Introductory Course
on Logic and Automata Theory

Introduction to type systems

Polyvios.Pratikakis@imag.fr

Based on slides by Jeff Foster, UMD

The need for types

- Consider the lambda calculus terms:
  - $\mathsf{false} = \lambda x.\lambda y.x$
  - $0 = \lambda x.\lambda y.x$ (Scott encoding)

- Everything is encoded using functions
  - One can easily misuse combinators
    - $\mathsf{false}\ 0$, or $\mathsf{if}\ 0\ \mathtt{then}\ \ldots$, etc...
  - It's no better than assembly language!

Type system

- A *type system* is some mechanism for distinguishing good programs from bad
  - Good programs are *well typed*
  - Bad programs are ill typed or not typeable
- Examples:
  - $0 + 1$ is well typed
  - $\mathsf{false}\ 0$ is ill typed: booleans cannot be applied to numbers
  - $1 + (\mathsf{if}\ \mathtt{true}\ \ \mathtt{then}\ 0\ \mathsf{else}\ \mathsf{false})$ is ill typed: cannot add a boolean to an integer

A definition

"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute."

– Benjamin Pierce, Types and Programming Languages

Simply-typed lambda calculus

- $e ::= n \mid x \mid \lambda x : \tau . e \mid e\ e$
  - Functions include the type of their argument
  - We don't need this yet, but it will be useful later
- $\tau ::= \mathsf{int} \mid \tau \rightarrow \tau$
  - $\tau_1 \rightarrow \tau_2$ is a the type of a function that, given an argument of type τ_1, returns a result of type τ_2
  - We say τ_1 is the *domain*, and τ_2 is the *range*

Typing judgements

- The type system will prove judgments of the form
  - $\Gamma \vdash e : \tau$
  - "In type environment Γ, expression e has type τ"

Type environments

- A *type environment* is a map from variables to types (similar to a symbol table)
  - $\emptyset$ is the empty type environment
    - A closed term e is well-typed if $\emptyset \vdash e : \tau$ for some τ
    - Also written as $\vdash e : \tau$
  - $\Gamma, x : \tau$ is just like Γ except x has type τ
    - The type of x in Γ is τ
    - For $z \neq x$, the type of z in $\Gamma, x : \tau$ is the type of z in Γ (we look up variables from the end)
- When we see a variable in a program, we look in the type environment to find its type

Type rules

$$\frac{}{\Gamma \vdash n : \mathsf{int}} \qquad \frac{x \in \mathsf{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau.e : \tau \to \tau'} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}$$

Example

Assume $\Gamma = (- : \mathsf{int} \to \mathsf{int})$

$$\dfrac{\dfrac{- \in \mathsf{dom}(\Gamma)}{\Gamma \vdash - : \mathsf{int} \to \mathsf{int}} \qquad \dfrac{}{\Gamma \vdash 3 : \mathsf{int}}}{\Gamma \vdash -\ 3 : \mathsf{int}}$$

An algorithm for type checking

- The above type rules are deterministic
  - For each syntactic form of a term e, only one rule is possible
- They define a natural type checking algorithm

TypeCheck : (type_env $\times$ expression) $\longrightarrow$ **type**

TypeCheck(Γ, n) = int

TypeCheck(Γ, x) = **if** $x \in$ dom(Γ) **then** $\Gamma(x)$ **else** fail

TypeCheck(Γ, $\lambda x : \tau.e$) = TypeCheck($(\Gamma, x : \tau)$, e)

TypeCheck(Γ, e_1 e_2) =

let τ_1 = TypeCheck(Γ, e_1) **in**

let τ_2 = TypeCheck(Γ, e_2) **in**

if dom(τ_1) = τ_2 **then** range(τ_1) **else** fail

Reminder: semantics

- Small-step, call-by-value semantics
  - If an expression is not a value, and cannot evaluate any more, we say it is *stuck*, e.g. $0\ 1$

$$\frac{}{(\lambda x : \tau . e_1)\ v_2 \rightarrow e_1[v_2/x]} \qquad \frac{e_1 \rightarrow e_1'}{e_1\ e_2 \rightarrow e_1'\ e_2}$$

$$\frac{e_2 \rightarrow e_2'}{v_1\ e_2 \rightarrow v_1\ e_2'}$$

where

$$\begin{array}{rcl} e & ::= & v \mid x \mid e\ e \\ v & ::= & n \mid \lambda x : \tau . e \end{array}$$

Progress theorem

- *If* $\vdash e : \tau$ *then either* e *is a value, or there exists* e' *such that* $e \to e'$
- Proof by induction on e
  - Base cases (values): $n, \lambda x : \tau.e$, trivially true
  - Case x: impossible, untypeable in empty environment
  - Inductive case: $e_1\ e_2$
    - If e_1 is not a value, apply the theorem inductively and then second semantic rule.
    - If e_1 is a value but e_2 is not, similar (using the third semantic rule)
    - If e_1 and e_2 are values, then e_1 has function type, and the only value with a function type is a lambda term, so we apply the first semantic rule

Preservation theorem

- *If* $\vdash e : \tau$ *and* $e \rightarrow e'$ *then* $\vdash e' : \tau$
- Proof by induction on $e \rightarrow e'$
  - In all cases (one for each rule), e has the form $e = e_1\ e_2$
  - Inversion: from $\vdash e_1\ e_2 : \tau$ we get $\vdash e_1 : \tau' \rightarrow \tau$ and $\vdash e_2 : \tau'$
  - Three semantic rules:
    - Second or third rule: apply induction on e_1 or e_2 respectively, and type-rule for application
    - Remaining case: $(\lambda x : \tau'.e)\ v \rightarrow e[v/x]$

Preservation continued

- Case $(\lambda x : \tau'.e)\ v \rightarrow e[v/x]$
  From hypothesis:

$$\frac{x : \tau' \vdash e : \tau}{\vdash \lambda x : \tau'.e : \tau' \rightarrow \tau}$$

- To finish preservation proof, we must prove $\vdash e[v/x] : \tau$.
- Susbstitution lemma

Substitution lemma

- *If* $\Gamma \vdash v : \tau$ *and* $\Gamma, x : \tau \vdash e : \tau'$*, then* $\Gamma \vdash e[v/x] : \tau'$
- Proof by induction on e (not shown)

- For lazy (call by name) semantics, substitution lemma is *If* $\Gamma \vdash e_1 : \tau$ *and* $\Gamma, x : \tau \vdash e : \tau'$*, then* $\Gamma \vdash e[e_1/x] : \tau'$

Soundness

- So far:
  - Progress: If $\vdash e : \tau$, then either e is a value, or there exists e' such that $e \rightarrow e'$
  - Preservation: If $\vdash e : \tau$ and $e \rightarrow e'$ then $\vdash e' : \tau$
- Putting these together, we get *soundness*
  - *If* $\vdash e : \tau$ *then either there exists a value* v *such that* $e \rightarrow^* v$ *or* e *doesn't terminate*
- What does this mean?
  - "Well-typed programs don't go wrong"
  - Evaluation never gets stuck

Product types (tuples)

$$e ::= \ldots \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e$$

$$\tau ::= \ldots \mid \tau \times \tau$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash (e_1, e_2) : \tau \times \tau'}$$

$$\frac{\Gamma \vdash e : \tau \times \tau'}{\Gamma \vdash \mathsf{fst}\ e : \tau} \qquad \frac{\Gamma \vdash e : \tau \times \tau'}{\Gamma \vdash \mathsf{snd}\ e : \tau'}$$

- Alternatively, using function signatures
  - $\mathsf{pair} : \tau \to \tau' \to \tau \times \tau'$
  - $\mathsf{fst} \ : \tau \times \tau' \to \tau$
  - $\mathsf{snd} \ : \tau \times \tau' \to \tau'$

Sum types

$$e ::= \ldots \mid \mathsf{inL}_{\tau_2}\ e \mid \mathsf{inR}_{\tau_1}\ e \mid (\mathsf{case}\ e\ \mathsf{of}\ x_1 : \tau_1 \rightarrow e_1 \mid x_2 : \tau_2 \rightarrow e_2)$$

$$\tau ::= \ldots \mid \tau + \tau$$

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{inL}_{\tau_2}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{inR}_{\tau_1}\ e : \tau_1 + \tau_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau_1 + \tau_2 \\ \Gamma, x_1 : \tau_1 \vdash e_1 : \tau \\ \Gamma, x_2 : \tau_2 \vdash e_2 : \tau\end{array}}{\Gamma \vdash (\mathsf{case}\ e\ \mathsf{of}\ x_1 : \tau_1 \rightarrow e_1 \mid x_2 : \tau_2 \rightarrow e_2) : \tau}$$

Self application and types

- Self application is not typeable in this system

$$\dfrac{\dfrac{\dfrac{\ldots}{\Gamma, x :? \vdash x : \tau \to \tau'} \qquad \dfrac{\ldots}{\Gamma, x :? \vdash x : \tau}}{\Gamma, x :? \vdash x\ x : \ldots}}{\Gamma \vdash \lambda x :?. x\ x : \ldots}$$

- We need a type τ such that $\tau = \tau \to \tau'$
- The simply-typed lambda calculus is *strongly normalizing*
  - Every program has a normal form
  - Or, every program halts!

Recursive types

- We can type self application if we have a type to represent the solution to equations like $\tau = \tau \to \tau'$
  - We define the type $\mu\alpha.\tau$ to be the solution to the (recursive) equation $\alpha = \tau$
  - Example: $\mu\alpha.\text{int} \to \alpha$

Folding/unfolding recursive types

- Inferred: We can check type equivalence with the previous definition
  - Standard unification, omit occurs checks (explained later)
- Alternative method: explicit fold/unfold
  - The programmer inserts explicit fold and unfold operations to expand/contract a “level” of the type
    - $\mathsf{unfold}\ \mu\alpha.\tau = \tau[\mu\alpha.\tau/\alpha]$
    - $\mathsf{fold}\ \tau[\mu\alpha.\tau/\alpha] = \mu\alpha.\tau$

Fold-based recursive types

$$e ::= \ldots \mid \mathsf{fold}\ e \mid \mathsf{unfold}\ e$$

$$\tau ::= \ldots \mid \mu\alpha.\tau$$

$$\frac{\Gamma \vdash e : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \vdash \mathsf{fold}\ e : \mu\alpha.\tau}$$

$$\frac{\Gamma \vdash e : \mu\alpha.\tau}{\Gamma \vdash \mathsf{unfold}\ e : \tau[\mu\alpha.\tau/\alpha]}$$

ML Datatypes

- Combines fold/unfold-style recursive and sum types
  - Each occurence of a type constructor when producing a value corresponds to $\mathsf{inR}/\mathsf{inL}$ and (if recursive,) also fold
  - Each occurence of a type constructor in a pattern match corresponds to a case and (if recursive,) at least one unfold

ML Datatypes examples

- **type** intlist =

  Int **of** int

  | Cons **of** int $*$ intlist

  is equivalent to $\mu\alpha.\mathsf{int} + (\mathsf{int} \times \alpha)$

- (Int 3)

  is equivalent to $\mathsf{fold}\ (\mathsf{inL}_{\mathsf{int} \times \mu\alpha.\mathsf{int} + (\mathsf{int} \times \alpha)} 3)$

More ML Datatype examples

- (Cons (42, (Int 3)))

  is equivalent to $\mathsf{fold}\ (\mathsf{inR}_{\mathsf{int}}(2, \mathsf{fold}\ (\mathsf{inL}_{\mathsf{int}\times\mu\alpha.\mathsf{int}+(\mathsf{int}\times\alpha)}3)))$

- match e **with**

  Int x $\longrightarrow e_1$

  | Cons x $\longrightarrow e_2$

  is equivalent to

$$\begin{array}{l} \mathsf{case}\ (\mathsf{unfold}\ e)\ \mathsf{of} \\ \quad x : \mathsf{int} \rightarrow e_1 \\ \quad \mid x : \mathsf{int} \times (\mu\alpha.\mathsf{int} + (\mathsf{int} \times \alpha)) \rightarrow e_2 \end{array}$$

Discussion

- In the pure lambda calculus, every term is typeable with recursive types
  - "Pure" means only including functions and variables (the calculus from the last class)
- Most languages have some kind of "recursive" type
  - Used to encode data structures like e.g. lists, trees, ...
- However, usually two recursive types are differentiated by name, even when they define the same structure
  - For example,

    ```
    struct foo { int x; struct foo *next; }
    ```

    is different from

    ```
    struct bar { int x; struct bar *next; }
    ```

Intermission

Next: Curry-Howard

correspondence

Classical propositional logic

- Formulas of the form

$$\phi ::= p \mid \bot \mid \phi \vee \phi \mid \phi \wedge \phi \mid \phi \rightarrow \phi$$

- Where $p \in \mathcal{P}$ is an atomic proposition, e.g. "Socrates is a man"
- Convenient abbreviations:
  - $\neg\phi$ means $\phi \rightarrow \bot$
  - $\phi \longleftrightarrow \phi'$ means $(\phi \rightarrow \phi') \wedge (\phi' \rightarrow \phi)$

Semantics of classical logic

- Interpretation $\mathfrak{m} : \mathcal{P} \longrightarrow \{\mathsf{true}, \mathsf{false}\}$

$$
\begin{array}{rcl}
[\![p]\!]^{\mathfrak{m}} & = & \mathfrak{m}(p) \\
[\![\bot]\!]^{\mathfrak{m}} & = & \mathsf{false} \\
[\![\phi \wedge \phi']\!]^{\mathfrak{m}} & = & [\![\phi]\!]^{\mathfrak{m}} \bar{\wedge} [\![\phi']\!]^{\mathfrak{m}} \\
[\![\phi \vee \phi']\!]^{\mathfrak{m}} & = & [\![\phi]\!]^{\mathfrak{m}} \bar{\vee} [\![\phi']\!]^{\mathfrak{m}} \\
[\![\phi \longrightarrow \phi']\!]^{\mathfrak{m}} & = & \bar{\neg}[\![\phi]\!]^{\mathfrak{m}} \bar{\vee} [\![\phi']\!]^{\mathfrak{m}}
\end{array}
$$

- Where $\bar{\wedge}, \bar{\vee}, \bar{\neg}$ are the standard boolean operations on $\{\mathsf{true}, \mathsf{false}\}$

Terminology

- A formula ϕ is *valid* if $[\![\phi]\!]^{\mathfrak{m}} = \mathsf{true}$ for all $\mathfrak{m}$
- A formula ϕ is *unsatisfiable* if $[\![\phi]\!]^{\mathfrak{m}} = \mathsf{false}$ for all $\mathfrak{m}$
- *Law of excluded middle*:
- Formula $\phi \vee \neg\phi$ is valid for any ϕ
- A *proof system* attempts to determine the validity of a formula

Proof theory for classical logic

- Proves judgements of the form $\Gamma \vdash \phi$:
  - For any interpretation, under assumption Γ, ϕ is true
- Syntactic deduction rules that produce “proof trees” of $\Gamma \vdash \phi$: *Natural deduction*
- Problem: classical proofs only address truth value, not constructive
- Example: “There are two irrational numbers x and y, such that x^y is rational”
  - Proof does not include much information

Intuitionistic logic

- Get rid of the law of excluded middle
- Notion of “truth” is not the same
  - A proposition is true, if we can construct a proof
  - Cannot assume predefined truth values without constructed proofs (no “either true or false”)
- Judgements are not expression of “truth”, they are constructions
  - $\vdash \phi$ means “there is a proof for ϕ”
  - $\vdash \phi \rightarrow \bot$ means “there is a refutation for ϕ”, not “there is no proof”
  - $\vdash (\phi \rightarrow \bot) \rightarrow \bot$ only means the absense of a refutation for ϕ, does not imply ϕ as in classical logic

Proofs in intuitionistic logic

$$\frac{}{\Gamma, \phi \vdash \phi} \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash \phi}$$

$$\frac{\Gamma \vdash \phi \qquad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi} \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi} \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi}$$

$$\frac{\Gamma \vdash \phi}{\Gamma \vdash \phi \vee \psi} \qquad \frac{\Gamma \vdash \psi}{\Gamma \vdash \phi \vee \psi} \qquad \frac{\begin{array}{c}\Gamma, \phi \vdash \rho \qquad \Gamma, \psi \vdash \rho \\ \Gamma \vdash \phi \vee \psi\end{array}}{\Gamma \vdash \rho}$$

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \rightarrow \psi} \qquad \frac{\Gamma \vdash \phi \rightarrow \psi \qquad \Gamma \vdash \phi}{\Gamma \vdash \psi}$$

Does that resemble anything?

Curry-Howard correspondence

- We can mechanically translate formulas ϕ into type τ for every ϕ and the reverse
  - E.g. replace $\wedge$ with $\times$, $\vee$ with $+$, $\ldots$
- *If* $\Gamma \vdash e : \tau$ *in simply-typed lambda calculus, and* τ *translates to* ϕ*, then* $\mathsf{range}(\Gamma) \vdash \phi$ *in intuitionistic logic*
- *If* $\Gamma \vdash \phi$ *in intuitionistic logic, and* ϕ *translates to* τ*, then there exists* e *and* Γ' *such that* $\mathsf{range}(\Gamma') = \Gamma$ *and* $\Gamma' \vdash e : \tau$
- Proof by induction on the derivation $\Gamma \vdash \phi$
  - Can be simplified by fixing the logic and type languages to match

Consequences

- Lambda terms encode proof trees
- Evaluation of lambda terms is proof simplification
- Automated proving by trying to construct a lambda term with the wanted type
- Verifying a proof is typechecking
  - Increased trust in complicated proofs when machine-verifiable
- Proof-carrying code
- Certifying compilers

So far. . .

- We have discussed simple types
  - Integers, functions, pairs, unions
  - Extension for recursive types
- Type systems have nice properties
  - Type checking is straightforward (needs annotations)
  - Well-typed programs don't go wrong (get stuck)
- But. . . , we cannot type-check all good programs
  - Sound, but not complete
  - Might reject correct programs (have false warnings)

Next: improving types

- How can we build more flexible type systems?
  - More programs type check
  - Type checking is still tractable
- How can we reduce the annotation burden?
  - Type inference: infer annotations automatically

Parametric polymorphism

- Observation: $\lambda x.x$ returns its argument exactly without any constraints on the type of x
  - The identity function works for any argument type
- We can express this with *universal quantification*
  - $\lambda x.x : \forall\alpha.\alpha \to \alpha$
  - For any type α the identity function has type $\alpha \to \alpha$
  - This is also known as *parametric polymorphism*

System F: annotated polymorphism

- We extend the previous system:

$$\tau ::= \mathsf{int} \mid \tau \to \tau \mid \alpha \mid \forall \alpha . \tau$$

$$e ::= n \mid x \mid \lambda x . e \mid e\ e \mid \Lambda \alpha . e \mid e[\tau]$$

- We add polymorphic types, and we add explicit *type abstraction* (generalization over all α) ...
  - Explicit creation of values ov polymorphic types
- ... and *type application* (instantiation of generalized types)
  - Explicitly marks code locations where a value of polymorphic type is used
- System by Girard, concurrently Reynolds

Defining polymorphic functions

- Polymorphic functions map types to terms
  - Normal functions map terms to terms
- Examples
  - $\Lambda\alpha.\lambda x : \alpha.x : \forall\alpha.\alpha \to \alpha$
  - $\Lambda\alpha.\Lambda\beta.\lambda x : \alpha.\lambda y : \beta.x : \forall\alpha.\forall\beta.\alpha \to \beta \to \alpha$
  - $\Lambda\alpha.\Lambda\beta.\lambda x : \alpha.\lambda y : \beta.y : \forall\alpha.\forall\beta.\alpha \to \beta \to \beta$

Instantiation

- When we use a parametric polymorphic type, we *apply* or *instantiate* it with a particular type
  - In System F this is done by hand
  - $(\Lambda\alpha.\lambda x : \alpha.x)[\tau_1] : \tau_1 \to \tau_1$
  - $(\Lambda\alpha.\lambda x : \alpha.x)[\tau_2] : \tau_2 \to \tau_2$
- This is where the term *parametric* comes from
  - The type $\forall\alpha.\alpha \to \alpha$ is a “function” in the domain of types
  - It is given a parameter at instantiation time

Type rules

$$\frac{\Gamma, \alpha \vdash e : \tau}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.\tau} \qquad \frac{\Gamma \vdash e : \forall\alpha.\tau}{\Gamma \vdash e[\tau'] : \tau[\tau'/\alpha]}$$

- Notice that there are no constructs for manipulating values of polymorphic type
  - This justifies instantiation with *any* type–that's what "for all" means
- We add α to Γ: we use this to ensure types are well-formed

Small-step semantics rules

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

- We have to extend the definition of substitution for types

Free variables (again)

- We need to perform substitutions on quantified types
  - Just like with lambda calculus, we need to think about free variables and avoid capturing with substitution
- Define the free variables of a type

$$\begin{array}{rcl} FV(\alpha) & = & \{\alpha\} \\ FV(c) & = & \emptyset \\ FV(\tau \rightarrow \tau') & = & FV(\tau) \cup FV(\tau') \\ FV(\forall \alpha.\tau) & = & FV(\tau) \setminus \{\alpha\} \end{array}$$

Substitution (again)

- We define $\tau[u/\alpha]$ as

$$
\begin{array}{rcll}
\alpha[u/\alpha] & = & u & \\
\beta[u/\alpha] & = & \beta & \text{where } \beta \neq \alpha \\
(\tau \rightarrow \tau')[u/\alpha] & = & \tau[u/\alpha] \rightarrow \tau'[u/\alpha] & \\
(\forall \beta . \tau)[u/\alpha] & = & \forall \beta . (\tau[u/\alpha]) & \text{where } \beta \neq \alpha \wedge \beta \notin FV(u)
\end{array}
$$

- We define $e[u/\alpha]$ as

$$
\begin{array}{rcll}
(\lambda x : \tau . e)[u/\alpha] & = & (\lambda x : \tau[u/\alpha] . e[u/\alpha] & \\
(\Lambda \beta . e)[u/\alpha] & = & \Lambda \beta . e[u/\alpha] & \text{where } \beta \neq \alpha \wedge \beta \notin FV(u) \\
(e_1 \; e_2)[u/\alpha] & = & e_1[u/\alpha] \; e_2[u/\alpha] & \\
x[u/\alpha] & = & x & \\
n[u/\alpha] & = & n &
\end{array}
$$

Type inference

- Consider the simply typed lambda calculus with integers
  - $e ::= n \mid x \mid \lambda x.\tau \mid e\ e$
  - Simple, no polymorphism
- Type inference: Given a bare term (no annotations), can we reconstruct a valid typing if there is one?

Type language

- Problem: consider the rule for functions

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau . e : \tau \rightarrow \tau'}$$

- Without type annotations on λ terms, where do we find τ?
  - We use *type variables* in place of as-yet-unknown types
    - $\tau ::= \alpha \mid \mathsf{int} \mid \tau \rightarrow \tau$
  - We generate *equality constraints* $\tau = \tau'$ among types and type variables
    - We solve the constraints to compute a typing

Type inference rules

$$\frac{}{\Gamma \vdash n : \mathsf{int}} \qquad \frac{x \in \mathsf{dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma, x : \alpha \vdash e : \tau' \qquad \alpha \text{ fresh}}{\Gamma \vdash \lambda x.e : \alpha \to \tau'} \qquad \frac{\begin{array}{cc}\Gamma \vdash e_1 : \tau_1 & \Gamma \vdash e_2 : \tau_2 \\ \tau_1 = \tau_2 \to \beta & \beta \text{ fresh}\end{array}}{\Gamma \vdash e_1\ e_2 : \beta}$$

Example

$$\dfrac{\dfrac{\Gamma, x : \alpha \vdash x : \alpha}{\Gamma \vdash (\lambda x.x) : \alpha \to \alpha} \qquad \Gamma \vdash 3 : \mathsf{int} \qquad \alpha \to \alpha = \mathsf{int} \to \beta}{\Gamma \vdash (\lambda x.x)\ 3 : \beta}$$

- We collect all constraints appearing in the derivation into a set C to be solved
- Here C contains $\alpha \to \alpha = \mathsf{int} \to \beta$
  - Solution: $\alpha = \beta = \mathsf{int}$
- So, this program is typeable
- We can create a typing by replacing variables in the derivation

Solving equality constraints

- We can solve the equality constraints using the following rewrite rules, which reduce a larger set of constraints to a smaller set
  - $C \cup \{\mathsf{int} = \mathsf{int}\} \Rightarrow C$
  - $C \cup \{\alpha = \tau\} \Rightarrow C[\tau/\alpha]$
  - $C \cup \{\tau = \alpha\} \Rightarrow C[\tau/\alpha]$
  - $C \cup \{\tau_1 \to \tau_2 = \tau_1' \to \tau_2'\} \Rightarrow C \cup \{\tau_1 = \tau_1'\} \cup \{\tau_2 = \tau_2'\}$
  - $C \cup \{\mathsf{int} = \tau_1 \to \tau_2\} \Rightarrow$ unsatisfiable
  - $C \cup \{\tau_1 \to \tau_2 = \mathsf{int}\} \Rightarrow$ unsatisfiable

Termination

- We can prove that the constraint solving algorithm terminates
- For each rewriting rule, either
  - We reduce the size of the constraint set
  - We reduce the number of “arrow” constructors in the constraint set
- As a result, the constraint always gets “smaller” and eventually becomes empty
  - A similar argument is made for strong normalization of the simply-typed lambda calculus

Occurs check

- We don't have recursive types, so we don't infer them
- In the operation $C[\tau/\alpha]$ we require that $\alpha \notin FV(\tau)$
- In practice, it may be better to allow $\alpha \in FV(\tau)$ and perform an *occurs check* at the end
  - Could be difficult to implement
  - Unification might not terminate without occurs check

Unifying a variable and a type

- Computing $C[\tau/\alpha]$ by substitution is inefficient
- Instead, use a union-find data structure to represent equal types
  - The terms are in a union-find forest
  - When a variable and a term are equated, we union them so they have the same equivalence class
  - If there is a concrete type in an equivalence class, use it to represent the class
  - Solution is the representative of each class at the end

Example

$$\alpha = \mathsf{int} \to \beta$$

$$\gamma = \mathsf{int} \to \mathsf{int}$$

$$\alpha = \gamma$$

Unification

- The process of finding a solution to a set of equality constraints is called *unification*
  - Original algorithm by Robinson (inefficient)
  - Often written in different form (algorithm W)
  - Usually solved on-line as type rules are applied

Discussion

- The algorithm we've given finds the *most general type* of a term
  - Any other type is "more specific"
  - Formally, any other valid type can be created from the most general type by applying a substitution on type variables
- All this is for a monomorphic type system, no quantification
  - Variables α stand for "some particular type"

Inference for polymorphism

- We would like to have the power of System F, and the ease of use of type inference
  - In short: given an untyped lambda calculus term, can we discover the annotations necessary for typing the term in System F, if such a typing is possible?
  - Unfortunately, no. This problem has been shown to be undecidable.
- Can we at least perform some kind of parametric polymorphism
  - Yes, a “sweet spot” was found by Hindley and Milner
  - Abstraction at let-statements, instantiation on each variable use
  - Used in ML

Curry-Howard extension

- Does the equivalent logic get extended using polymorphism?
  - More than enough to encode first-order logic
  - A subset of System F (with several restrictions) is equivalent to First Order Logic
- Actually, we can use System F as a common language for polymorphic lambda calculus and for second-order propositional logic!

Other extensions

- Higher-order logic is encodable in System F_ω
- Further extension using the Calculus of Constructions can encode even more complex logics
- Inductive types can encode algebraic data types, inductive proofs
- Combining the calculus of constructions with inductive types: The Calculus of Inductive Constructions:
  - Proof language used in the Coq theorem prover
  - Can encode and reason about other logics, type systems
  - Can also be used to reason about Classical Logic
    - Just make the law of excluded middle an axiom