Types

Robin Milner in Computing Tomorrow:

“One of the most helpful concepts in the whole of programming is the notion of type, used to classify the kinds of object which are manipulated. A significant proportion of programming mistakes are detected by an implementation which does type-checking before it runs any program.”
What Are Types Good For?

- Detect errors via type-checking (prevent multiplication of an integer by a bool)
- Abstraction and Interfaces (programmer 1: “please give me a value in mph”; programmer 2: “I give you a value in kmph”)
- Documentation (useful hints about intended use which is kept consistent with the changes of the program)
- Efficiency (if I know a value is an int, I can compile to use machine registers)
Definition of Types

\[ T ::= X \mid T_1 \rightarrow T_2 \]

- Type variables and function types
Definition of Types

\[ T ::= X \mid T_1 \to T_2 \]

- Type variables and function types

nominal_datatype ty =
  \( \text{TVar "string"} \)
  \| \( \text{TArr "ty" "ty" (_ \to _)} \)
Definition of Types

\[ T ::= X \mid T_1 \rightarrow T_2 \]

- Type variables and function types
  
  ```
  nominal_datatype ty =
  TVar "string"
  | TArr "ty" "ty" (_, _)
  ```

- In real programming languages you also have primitive types, for example `bool`, `string`, `int`, and type constructors for example `_ \times _`. 
Type Contexts

The type of variables will be explicitly given in a typing-context. They are lists of (variable,type)-pairs:

\[ \Gamma = [(x, T_1), (y, T_2), (z, T_3)] \]
Type Contexts

The type of variables will be explicitly given in a typing-context. They are lists of (variable,type)-pairs:

\[ \Gamma = [(x, T_1), (y, T_2), (z, T_3)] \]

we write them as

\[ \Gamma = [x : T_1, y : T_2, z : T_3] \]
Type Contexts

The type of variables will be explicitly given in a typing-context. They are lists of (variable,type)-pairs:

\[ \Gamma = [(x, T_1), (y, T_2), (z, T_3)] \]

we write them as

\[ \Gamma = [x : T_1, y : T_2, z : T_3] \]

inductive

valid :: "(name×ty) list ⇒ bool"

where

v1[intro]: "valid []"

| v2[intro]: "[valid \( \Gamma \); x#\( \Gamma \)]⇒ valid ((x,T)#\( \Gamma \))"
Typing Rules

inductive

typing :: "(name × ty) list ⇒ lam ⇒ ty ⇒ bool" ("_ ⊢ _ : _")

where

\[\text{\textbf{tVar: } } [\text{valid } \Gamma; (x, T) ∈ \text{set } \Gamma] \implies \Gamma \vdash \text{Var } x : T\]

\[\text{\textbf{tApp: } } [\Gamma \vdash t_1 : T_1 \rightarrow T_2; \Gamma \vdash t_2 : T_1] \implies \Gamma \vdash \text{App } t_1 \ t_2 : T_2\]

\[\text{\textbf{tLam: } } [x \# \Gamma; (x, T_1)\# \Gamma \vdash t : T_2] \implies \Gamma \vdash \text{Lam } [x].t : T_1 \rightarrow T_2\]
Typing Rules

inductive
typing :: "(name × ty) list ⇒ lam ⇒ ty ⇒ bool" ("_ ⊢ _ : _")
where
  tVar: "[valid Γ; (x, T) ∈ set Γ] ⊢ Γ ⊢ Var x : T"
| tApp: "[Γ ⊢ t₁ : T₁ → T₂; Γ ⊢ t₂ : T₁] ⊢ Γ ⊢ App t₁ t₂ : T₂"
| tLam: "[x # Γ; (x, T₁) # Γ ⊢ t : T₂] ⊢ Γ ⊢ Lam [x].t : T₁ → T₂"

An interesting property is type soundness:

- If Γ ⊢ t₁ : T and t₁ ⟩ β ⟩ t₂ then Γ ⊢ t₂ : T.
- If Γ ⊢ t₁ : T and t₁ ↓ t₂ then Γ ⊢ t₂ : T.
Weakening Lemma

abbreviation
"sub_cxt" :: "(name × ty) list ⇒ (name × ty) list ⇒ bool" ("_ ⊆ _")

where
"Γ₁ ⊆ Γ₂ ≡ ∀ x T. (x, T) ∈ set Γ₁ ⟷ (x, T) ∈ set Γ₂"

Weakening Lemma:

- If Γ₁ ⊢ t : T, Γ₁ ⊆ Γ₂ and valid Γ₂ then Γ₂ ⊢ t : T.
Proof

lemma weakening_lemma:
  fixes Γ₁ Γ₂ : "(name × ty) list"
  assumes a: "Γ₁ ⊢ t : T" "valid Γ₂" "Γ₁ ⊆ Γ₂"
  shows "Γ₂ ⊢ t : T"

using a

proof (induct arbitrary: Γ₂)
  case (tVar Γ₁ × T)
  have "Γ₁ ⊆ Γ₂" by fact
  moreover
  have "valid Γ₂" by fact
  moreover
  have "(x, T) ∈ (set Γ₁)" by fact
  ultimately show "Γ₂ ⊢ Var x : T" by auto

next
lemma weakening_lemma:
    fixes $\Gamma_1 \Gamma_2 :: \text{"(name \times ty) list"}
    assumes a: "\Gamma_1 \vdash t : T" "valid \Gamma_2" "\Gamma_1 \subseteq \Gamma_2"
    shows "\Gamma_2 \vdash t : T"
using a
proof (induct arbitrary: $\Gamma_2$)
case (tLam x $\Gamma_1 T_1 t T_2$)
  have a0: "x\#\Gamma_1" by fact
  have a1: "(x,T_1)\#\Gamma_1 \vdash t : T_2" by fact
  have a2: "\Gamma_1 \subseteq \Gamma_2" by fact
  have a3: "valid \Gamma_2" by fact
  have ih: "\bigwedge \Gamma_3. [valid \Gamma_3; (x,T_1)\#\Gamma_1 \subseteq \Gamma_3] \implies \Gamma_3 \vdash t : T_2"
  have "(x,T_1)\#\Gamma_1 \subseteq (x,T_1)\#\Gamma_2" using a2 by simp
  moreover
  have "valid ((x,T_1)\#\Gamma_2)" using v2
lemma weakening_lemma:
  fixes $\Gamma_1 \Gamma_2 :: \text{"(name \times ty) list"}$
assumes a: "$\Gamma_1 \vdash t : T" "valid $\Gamma_2" "$\Gamma_1 \subseteq \Gamma_2"
shows "$\Gamma_2 \vdash t : T"
using a
proof (nominal_induct $\Gamma_1 \vdash T$ avoiding: $\Gamma_2$
  rule: typing.strong_induct)
case ($t\text{Lam \langle x \rangle} \Gamma_1 \Gamma_1 \vdash T_1 \vdash T_2$)
  have vc: "$x \notin \Gamma_2" by fact
  have ih: "$\text{valid ((x,T_1)\#}\Gamma_2); (x,T_1)\#\Gamma_1 \subseteq (x,T_1)\#\Gamma_2]$
    $\implies (x,T_1)\#\Gamma_2 \vdash t : T_2" by fact
  have "$\Gamma_1 \subseteq \Gamma_2" by fact
  then have "$((x,T_1)\#\Gamma_1 \subseteq (x,T_1)\#\Gamma_2" by simp
moreover
  have "valid $\Gamma_2" by fact
  then have "valid ((x,T_1)\#\Gamma_2)" using vc by (simp add: v2)
ultimately have "((x,T_1)\#\Gamma_2) \vdash t : T_2" using ih by simp
with vc show "$\Gamma_2 \vdash \text{Lam \langle x \rangle}.t : T_1 \rightarrow T_2" by auto
Can we always do that?

inductive
strip :: "lam ⇒ lam ⇒ bool" ("_ ⊢ _")
where
sVar: "(Var x) ⊢ (Var x)"
sApp: "(App t1 t2) ⊢ (App t1 t2)"
sLam: "t ⊢ t' ⊢ (Lam [x]. t) ⊢ t'"

<table>
<thead>
<tr>
<th>Var x ⊢ Var x</th>
<th>App t1 t2 ⊢ App t1 t2</th>
</tr>
</thead>
<tbody>
<tr>
<td>t ⊢ t'</td>
<td>Lam [x]. t ⊢ t'</td>
</tr>
</tbody>
</table>
lemma faulty:
fixes x::"name"
assumes a: "t ⟷ t'"
shows "x ∉ t ⟷ x ∉ t'"
using a by (nominal_induct t t'≡ t' avoiding: t'
  rule: strip.strong_induct)
  (auto simp add: abs_fresh)

lemma false:
shows "False"
proof -
  fix x
  have "Lam [x].(Var x) ⟷ (Var x)" by (auto intro: strip.intros)
  moreover
  have "x ∉ Lam [x].(Var x)" by (simp add: abs_fresh)
  ultimately have "x ∉ (Var x)" by (simp only: faulty)
  then show "False" by (simp add: fresh_atm)
qed
If we want that a binder avoids something in a rule induction, then this binder cannot be free in the conclusion.

\[
\begin{align*}
\text{Var } x & \quad \leftrightarrow \quad \text{Var } x \\
\text{App } t_1 \ t_2 & \quad \leftrightarrow \quad \text{App } t_1 \ t_2 \\
\top & \quad \leftrightarrow \quad t' \\
\text{Lam } [x].\top & \quad \leftrightarrow \quad t'
\end{align*}
\]
If we want that a binder avoids something in a rule induction, then this binder cannot be free in the conclusion.

\[
\begin{align*}
\text{valid } \Gamma & \quad (x, T) \in \text{set } \Gamma \\
\Gamma & \vdash \text{Var } x : T \\
\Gamma & \vdash t_1 : T_1 \rightarrow T_2 \\
\Gamma & \vdash t_2 : T_1 \\
\Gamma & \vdash \text{App } t_1 \ t_2 : T_2 \\
\times \# \Gamma & \quad (x, T_1)\# \Gamma \vdash t : T_2 \\
\Gamma & \vdash \text{Lam } [x].t : T_1 \rightarrow T_2
\end{align*}
\]
Safety Conditions

If we want that a binder avoids something in a rule induction, then this binder cannot be free in the conclusion.

It also depends on the inductive definitions to be equivariant:

- \( \pi \cdot t[x::=s] = (\pi \cdot t)[(\pi \cdot x)::=(\pi \cdot s)] \)
- If valid \( \Gamma \) then valid \( (\pi \cdot \Gamma) \)
- If \( \Gamma \vdash t : \tau \) then \( \pi \cdot \Gamma \vdash \pi \cdot t : \pi \cdot \tau \)
Every typable lambda-term is strongly normalising (that is every beta-reduction sequence terminates). Proofs of this property are notoriously difficult.

Type soundness:
If $\Gamma \vdash t_1 : T$ and $t_1 \xrightarrow{\beta} t_2$ then $\Gamma \vdash t_2 : T$.

Type soundness needs a lemma:
If $(x, \top)\#\Gamma \vdash t_1 : T'$ and $\Gamma \vdash t_2 : T$ then $\Gamma \vdash t_1[x::=t_2] : T'$. 
Inversions

We also need the following inversion lemmas

**lemma Ty_Lam_inversion:**

*assumes* ty: "Γ ⊢ Lam [x].t : T" and fc: "x ∈ Γ"

*shows* "∃ T₁ T₂. T = T₁ → T₂ ∧ (x,T₁) ∈ Γ ⊢ t : T₂"

*using* ty fc

*by* (cases rule: typing.strong_cases)

(auto simp add: alpha lam.inject ty.inject abs_fresh
   ty_fresh)

**lemma Beta_Lam_inversion:**

*assumes* red: "Lam [x].t ⊢β s" and fc: "x ∈ s"

*shows* "∃ t'. s = Lam [x].t' ∧ t ⊢β t'"

*using* red fc

*by* (cases rule: beta_red.strong_cases)

(auto simp add: alpha lam.inject abs_fresh)
Inversions

lemma Beta_App_inversion:

assumes red: "App (Lam [x].t) s \rightarrow_{\beta} r" and fc: "x \notin (s,r)"

shows "(\exists t'. r = App (Lam [x].t') s \land t \rightarrow_{\beta} t') \lor
(\exists s'. r = App (Lam [x].t) s' \land s \rightarrow_{\beta} s') \lor
(r = t[x::=s])"

using red fc

by (cases rule: beta_red.strong_cases)
  (auto dest: Beta_Lam_inversion simp add: alpha lam.inject
  abs_fresh fresh_prod)
Points to Take Home

- Types and type-systems for programming languages are an important branch of programming language research. Types ensure safety properties that can be checked at compile-time.

- Type safety proofs for real languages are complex and need computer support.

- Types in theorem provers usually ensure consistency.

- The variable convention in proofs by rule induction can lead to faulty proofs (the rules need to satisfy additional constraints).