nat}\,{\to}\mathsf{I}}{\vdash \lambda x.\lambda y.(x\,y)\,y :: (Nat \to Nat \to Nat) \to Nat \to Nat}\,{\to}\mathsf{I}$$

Figure 1: An example of type checking.

- In theory, all dynamic properties can be captured by a type system that is expressive enough. However, such a type system could be very tedious to use. Design of such systems is still an active topic.
- In practise, we try to capture some reasonable type errors ($1 + \text{'}c\text{'}$) and leave some to runtime ($1/0$), and still call the language strongly typed.

2.4 More Datatypes

Products

- To add more datatypes to the language, just add the corresponding introduction and elimination rules.
- For pairs (product type), we have

$$\dfrac{\Gamma \vdash e_1 :: \sigma \qquad \Gamma \vdash e_2 :: \tau}{\Gamma \vdash (e_1, e_2) :: (\sigma, \tau)}\,{\times}\mathsf{I}$$

$$\dfrac{\Gamma \vdash e :: (\sigma, \tau)}{\Gamma \vdash \mathit{fst}\ e :: \sigma}\,{\times}E_1 \qquad \dfrac{\Gamma \vdash e :: (\sigma, \tau)}{\Gamma \vdash \mathit{snd}\ e :: \tau}\,{\times}E_2$$

- Products are like "struct" in C.

Sum

- There is a type we should have talked more about: $\mathbf{data}\ \mathit{Either}\ a\ b = \mathit{Left}\ a \mid \mathit{Right}\ b$. We will abbreviate $\mathit{Either}\ a\ b$ to $a + b$, Left to L, Right to R.
- Typing rules:

$$\dfrac{\Gamma \vdash e :: \sigma}{\Gamma \vdash \mathit{Left}\ e :: \sigma + \tau}\,{+}I_1 \qquad \dfrac{\Gamma \vdash e :: \tau}{\Gamma \vdash \mathit{Right}\ e :: \sigma + \tau}\,{+}I_2$$

$$\dfrac{\Gamma \vdash e :: \sigma + \tau \qquad \Gamma, x :: \sigma \vdash e_1 :: \gamma \qquad \Gamma, y :: \tau \vdash e_2 :: \gamma}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ L\,x \to e_1; R\,y \to e_2 :: \gamma}\,{+}\mathsf{E}$$

- Sums are like "union" in C.

Unit and Empty

- The unit type in Haskell is written $()$. It has only one element, also written $()$.

$$\dfrac{}{\Gamma \vdash () :: ()}\,\text{Unit-I}$$

- The empty type consists of no term. You can define it in Haskell by $\mathbf{data}\ \mathit{Empty}$. There is only an elimination rule:

$$\dfrac{\Gamma \vdash e_1 :: \mathit{Empty}}{\Gamma \vdash e_2 :: \tau}\,\text{Empty-E}$$

That is, if you manage to construct a term e_1 having type Empty (which cannot happen), you can assign e_2 any type.

3 Curry-Howard Isomorphism

Proof Terms

But aren't they just natural deduction rules annotated by terms?

$$\dfrac{x :: \tau \in \Gamma}{\Gamma \vdash x :: \tau}\,\mathsf{Var} \qquad \dfrac{\Gamma, x :: \sigma \vdash e :: \tau}{\Gamma \vdash \lambda x.e :: \sigma \to \tau}\,{\to}\mathsf{I}$$

$$\dfrac{\Gamma \vdash e_1 :: \sigma \to \tau \qquad \Gamma \vdash e_2 :: \sigma}{\Gamma \vdash e_1\,e_2 :: \tau}\,{\to}\mathsf{E}$$

$$\dfrac{}{\Gamma \vdash () :: ()}\,\text{Unit-I} \qquad \dfrac{\Gamma \vdash e_1 :: \mathit{Empty}}{\Gamma \vdash e_2 :: \tau}\,\text{Empty-E}$$

Proof Terms

$$\dfrac{\Gamma \vdash e_1 :: \sigma \qquad \Gamma \vdash e_2 :: \tau}{\Gamma \vdash (e_1, e_2) :: (\sigma, \tau)}\,{\times}\mathsf{I}$$

$$\dfrac{\Gamma \vdash e :: (\sigma, \tau)}{\Gamma \vdash \mathit{fst}\ e :: \sigma}\,{\times}E_1 \qquad \dfrac{\Gamma \vdash e :: (\sigma, \tau)}{\Gamma \vdash \mathit{snd}\ e :: \tau}\,{\times}E_2$$

$$\dfrac{\Gamma \vdash e :: \sigma}{\Gamma \vdash \mathit{Left}\ e :: \sigma + \tau}\,{+}I_1 \qquad \dfrac{\Gamma \vdash e :: \tau}{\Gamma \vdash \mathit{Right}\ e :: \sigma + \tau}\,{+}I_2$$

$$\dfrac{\Gamma \vdash e :: \sigma + \tau \qquad \Gamma, x :: \sigma \vdash e_1 :: \gamma \qquad \Gamma, y :: \tau \vdash e_2 :: \gamma}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ L\,x \to e_1; R\,y \to e_2 :: \gamma}\,{+}\mathsf{E}$$

Programs are Proofs

See Figure 2.

Programs are Proofs

See Figure 3.

Curry-Howard Isomorphism

- It was noticed that programs and proofs have such correspondence. Types are propositions, programs are proofs.
- Logic is thus given a computational meaning.
  - A proof of $P \Rightarrow Q$, for example, is a function that takes a proof of P and produces a proof of Q;
  - A proof of $P \wedge Q$ is a pair consisting a proof of P and a proof of Q, etc.
- "Given a proposition, find a proof" is the type inhabitation problem.
- β reduction is proof reduction (proof simplification).
- Propositional logic is a simple logic with nice properties: every true proposition has a proof, etc. This nice properties carries over to simply typed λ calculus.
- There are stronger logic, though. When we design a new type system, we often ask ourselves what logic it corresponds to, and vice versa.

More Expressive Logic/Type Systems

- Second-order logic: allowing $\forall$ that quantifies over propositions (types):
  - e.g. $\forall a.(\forall b.b \to b) \to a \to a$.
  - That gives us polymorphic types.
  - Haskell's (original) Hindley-Milner type system is a more restrictive version that allows $\forall$ only at the outer-most level.
- First-order logic: allowing $\forall$ that quantifies over terms:
  - e.g $\forall m, n \in Nat.m \leqslant n \to 1 + m \leqslant 1 + n$.
  - *Dependent type*. A very expressive type system in which you may express various properties: e.g. a function returns a sorted list.

More Expressive Logic/Type Systems

- Allowing $\exists$: existential type. Used to express abstract datatypes.
  - $\exists a.a \wedge (a \to a \to a)$: some type that has a base element and an "addition" operator.
- Subtyping: $Nat \leqslant Real \leqslant Complex$... related to ad-hoc polymorphism.
- If we allow everything in the typing context to be used exactly once, e.g:

$$\frac{\Gamma_1 \vdash e_1 :: \sigma \to \tau \qquad \Gamma_2 \vdash e_2 :: \sigma}{\Gamma_1 \uplus \Gamma_2 \vdash e_1\, e_2 :: \tau} \to \mathsf{E}$$

  - where $\Gamma_1 \uplus \Gamma_2$ denotes disjoint union — Γ_1 and Γ_2 must not share elements,
  - we get *linear type*. Used to reason about usage of resources.

More Expressive Logic/Type Systems

- Recall the law of excluded middle:

$$\frac{}{\Gamma \vdash P \vee \neg P}\ \text{Excluded-Middle}$$

  - If we want such a rule for the types, what is the corresponding term?
  - It has to do with *continuations* — think of it as a kind of "go-to".
- And many more. Research on types is still a hot topic.

4 Polymorphic λ Calculus

Polymorphism

- Allowing values of different types to be handled through a uniform interface.
- Christopher Strachey descriped two kinds of polymorphism:
- *Ad-hoc* polymorphism: allowing potentially different code (e.g. $+$ for Int and $Float$) to "look the same".
  - e.g. function overloading, and method overloading in many OO languages.
  - e.g. type classes ($Eq\ a \Rightarrow \ldots$) in Haskell.

Let $\Gamma = P \to Q \to R, P \to Q, P$. Prove that $(P \to Q \to R) \to (P \to Q) \to P \to R$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{P \to Q \to R \in \Gamma}{\Gamma \vdash P \to Q \to R}\,\mathsf{Hyp} \quad \dfrac{P \in \Gamma}{\Gamma \vdash P}\,\mathsf{Hyp}}{\Gamma \vdash Q \to R}\,{\to}\mathsf{E} \quad \dfrac{\dfrac{P \to Q \in \Gamma}{\Gamma \vdash P \to Q}\,\mathsf{Hyp} \quad \dfrac{P \in \Gamma}{\Gamma \vdash P}\,\mathsf{Hyp}}{\Gamma \vdash Q}\,{\to}\mathsf{E}}{\Gamma \vdash R}\,{\to}\mathsf{E}}{P \to Q \to R, P \to Q \vdash P \to R}\,{\to}\mathsf{I}}{P \to Q \to R \vdash (P \to Q) \to P \to R}\,{\to}\mathsf{I}}{\vdash (P \to Q \to R) \to (P \to Q) \to P \to R}\,{\to}\mathsf{I}$$

Figure 2: Proving $(P \to Q \to R) \to (P \to Q) \to P \to R$

Let $\Gamma = f :: P \to Q \to R, g :: P \to Q, x :: P$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{f :: P \to Q \to R \in \Gamma}{\Gamma \vdash f :: P \to Q \to R}\,\mathsf{Var} \quad \dfrac{x :: P \in \Gamma}{\Gamma \vdash x :: P}\,\mathsf{Var}}{\Gamma \vdash f\,x :: Q \to R}\,{\to}\mathsf{E} \quad \dfrac{\dfrac{g :: P \to Q \in \Gamma}{\Gamma \vdash g :: P \to Q}\,\mathsf{Var} \quad \dfrac{x :: P \in \Gamma}{\Gamma \vdash x :: P}\,\mathsf{Var}}{\Gamma \vdash g\,x :: Q}\,{\to}\mathsf{E}}{\Gamma \vdash f\,x\,(g\,x) :: R}\,{\to}\mathsf{E}}{f :: P \to Q \to R, g :: P \to Q \vdash \lambda x.f\,x\,(g\,x) :: P \to R}\,{\to}\mathsf{I}}{f :: P \to Q \to R \vdash \lambda g.\lambda x.f\,x\,(g\,x) :: (P \to Q) \to P \to R}\,{\to}\mathsf{I}}{\vdash \lambda f.\lambda g.\lambda x.f\,x\,(g\,x) :: (P \to Q \to R) \to (P \to Q) \to P \to R}\,{\to}\mathsf{I}$$

Figure 3: The term $\lambda f.\lambda g.\lambda x.f\,x\,(g\,x)$ sufficiently records the proof, from which we can reconstruct the proof tree.

- *Parametric* polymorphism: allowing the same piece of code, which does not depend on the type of the input data, to be used on a wide range of types.
  - e.g. $reverse :: List\ a \to List\ a$ in Haskell.
- We will only briefly talk about the second kind.

Polymorphic λ Calculus (System F)

- Proposed by Girard.
- An additional construct in the syntax of types (where T ranges over type variables):

$$\begin{aligned}\tau = {} & Unit \mid Empty \mid Nat \mid T \\ & \mid \tau \to \tau \mid (\tau, \tau) \mid \tau + \tau \mid \forall T.\tau\end{aligned}$$

- And two additional construct of terms:

$$\dfrac{\Gamma \vdash e :: \tau \qquad a \text{ not free in } \Gamma}{\Gamma \vdash \Lambda a.e :: \forall a.\tau}\,\forall I$$

$$\dfrac{\Gamma \vdash e :: \forall a.\tau}{\Gamma \vdash e\ \sigma :: \tau[a \backslash \sigma]}\,\forall E$$

  Terms may take types as arguments!
- One more reduction rule: $(\Lambda a.e)\ \sigma \xrightarrow{\beta} e[a \backslash \sigma]$.

Example: Polymorphic *const*

Recall the function $const\ x\ y = x$ in Haskell. The corresponding function in System F:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x :: a \in \{x :: a, y :: b\}}{x :: a, y :: b \vdash x :: a}\,\mathsf{Var}}{x :: a \vdash \lambda(y :: b).x :: b \to a}\,{\to}I}{\vdash \lambda(x :: a).\lambda(y :: b).x :: a \to b \to a}\,{\to}I}{\vdash \Lambda b.\lambda(x :: a).\lambda(y :: b).x :: \forall b.a \to b \to a}\,\forall I}{\vdash \Lambda a.\Lambda b.\lambda(x :: a).\lambda(y :: b).x :: \forall a.\forall b.a \to b \to a}\,\forall I$$

Example: Using Polymorphic Functions

See Figure 4.

Second-Order Logic

- Recall Curry-Howard isomorphism? What logic does this type system correspond to?
- Ans: second-order (intuitionistic) logic. That is, propositional logic extended with $\forall$, and in all $\forall a$, a is a type (proposition).
  - There ought to be an $\exists$ operator too, but it can be simulated by $\forall$ and is often omitted.

Let $\Gamma = f :: \forall a.a \to a$. Abbreviate $Bool$ to $\mathbb{B}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{f :: \forall a.a \to a \in \Gamma}{\Gamma \vdash f :: \forall a.a \to a}\,\text{Var}}{\Gamma \vdash f\ Nat :: Nat \to Nat}\,\forall E \quad \dfrac{}{\Gamma \vdash 3 :: Nat}\,\text{Nat}}{\Gamma \vdash f\ Nat\ 3 :: Nat}\,\to E \quad \dfrac{\text{(omitted)}}{\Gamma \vdash f\ \mathbb{B}\ True :: \mathbb{B}}}{\Gamma \vdash (f\ Nat\ 3, f\ \mathbb{B}\ True) :: (Nat, \mathbb{B})}\,\times I}{\vdash \lambda f.(f\ Nat\ 3, f\ \mathbb{B}\ True) :: (\forall a.a \to a) \to (Nat, \mathbb{B})}\,\to I$$

Figure 4: Example: type checking a polymorphic function.

- 2nd-order logic: very expressive. You can encode all inductive and coinductive datatypes in it! (Search for Church encoding.)
- Sound. But no deductive system for it can be complete — there are true propositions that cannot be proved. Thus type inhabitance for it is undecidable.

Second-Order Logic/Polymorphic λ Calculus

- Type inference is undecidable.
- Type checking is decidable — for the Church style (where λ abound variables are annotated with types).
  - For Curry style, even type checking is undecidable.

Polymorphic Datatypes

- When we define a datatype $\textbf{data}\ Nat = Zero \mid Suc\ Nat$ in Haskell, we have introduced:
  - a type Nat,
  - two data constructors $Zero :: Nat$ and $Suc :: Nat \to Nat$.
- When we define a polymorphic data type $\textbf{data}\ List\ a = [\,] \mid a : List\ a$ in Haskell, we have introduced:
  - a type constructor $List$ — a function from a type to a type, e.g. from Int to $List\ Int$.
  - two data constructors $[\,]\_ :: \forall a.List\ a$, and $(:\_) :: \forall a.a \to List\ a \to List\ a$.
  - To build a list of Int we should have written, e.g, $1 :_{Int} 2 :_{Int} 3 :_{Int} [\,]_{Int}$. But in Haskell we always omit the type application (since they can be inferred).

Comparison

- Type of a polymorphic function in Haskell, e.g, $zip :: List\ a \to List\ b \to List\ (a, b)$, should actually be $\forall a.\forall b.List\ a \to List\ b \to List\ (a, b)$.
- In Haskell we omit all the type applications. E.g. we say $zip\ [1, 2]$ "ab" instead of $zip\ Nat\ Char\ [1, 2]$ "ab".
- Finally, Haskell uses a weaker system of polymorphic type:
  - Names of types starting with lower-case characters are assumed to be $\forall$-quantified. Another way to say that is "lower-cased types are type variables."
  - All $\forall$s appear at outer-most positions only. Thus $List\ a \to List\ b \to List\ (a, b)$ is seen as $\forall a.\forall b.List\ a \to List\ b \to List\ (a, b)$.
  - The type $(\forall a.a \to a) \to (Nat, \mathbb{B})$, which we have seen previously, is not allowed in (standard) Haskell 98!
  - Thus the $\forall$ symbol is not explicit written.
- Why these restrictions? To allow type inference!

5 Hindley-Milner Style Type Inference

Type Inference

Example: find τ such that $\vdash \lambda x.\lambda y.x :: \tau$. Note that x and y are no longer annotated with types. We have to somehow find them out.

$$\dfrac{\dfrac{\dfrac{x :: e \in \{x :: b, y :: d\}}{x :: b, y :: d \vdash x :: e}\,\text{Var}}{x :: b \vdash \lambda y.x :: c}\,\to I}{\vdash \lambda x.\lambda y.x :: a}\,\to I$$

$a = b \to c$

$c = d \to e$

$e = b$

Thus $\tau = \forall b. \forall d. b \to d \to b$.

Hindley-Milner Style Type Inference

- Assume the unknown types to be type variables.
- Proceed with the typing rules of simply typed λ calculus, and use a *unification engine* to discover constraints between the type variables.
  - Algorithms for unification can be quite nontrivial. We do not go into the details for this course, and rely merely on your intuition to perform the unification manually.
- If the procedure succeeds, $\forall$-quantify all the unconstrained variables.
- The procedure fails if we encounter circular constraints: $a = \ldots a \ldots$.

Example of a Type Error

$$\dfrac{\dfrac{\dfrac{f :: e \to c \in \{f :: b\}}{f :: b \vdash f :: e \to c}\,\text{Var} \quad \dfrac{f :: e \in \{f :: b\}}{f :: b \vdash f :: e}\,\text{Var}}{f :: b \vdash f\ f :: c}\,\to E}{\vdash \lambda f. f\ f :: a}\,\to I$$

$a = b \to c$

$b = e \to c$

$b = e$

The constraints imply $e = e \to c$, a circular type. So we signal a type error.

Hindley-Milner Type Inference

- The Hindley-Milner system is essentially a monomorphic type system disguised as polymorphic.
- By doing so it found a nice balance — limited polymorphism, with type inference.
- Adopted by some early typed functional languages. It was believed that programmers no longer need to write types.
- Later, people were not satisfied with its limitation, and the Haskell type system was extended with more features (e.g. those more like System F, type classes, etc) but we lost full type inference.

More on Types

- Many other aspects on types that we won't have time to talk about:
- "free theorems" of polymorphic functions.
  - What can we say about a function $f :: \forall a. List\ a \to a$?
  - $f \cdot map\ g = g \cdot f$.
- Existential type ($\exists$) is dual to $\forall$, and implements abstract data types (e.g $\exists t. Printable\ t$).
- Subtyping: $Nat \leqslant Real \leqslant Complex$... related to ad-hoc polymorphism.
- What type corresponds to first order logic? Dependent type, a highly expressive type system.
- And many more. Research on types is still a hot topic.