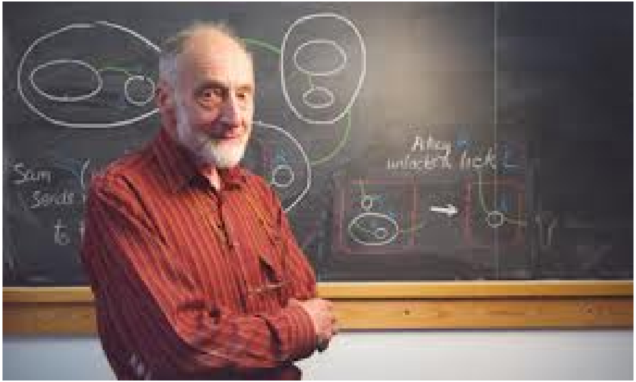
CIS 352

Types

Jim Royer

March 22, 2019



"Well-typed programs cannot go wrong."
— Robin Milner

References

- !!!! Slides for: *Language Semantics and Implementation Lectures 13 & 16*, by Richard Mayr and Colin Stirling, School of Informatics, University of Edinburgh, 2013. http://www.inf.ed.ac.uk/teaching/courses/lsi/13Lsi13-14-nup.pdf
- *Introduction to the semantics of programming languages: A simple functional language*, by Matthew Hennessy, Trinity College Dublin, University of Dublin, 2014–2015. https://www.scss.tcd.ie/Matthew.Hennessy/splexternal2015/LectureNotes/Notes14%20copy.pdf
- *Lecture Notes on Semantics of Programming Languages* by Andrew Pitts, Computer Laboratory, University of Cambridge, 1997-2002. http://www.inf.ed.ac.uk/teaching/courses/lsi/sempl.pdf.
- *Type Inference* by Michael Clarkson, Computer Science Dept., Cornell University, 2016. http://www.cs.cornell.edu/courses/cs3110/2016fa/l/17-inference/notes.html
- *Polymorphic Type Inference* by Tony Field, Department of Computing, Imperial College London, 2012. http://wp.doc.ic.ac.uk/ajf/type-inference

A type-system for LFP$^+$, 1

LFP$^+$ Types

$$
\begin{array}{rll}
\tau ::= & \mathbf{int} & \text{integers} \\
\mid & \mathbf{bool} & \text{booleans} \\
\mid & \mathbf{loc} & \text{locations} \\
\mid & \mathbf{cmd} & \text{commands} \\
\mid & \tau \to \tau' & \text{functions}
\end{array}
$$

Definition

(a) An LFP$^+$-expression e is ***closed*** $\iff fv(e) = \emptyset$.

(b) Suppose e is closed. Then $\vdash e : \tau$ is a ***type judgment*** that asserts e can be assigned type τ. *(At the moment the "$\vdash$" is just decoration.)*

Goal: We want $e : \tau$ to entail that e is

"an expression with all the properties demanded by τ."

A type-system for LFP$^+$, 2

Goal: We want $e : \tau$ to entail that e is
"an expression with all the properties demanded by τ."

Example

If $\vdash e_0 : \tau_1 \to \tau_2$ and $\vdash e_1 : \tau_1$, then $(e_0\ e_1)$'s value should be of type τ_2.

- The typable LFP$^+$-expressions should be well-behaved.
  (Details later.)
- Junk LFP$^+$-expressions *(e.g.,* $\mathbf{skip} + 3$*)* should be untypable.
- However, some funky, but non-junky LFP$^+$-expressions will also be untypable.
  (This provides employment opportunities for type-theorists.)

Example type judgments we want

$$\vdash 3 + !\ell : \mathbf{int}$$

$$\vdash \lambda x.x : \mathbf{int} \to \mathbf{int}$$

$$\vdash \lambda x.x : \mathbf{cmd} \to \mathbf{cmd}$$

$$\vdash \lambda x.(x := 0) : \mathbf{loc} \to \mathbf{cmd}$$

$$\vdash \lambda x.\lambda y.(x := y) : \mathbf{loc} \to \mathbf{int} \to \mathbf{cmd}$$

$$\vdash (\lambda x.\lambda y.(x := y))\ \ell : \mathbf{int} \to \mathbf{cmd}$$

$$\vdash \mathbf{rec}\, f.(\lambda x.\ \mathbf{if}\ x = 0\ \mathbf{then}\ 1\ \mathbf{else}\ x * f(x-1)) : \mathbf{int} \to \mathbf{int}$$

How to assign types?

- Rules! *(In this course, what else would you expect?)*
- We have to handle open expressions,
  i.e., expressions with free variables.
- Where do the free variables get their types?
- *Type context* = a dictionary of variables and their types.
- *E.g.*, $\Gamma = x\colon \textbf{int},\ f\colon \textbf{bool}{\to}\textbf{int}$
- General type judgments

$$\Gamma \vdash e : \tau \qquad \equiv \qquad \text{under context } \Gamma,\ \ e \text{ can be assigned type } \tau$$

- Rule format:

$$\textit{name:}\ \frac{\cdots\ \textit{premises}\ \cdots}{\Gamma \vdash e : \tau}\ (\text{side condition})$$

LFP$^+$ typing rules, 1

$$:\text{-}int: \quad \frac{}{\Gamma \vdash n\colon \mathbf{int}} \ (n \in \mathbb{Z})$$

$$:\text{-}bool: \quad \frac{}{\Gamma \vdash b\colon \mathbf{bool}} \ (b \in \mathbb{B})$$

$$:\text{-}loc: \quad \frac{}{\Gamma \vdash \ell\colon \mathbf{loc}} \ (\ell \in \mathbb{L})$$

$$:\text{-}iop: \quad \frac{\Gamma \vdash e_1\colon \mathbf{int} \quad \Gamma \vdash e_2\colon \mathbf{int}}{\Gamma \vdash e_1 \ iop \ e_2\colon \mathbf{int}}$$

$$:\text{-}cop: \quad \frac{\Gamma \vdash e_1\colon \mathbf{int} \quad \Gamma \vdash e_2\colon \mathbf{int}}{\Gamma \vdash e_1 \ cop \ e_2\colon \mathbf{bool}}$$

$$:\text{-}skip: \quad \frac{}{\Gamma \vdash \mathbf{skip} : \mathbf{cmd}}$$

$$:\text{-}if: \quad \frac{\Gamma \vdash e_o\colon \mathbf{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{if} \ e_0 \ \mathbf{then} \ e_1 \ \mathbf{else} \ e_2 : \tau}$$

$$:\text{-}get: \quad \frac{\Gamma \vdash e :: \mathbf{loc}}{\Gamma \vdash !e\colon \mathbf{int}}$$

$$:\text{-}set: \quad \frac{\Gamma \vdash e_1\colon \mathbf{loc} \quad \Gamma \vdash e_2\colon \mathbf{int}}{\Gamma \vdash e_1 := e_2\colon \mathbf{cmd}}$$

$$:\text{-}seq: \quad \frac{\Gamma \vdash e_1\colon \mathbf{cmd} \quad \Gamma \vdash e_2\colon \mathbf{cmd}}{\Gamma \vdash e_1;\ e_2\colon \mathbf{cmd}}$$

$$:\text{-}whi: \quad \frac{\Gamma \vdash e_1\colon \mathbf{bool} \quad \Gamma \vdash e_2\colon \mathbf{cmd}}{\Gamma \vdash \mathbf{while} \ e_1 \ \mathbf{do} \ e_2\colon \mathbf{cmd}}$$

$iop \in \{+, -, *\}$ $\qquad$ $cop \in \{=, \neq, <, >, \leq, \geq\}$

LFP$^+$ typing rules, 2

:-var: $$\frac{}{\Gamma,\ x:\tau \vdash x:\tau}$$

:-fn: $$\frac{\Gamma,\ x:\tau \vdash e':\tau'}{\Gamma \vdash \lambda x.e':\tau \to \tau'}$$

:-app: $$\frac{\Gamma \vdash e_1:\tau \to \tau' \qquad \Gamma \vdash e_2:\tau}{\Gamma \vdash (e_1\ e_2):\tau'}$$

:-rec: $$\frac{\Gamma,\ x:\tau \vdash e:\tau}{\Gamma \vdash \mathbf{rec}\ x.e:\tau}$$

Notes

- $\Gamma, x:\tau \quad \equiv \quad \Gamma \cup \{\, x:\tau \,\}$
- $\vdash e:\tau \quad \equiv \quad \emptyset \vdash e:\tau.$

Sample Derivations

$$\textit{+:}\ \dfrac{\textit{var:}\ \dfrac{}{x\colon \mathbf{int},\, y\colon \mathbf{int} \vdash x\colon \mathbf{int}} \qquad \textit{∗:}\ \dfrac{\textit{int:}\ \dfrac{}{x\colon \mathbf{int},\, y\colon \mathbf{int} \vdash 3\colon \mathbf{int}} \qquad \textit{var:}\ \dfrac{}{x\colon \mathbf{int},\, y\colon \mathbf{int} \vdash y\colon \mathbf{int}}}{x\colon \mathbf{int},\, y\colon \mathbf{int} \vdash (3 * y)\colon \mathbf{int}}}{x\colon \mathbf{int},\, y\colon \mathbf{int} \vdash (x + (3 * y))\colon \mathbf{int}}$$

$$\textit{fn:}\ \dfrac{\textit{fn:}\ \dfrac{\textit{+:}\ \dfrac{\textit{var:}\ \dfrac{}{x\colon \mathbf{int},\, y\colon \mathbf{int} \vdash x\colon \mathbf{int}} \qquad \textit{∗:}\ \dfrac{\textit{int:}\ \dfrac{}{x\colon \mathbf{int},\, y\colon \mathbf{int} \vdash 3\colon \mathbf{int}} \qquad \textit{var:}\ \dfrac{}{x\colon \mathbf{int},\, y\colon \mathbf{int} \vdash y\colon \mathbf{int}}}{x\colon \mathbf{int},\, y\colon \mathbf{int} \vdash (3 * y)\colon \mathbf{int}}}{x\colon \mathbf{int},\, y\colon \mathbf{int} \vdash (x + (3 * y))\colon \mathbf{int}}}{x\colon \mathbf{int} \vdash \lambda y.(x + (3 * y))\colon \mathbf{int} \to \mathbf{int}}}{\vdash \lambda x.\lambda y.(x + (3 * y))\colon \mathbf{int} \to \mathbf{int} \to \mathbf{int}}$$

Class Exercise

Derive each of:

$$\vdash 3+!\ell : \textbf{int}$$

$$\vdash \lambda x.x : \textbf{int} \rightarrow \textbf{int}$$

$$\vdash \lambda x.x : \textbf{cmd} \rightarrow \textbf{cmd}$$

$$\vdash \lambda x.(x := 0) : \textbf{loc} \rightarrow \textbf{cmd}$$

$$\vdash \lambda x.\lambda y.(x := y) : \textbf{loc} \rightarrow \textbf{int} \rightarrow \textbf{cmd}$$

$$\vdash (\lambda x.\lambda y.(x := y))\ \ell : \textbf{int} \rightarrow \textbf{cmd}$$

$$\vdash \textbf{rec}\, f.(\lambda x.\ \textbf{if}\ x = 0\ \textbf{then}\ 1\ \textbf{else}\ x * f(x-1)) : \textbf{int} \rightarrow \textbf{int}$$

Properties of the type system

Proposition

a *Declaration lemma*
Suppose: $\Gamma \vdash e : \tau$. **Then:** $fv(e) \subseteq \mathrm{dom}(\Gamma)$.

b *Weakening*
Suppose: $\Gamma \vdash e : \tau$ and $x \notin \mathrm{dom}(\Gamma)$. **Then:** $\Gamma, x : \sigma \vdash e : \tau$.

c *Strengthening*
Suppose: $\Gamma, x : \sigma \vdash e : \tau$ and $x \notin fv(e)$. **Then:** $\Gamma \vdash e : \tau$.

d *Substitution typing lemma*
Suppose: $\Gamma, x : \sigma \vdash e : \tau$ and $\Gamma \vdash e' : \sigma$. **Then:** $\Gamma \vdash e[e'/x] : \tau$.

e *Contraction*
Suppose: $\Gamma,\ x : \sigma,\ y : \sigma \vdash e : \tau$. **Then:** $\Gamma,\ x : \sigma \vdash e[x/y] : \tau$.

Subject Reduction

Subject Reduction $\equiv$ a type-τ expression evaluates to a type-τ value

Call-by-name subject reduction

Suppose: $\vdash e : \tau$ and $\langle e, s \rangle \Downarrow_{\mathsf{N}} \langle v, s' \rangle$. **Then:** $\vdash v : \tau$.

Call-by-value subject reduction

Suppose: $\vdash e : \tau$ and $\langle e, s \rangle \Downarrow_{\mathsf{V}} \langle v, s' \rangle$. **Then:** $\vdash v : \tau$.

Side-effect-free-ness for call-by-name LFP$^+$ expressions

Theorem

If $\vdash e : \tau$ *where* $\tau \neq \mathbf{cmd}$ *and* $\langle\, e, s \,\rangle \Downarrow_{\mathsf{N}} \langle\, v, s' \,\rangle$, *then* $s = s'$.

The proof is a tricky structural induction.

The call-by-value version of the theorem is ***false***. E.g.,

$$\langle\, (\lambda x.0)\ (\ell := 1), \{\, \ell \mapsto 0 \,\} \,\rangle \Downarrow_{\mathsf{V}} \langle\, 0, \{\, \ell \mapsto 1 \,\} \,\rangle.$$

Type checking and type inference

Type checking

Q: Given Γ, e and τ, how do we verify $\Gamma \vdash e : \tau$?

A: Use the typing rules.

Type inference

Q: Given Γ and e, how do we figure out a τ such that $\Gamma \vdash e : \tau$?

A_1: Use the draw-the-owl technique.

A_2: Gather and solve type constraints.

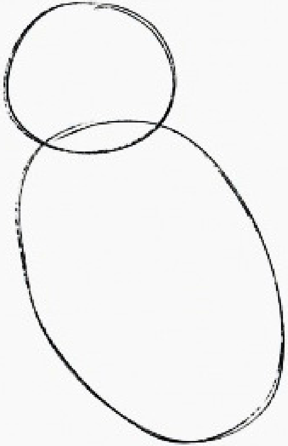


Fig 1. Draw two circles

Fig 2. Draw the rest of the damn Owl

Infering $\lambda x.(+\ 5\ x) : \textbf{int}{\to}\textbf{int}$

Example lifted from http://www.cs.cornell.edu/courses/cs3110/2016fa/l/17-inference/lec.pdf.

	Subexpression	Prelimary Type
a.	$\lambda x.(((+)\ 5)\ x)$	R
b.	x	S
c.	$(((+)\ 5)\ x))$	T
d.	$((+)\ 5)$	U
e.	$(+)$	$\textbf{int}{\to}\textbf{int}{\to}\textbf{int}$
f.	5	$\textbf{int}$
g.	x	V

Constraints

1. $S = V$ (b & g)
2. $R = S \to T$ (a, b, & c)
3. $U = V \to T$ (c, d, & g)
4. $\textbf{int}{\to}\textbf{int}{\to}\textbf{int} = \textbf{int}{\to}\ U$ (d, e, & f)

Infering $\lambda x.(+\ 5\ x) : \textbf{int}{\to}\textbf{int}$

	Subexpression	**Prelimary Type**
a.	$\lambda x.(((+)\ 5)\ x)$	R
b.	x	S
c.	$(((+)\ 5)\ x))$	T
d.	$((+)\ 5)$	U
e.	$(+)$	$\textbf{int}{\to}\textbf{int}{\to}\textbf{int}$
f.	5	$\textbf{int}$
g.	x	V

Constraints

1. $S = V$ (b & g)
2. $R = S \to T$ (a, b, & c)
3. $U = V \to T$ (c, d, & g)
4. $\textbf{int}{\to}\textbf{int}{\to}\textbf{int} = \textbf{int}{\to}\ U$ (d, e, & f)

Solving the Constraints: Step 1

Use $S = V$ to eliminate S by $[V/S]$.

$$R = {\color{red}V} \to T$$

$$U = V \to T$$

$$\textbf{int}{\to}\textbf{int}{\to}\textbf{int} = \textbf{int}{\to}\ U$$

Infering $\lambda x.(+\ 5\ x) : \mathbf{int}{\to}\mathbf{int}$

	Subexpression	Prelimary Type
a.	$\lambda x.(((+)\ 5)\ x)$	R
b.	x	S
c.	$(((+)\ 5)\ x))$	T
d.	$((+)\ 5)$	U
e.	$(+)$	$\mathbf{int}{\to}\mathbf{int}{\to}\mathbf{int}$
f.	5	$\mathbf{int}$
g.	x	V

Constraints

1. $S = V$ (b & g)
2. $R = S \to T$ (a, b, & c)
3. $U = V \to T$ (c, d, & g)
4. $\mathbf{int}{\to}\mathbf{int}{\to}\mathbf{int} = \mathbf{int}{\to}\ U$ (d, e, & f)

Solving the Constraints: Step 2

Use $U = V \to T$ to eliminate U by $[(V \to T)/U]$.

$$R = \ V \to T$$

$$\mathbf{int}{\to}\mathbf{int}{\to}\mathbf{int} = \mathbf{int}{\to}\ V \to T.$$

Infering $\lambda x.(+\ 5\ x) : \mathbf{int}{\to}\mathbf{int}$

	Subexpression	**Prelimary Type**
a.	$\lambda x.(((+)\ 5)\ x)$	R
b.	x	S
c.	$(((+)\ 5)\ x))$	T
d.	$((+)\ 5)$	U
e.	$(+)$	$\mathbf{int}{\to}\mathbf{int}{\to}\mathbf{int}$
f.	5	$\mathbf{int}$
g.	x	V

Constraints

1. $S = V$ (b & g)
2. $R = S \to T$ (a, b, & c)
3. $U = V \to T$ (c, d, & g)
4. $\mathbf{int}{\to}\mathbf{int}{\to}\mathbf{int} = \mathbf{int}{\to}\ U$ (d, e, & f)

Solving the Constraints: Step 3

Use $\mathbf{int}{\to}\mathbf{int}{\to}\mathbf{int}{=}\ \mathbf{int}{\to}\ V \to T$ to eliminate V and T.
I.e., $[\mathbf{int}\ / V]$ and $[\mathbf{int}\ / T]$.

$$R = \mathbf{int}{\to}\mathbf{int}$$

Gathering type constraints for a definition

- Initially do variable substitutions on the expression to make sure bound variable names are distinct. E.g.,

$$\textbf{let } x = 15 \textbf{ in } (\textbf{let } x = (+\ x\ 3) \textbf{ in } (*\ 2\ x))$$

$$\rightsquigarrow$$

$$\textbf{let } x = 15 \textbf{ in } (\textbf{let } y = (+\ x\ 3) \textbf{ in } (*\ 2\ y))$$

- Assign a preliminary type to every subexpression.
  - If the subexpression is a constant (e.g., 5) or a primitive function (e.g., +), use it's preassigned type.
  - Otherwise, use a fresh type variable.
- Use the "shape" of the expressions to generate constraints.
  - E.g., for $((\lambda x.e)\ 5)$ with $5 : \textbf{int}$, we obtain x's type = **int**.

Gathering type constraints for a definition

- Initially do variable substitutions on the expression to make sure bound variable names are distinct. E.g.,

$$\textbf{let } x = 15 \textbf{ in } (\textbf{let } x = (+\ x\ 3) \textbf{ in } (*\ 2\ x))$$
$$\rightsquigarrow$$
$$\textbf{let } x = 15 \textbf{ in } (\textbf{let } y = (+\ x\ 3) \textbf{ in } (*\ 2\ y))$$

- Assign a preliminary type to every subexpression.
  - If the subexpression is a constant (e.g., 5) or a primitive function (e.g., +), use it's preassigned type.
  - Otherwise, use a fresh type variable.
- Use the "shape" of the expressions to generate constraints.
  - E.g., for $((\lambda x.e)\ 5)$ with $5 : \textbf{int}$, we obtain x's type = **int**.

Infer:

- $\vdash\ apply : (a \to b) \to a \to b$ where $apply = \lambda f.\lambda x.(f\ x)$.
- $apply : (a \to b) \to a \to b,\ g\colon \textbf{int}{\to}\textbf{int} \vdash\ (apply\ g\ 5)\colon \textbf{int}$
- $apply : (a \to b) \to a \to b,\ not\colon \textbf{bool}{\to}\textbf{bool} \vdash\ (apply\ not\ tt)\colon \textbf{bool}$

These are worked out in Clarkson's notes.

Collection more formally

$$\begin{aligned} D &: \text{variables} \to \text{types} & \qquad D(x) &= \text{type var of arg } x \\ U &: \text{expressions} \to \text{types} & \qquad U(e) &= \text{type var assigned to exp } e \end{aligned}$$

- At a variable use x: $U(x) = D(x)$.
- At a function application $(e_1\ e_2)$: $U(e_1) = U(e_2) \to U(e_1\ e_2)$.
- At an anon. function $\lambda x.e$: $U(\lambda x.e) = D(x) \to U(e)$.
- Etc.
- Unioned with constraints collected at each subexpression.

Exercise: Try this on $\lambda x.\lambda y.x$.

Solving constraints

- A set of constraints is really a set of equations.
- The solution of these equations yields the type of the main expression.
- To to this we use a variation of Robinson's *unification algorithm*

  — Work done at SU in the mid-1960s.
- More on this next time.

Field's set-up

```
data Expr = Number Int        --  Expressions
          | Boolean Bool
          | Id String
          | Prim String
          | Cond Expr Expr Expr
          | App Expr Expr
          | Fun String Expr -- User defined functions
          deriving (Eq, Ord, Show)

data Type = TInt              --  Types
          | TBool
          | TFun Type Type
          | TErr
          | TVar String -- Type variables
          deriving (Eq, Ord, Show)
```

Field's set-up

```
data Expr = Number Int      --  Expressions
          | Boolean Bool
          | Id String
          | Prim String
          | Cond Expr Expr Expr
          | App Expr Expr
          | Fun String Expr -- User defined functions
          deriving (Eq, Ord, Show)

data Type = TInt            --  Types
          | TBool
          | TFun Type Type
          | TErr
          | TVar String -- Type variables
          deriving (Eq, Ord, Show)
```

```
type TypeTable = [(String, Type)]
type TEnv = TypeTable

-- A sample type-environment
te = [("x",TInt), ("y",TInt), ("f",TFun TInt TInt)]

-- Built-in function types...
primTypes :: TypeTable
primTypes
  = [ ("+", TFun TInt (TFun TInt TInt))
    , (">", TFun TInt (TFun TInt TBool)),
    , ("==", TFun TInt (TFun TInt TBool)),
    , ("not", TFun TBool TBool)
    ]
```

The easy case: Monomorphic types

Suppose our programs are missing

- user defined functions, i.e. no `Fun`-expressions, and
- type variables.

The we can use the follows rules (from Field):

1. Constants: (numbers and booleans): trivial.
2. Identifiers: look up the type in the *type environment*
   — a table of identifiers and their types
3. Primitives: they have preassigned types, `primTypes`
4. Conditionals: The test expression is Boolean and the two branches must identical types. If not, assign the type `TErr`. ($\star$)
5. Applications: The function should have type `TFun` t t' and the argument should have type t, in which case the result has type t'; o/w `TErr`.

Monomorphic type inference: Example from Field

1. Constants: (numbers and booleans): trivial.
2. Identifiers: look up the type in the *type environment*
3. Primitives: they have preassigned types, `primTypes`
4. Conditionals: The test expression is Boolean and the two branches must identical types. If not, assign the type `TErr`. ($\star$)
5. Applications: The function should have type `TFun` t t' and the argument should have type t, in which case the result has type t'; o/w `TErr`.

	Subexpression	Inferred type	Comment
1	`Number 1`	`TInt`	By rule 1
2	`Id "x"`	`TInt`	By rule 2
3	`Id "y"`	`TInt`	By rule 2
4	`Prim "+"`	`TFun TInt (TFun TInt TInt)`	By rule 3
5	`App (Prim "+") (Id "y")`	`TFun TInt TInt`	See *
6	`App (App (Prim "+") (Id "y")) (Number 1)`	`TInt`	See **

Type Variables

With type variables, checking if two types are “the same” is more involved.

Example

Suppose that for e = `(Cond tst thenExp elseExp)`, we infer

- `thenExp` has type `(TFun TInt (TVar "a"))`
- `elseExp` has type `(TFun (TVar "b") TBool)`

These types are not identical, but they are ***unifiable*** by requiring

`"a"` $\mapsto$ `TBool`   `"b"` $\mapsto$ `TInt`

The (successful) result of a unification is a unifying type substitution.

```
type TypeTable = [(String, Type)]
type Sub       = TypeTable
tsub :: Sub
tsub = [("a",TBool),("b",TInt)]
```

IMPORTANT: A unification can *fail*.

The Martelli-Montanari Unification Algorithm, I

- The algorithm operates on
  - A list of pairs of types $[(t_1, t_1'), \ldots, (t_n, t_n')]$ to unify, and
  - a type substitution, s, which records prior committments.
- Initially, $[(t, t')]$ is the list of pairs and $s = []$.
- The algorithm either

  returns the final version of s (***the unifying substitution***)

  or else reports failure.

The Martelli-Montanari Unification Algorithm, II

A step of the algorithm where

- the list of pairs $= ((t_1, t_1') : tps)$
- the type subst. $= s$

Case: $t_1 = t_1' = \texttt{TInt}$.
Then continue with
the list of pairs = *tps*
the type subst. = s.

Case: $t_1 = (\texttt{TFun}\ t_a\ t_b)$ and $t_1' = (\texttt{TFun}\ t_a'\ t_b')$
Then continue with
the list of pairs = $(t_a, t_a') : (t_b, t_b') : tps$
the type subst. = s

Case: $t_1 = (\texttt{TVar}\ v)$ and $t_1' = (\texttt{TVar}\ v')$ with $v = v'$.
Then continue with
the list of pairs = *tps*
the type subst. = s

More . . .

The Martelli-Montanari Unification Algorithm, III

Case: One of t_1 or t_1' is of the form `(TVar` v`)` and the other is some type t' where $t' \neq$ `(TVar` v`)`. Then

- the type subst. = $(v, t') : s$.
- the *tps* list is updated with the substitution $[(v, t')]$ applied to each pair in the list.
  - This can fail because of an ***occurs-check***.
  - More on this in a moment.

Case: Otherwise. Then report failure.

The Algorithm

Initially, $tps = [(t, t')]$ and $s = []$.
Keep applying the step-algorithm until $tps = []$,
in which case return s = ***the unifying substition***,
unless you failed along the way.

An example from Field

Suppose

```
tps = [(TVar "a", TInt),
       (TBool, TBool),
       (TFun (TVar "a") (TVar "b"), TVar "c"),
       (TInt, TVar "a")]
```

Then a `"a"` $\mapsto$ `TInt` substitution on `(tail tps)` yields

```
tps = [(TBool, TBool),
       (TFun TInt (TVar "b"), TVar "c"),
       (TInt, TInt)]
```

Which is the new *tps*.

Important point: **Occurs check**

Before we attempt a substitution $[(v, t)]$,
we check if the type variable v occurs in t;
if it does, unification ***FAILS!!!***

Why? Because otherwise we may have an infinite type (a no-no).

Example

The only solution to

$$a \equiv \texttt{Int} \to a$$

is the infinite type

$$\texttt{Int} \to \texttt{Int} \to \texttt{Int} \to \texttt{Int} \to \texttt{Int} \to \texttt{Int} \to \texttt{Int} \to \ldots$$

Unification examples from Field

t	t'	unify t t'
TFun (TVar "a") TInt	TVar "b"	Just [("b",TFun (TVar "a") TInt)]
TFun TBool TBool	TFun TBool TBool	Just []
TFun (TVar "a") TInt	TFun TBool TInt	Just [("a",TBool)]
TBool	TFun TInt TBool	Nothing
TFun (TVar "a") TInt	TFun TBool (TVar "b")	Just [("b",TInt),("a",TBool)]
TFun (TVar "a") (TVar "a")	TVar "a"	Nothing