Recapitulation from Previous Lectures

Recap: Inductively-defined Relations and Sets

Example: define relation $R \subseteq \mathbb{Z} \times \mathbb{Z}$ as the ***smallest*** set satisfying:

$$\frac{}{(0,0) \in R} \text{ (zero)} \qquad \frac{(x,y) \in R}{(x,y+1) \in R} \text{ (increase right)}$$

$$\frac{(x,y) \in R}{(x+1,y+1) \in R} \text{ (increase both)} \qquad \frac{(x,y) \in R}{(x-1,y-1) \in R} \text{ (decrease both)}$$

This is a way of expressing: $R = \{\ (x,y)\ \mid\ x \leq y\ \}$

Proof: establish two directions

- if there exists a derivation, then $x \leq y$ (by induction on derivation)
- if $x \leq y$, then there exists a derivation (provide algorithm that finds a derivation)

Recap: Operational Semantics

Dynamic semantics of the language given by

inductively-defined rules of ***operational semantics***

Rules for **if**:

$$\frac{b \rightsquigarrow b'}{(\textbf{if}\ (b)\ t_1\ \textbf{else}\ t_2) \rightsquigarrow (\textbf{if}\ (b')\ t_1\ \textbf{else}\ t_2)}$$

$$\frac{}{(\textbf{if}\ (\mathit{true})\ t_1\ \textbf{else}\ t_2) \rightsquigarrow t_1}$$

$$\frac{}{(\textbf{if}\ (\mathit{false})\ t_1\ \textbf{else}\ t_2) \rightsquigarrow t_2}$$

Example: $\mathit{fact}(2-1) \rightsquigarrow \mathit{fact}(1) \rightsquigarrow (\textbf{if}\ (1 \leq 1)\ 1\ \textbf{else}\ 1 * \mathit{fact}(1-1)) \rightsquigarrow$
$(\textbf{if}\ (\mathit{true})\ 1\ \textbf{else}\ 1 * \mathit{fact}(1-1)) \rightsquigarrow 1$

Conclusion: $\mathit{fact}(2-1) \overset{*}{\rightsquigarrow} 1$

Recap: Typing Rules

Static semantics of the language given by inductively-defined ***typing rules***

General form $\Gamma \vdash t : T$ Example rules:

$$\frac{\Gamma \vdash b : \mathit{Bool}, \quad \Gamma \vdash t_1 : \tau, \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash (\textbf{if } (b) \ t_1 \textbf{ else } t_2) : \tau} \qquad \frac{n \text{ is an integer literal}}{n : \mathsf{Int}}\ \text{Lit}$$

$$\frac{(x, \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \cdot (x : T_1) \vdash t_2 : T_2}{\Gamma \vdash ((x : T_1) \Rightarrow t_2) : (T_1) \Rightarrow T_2}$$

$$\frac{\Gamma \vdash t_f : \tau_1 \times \cdots \times \tau_n \to \tau_0, \quad \Gamma \vdash t_1 : \tau_1, \ \ldots, \ \Gamma \vdash t_n : \tau_n}{\Gamma \vdash t_f(t_1, \ldots, t_n) : \tau_0}$$

Recap: Typing and Evaluation Derivations

We consider ***derivation trees*** of inductive definition, such as:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \cdot (x, \mathsf{Int}) \vdash x : \mathsf{Int}}\,\mathsf{Var} \quad \dfrac{}{\Gamma \cdot (x, \mathsf{Int}) \vdash 1 : \mathsf{Int}}\,\mathsf{Lit}}{\Gamma \cdot (x, \mathsf{Int}) \vdash x + 1 : \mathsf{Int}}\,\mathsf{Add}}{\Gamma \vdash ((x : \mathit{Int}) \Rightarrow x + 1) : (\mathsf{Int}) \Rightarrow \mathsf{Int}}\,\mathsf{Lam} \qquad \Gamma \vdash 42 : \mathsf{Int}}{\Gamma \vdash ((x : \mathit{Int}) \Rightarrow x + 1)(42) : \mathsf{Int}}\,\mathsf{Call}$$

Derivation trees (or just ***derivations***) of typing and evaluation judgments are manipulated in proofs of the type system's properties

Induction principle: Like for any other tree, consider all node and leaf forms, and assume subtrees (sub-derivations) satisfy the property

Recap: Formal Soundness of a Type System

Soundness theorem: *if program type checks, its evaluation "does not get stuck".*

Proof uses the following two lemmas (common approach):

- **Progress:** *if a program type checks, it is not stuck.*

  If $\Gamma \vdash t : \tau$, then *either*

  t is a constant (execution is done)

  or there exists a t' such that $t \rightsquigarrow t'$

- **Preservation:** *if a program type checks and makes one " $\rightsquigarrow$ " step,*

  then the result still type checks.

  In our simple system: it type checks *and has the same type*:

  if $\Gamma \vdash t : \tau$ and $t \rightsquigarrow t'$, then $\Gamma \vdash t' : \tau$.

Type Inference

Type Inference – Motivation

Problem: writing type annotations is ***boring***; gets in the way of productivity/clarity

Type Inference – Motivation

Problem: writing type annotations is ***boring***; gets in the way of productivity/clarity

Example Java:

```
final Map<Integer, List<String>> xs =
  new HashMap<>();
for (Pair<Int, List<String>> i_ys : input) {
    List<String> ls = xs.get(i_ys.get1());
    if (ls == null)
      xs.put(i_ys.get1(), i_ys.get2());
    else ls.addAll(i_ys.get2());
}
```

Type Inference – Motivation

Problem: writing type annotations is ***boring***; gets in the way of productivity/clarity

Example Java:

```
final Map<Integer, List<String>> xs =
  new HashMap<>();
for (Pair<Int, List<String>> i_ys : input) {
    List<String> ls = xs.get(i_ys.get1());
    if (ls == null)
      xs.put(i_ys.get1(), i_ys.get2());
    else ls.addAll(i_ys.get2());
}
```

Corresponding Scala:

```
val xs =
  HashMap.empty[Int, Buffer[String]]
for i_ys <- input do
    val ls = xs.get(i_ys._1)
    if ls.isEmpty
      then xs.put(i_ys._1, i_ys._2)
      else ls.get ++= i_ys._2
```

Type Inference – Motivation

Problem: writing type annotations is ***boring***; gets in the way of productivity/clarity

Example Java:

```
final Map<Integer, List<String>> xs =
  new HashMap<>();
for (Pair<Int, List<String>> i_ys : input) {
    List<String> ls = xs.get(i_ys.get1());
    if (ls == null)
      xs.put(i_ys.get1(), i_ys.get2());
    else ls.addAll(i_ys.get2());
}
```

Corresponding Scala:

```
val xs =
  HashMap.empty[Int, Buffer[String]]
for i_ys <- input do
    val ls = xs.get(i_ys._1)
    if ls.isEmpty
      then xs.put(i_ys._1, i_ys._2)
      else ls.get ++= i_ys._2
```

Which one is better? What is different?

Type Inference – Motivation

Problem: writing type annotations is ***boring***; gets in the way of productivity/clarity

Example Java:

```
final Map<Integer, List<String>> xs =
  new HashMap<>();
for (Pair<Int, List<String>> i_ys : input) {
    List<String> ls = xs.get(i_ys.get1());
    if (ls == null)
      xs.put(i_ys.get1(), i_ys.get2());
    else ls.addAll(i_ys.get2());
}
```

Corresponding Scala:

```
val xs =
  HashMap.empty[Int, Buffer[String]]
for i_ys <- input do
    val ls = xs.get(i_ys._1)
    if ls.isEmpty
      then xs.put(i_ys._1, i_ys._2)
      else ls.get ++= i_ys._2
```

Which one is better? What is different?

Scala *looks* like a dynamic/scripting language, though *it is* statically-typed

⇒ Mainly thanks to ***type inference***.

Different Styles of Type Inference

Large ***design space*** of type inference styles/approaches

Different Styles of Type Inference

Large ***design space*** of type inference styles/approaches

Oversimplified:

style	**examples**	**pros**	**cons**
None	Pascal, C	easy implementation	painful
Bottom-up	C#, Java, modern C++	not too hard to impl.	still painful
Global	SML, OCaml, Haskell	no annotations needed	limited types
Hybrid	Scala, TypeScript	rich types, few annot.	corner cases

Different Styles of Type Inference

Large ***design space*** of type inference styles/approaches

Oversimplified:

style	examples	pros	cons
None	Pascal, C	easy implementation	painful
Bottom-up	C#, Java, modern C++	not too hard to impl.	still painful
Global	SML, OCaml, Haskell	no annotations needed	limited types
Hybrid	Scala, TypeScript	rich types, few annot.	corner cases

More powerful type inference often requires *less powerful* type system

to keep the algorithm decidable and tractable

e.g., no subtyping in ML languages

"Global" Type Inference

Languages such as Haskell, ML, OCaml support ***global*** inference of types in most cases

Allows writing programs without type annotations. In Amy syntax:

```
def message(s, verbose) = {
  if (verbose > 1) { print(s) } else { print(".") }
}
```

Infer types of parameters and result, then ***check*** the program type checks

If not possible to find types, type checker still complains

Pro: as concise code as an untyped language

Pro: type inference still catches meaningless programs

Today: explain how to do such type inference, for simple types

Intuition

```
def message(s, verbose) = {
  if (verbose > 1) { print(s) } else { print(".") }
}
```

Intuition

```
def message(s, verbose) = {
  if (verbose > 1) { print(s) } else { print(".") }
}
```

$$\frac{(>) : \mathit{Int} \times \mathit{Int} \to \mathit{Bool},\ \mathit{verbose} : \alpha_{\mathit{verbose}},\ 1 : \mathit{Int}}{(\mathit{verbose} > 1) : \mathit{Bool}}$$

Intuition

```
def message(s, verbose) = {
  if (verbose > 1) { print(s) } else { print(".") }
}
```

$$\frac{(>) : \mathit{Int} \times \mathit{Int} \to \mathit{Bool},\ \mathit{verbose} : \alpha_{\mathit{verbose}},\ 1 : \mathit{Int}}{(\mathit{verbose} > 1) : \mathit{Bool}}$$

so $\alpha_{\mathit{verbose}} = \mathit{Int}$, for application of $>$ to make sense.

Intuition

```
def message(s, verbose) = {
  if (verbose > 1) { print(s) } else { print(".") }
}
```

$$\frac{(>) : \mathit{Int} \times \mathit{Int} \to \mathit{Bool},\ \mathit{verbose} : \alpha_{\mathit{verbose}},\ 1 : \mathit{Int}}{(\mathit{verbose} > 1) : \mathit{Bool}}$$

so $\alpha_{\mathit{verbose}} = \mathit{Int}$, for application of $>$ to make sense.

$$\frac{\mathit{print} : \mathit{String} \to \mathit{Unit},\ s : \alpha_s}{\mathit{print}(s) : \mathit{Unit}}$$

Intuition

```
def message(s, verbose) = {
  if (verbose > 1) { print(s) } else { print(".") }
}
```

$$\frac{(>) : \mathit{Int} \times \mathit{Int} \to \mathit{Bool},\ \mathit{verbose} : \alpha_{\mathit{verbose}},\ 1 : \mathit{Int}}{(\mathit{verbose} > 1) : \mathit{Bool}}$$

so $\alpha_{\mathit{verbose}} = \mathit{Int}$, for application of $>$ to make sense.

$$\frac{\mathit{print} : \mathit{String} \to \mathit{Unit},\ s : \alpha_s}{\mathit{print}(s) : \mathit{Unit}}$$

so $\alpha_s = \mathit{String}$, for application of *print* to make sense.

Key Ideas

```
def message(s, verbose) = {
  if (verbose > 1) { print(s) } else { print(".") }
}
```

Both branches return Unit, and so should `message`.

Key Ideas

```
def message(s, verbose) = {
  if (verbose > 1) { print(s) } else { print(".") }
}
```

Both branches return Unit, and so should `message`.

Strategy for type inference:

1. Use **type variables** (e.g. $\alpha_{verbose}$, α_s) to denote unknown types

Key Ideas

```
def message(s, verbose) = {
  if (verbose > 1) { print(s) } else { print(".") }
}
```

Both branches return Unit, and so should `message`.

Strategy for type inference:

1. Use **type variables** (e.g. $\alpha_{verbose}$, α_s) to denote unknown types
2. Use type checking rules to derive **constraints** among type variables (e.g., arguments have expected types)

Key Ideas

```
def message(s, verbose) = {
  if (verbose > 1) { print(s) } else { print(".") }
}
```

Both branches return Unit, and so should `message`.

Strategy for type inference:

1. Use **type variables** (e.g. $\alpha_{verbose}$, α_s) to denote unknown types
2. Use type checking rules to derive **constraints** among type variables (e.g., arguments have expected types)
3. Use a **unification algorithm** to solve the constraints

Small language with tuples and functions

Types are:

1. primitive types: Int, Bool, String, Unit

Small language with tuples and functions

Types are:

1. primitive types: Int, Bool, String, Unit
2. type constructors:
   - Pair[A,B] or (A,B) denotes set of pairs
   - Function[A,B] or $A \Rightarrow B$ denotes functions from A to B

Small language with tuples and functions

Types are:

1. primitive types: Int, Bool, String, Unit
2. type constructors:
   - Pair[A,B] or (A,B) denotes set of pairs
   - Function[A,B] or $A \Rightarrow B$ denotes functions from A to B

Abstract syntax of types:

$$T ::= \mathit{Int} \mid \mathit{Bool} \mid \mathit{String} \mid \mathit{Unit} \mid (T_1, T_2) \mid (T_1 \Rightarrow T_2)$$

Small language with tuples and functions

Types are:

1. primitive types: Int, Bool, String, Unit
2. type constructors:
   - Pair[A,B] or (A,B) denotes set of pairs
   - Function[A,B] or $A \Rightarrow B$ denotes functions from A to B

Abstract syntax of types:

$$T ::= \mathit{Int} \mid \mathit{Bool} \mid \mathit{String} \mid \mathit{Unit} \mid (T_1, T_2) \mid (T_1 \Rightarrow T_2)$$

Terms include pairs and anonymous functions (x denotes variables, c literals):

$$t ::= x \mid c \mid f(t_1, \ldots, t_n) \mid \textbf{if } (t)\ t_1 \textbf{ else } t_2 \mid (t_1, t_2) \mid (x \Rightarrow t)$$

Small language with tuples and functions

Types are:

1. primitive types: Int, Bool, String, Unit
2. type constructors:
   - Pair[A,B] or (A,B) denotes set of pairs
   - Function[A,B] or $A \Rightarrow B$ denotes functions from A to B

Abstract syntax of types:

$$T ::= \mathit{Int} \mid \mathit{Bool} \mid \mathit{String} \mid \mathit{Unit} \mid (T_1, T_2) \mid (T_1 \Rightarrow T_2)$$

Terms include pairs and anonymous functions (x denotes variables, c literals):

$$t ::= x \mid c \mid f(t_1, \ldots, t_n) \mid \textbf{if } (t)\ t_1 \textbf{ else } t_2 \mid (t_1, t_2) \mid (x \Rightarrow t)$$

if $t = (x, y)$ then $t.\_1 = x$ and $t.\_2 = y$

For values and types, (x, y, z) is shorthand for $(x, (y, z))$

Type Rules

Rule for conditionals:

$$\frac{\Gamma \vdash b : \mathit{Bool} \quad \Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T}{\Gamma \vdash (\textbf{if } (b)\ t_1 \textbf{ else } t_2) : T}$$

Rules for variables:

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

Rules for constants:

$$\frac{}{\text{"..."} : \mathit{String}} \qquad \frac{}{\mathit{true} : \mathit{Bool}} \qquad \frac{}{\mathit{false} : \mathit{Bool}} \qquad \ldots$$

Rules for Pairs

$$\frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (t_1, t_2) : (T_1, T_2)}$$

If the first component t_1 has type T_1 and the second component t_2 has type T_2 then the pair (t_1, t_2) has the type (T_1, T_2).

$$\frac{\Gamma \vdash t : (T_1, T_2)}{\Gamma \vdash t.\_1 : T_1}$$

$$\frac{\Gamma \vdash t : (T_1, T_2)}{\Gamma \vdash t.\_2 : T_2}$$

Functions of One argument

$$\frac{\Gamma \vdash f : T \Rightarrow T_0 \quad \Gamma \vdash t : T}{\Gamma \vdash f(t) : T_0}$$

Functions of One argument

$$\frac{\Gamma \vdash f : T \Rightarrow T_0 \quad \Gamma \vdash t : T}{\Gamma \vdash f(t) : T_0}$$

Why only one argument?

Functions of One argument

$$\frac{\Gamma \vdash f : T \Rightarrow T_0 \quad \Gamma \vdash t : T}{\Gamma \vdash f(t) : T_0}$$

Why only one argument?

Note that T can be a tuple $(T_1, \ldots, T_n)$, so we can derive:

$$\frac{\dfrac{\Gamma \vdash t_1 : T_1 \ \ldots \ \Gamma \vdash t_n : T_n \quad \Gamma \vdash f : (T_1, \ldots, T_n) \Rightarrow T_0}{\Gamma \vdash (t_1, \ldots, t_n) : (T_1, \ldots, T_n) \quad \Gamma \vdash f : (T_1, \ldots, T_n) \Rightarrow T_0}}{\Gamma \vdash f((t_1, \ldots, t_n)) : T_0}$$

Rules for Anonymous Function

$$\frac{\Gamma \cdot (x,\, T_1) \vdash t : T_2}{\Gamma \vdash (x \Rightarrow t) : (T_1 \Rightarrow T_2)}$$

What does this rule say?

Rules for Anonymous Function

$$\frac{\Gamma \cdot (x,\, T_1) \vdash t : T_2}{\Gamma \vdash (x \Rightarrow t) : (T_1 \Rightarrow T_2)}$$

What does this rule say?

Anonymous function $x \Rightarrow t$ that maps x to t, has a function type

Rules for Anonymous Function

$$\frac{\Gamma \cdot (x,\, T_1) \vdash t : T_2}{\Gamma \vdash (x \Rightarrow t) : (T_1 \Rightarrow T_2)}$$

What does this rule say?

Anonymous function $x \Rightarrow t$ that maps x to t, has a function type $T_1 \Rightarrow T_2$

where T_1 is the type of x and T_2 is the type of t.

Rules for Anonymous Function

$$\frac{\Gamma \cdot (x, T_1) \vdash t : T_2}{\Gamma \vdash (x \Rightarrow t) : (T_1 \Rightarrow T_2)}$$

What does this rule say?

Anonymous function $x \Rightarrow t$ that maps x to t, has a function type $T_1 \Rightarrow T_2$

where T_1 is the type of x and T_2 is the type of t.

Within t, there may be uses of x, which has some type T_1.

This is why Γ is extended with binding of x to T_1 when type checking t.

Example

Program without type annotations:

```
def translatorFactory(dx, dy) = {
  p ⇒ (p._1 + dx, p._2 + dy)  // returns anonymous function
}
def upTranslator = translatorFactory(0, 100)
def test = upTranslator((3, 5)) // computes (3, 105)
```

Type inference can find types that correspond to this annotated program:

Example

Program without type annotations:

```
def translatorFactory(dx, dy) = {
  p ⇒ (p._1 + dx, p._2 + dy)  // returns anonymous function
}
def upTranslator = translatorFactory(0, 100)
def test = upTranslator((3, 5)) // computes (3, 105)
```

Type inference can find types that correspond to this annotated program:

```
def translatorFactory(dx: Int, dy: Int): (Int,Int) ⇒ (Int,Int) = {
  p ⇒ (p._1 + dx, p._2 + dy)  }
def upTranslator : (Int,Int) ⇒ (Int,Int) = translatorFactory(0, 100)
def test: (Int,Int) = upTranslator((3, 5))
```

Are our inferred types correct?

```
def translatorFactory(dx: Int, dy: Int): (Int,Int) ⇒ (Int,Int) = {
  p ⇒ (p._1 + dx, p._2 + dy)  }
def upTranslator : (Int,Int) ⇒ (Int,Int) = translatorFactory(0, 100)
def test: (Int,Int) = upTranslator((3, 5))
```

$$\Gamma \vdash p \Rightarrow (p.\_1 + dx, p.\_2 + dy) \quad : \quad (Int, Int) \Rightarrow (Int, Int)$$

From Type Checking to Type Inference

```
def translatorFactory(dx: Int, dy: Int): (Int,Int) ⇒ (Int,Int) = {
  p ⇒ (p._1 + dx, p._2 + dy)  }
def upTranslator : (Int,Int) ⇒ (Int,Int) = translatorFactory(0, 100)
def test: (Int,Int) = upTranslator((3, 5))
```

Example steps in type checking the body. Let $\Gamma' = \Gamma \cdot (p, (Int, Int))$

$$\dfrac{\dfrac{\dfrac{\Gamma' \vdash p.\_1 : Int \quad \Gamma' \vdash dx : Int}{\Gamma' \vdash (p.\_1 + dx) : Int \quad \ldots}}{\Gamma' \vdash (p.\_1 + dx, p.\_2 + dy) : (Int, Int)}}{\Gamma \vdash p \Rightarrow (p.\_1 + dx, p.\_2 + dy) \;\; : \;\; (Int, Int) \Rightarrow (Int, Int)}$$

How does type inference discover $dx : Int$?

⇒ construct derivation tree keeping type of dx symbolic
until some derivation step tells us what it must be.

Here, $+$ expects two integers in $p.\_1 + dx$

Generating Constraints During Type Inference

```
def translatorFactory(dx, dy) = { p ⇒ (p._1 + dx, p._2 + dy) }
```

Let $\Gamma_1 = \Gamma \cdot (p, \alpha_p)$ where type variable α_p to be determined later

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash p : \alpha_p \quad \alpha_p = (\alpha_3, \alpha_4)}{\Gamma_1 \vdash p.\_1 : \alpha_3 \quad \Gamma_1 \vdash dx : \alpha_{dx} \quad \Gamma_1 \vdash (+) : (Int, Int) \to Int}}{\Gamma_1 \vdash p.\_1 + dx \ : \ Int \quad \alpha_3 = Int,\ \alpha_{dx} = Int}}{\Gamma_1 \vdash (p.\_1 + dx,\ p.\_2 + dy) : (Int, Int)}}{\Gamma \vdash (p \Rightarrow (p.\_1 + dx, p.\_2 + dy)) : \ \alpha_p \Rightarrow (Int, Int)}$$

Generating Constraints During Type Inference

```
def translatorFactory(dx, dy) = { p ⇒ (p._1 + dx, p._2 + dy) }
```

Let $\Gamma_1 = \Gamma \cdot (p, \alpha_p)$ where type variable α_p to be determined later

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash p : \alpha_p \quad \alpha_p = (\alpha_3, \alpha_4)}{\Gamma_1 \vdash p.\_1 : \alpha_3 \quad \Gamma_1 \vdash dx : \alpha_{dx} \quad \Gamma_1 \vdash (+) : (Int, Int) \to Int}}{\Gamma_1 \vdash p.\_1 + dx \ : \ Int \quad \alpha_3 = Int,\ \alpha_{dx} = Int}}{\Gamma_1 \vdash (p.\_1 + dx,\ p.\_2 + dy) : (Int, Int)}}{\Gamma \vdash (p \Rightarrow (p.\_1 + dx, p.\_2 + dy)) : \ \alpha_p \Rightarrow (Int, Int)}$$

Analogously, for the second component of the pair, we derive Int and $\alpha_4 = Int$ from other branches of the derivation tree.

From these constraints, it follows that the result type is

$$\alpha_p \Rightarrow (Int, Int) \quad = \quad (\alpha_3, \alpha_4) \Rightarrow (Int, Int) \quad = \quad (Int, Int) \Rightarrow (Int, Int)$$

Constraints

Generate fresh type variable when type is unknown. Collect these constraints:

AST node	inferred sub-expr types	inferred type	inferred constraints
$f(t)$	$(f : T_f)(t : T)$	α	$T_f = (T \Rightarrow \alpha)$ α fresh
$x \Rightarrow t$	$((x : \alpha_x) \Rightarrow (t : T))$	$(\alpha_x \Rightarrow T)$	$\Gamma \cdot (x, \alpha_x) \vdash t$ α_x fresh
(t_1, t_2)	$(t_1 : T_1, t_2 : T_2)$	(T_1, T_2)	
$t.\_1$	$(t : \tau).\_1$	α_1	$\tau = (\alpha_1, \alpha_2)$ α_1, α_2 fresh
$t.\_2$	$(t : \tau).\_2$	α_2	$\tau = (\alpha_1, \alpha_2)$ α_1, α_2 fresh
x	x	α_x	where $\Gamma(x) = \alpha_x$
true	*true*	*Bool*	
false	*false*	*Bool*	
n	n	*Int*	
"..."	"..."	*String*	
if $(b : T_b)$ $t_1 : T_1$ **else** $t_2 : T_2$		T_1	$T_b = Bool,\ T_1 = T_2$

Summary of type inference

1. Introduce type variable whenever type not locally known
2. Use typing rules to derive constraints between inferred subexpression types
3. Solve resulting set of type equations

Solving equations on simple types: unification (as in Prolog)

Types in equations have the following syntax:

$$T \quad ::= \quad \alpha \mid \mathit{Int} \mid \mathit{Bool} \mid \mathit{String} \mid \mathit{Unit} \mid (T_1, T_2) \mid (T_1 \Rightarrow T_2)$$

We assume that

- primitive, pair, and function types are all distinct/disjoint
- two pair types are equal iff their corresponding component types are equal
- two function types are equal iff their argument and result types are equal

Idea: eliminate variables, decompose pair and function equalities.

Algorithm works for any *term algebra* (algebra of syntactic terms)

- Pair[A,B] and Function[A,B] are two distinct binary term constructors
- Int, Bool, String are distinct nullary constructors

Analogy: Solving Equations over Non-negative Integers

Inspiration: Gaussian elimination to solve the system of equations:

$$\begin{aligned} x+y+z &= 5 \\ x+2y+z &= 6 \\ 2x+y+2z &= 5 \end{aligned}$$

For example, we can express x and substitute:

$$\begin{aligned} x &= 5-y-z \\ (5-y-z)+2y+z &= 6 \\ 2(5-y-z)+y+z &= 5 \end{aligned} \quad \text{i.e.,} \quad \begin{aligned} x &= 5-y-z \\ y &= 1 \\ y+z &= 5 \end{aligned}$$

Here, $y=1,\ z=4,\ x=0$ is unique solution

Some systems have infinitely many solutions; some have none

Over non-negative integers, $x = x+y+1$ has no solutions

Unification Algorithm

Apply following rules as long as current set of equations changes:

Orient: Replace $T = \alpha$ with $\alpha = T$ when T not a type variable

Delete useless: Remove $T = T$ (both sides syntactically equal)

Eliminate: Given $\alpha = T$ where α does not occur in T,
replace α with T in all remaining equations

Occurs check: Given $\alpha = T$ where α occurs in T, report error (no solutions)

Decompose pairs: Replace $(T_1, T_2) = (T_1', T_2')$ with $T_1 = T_1'$ and $T_2 = T_2'$

Decompose functions: Replace $(T_1 \Rightarrow T_2) = (T_1' \Rightarrow T_2')$ with $T_1 = T_1'$ and $T_2 = T_2'$.

Decomposition clash (remaining cases): Given $T_1 = T_2$
where T_1 and T_2 have different constructors, report clash (no solution)

Examples: $(T_1, T_2) = (T_1' \Rightarrow T_2')$; $Int = (T_1, T_2)$; $Int = Bool$; $(T_1 \Rightarrow T_2) = String$

Properties of Unification

Algorithm always terminates.

Running time almost linear given the right data structures (using lazy substitution)

If error reported, equations have no solution

i.e., no type annotations can make program type check

Otherwise, the equations have one or more solutions. Note that a variable that appears on left of equation does not appear on the right (else the eliminate rule would apply). Call a variable that only appears on the right a *parameter*.

If there are no parameters, there is exactly one solution. Otherwise, for each assignment of types to parameters we obtain a solution. Moreover, all solutions are obtained by instantiating parameters.

Therefore, the result of the unification algorithm describes all possible ways to assign simple types to the program.

Use the algorithm to infer the type of rightNest

```
def rightNest(t) = { (t._1._1, (t._1._2, t._2)) }

def test1 = rightNest(((1, 2), 3))  // computes (1,(2,3))
```

Use the algorithm to infer the type of rightNest

```
def rightNest(t) = { (t._1._1, (t._1._2, t._2)) }

def test1 = rightNest(((1, 2), 3))  // computes (1,(2,3))
```

Inferred types for each sub-expression, where $T_1 = (\alpha_3, \alpha_4)$:

$$\Big(\ ((t:\alpha_t).\_1:\alpha_1).\_1:\alpha_2,\ \Big(\ ((t:\alpha_t).\_1:\alpha_1).\_2:\alpha_3,\ \ (t:\alpha_t).\_2:\alpha_4\ \Big)\ :\ T_1\ \Big)\ :\ (\alpha_2, T_1)$$

Use the algorithm to infer the type of rightNest

```
def rightNest(t) = { (t._1._1, (t._1._2, t._2)) }

def test1 = rightNest(((1, 2), 3))  // computes (1,(2,3))
```

Inferred types for each sub-expression, where $T_1 = (\alpha_3, \alpha_4)$:

$$\Big(\ ((t:\alpha_t).\_1:\alpha_1).\_1:\alpha_2,\ \Big(\ ((t:\alpha_t).\_1:\alpha_1).\_2:\alpha_3,\ \ (t:\alpha_t).\_2:\alpha_4\ \Big)\ :\ T_1\ \Big)\ :\ (\alpha_2, T_1)$$

$$\left|\begin{array}{l} \alpha_t = (\alpha_1, \alpha_{10}) \\ \alpha_1 = (\alpha_2, \alpha_{20}) \\ \alpha_t = (\alpha_1, \alpha_{30}) \\ \alpha_1 = (\alpha_{40}, \alpha_3) \\ \alpha_t = (\alpha_{50}, \alpha_4) \end{array}\right|$$

Use the algorithm to infer the type of rightNest

```
def rightNest(t) = { (t._1._1, (t._1._2, t._2)) }

def test1 = rightNest(((1, 2), 3))  // computes (1,(2,3))
```

Inferred types for each sub-expression, where $T_1 = (\alpha_3, \alpha_4)$:

$$\big(\ ((t:\alpha_t).\_1:\alpha_1).\_1:\alpha_2,\ \big(\ ((t:\alpha_t).\_1:\alpha_1).\_2:\alpha_3,\ \ (t:\alpha_t).\_2:\alpha_4\ \big)\ :\ T_1\ \big)\ :\ (\alpha_2, T_1)$$

$$\left|\begin{array}{l} \alpha_t = (\alpha_1, \alpha_{10}) \\ \alpha_1 = (\alpha_2, \alpha_{20}) \\ \alpha_t = (\alpha_1, \alpha_{30}) \\ \alpha_1 = (\alpha_{40}, \alpha_3) \\ \alpha_t = (\alpha_{50}, \alpha_4) \end{array}\right| \Rightarrow \left|\begin{array}{l} \alpha_t = (\alpha_1, \alpha_{10}) \\ \alpha_1 = (\alpha_2, \alpha_{20}) \\ (\alpha_1, \alpha_{10}) = (\alpha_1, \alpha_{30}) \\ \alpha_1 = (\alpha_{40}, \alpha_3) \\ (\alpha_1, \alpha_{10}) = (\alpha_{50}, \alpha_4) \end{array}\right|$$

Use the algorithm to infer the type of rightNest

```
def rightNest(t) = { (t._1._1, (t._1._2, t._2)) }

def test1 = rightNest(((1, 2), 3))  // computes (1,(2,3))
```

Inferred types for each sub-expression, where $T_1 = (\alpha_3, \alpha_4)$:

$$\big(\ ((t:\alpha_t).\_1:\alpha_1).\_1:\alpha_2,\ \big(\ ((t:\alpha_t).\_1:\alpha_1).\_2:\alpha_3,\ \ (t:\alpha_t).\_2:\alpha_4\ \big)\ :\ T_1\ \big)\ :\ (\alpha_2, T_1)$$

$$\left|\begin{array}{l}\alpha_t = (\alpha_1, \alpha_{10})\\ \alpha_1 = (\alpha_2, \alpha_{20})\\ \alpha_t = (\alpha_1, \alpha_{30})\\ \alpha_1 = (\alpha_{40}, \alpha_3)\\ \alpha_t = (\alpha_{50}, \alpha_4)\end{array}\right| \Rightarrow \left|\begin{array}{l}\alpha_t = (\alpha_1, \alpha_{10})\\ \alpha_1 = (\alpha_2, \alpha_{20})\\ (\alpha_1, \alpha_{10}) = (\alpha_1, \alpha_{30})\\ \alpha_1 = (\alpha_{40}, \alpha_3)\\ (\alpha_1, \alpha_{10}) = (\alpha_{50}, \alpha_4)\end{array}\right| \Rightarrow \left|\begin{array}{l}\alpha_t = (\alpha_1, \alpha_{10})\\ \alpha_1 = (\alpha_2, \alpha_{20})\\ \alpha_1 = \alpha_1;\ \alpha_{10} = \alpha_{30}\\ \alpha_1 = (\alpha_{40}, \alpha_3)\\ (\alpha_1, \alpha_{10}) = (\alpha_{50}, \alpha_4)\end{array}\right| \Rightarrow$$

Applying Unification Rules Some More

$$\left|\begin{array}{l}\alpha_t = (\alpha_1, \alpha_{10}) \\ \alpha_1 = (\alpha_2, \alpha_{20}) \\ \alpha_{10} = \alpha_{30} \\ \alpha_1 = (\alpha_{40}, \alpha_3) \\ (\alpha_1, \alpha_{10}) = (\alpha_{50}, \alpha_4)\end{array}\right| \Rightarrow \left|\begin{array}{l}\alpha_t = (\alpha_1, \alpha_{10}) \\ \alpha_1 = (\alpha_2, \alpha_{20}) \\ \alpha_{10} = \alpha_{30} \\ \alpha_1 = (\alpha_{40}, \alpha_3) \\ \alpha_1 = \alpha_{50};\ \alpha_{10} = \alpha_4\end{array}\right| \Rightarrow \left|\begin{array}{l}\alpha_t = (\alpha_1, \alpha_4) \\ \alpha_1 = (\alpha_2, \alpha_{20}) \\ \alpha_4 = \alpha_{30} \\ \alpha_1 = (\alpha_{40}, \alpha_3) \\ \alpha_1 = \alpha_{50}, \alpha_{10} = \alpha_4\end{array}\right| \Rightarrow$$

$$\left|\begin{array}{l}\alpha_t = (\alpha_1, \alpha_4) \\ \alpha_1 = (\alpha_2, \alpha_{20}) \\ \alpha_{30} = \alpha_4 \\ \alpha_1 = (\alpha_{40}, \alpha_3) \\ \alpha_{50} = \alpha_1, \alpha_{10} = \alpha_4\end{array}\right| \Rightarrow \left|\begin{array}{l}\alpha_t = ((\alpha_2, \alpha_{20}), \alpha_4) \\ \alpha_1 = (\alpha_2, \alpha_{20}) \\ \alpha_{30} = \alpha_4 \\ (\alpha_2, \alpha_{20}) = (\alpha_{40}, \alpha_3) \\ \alpha_{50} = (\alpha_2, \alpha_{20});\ \alpha_{10} = \alpha_4\end{array}\right| \Rightarrow \left|\begin{array}{l}\alpha_t = ((\alpha_2, \alpha_{20}), \alpha_4) \\ \alpha_1 = (\alpha_2, \alpha_{20}) \\ \alpha_{30} = \alpha_4 \\ \alpha_2 = \alpha_{40};\ \alpha_{20} = \alpha_3 \\ \alpha_{50} = (\alpha_2, \alpha_{20}), \alpha_{10} = \alpha_4\end{array}\right| \Rightarrow$$

And More

$$\left|\begin{array}{l}\alpha_t = ((\alpha_2,\alpha_3),\alpha_4)\\ \alpha_1 = (\alpha_2,\alpha_3)\\ \alpha_{30} = \alpha_4\\ \alpha_2 = \alpha_{40};\ \alpha_{20} = \alpha_3\\ \alpha_{50} = (\alpha_2,\alpha_3);\ \alpha_{10} = \alpha_4\end{array}\right| \Rightarrow \left|\begin{array}{l}\alpha_t = ((\alpha_2,\alpha_3),\alpha_4)\\ \alpha_1 = (\alpha_2,\alpha_3)\\ \alpha_{30} = \alpha_4\\ \alpha_{40} = \alpha_2, \alpha_{20} = \alpha_3\\ \alpha_{50} = (\alpha_2,\alpha_3), \alpha_{10} = \alpha_4\end{array}\right| \Rightarrow \left|\begin{array}{l}\alpha_t = ((\alpha_2,\alpha_3),\alpha_4)\\ \alpha_1 = (\alpha_2,\alpha_3)\\ \alpha_{30} = \alpha_4\\ \alpha_{40} = \alpha_2, \alpha_{20} = \alpha_3\\ \alpha_{50} = (\alpha_2,\alpha_3), \alpha_{10} = \alpha_4\end{array}\right|$$

No more rules apply. Variables on

- right-hand sides: $\alpha_2, \alpha_3, \alpha_4$
- left-hand sides: all others

Argument type is $\alpha_t = ((\alpha_2,\alpha_3),\alpha_4)$; result type is $(\alpha_2,(\alpha_3,\alpha_4))$

So rightNest has type $((\alpha_2,\alpha_3),\alpha_4) \Rightarrow (\alpha_2,(\alpha_3,\alpha_4))$

Types for $\alpha_2, \alpha_3, \alpha_4$ can be picked arbitrarily — infinitely many solutions

Adding Constraints for Function Call

We have:

$$\text{rightNest} : ((\alpha_2, \alpha_3), \alpha_4) \Rightarrow (\alpha_2, (\alpha_3, \alpha_4))$$

Adding Constraints for Function Call

We have:

$$\text{rightNest} : ((\alpha_2, \alpha_3), \alpha_4) \Rightarrow (\alpha_2, (\alpha_3, \alpha_4))$$

Given call rightNest(((1, 2), 3)), we add constraints equivalent to

$$(\alpha_2, \alpha_3), \alpha_4) = ((\mathit{Int}, \mathit{Int}), \mathit{Int})$$

Adding Constraints for Function Call

We have:

$$\text{rightNest} : ((\alpha_2, \alpha_3), \alpha_4) \Rightarrow (\alpha_2, (\alpha_3, \alpha_4))$$

Given call rightNest(((1, 2), 3)), we add constraints equivalent to

$$(\alpha_2, \alpha_3), \alpha_4) = ((\mathit{Int}, \mathit{Int}), \mathit{Int})$$

Thus we conclude $\alpha_2 = \mathit{Int}$; $\alpha_3 = \mathit{Int}$; $\alpha_4 = \mathit{Int}$. Given that

$$\mathit{rightNest}(((1,2),3)) : (\alpha_2, (\alpha_3, \alpha_4))$$

we conclude

$$\mathit{rightNest}(((1,2),3)) : (\mathit{Int}, (\mathit{Int}, \mathit{Int}))$$

What happens in this case?

```
def rightNest(t) = { (t._1._1, (t._1._2, t._2)) }
def test1 = rightNest(((1, 2), 3))
def test2 = rightNest((false , true), false)
```

$$((\tau_2, \tau_3), \tau_4) = ((\mathit{Int}, \mathit{Int}), \mathit{Int}) \quad \text{because of test1}$$
$$((\tau_2, \tau_3), \tau_4) = ((\mathit{Bool}, \mathit{Bool}), \mathit{Bool}) \quad \text{because of test2}$$

which implies $\mathit{Int} = \mathit{Bool}$ and is contradictory.

Program ***fails to type check***:

type of argument t equated to both Int and $\mathit{Bool} \Rightarrow$ *inconsistent*

Pity, because we could *copy* rightNest as rightNest2

then call rightNest2((false, true), false),

and everything would work (same dynamic semantics)

More flexibility through generalization

```
def rightNest(t) = { (t._1._1, (t._1._2, t._2)) }
def test1 = rightNest(((1, 2), 3))
def test2 = rightNest((false , true), false)
```

After rightNest type inference, first ***generalize*** free type variables into a *type scheme*:

$$\forall a, b, c.\ ((a, b), c) \Rightarrow (a, (b, c))$$

Each time we use the function, replace quantified variables with fresh variables

Use in test1:

$$((a_1, b_1), c_1) \Rightarrow (a_1, (b_1, c_1)) \qquad \Longrightarrow \qquad a_1 = Int\,;\ b_1 = Int\,;\ c_1 = Int$$

Use in test2:

$$((a_2, b_2), c_2) \Rightarrow (a_2, (b_2, c_2)) \qquad \Longrightarrow \qquad a_2 = Bool\,;\ b_2 = Bool\,;\ c_2 = Bool$$

More flexibility through generalization

```
def rightNest(t) = { (t._1._1, (t._1._2, t._2)) }
def test1 = rightNest(((1, 2), 3))
def test2 = rightNest((false , true), false)
```

Now the program type checks and its types are inferred as follows:

```
def rightNest[A, B, C](t : ((A, B), C)) : (A, (B, C)) =
  { (t._1._1, (t._1._2, t._2)) }

def test1 : (Int, (Int, Int)) =
  rightNest[Int, Int, Int](((1, 2), 3))

def test2 : (Bool, (Bool, Bool))=
  rightNest[Bool, Bool, Bool]((false , true), false)
```

Global Type Inference: Let Polymorphism

ML languages (OCaml, Haskell) support ***let polymorphism***:

any nested binding of a function can be assigned a polymorphic type

Global Type Inference: Let Polymorphism

ML languages (OCaml, Haskell) support ***let polymorphism***:

any nested binding of a function can be assigned a polymorphic type

Example in ML syntax:

```
let foo x y =
  let helper z f = if x then f y else z
  in (helper 1 (fun a → 2), helper y (fun a → a))
```

Global Type Inference: Let Polymorphism

ML languages (OCaml, Haskell) support ***let polymorphism***:

any nested binding of a function can be assigned a polymorphic type

Example in ML syntax:

```
let foo x y =
  let helper z f = if x then f y else z
  in (helper 1 (fun a → 2), helper y (fun a → a))
```

Inferred type:

$$foo : \forall \alpha.\ Bool \rightarrow \alpha \rightarrow (Int,\ \alpha)$$

Notion of *Principal Type*

A type system has the ***principal type*** property if,

for every term t and type T' so that $\Gamma \vdash t : T$,

Notion of *Principal Type*

A type system has the ***principal type*** property if,

for every term t and type T' so that $\Gamma \vdash t : T$,

there is a T' such that $\Gamma \vdash t : T'$

and T' ***subsumes*** (i.e., is *as general as*) T

(T may be the same as T')

Notion of *Principal Type*

A type system has the ***principal type*** property if,

for every term t and type T' so that $\Gamma \vdash t : T$,

there is a T' such that $\Gamma \vdash t : T'$

and T' ***subsumes*** (i.e., is *as general as*) T

(T may be the same as T')

For example, $x \Rightarrow (x, x)$ can be assigned type $T = Int \Rightarrow (Int, Int)$,

but the *more general* type $T' = \forall \alpha.\ \alpha \Rightarrow (\alpha, \alpha)$ *subsumes* T

Notion of *Principal Type*

A type system has the ***principal type*** property if,

for every term t and type T' so that $\Gamma \vdash t : T$,

there is a T' such that $\Gamma \vdash t : T'$

and T' ***subsumes*** (i.e., is *as general as*) T

(T may be the same as T')

For example, $x \Rightarrow (x, x)$ can be assigned type $T = Int \Rightarrow (Int, Int)$,

but the *more general* type $T' = \forall \alpha.\ \alpha \Rightarrow (\alpha, \alpha)$ *subsumes* T

We say that $T' \leq T$ — i.e., T' is a *subtype* of T

Principal Type Inference

A ***principal type inference*** algorithm

is an algorithm that always infers the *principal type* for any given term

Principal Type Inference

A ***principal type inference*** algorithm

is an algorithm that always infers the *principal type* for any given term

In such languages, adding type annotations never make *more* programs type check

Principal Type Inference

A ***principal type inference*** algorithm

is an algorithm that always infers the *principal type* for any given term

In such languages, adding type annotations never make *more* programs type check

The basic "ML" language

forming the core of SML, OCaml, and Haskell,

has this property

Most languages (Scala, C#, TypeScript, etc.) are *far* from having the property

Global Type Inference: Subtyping

Principal global type inference is actually possible with subtyping,
though things become quite complicated

If interested, see: the Simple-sub algorithm
`https://github.com/LPTK/simple-sub`