Introduction to type systems

Polyvios.Pratikakis@imag.fr

Based on slides by Jeff Foster, UMD
The need for types

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

- Everything is encoded using functions
  - One can easily misuse combinators
    - \( \text{false} \), \( \text{0} \), or if \( \text{0} \) 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
- $\text{false} \ 0$ is ill typed: booleans cannot be applied to numbers
- $1 + (\text{if true then} \ 0 \ \text{else false})$ is ill typed: cannot add a boolean to an integer
“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

\[ \tau ::= \text{int} \mid \tau \rightarrow \tau \]

\[ \tau_1 \rightarrow \tau_2 \] is a the type of a function that, given an argument of type \( \tau_1 \), returns a result of type \( \tau_2 \)

We say \( \tau_1 \) is the domain, and \( \tau_2 \) is the range
Typing judgements

The type system will prove judgments of the form

\[ \Gamma \vdash e : \tau \]

“In type environment \( \Gamma \), expression \( e \) has type \( \tau \)”
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 $\tau$
- Also written as $\vdash e : \tau$
- $\Gamma, x : \tau$ is just like $\Gamma$ except $x$ has type $\tau$
- The type of $x$ in $\Gamma$ is $\tau$
- For $z \neq x$, the type of $z$ in $\Gamma, x : \tau$ is the type of $z$ in $\Gamma$ (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

\[ \Gamma \vdash n : \text{int} \]

\[ \Gamma, x : \tau \vdash e : \tau' \quad \Gamma \vdash \lambda x : \tau. e : \tau \rightarrow \tau' \]

\[ x \in \text{dom}(\Gamma) \]

\[ \Gamma \vdash x : \Gamma(x) \]

\[ \Gamma \vdash e_1 : \tau \rightarrow \tau' \]

\[ \Gamma \vdash e_2 : \tau \]

\[ \Gamma \vdash e_1 \ e_2 : \tau' \]
Example

Assume $\Gamma = (- : \text{int} \rightarrow \text{int})$

\[
\frac{- \in \text{dom}(\Gamma)}{\Gamma \vdash - : \text{int} \rightarrow \text{int}} \quad \frac{\Gamma \vdash 3 : \text{int}}{\Gamma \vdash - 3 : \text{int}}
\]
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

$\text{TypeCheck} : (\text{type_env} \times \text{expression}) \rightarrow \text{type}$

- $\text{TypeCheck}(\Gamma, x) = \text{if } x \in \text{dom}(\Gamma) \text{ then } \Gamma(x) \text{ else fail}$
- $\text{TypeCheck}(\Gamma, \lambda x : \tau. e) = \text{TypeCheck}((\Gamma, x : \tau), e)$
- $\text{TypeCheck}(\Gamma, e_1 \ e_2) =$
  - $\text{let } \tau_1 = \text{TypeCheck}(\Gamma, e_1) \text{ in}$
  - $\text{let } \tau_2 = \text{TypeCheck}(\Gamma, e_2) \text{ in}$
  - $\text{if } \text{dom}(\tau_1) = \tau_2 \text{ then } \text{range}(\tau_1) \text{ else fail}$
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. $0 \ 1$

\[
(\lambda x : \tau. e_1) \ v_2 \rightarrow e_1[v_2/x]
\]

\[
\frac{e_1 \rightarrow e'_1}{e_1 \ e_2 \rightarrow e'_1 \ e_2}
\]

where

\[
e \ ::= \ v \mid x \mid e \ e
\]

\[
v \ ::= \ n \mid \lambda x : \tau. e
\]
Progress theorem

If $\vdash e : \tau$ then either $e$ is a value, or there exists $e'$ such that $e \rightarrow e'$

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 $e = e_1 \cdot e_2$
- Inversion: from $\vdash e_1 \cdot e_2 : \tau$ 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:

\[
\begin{align*}
x : \tau' & \vdash e : \tau \\
\vdash \lambda x : \tau'. e : \tau' \rightarrow \tau
\end{align*}
\]

To finish preservation proof, we must prove \(\vdash e[v/x] : \tau\).

Substitution 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)

\[ e ::= \ldots \mid \text{fst } e \mid \text{snd } e \]
\[ \tau ::= \ldots \mid \tau \times \tau \]

\[ \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash (e_1, e_2) : \tau \times \tau'} \]

- Alternatively, using function signatures
  - \text{pair} : \tau \rightarrow \tau' \rightarrow \tau \times \tau'
  - \text{fst} : \tau \times \tau' \rightarrow \tau
  - \text{snd} : \tau \times \tau' \rightarrow \tau'
Sum types

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

\[ \tau ::= \ldots \mid \tau + \tau \]

\[ \Gamma \vdash e : \tau_1 \quad \Gamma \vdash e : \tau_2 \]

\[ \frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash e : \tau_2}{\Gamma \vdash \text{inL}_{\tau_2} e : \tau_1 + \tau_2} \quad \frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash e : \tau_2}{\Gamma \vdash \text{inR}_{\tau_1} e : \tau_1 + \tau_2} \]

\[ \frac{\Gamma, x_1 : \tau_1 \vdash e_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash e_2 : \tau}{\Gamma \vdash (\text{case } e \text{ of } x_1 : \tau_1 \rightarrow e_1 \mid x_2 : \tau_2 \rightarrow e_2) : \tau} \]
Self application and types

- Self application is not typeable in this system

\[ \Gamma, x : \tau \vdash x : \tau \rightarrow \tau' \]
\[ \Gamma, x : \tau \vdash x : \ldots \]
\[ \Gamma \vdash \lambda x . x : \ldots \]

- We need a type \( \tau \) such that \( \tau = \tau \rightarrow \tau' \)

- 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 $\tau = \tau \rightarrow \tau'$
- We define the type $\mu \alpha. \tau$ to be the solution to the (recursive) equation $\alpha = \tau$
- Example: $\mu \alpha. \text{int} \rightarrow \alpha$
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
- Unfold \( \mu \alpha. \tau = \tau[\mu \alpha. \tau / \alpha] \)
- Fold \( \tau[\mu \alpha. \tau / \alpha] = \mu \alpha. \tau \)
Fold-based recursive types

\[ e ::= \ldots | \text{fold } e | \text{unfold } e \]

\[ \tau ::= \ldots | \mu \alpha.\tau \]

\[ \Gamma \vdash e : \tau[\mu \alpha.\tau/\alpha] \]

\[ \Gamma \vdash \text{fold } e : \mu \alpha.\tau \]

\[ \Gamma \vdash e : \mu \alpha.\tau \]

\[ \Gamma \vdash \text{unfold } e : \tau[\mu \alpha.\tau/\alpha] \]
ML Datatypes

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

- **type intlist =**
  
  Int of int
  
  | Cons of int * intlist

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

- (Int 3)

  is equivalent to \( \text{fold} \ (\text{inL} \text{int} \times \mu \alpha. \text{int} + (\text{int} \times \alpha) ^ 3) \)
More ML Datatype examples

(Cons (42, (Int 3)))

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

match $e$ with

\begin{align*}
\text{Int } x & \rightarrow e_1 \\
\text{Cons } x & \rightarrow e_2
\end{align*}

is equivalent to

\[
\text{case } (\text{unfold } e) \text{ of } \\
\quad x : \text{int} \rightarrow e_1 \\
\quad | \ x : \text{int} \times (\mu \alpha.\text{int} + (\text{int} \times \alpha)) \rightarrow e_2
\]
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,
    ```c
    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
  \[ \phi ::= p \mid \bot \mid \phi \lor \phi \mid \phi \land \phi \mid \phi \rightarrow \phi \]

- Where \( p \in \mathcal{P} \) is an atomic proposition, e.g. “Socrates is a man”

- Convenient abbreviations:
  - \( \neg \phi \) means \( \phi \rightarrow \bot \)
  - \( \phi \leftrightarrow \phi' \) means \( (\phi \rightarrow \phi') \land (\phi' \rightarrow \phi) \)
Semantics of classical logic

Interpretation \( m : \mathcal{P} \to \{\text{true, false}\} \)

\[
\begin{align*}
[\mathcal{P}]^m &= m(p) \\
[\bot]^m &= \text{false} \\
[\phi \land \phi']^m &= [\phi]^m \land [\phi']^m \\
[\phi \lor \phi']^m &= [\phi]^m \lor [\phi']^m \\
[\phi \to \phi']^m &= \neg [\phi]^m \lor [\phi']^m
\end{align*}
\]

Where \( \bar{\land}, \bar{\lor}, \bar{\neg} \) are the standard boolean operations on \( \{\text{true, false}\} \)
Terminology

- A formula $\phi$ is valid if $[\phi]^m = \text{true}$ for all $m$
- A formula $\phi$ is unsatisfiable if $[\phi]^m = \text{false}$ for all $m$
- Law of excluded middle:
  - Formula $\phi \lor \neg\phi$ is valid for any $\phi$
- 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 $\Gamma$, $\phi$ 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 $xy$ 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 $\phi$”
  - $\vdash \phi \rightarrow \bot$ means “there is a refutation for $\phi$”, not “there is no proof”
  - $\vdash (\phi \rightarrow \bot) \rightarrow \bot$ only means the absense of a refutation for $\phi$, does not imply $\phi$ as in classical logic
Proofs in intuitionistic logic

\[ \Gamma, \phi \vdash \phi \]

\[ \Gamma, \phi \vdash \psi \]

\[ \Gamma \vdash \phi \wedge \psi \]

\[ \Gamma \vdash \phi \]

\[ \Gamma \vdash \psi \]

\[ \Gamma \vdash \phi \lor \psi \]

\[ \Gamma \vdash \phi \rightarrow \psi \]

\[ \Gamma, \phi \vdash \rho \]

\[ \Gamma, \psi \vdash \rho \]

\[ \Gamma \vdash \rho \]

\[ \Gamma \vdash \phi \wedge \psi \]

\[ \Gamma \vdash \phi \lor \psi \]

\[ \Gamma \vdash \phi \rightarrow \psi \]

\[ \Gamma \vdash \phi \]

\[ \Gamma \vdash \psi \]

\[ \Gamma \vdash \phi \rightarrow \psi \]

\[ \Gamma \vdash \phi \]

\[ \Gamma \vdash \psi \]

\[ \Gamma \vdash \phi \rightarrow \psi \]

Does that resemble anything?
Curry-Howard correspondence

We can mechanically translate formulas $\phi$ into type $\tau$ for every $\phi$ and the reverse

- E.g. replace $\land$ with $\times$, $\lor$ with $+$, . . .

If $\Gamma \vdash e : \tau$ in simply-typed lambda calculus, and $\tau$ translates to $\phi$, then $\text{range}(\Gamma) \vdash \phi$ in intuitionistic logic

If $\Gamma \vdash \phi$ in intuitionistic logic, and $\phi$ translates to $\tau$, then there exists $e$ and $\Gamma'$ such that $\text{range}(\Gamma') = \Gamma$ 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*
  - $\lambda x.x : \forall \alpha. \alpha \rightarrow \alpha$
  - For any type $\alpha$ the identity function has type $\alpha \rightarrow \alpha$
- This is also known as *parametric polymorphism*
System F: annotated polymorphism

We extend the previous system:

\[ \tau ::= \text{int} \mid \tau \rightarrow \tau \mid \alpha \mid \forall \alpha.\tau \]

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

- We add polymorphic types, and we add explicit type abstraction (generalization over all \( \alpha \)) ...
- 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

\[ \Lambda \alpha . \lambda x : \alpha. x : \forall \alpha . \alpha \rightarrow \alpha \]
\[ \Lambda \alpha . \Lambda \beta . \lambda x : \alpha. \lambda y : \beta . x : \forall \alpha . \forall \beta . \alpha \rightarrow \beta \rightarrow \alpha \]
\[ \Lambda \alpha . \Lambda \beta . \lambda x : \alpha. \lambda y : \beta . y : \forall \alpha . \forall \beta . \alpha \rightarrow \beta \rightarrow \beta \]
When we use a parametric polymorphic type, we apply or instantiate it with a particular type.

In System F this is done by hand:

- \((\Lambda \alpha. \lambda x : \alpha. x)[\tau_1] : \tau_1 \rightarrow \tau_1\)
- \((\Lambda \alpha. \lambda x : \alpha. x)[\tau_2] : \tau_2 \rightarrow \tau_2\)

This is where the term *parametric* comes from:

- The type \(\forall \alpha. \alpha \rightarrow \alpha\) is a “function” in the domain of types.
- It is given a parameter at instantiation time.
Type rules

\[ \Gamma, \alpha \vdash e : \tau \]
\[ \Gamma \vdash \Lambda \alpha. e : \forall \alpha. \tau \]
\[ \Gamma \vdash e : \forall \alpha. \tau \]
\[ \Gamma \vdash e[\tau'] : \tau'[\tau'/\alpha] \]

- Notice that there are no constructs for manipulating values of polymorphic type
- This justifies instantiation with *any* type—what “for all” means
- We add \( \alpha \) to \( \Gamma \): we use this to ensure types are well-formed
Small-step semantics rules

\[
(\Lambda \alpha.e)[\tau] \rightarrow e[\tau/\alpha] \quad e \rightarrow e'
\]

\[
e[\tau] \rightarrow e'[\tau]
\]

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

\[
FV(\alpha) = \{\alpha\} \\
FV(c) = \emptyset \\
FV(\tau \rightarrow \tau') = FV(\tau) \cup FV(\tau') \\
FV(\forall \alpha. \tau) = FV(\tau) \setminus \{\alpha\}
\]
Substitution (again)

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

\[
\alpha[u/\alpha] = u \\
\beta[u/\alpha] = \beta \\
(\tau \rightarrow \tau')[u/\alpha] = \tau[u/\alpha] \rightarrow \tau'[u/\alpha] \\
(\forall \beta.\tau)[u/\alpha] = \forall \beta. (\tau[u/\alpha]) \\
\text{where } \beta \neq \alpha
\]

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

\[
(\lambda x : \tau.e)[u/\alpha] = (\lambda x : \tau[u/\alpha].e[u/\alpha] \\
(\Lambda \beta.e)[u/\alpha] = \Lambda \beta.e[u/\alpha] \\
(e_1 e_2)[u/\alpha] = e_1[u/\alpha] e_2[u/\alpha] \\
x[u/\alpha] = x \\
n[u/\alpha] = n \\
\text{where } \beta \neq \alpha \land \beta \notin FV(u)
\]
Type inference

Consider the simply typed lambda calculus with integers

\[ e ::= n \mid x \mid \lambda x.\tau \mid ee \]

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 \( \lambda \) terms, where do we find \( \tau \)?

- We use type variables in place of as-yet-unknown types
- \( \tau ::= \alpha \mid \text{int} \mid \tau \rightarrow \tau \)
- We generate equality constraints \( \tau = \tau' \) among types and type variables
- We solve the constraints to compute a typing
Type inference rules

\[ \Gamma \vdash n : \text{int} \]

\[ \Gamma, x : \alpha \vdash e : \tau' \quad \alpha \text{ fresh} \]
\[ \Gamma \vdash \lambda x. e : \alpha \rightarrow \tau' \]

\[ x \in \text{dom}(\Gamma) \]
\[ \Gamma \vdash x : \Gamma(x) \]

\[ \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 = \tau_2 \rightarrow \beta \quad \beta \text{ fresh} \]
\[ \Gamma \vdash e_1 e_2 : \beta \]
Example

\[
\begin{align*}
\Gamma, x : \alpha & \vdash x : \alpha \\
\hline
\Gamma \vdash (\lambda x.x) : \alpha \rightarrow \alpha \\
\Gamma \vdash 3 : \text{int} & \quad \alpha \rightarrow \alpha = \text{int} \rightarrow \beta \\
\hline
\Gamma \vdash (\lambda x.x) \ 3 : \beta 
\end{align*}
\]

- We collect all constraints appearing in the derivation into a set \( C \) to be solved
- Here \( C \) contains \( \alpha \rightarrow \alpha = \text{int} \rightarrow \beta \)
- Solution: \( \alpha = \beta = \text{int} \)
- 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:

- \( C \cup \{ \text{int} = \text{int} \} \Rightarrow C \)
- \( C \cup \{ \alpha = \tau \} \Rightarrow C[\tau/\alpha] \)
- \( C \cup \{ \tau = \alpha \} \Rightarrow C[\tau/\alpha] \)
- \( C \cup \{ \tau_1 \rightarrow \tau_2 = \tau'_1 \rightarrow \tau'_2 \} \Rightarrow C \cup \{ \tau_1 = \tau'_1 \} \cup \{ \tau_2 = \tau'_2 \} \)
- \( C \cup \{ \text{int} = \tau_1 \rightarrow \tau_2 \} \Rightarrow \text{unsatisfiable} \)
- \( C \cup \{ \tau_1 \rightarrow \tau_2 = \text{int} \} \Rightarrow \text{unsatisfiable} \)
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

\[ \alpha = \text{int} \rightarrow \beta \]

\[ \gamma = \text{int} \rightarrow \text{int} \]

\[ \alpha = \gamma \]
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 $\alpha$ stand for “some particular type”
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 $\text{F}_\omega$
- 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