Part VI: Type Systems

- Previously, we have considered statically untyped languages.
- We now look at type systems
- A type system is a set of rules which assigns types to parts of programs.
- If some part of a program cannot be assigned a type, the program is rejected with a type error.
- In the following, we will look at some simple type systems and their properties.

Strong/Weak typing vs Static/Dynamic typing

- A language is *strongly typed* or *safe* if violation of its rules will lead to an error, rather than leading to unspecified behavior. Otherwise, we say that the language is *weakly typed* or *untyped*.
- A language is *statically typed* if there is a type system which will disallow certain programs before such programs are run.
- Static and strong typing are not the same:

	strong	weak
static		
dynamic		

- Usually, some safety checks are left until run-time (example: array bounds checking) because they are impractical at compile-time.
- There are languages which guarantee complete safety at compile time, e.g. Martin Loef's type theories.
- But these require proofs of safety properties to be encoded explicitly in the program.

Questions

- What to types signify or guarantee? ($\Rightarrow$ *type soundness*).
- Is *type checking* or *type reconstruction* possible?

The Simply Typed λ Calculus

- Task: Add types to λ calculus.
- Two versions:
  - With explicitly given parameter types (Church)
  - Without (Curry)
- We present the Curry version here.

Alphabets

Variables	x, y, z		
Type Variables	a, b, c	Type Constructors	K

In practice, we use arbitrary words instead of single letters.

We write type constructors in upper case, type variables in lower case.

Examples:

$x, f, true, width$	Variables
a, b, c	Type variables
$Boolean, List$	Type constructors

Syntax

Terms	E, F	$::= x$	Variable	
		$\mid$	$\lambda x.E$	Abstraction
		$\mid$	$F\ E$	Application
Types	T, U	$::=$	a	Type variable
		$\mid$	$T \to U$	Function type
		$\mid$	$K[T_1, \ldots, T_n]$	Data type

For type constructors without parameters, we simply write K instead of $K[]$.

Function arrows associate to the right: $T_1 \to T_2 \to T_3 \ = \ T_1 \to (T_2 \to T_3)$

Type Assignments

$True$	:	$Boolean$
Nil	:	$List[Boolean]$
$\lambda x.x$	:	$Boolean \to Boolean$
	:	$List[Boolean] \to List[Boolean]$
	:	$a \to a$
$\lambda x.\lambda y.x$	:	$a \to b \to a$
$(\lambda x.\lambda y.x) True Nil$	:	$Boolean$
$(\lambda f.\lambda x.fx)$	:	
$(\lambda f.\lambda g.\lambda x.f(gx)$	:	
x	:	
$\lambda x.xy$	:	

Type Judgments

... are of the form $\Gamma \vdash E : T$.

where $\Gamma = (x_1 : T_1, \ldots, x_n : T_n)$ is a *type environment* consisting of a series of variable/type bindings, one for each free variable $x \in \text{fn}(E)$.

Read: "Under assumptions Γ, E has type T".

Special case for closed terms (i.e. $\text{fn}(E) = \emptyset$):

$$\vdash E : T \qquad \text{" } E \text{ has type } T \text{ "}$$

Deduction Systems

A Deduction system defines a formal language of *judgments* $\mathcal{J}$, together with rules which let one decide whether a judgment is *derivable* or not.

Rules take the form of axioms $\mathcal{J}$ and of deduction rules

$$\frac{\mathcal{J}_1 \quad \ldots \quad \mathcal{J}_n}{\mathcal{J}'}$$

A judgment $\mathcal{J}$ is derivable iff there is a proof tree such that

- Each leaf of the tree is an instance of an axiom.
- Each internal node of the tree is an instance of a deduction rule
- The root of the tree is the judgment $\mathcal{J}$.

Intuitionistic Logic

The first deduction systems have been developed for logic calculi.

Example: positive intuitionistic logic.

Let P, Q range over propositions with constant **true**, operators $\wedge$, $\vee$ and $\Rightarrow$ (missing is **false**, $\neg$).

Let Π be a *hypothesis*, i.e. a set of propositions which is assumed to be true.

Problem: How to decide whether Π implies P.

Solution: Give a deduction system for judgments of the form $\Pi \vdash P$.

Rules of Intuitionistic Logic

$$(\textsc{True})\quad \Pi \vdash \mathbf{true} \qquad\qquad (\textsc{Taut})\quad \Pi \vdash P \quad (P \in \Pi)$$

$$(\wedge\mathrm{I})\ \frac{\Pi \vdash P \qquad \Pi \vdash Q}{\Pi \vdash P \wedge Q} \qquad (\wedge\mathrm{E})\ \frac{\Pi \vdash P \wedge Q}{\Pi \vdash P} \qquad \frac{\Pi \vdash P \wedge Q}{\Pi \vdash Q}$$

$$(\Rightarrow\mathrm{I})\ \frac{\Pi, P \vdash Q}{\Pi \vdash P \Rightarrow Q} \qquad (\Rightarrow\mathrm{E})\ \frac{\Pi \vdash P \Rightarrow Q \qquad \Pi \vdash P}{\Pi \vdash Q}$$

$$(\vee\mathrm{I})\ \frac{\Pi \vdash P}{\Pi \vdash P \vee Q} \qquad \frac{\Pi \vdash Q}{\Pi \vdash P \vee Q}$$

$$(\vee\mathrm{E})\ \frac{\Pi \vdash P \vee Q \quad \Pi, P \vdash R \quad \Pi, Q \vdash R}{\Pi \vdash R}$$

Example Proof

Let $\Pi \stackrel{\text{def}}{\equiv} (P \vee Q), (P \Rightarrow Q)$.

Then:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\Pi \vdash P \vee Q} \qquad \cfrac{\cfrac{\Pi, P \vdash (P \Rightarrow R) \qquad \Pi, P \vdash P}{\Pi, P \vdash R}}{\Pi, P \vdash Q \vee R} \qquad \cfrac{\Pi, Q \vdash Q}{\Pi, Q \vdash Q \vee R}}{\Pi \vdash Q \vee R}}{(P \vee Q) \vdash (P \Rightarrow R) \Rightarrow Q \vee R}}{\vdash (P \vee Q) \Rightarrow ((P \Rightarrow R) \Rightarrow Q \vee R)}}{}$$

How to derive type judgments

Assume as given for each constant C a set **typeof**(C) of types.

Then we can derive type judgments by the following rules.

$$(\textsc{Var})\quad \Gamma, x : T, \Gamma' \;\vdash\; x : T \qquad (x \notin \mathrm{dom}(\Gamma'))$$

$$(\to\mathrm{I})\ \frac{\Gamma, x : T \;\vdash\; E : U}{\Gamma \;\vdash\; \lambda x.E : T \to U}$$

$$(\to\mathrm{E})\ \frac{\Gamma \;\vdash\; M : T \to U \qquad \Gamma \;\vdash\; N : T}{\Gamma \;\vdash\; M\ N : U}$$

Examples

id	$\equiv$	λx.x	: ?
apply	$\equiv$	λf.λx. f x	: ?
twice	$\equiv$	λf.λx. f (f x)	:
compose	$\equiv$	λf.λg. f (g x)	:

Exercise: : Construct proofs for these judgements.

Constants and Polymorphism

Nearly all programs are not closed terms but make use of predefined constants such as true or **if**.

We'd like to add these to an *initial environment* which is used to type whole programs.

But some constants have multiple types.

Example:

Nil : List[Int]
Nil : List[List[Int]]
Nil : List[a]

We subsume all of these types by a *type scheme* (or: *polymorphic type*).

$$\text{Type Scheme}\quad S \quad ::= \quad T \ \mid\ \forall a.S$$

Instantiation

Type schemes can be instantiated by the following elimination rule:

$$(\forall\mathrm{E})\ \frac{\Gamma \;\vdash\; E : \forall a.S}{\Gamma \;\vdash\; E : [T/a]S}$$

($[T/a]$ is *substitution*. $\forall a.S$ is called a *type scheme* or *polymorphic type*).

Example:

$$\frac{\Gamma \;\vdash\; \mathit{Nil} : \forall a.\mathit{List}[a]}{\Gamma \;\vdash\; \mathit{Nil} : \mathit{List}[\mathit{Int}]}$$

For the moment, we will admit polymorphism only for predefined constants, therefore an introduction rule for ∀is missing.

Some Useful Constants and their types

Nil	: ∀a.List[a]
Cons	: ∀a.a → List[a] → List[a]
head	: ∀a.List[a] → a
tail	: ∀a.List[a] → List[a]
isEmpty	: ∀a.List[a] → Boolean
true	: Boolean
false	: Boolean
if	: ∀a. Boolean → a → a → a
0, 1, 2, ...	: Int
plus	: Int → Int → Int
eq	: Int → Int → Boolean
fix	: ∀a. (a → a) → a

Example

Let

```
length =
  fix (λ length.λ xs.
    if (isEmpty xs)
      0
      (plus 1 (length (tail (xs)))))
```

Show

$\vdash$ length: List[a] → Int

Question: What do types signify?

Answer: "Types are sets of values". E.g.

$$T \quad \approx \quad \{V \mid V : T\}$$

where

$$\text{Value} \quad V \quad ::= \quad x \mid \lambda x.E$$

Question: Why is this useful?

Answer: Type judgements are preserved under reduction.

Subject Reduction and Type Soundness

Theorem: (Subject-Reduction) $\Gamma \vdash E : T$ and $E \twoheadrightarrow F$ imply $\Gamma \vdash F : T$.

Note: the converse of subject-reduction does not hold. I.e.

$$\Gamma \vdash F : T \wedge E \twoheadrightarrow F \quad \not\Rightarrow \quad \Gamma \vdash E : T$$

Definition: A language of terms E is *type-sound* if whenever $\vdash E : T$ then either E diverges or E reduces to a value of type T.

Type soundness is more than subject-reduction, since subject reduction still admits reduction of terms to "get stuck" in a non-value.

Theorem: Simply-typed lambda calculus is type-sound.

Product- and Sum-Types

In system discussed so far does not yet have types for products and sums.

The *product type* $T \times U$ represents all pairs whose first component is of type T and whose second component is of type U.

Problem: Design syntax and typing rules for formation of pairs and operations on them.

The *sum type* $T + U$ represents a tagged union of the types T and U.

Problem: Design syntax and typing rules for formation of tagged unions and operations on them.

The Curry-Howard Isomorphism

The deduction system for intuitionistic logic and the deduction system for simply typesd lambda calculus are remarkably similar!

We observe:

$$\begin{array}{rcl} \text{Formula} & \approx & \text{Type} \\ \text{Hypothesis} & \approx & \text{Type environment} \\ \Rightarrow & \approx & \rightarrow \\ \wedge & \approx & \times \\ \vee & \approx & + \end{array}$$

If types in lambda calculus are formulas of logic, what are the terms of lambda calculus?

Terms Are Proofs

Given a type judgement $\Gamma \vdash E : T$, we can interprete E as a proof of the formula represented by T.

Example 1: The deduction rule

$$\frac{\Gamma \vdash E_1 : T_1 \qquad \Gamma \vdash E_2 : T_2}{\Gamma \vdash (E_1, E_2) : T_1 \times T_2}$$

can be interpreted as:

"Given a proof E_1 of T_1 and a proof E_2 of T_2 we combine the two proofs to yield a proof of $T_1 \wedge T_2$."

Logical Frameworks

The Curry-Howard Isomorphism is used in interactive theorem provers such as LCF, ELF, HOL, Isabelle.

The user of such a prover encodes a proposition as a type and then proves the proposition by presenting a term which has this type.

Type systems are usually richer than the one we have seen – in particular they admit often admit types which depend on values.

Example: $\mathit{Array}(N)$ – the types of arrays with length N.

Type Checking and Type Reconstruction

We now come to the question of type checking and type reconstruction.

Type checking:

Given Γ, E and T, check whether $\Gamma \vdash E : T$

Type reconstruction:

Given Γ and E, find a type T such that $\Gamma \vdash E : T$

Type checking and reconstruction seem difficult since parameters in lambda calculus do not carry their types with them.

Type reconstruction also suffers from the problem that a term can have many types.

Idea: : We construct all type derivations in parallel, reducing type reconstruction to a unification problem.

From Judgements to Equations

$$TP : \mathit{Judgement} \rightarrow \mathit{Equations}$$

$$
\begin{array}{l}
TP(\Gamma \vdash E : T) = \\
\quad \textbf{case } E \textbf{ of} \\
\qquad \begin{array}{lcl}
x & : & \{\Gamma(x) \,\hat{=}\, T\} \\
\lambda x.E' & : & \textbf{let } a, b \text{ fresh } \textbf{in} \\
 & & \{(a \rightarrow b) \,\hat{=}\, T\} \quad \cup \\
 & & TP(\Gamma, x : a \vdash E' : b) \\
E\ E' & : & \textbf{let } a \text{ fresh } \textbf{in} \\
 & & TP(\Gamma \vdash E : a \rightarrow T) \quad \cup \\
 & & TP(\Gamma \vdash E' : a)
\end{array}
\end{array}
$$

Constants

Constants are treated as variables in the initial environment.

However, we have to make sure we create a new instance of their type as follows:

$$
\begin{array}{l}
\mathit{newInstance}(\forall a_1, \ldots, a_n.S) = \\
\quad \textbf{let } b_1, \ldots, b_n \text{ fresh } \textbf{in} \\
\quad [b_1/a_1, \ldots, b_n/a_n]S
\end{array}
$$

$$
\begin{array}{l}
TP(\Gamma \vdash E : T) = \\
\quad \textbf{case } E \textbf{ of} \\
\qquad x \quad : \quad \{\mathit{newInstance}(\Gamma(x)) \,\hat{=}\, T\} \\
\qquad \ldots
\end{array}
$$

Soundness and Completeness I

Definition: In general, a type reconstruction algorithm $\mathcal{A}$ assigns to an environment $\Gamma$and a term E a set of types $\mathcal{A}(\Gamma, E)$.

The algorithm is *sound* if for every type $T \in \mathcal{A}(\Gamma, E)$ we can prove the judgement $\Gamma \vdash E : T$.

The algorithm is *complete* if for every provable judgement $\Gamma \vdash E : T$ we have that $T \in \mathcal{A}(\Gamma, E)$.

Theorem: TP is sound and complete. Specifically:

$$\Gamma \vdash E : T \quad \text{iff} \quad \exists \overline{b}.[T/a]EQNS$$

where

a is a new type variable

$EQNS = TP(\Gamma \vdash E : a)$

$\overline{b} = \mathrm{tv}(EQNS) \backslash \mathrm{tv}(\Gamma)$

Here, tv denotes the set of free type varibales (of a term, and environment, an equation set).

Type Reconstruction and Unification

Problem: : Transform set of equations

$$\{T_i \doteq U_i\}_{i=1,\,\ldots,\,m}$$

into equivalent substitution

$$\{a_j \doteq T'_j\}_{j=1,\,\ldots,\,n}$$

where type variables do not appear recursively on their right hand sides (directly or indirectly). That is:

$$a_j \notin \mathrm{tv}(T'_k) \quad \text{for } j = 1, \ldots, n, k = j, \ldots, n$$

Substitutions

A *substitution* s is an idempotent mapping from type variables to types which maps all but a finite number of type variables to themselves.

We often represent a substitution is as set of equations $a \doteq T$ with a not in $\mathrm{tv}(T)$.

Substitutions can be generalized to mappings from types to types by defining

$$\begin{array}{lcl} s(T \to U) & = & sT \to sU \\ s(K[T_1, \ldots, T_n]) & = & K[sT_1, \ldots, sT_n] \end{array}$$

Substitutions are idempotent mappings from types to types, i.e. $s(s(T)) = s(T)$. (why?)

The $\circ$operator denotes composition of substitutions (or other functions): $(f \circ g)\ x \ = \ f(gx)$.

A Unification Algorithm

We present an incremental version of Robinson's algorithm (1965).

$$\begin{array}{lcll} mgu & : & (Type \doteq Type) \to Subst \to Subst & \\ mgu(T \doteq U)\ s & = & mgu'(sT \doteq sU)\ s & \\ mgu'(a \doteq a)\ s & = & s & \\ mgu'(a \doteq T)\ s & = & s \cup \{a \doteq T\} & \textbf{if } a \notin tv(T) \\ mgu'(T \doteq a)\ s & = & s \cup \{a \doteq T\} & \textbf{if } a \notin tv(T) \\ mgu'(T \to T' \doteq U \to U')\ s & = & (mgu(T' \doteq U') \circ mgu(T \doteq U))\ s & \\ \multicolumn{4}{l}{mgu'(K[T_1, \ldots, T_n] \doteq K[U_1, \ldots, U_n])\ s} \\ & = & (mgu(T_n \doteq U_n) \circ \ldots \circ mgu(T_1 \doteq U_1))\ s & \\ mgu'(T \doteq U)\ s & = & error & \text{in all other cases} \end{array}$$

Soundness and Completeness of Unification

Definition: A substitution u is a *unifier* of a set of equations $\{T_i \doteq U_i\}_{i=1,\dots,m}$ if $uT_i = uU_i$, for all i. It is a *most general unifier* if for every other unifier u' of the same equations there exists a substitution s such that $u' = s \circ u$.

Theorem: Given a set of equations $EQNS$. If $EQNS$ has a unifier then $mgu\ EQNS\ \{\}$ computes the most general unifier of $EQNS$. If $EQNS$ has no unifier then $mgu\ EQNS\ \{\}$ fails.

From Judgements to Substitutions

$$TP : Judgement \to Subst \to Subst$$

$$
\begin{array}{l}
TP(\Gamma \vdash E : T) = \\
\quad \textbf{case } E \textbf{ of} \\
\qquad \begin{array}{lcl}
x & : & \mathrm{mgu}(newInstance(\Gamma x) \doteq T) \\
\lambda x.E' & : & \textbf{let } t, u \text{ fresh } \textbf{in} \\
 & & \mathrm{mgu}((t \to u) \doteq T) \quad \circ \\
 & & TP(\Gamma, x : t \vdash E' : u) \\
E\ E' & : & \textbf{let } t \text{ fresh } \textbf{in} \\
 & & TP(\Gamma \vdash E : a \to T) \quad \circ \\
 & & TP(\Gamma \vdash E' : a)
\end{array}
\end{array}
$$

Soundness and Completeness II

One can show by comparison with the previous algorithm:

Theorem: TP is sound and complete. Specifically:

$$
\begin{array}{l}
\Gamma \vdash E : T \quad \text{iff} \quad T = r(s(t)) \\
\qquad\qquad \textbf{where} \\
\qquad\qquad\quad t \text{ is a new type variable} \\
\qquad\qquad\quad s = TP\ (\Gamma \vdash E : t)\ \{\} \\
\qquad\qquad\quad r \text{ is a substitution on } \mathrm{tv}(s\ t) \backslash \mathrm{tv}(s\ \Gamma)
\end{array}
$$

Strong Normalization

Question: Can Ω be given a type?

$$\Omega \quad = \quad (\lambda x.xx)(\lambda x.xx) :?$$

What about Y?

Self-application is not typable!

In fact, we have more:

Theorem: (Strong Normalization) If $\vdash E : T$, then there is a value V such that $E \twoheadrightarrow V$.

Corollary: Simply typed lambda calculus is not Turing complete.