Last time on **Types**...

- *Principal type schemes*

Principal type schemes for closed expressions

A closed type scheme $\forall A\,(\tau)$ is the principal type scheme of a closed Mini-ML expression M if

(a) $\vdash M : \forall A\,(\tau)$

(b) for any other closed type scheme $\forall A'\,(\tau')$,
if $\vdash M : \forall A'\,(\tau')$, then $\forall A\,(\tau) \succ \tau'$

Theorem (Hindley; Damas-Milner)

slide 26 (p.19)

Theorem

If the closed Mini-ML expression M is typeable (i.e. $\vdash M : \sigma$ holds for some type scheme σ), then there is a principal type scheme for M.

Last time on **Types**...

- *Principal type schemes*
- *MGUs* (most general unifiers)

There is an algorithm *mgu* which when input two Mini-ML types τ_1 and τ_2 decides whether τ_1 and τ_2 are unifiable, i.e. whether there exists a type-substitution $S \in \mathrm{Sub}$ with

(a) $S(\tau_1) = S(\tau_2)$.

Moreover, if they are unifiable, $mgu(\tau_1, \tau_2)$ returns the most general unifier—an S satisfying both (a) and

(b) for all $S' \in \mathrm{Sub}$, if $S'(\tau_1) = S'(\tau_2)$, then $S' = TS$ for some $T \in \mathrm{Sub}$
(any other substitution S' can be factored through S, by specialising S with T)

By convention $mgu(\tau_1, \tau_2) = \mathit{FAIL}$ if (and only if) τ_1 and τ_2 are not unifiable.

Last time on **Types**...

- *Principal type schemes*
- *MGUs* (most general unifiers)
- *Type inference algorithm* (pt) [also called "Algorithm $\mathcal{W}$"]

Some of the clauses in a definition of *pt*

Function abstractions: $pt(\Gamma \vdash \lambda x(M) : ?) \stackrel{\text{def}}{=}$

$$\begin{array}{l} \text{let } \alpha = \text{fresh in} \\ \text{let } (S, \tau) = pt(\Gamma, x : \alpha \vdash M : ?) \text{ in } (S, S(\alpha) \to \tau) \end{array}$$

Function applications: $pt(\Gamma \vdash M_1\, M_2 : ?) \stackrel{\text{def}}{=}$

$$\begin{array}{l} \text{let } (S_1, \tau_1) = pt(\Gamma \vdash M_1 : ?) \text{ in} \\ \text{let } (S_2, \tau_2) = pt(S_1\, \Gamma \vdash M_2 : ?) \text{ in} \\ \text{let } \alpha = \text{fresh in} \\ \text{let } S_3 = mgu(S_2\, \tau_1, \tau_2 \to \alpha) \text{ in } (S_3 S_2 S_1, S_3(\alpha)) \end{array}$$

A rough guide to constructing Algorithm $\mathcal{W}$ (pt)

$$pt(\Gamma \vdash e :?) = (S, \tau)$$

- Recursively apply on sub terms - see type rules
  - thread substitutions through
  - collect substitutions at the end
- When types need to agree (see type rules), use *mgu*
- When types are unknown, generate a **fresh** type variable

This time on **Types**...

- Extend Mini-ML with references
- Type soundness lost.
- Fix type system; type soundness regained.

Formal type systems

- Constitute the precise, mathematical characterisation of informal type systems (such as occur in the manuals of most typed languages.)
- Basis for type soundness theorems: 'any well-typed program cannot produce run-time errors (of some specified kind)'.
- Can decouple specification of typing aspects of a language from algorithmic concerns: the formal type system can define typing independently of particular implementations of type-checking algorithms.

ML types and expressions for mutable references

$$
\begin{array}{rcll}
\tau & ::= & \dots & \\
 & \mid & \mathit{unit} & \text{unit type} \\
 & \mid & \tau\ \mathit{ref} & \text{reference type.} \\
 & & & \\
M & ::= & \dots & \\
 & \mid & () & \text{unit value} \\
 & \mid & \mathtt{ref}\ M & \text{reference creation} \\
 & \mid & !M & \text{dereference} \\
 & \mid & M := M & \text{assignment}
\end{array}
$$

Midi-ML's extra typing rules

$$\Gamma \vdash () : \mathit{unit} \tag{unit}$$

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \texttt{ref}\ M : \tau\ \mathit{ref}} \tag{ref}$$

$$\frac{\Gamma \vdash M : \tau\ \mathit{ref}}{\Gamma \vdash\ !M : \tau} \tag{get}$$

$$\frac{\Gamma \vdash M_1 : \tau\ \mathit{ref} \qquad \Gamma \vdash M_2 : \tau}{\Gamma \vdash M_1 := M_2 : \mathit{unit}} \tag{set}$$

Example

The expression

$$
\begin{array}{l}
\texttt{let}\, r = \texttt{ref}\, \lambda x(x)\, \texttt{in} \\
\quad \texttt{let}\, u = (r := \lambda x'(\texttt{ref}\, !x'))\, \texttt{in} \\
\qquad (!r)()
\end{array}
$$

has type *unit*.

$$\text{let } r = \overbrace{\text{ref } \lambda x(x)}^{①} \text{ in}$$

$$\quad \text{let } u = (\underbrace{r}_{3} := \underbrace{\lambda x'(\text{ref } ! x')}_{②}) \text{ in}$$

$$\quad (!\underbrace{r}_{④})\,()$$

Work out the types for ① – ④ and the type ~~sha~~ scheme for r in the body of the (outer) let.

① $(\alpha \to \alpha)\,\text{ref}$

② $\beta\,\text{ref} \to \beta\,\text{ref}$

③ $(\beta\,\text{ref} \to \beta\,\text{ref})\,\text{ref}$

④ $(\text{unit} \to \text{unit})\,\text{ref}$

$$r : \forall \alpha . (\alpha \to \alpha)\,\text{ref}$$

Later

with (letv) [value restriction] rule for Midi-ML then

$$r : \forall \emptyset . (\alpha \to \alpha)\,\text{ref}$$

Midi-ML transition system

extra

$$\langle M, s \rangle \to \langle M', s' \rangle$$

$$\textit{or} \quad \langle M, s \rangle \to \mathit{FAIL}$$

where $\mathit{fv}(M) \subseteq \mathit{dom}(s)$.

- M, M' range over Midi-ML expressions
- s, s' range over states

  (finite functions mapping variables to values)

$$\{x_1 \mapsto V_1, \ldots, x_n \mapsto V_n\}$$

Midi-ML transitions involving references

$$\langle !x, s\rangle \to \langle s(x), s\rangle \quad \text{if } x \in dom(s)$$

$$\langle !V, s\rangle \to FAIL \quad \text{if } V \text{ not a variable}$$

$$\langle x := V', s\rangle \to \langle (), s[x \mapsto V']\rangle$$

$$\langle V := V', s\rangle \to FAIL \quad \text{if } V \text{ not a variable}$$

$$\langle \texttt{ref}\ V, s\rangle \to \langle x, s[x \mapsto V]\rangle \quad \text{if } x \notin dom(s)$$

where V ranges over values:

$$V ::= x \mid \lambda x(M) \mid () \mid \texttt{true} \mid \texttt{false} \mid \texttt{nil} \mid V :: V$$

Value-restricted typing rule for `let`-expressions

$$\frac{\Gamma \vdash M_1 : \tau_1 \qquad \Gamma, x : \forall A\,(\tau_1) \vdash M_2 : \tau_2}{\Gamma \vdash \mathtt{let}\, x = M_1\, \mathtt{in}\, M_2 : \tau_2}\ (\dagger) \qquad \text{(letv)}$$

($\dagger$) provided $x \notin dom(\Gamma)$ and

$$A = \begin{cases} \{\,\} & \text{if } M_1 \text{ is not a value} \\ ftv(\tau_1) - ftv(\Gamma) & \text{if } M_1 \text{ is a value} \end{cases}$$

(Recall that values are given by
$V ::= x \mid \lambda x(M) \mid () \mid \mathtt{true} \mid \mathtt{false} \mid \mathtt{nil} \mid V :: V$.)

$$\left\langle \begin{array}{l} \text{let } r = \text{ref } \lambda x(x) \text{ in} \\ \text{let } u = (r := \lambda x'(\text{ref } !x')) \text{ in} \\ (!r)() \end{array}, \{\} \right\rangle \longrightarrow^*$$

$$\left\langle \begin{array}{l} \text{let } u = (r := \lambda x'(\text{ref } !x')) \text{ in} \\ (!r)() \end{array}, \{r \mapsto \lambda x(x)\} \right\rangle \longrightarrow^*$$

$$\langle (!r)(), \{r \mapsto \lambda x'(\text{ref } !x')\} \rangle \longrightarrow$$

$$\langle (\lambda x'(\text{ref } !x'))(), \{r \mapsto \lambda x'(\text{ref } !x')\} \rangle \longrightarrow$$

$$\langle \text{ref } !(), \{r \mapsto \lambda x'(\text{ref } !x')\} \rangle \longrightarrow$$

FAIL

Type soundness for Midi-ML with the value restriction

For any closed Midi-ML expression M, if there is some type scheme σ for which

$$\vdash M : \sigma$$

is provable in the value-restricted type system (axioms and rules on Slides 7–8, 2 and 1), then evaluation of M does not fail, i.e. there is no sequence of transitions of the form

$$\langle M, \{\,\} \rangle \to \cdots \to \mathit{FAIL}$$

for the transition system $\to$ defined in Figure 4 (of the notes) (where $\{\,\}$ denotes the empty state).

note: with the (letv) rule, some Mini-ML expressions that were typeable become untypeable in Midi-ML, e.g.,

$$\texttt{let}\, f = (\lambda x(x))(\lambda y(y))\,\texttt{in}\,(f\,\texttt{true}) :: (f\,\texttt{nil})$$

(but we can often avoid this using η-expansion and β-reduction).

Next time on **Types**...

Polymorphic λ-calculus

(polymorphic λ-binding). Let's us type:

$$\lambda f((f\,\mathtt{true}) :: (f\,\mathtt{nil}))$$