Type soundness

If

• $\Gamma \vdash e : \tau$
• $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$
• $\Gamma$, $\rho$, and $\sigma$ are consistent,

then

$\tau$ predicts $v$

Consistency: $\text{dom } \Gamma = \text{dom } \rho$, and
$\forall x \in \text{dom } \Gamma : \Gamma(x)$ predicts $\sigma(\rho(x))$.

Sample predictions: int predicts 7, bool predicts #t
Understanding language design

Questions about types never seen before:
- What types can I make?
- What syntax goes with each form?
- What functions?
- What about user-defined types?

Examples: pointer, struct, function, record
Talking type theory

Formation: make new types

Introduction: make new values

Elimination: observe ("take apart") existing values
Types and their C constructs

<table>
<thead>
<tr>
<th>Type</th>
<th>Produce</th>
<th>Consume</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>struct</strong></td>
<td><strong>Introduce</strong></td>
<td><strong>Eliminate</strong></td>
</tr>
<tr>
<td>(definition form only)</td>
<td></td>
<td>dot notation</td>
</tr>
<tr>
<td><em>e</em>.next, e-&gt;next</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>pointer</strong></td>
<td>&amp;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>*</td>
</tr>
<tr>
<td><strong>function</strong></td>
<td>(definition form only)</td>
<td>application</td>
</tr>
</tbody>
</table>
## Types and their $\mu$Scheme constructs

<table>
<thead>
<tr>
<th>Type</th>
<th>Produce</th>
<th>Consume</th>
</tr>
</thead>
<tbody>
<tr>
<td>record</td>
<td>constructor</td>
<td>accessor functions</td>
</tr>
<tr>
<td>function</td>
<td>lambda</td>
<td>application</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type</th>
<th>Produce</th>
<th>Consume</th>
</tr>
</thead>
<tbody>
<tr>
<td>record</td>
<td>introduce</td>
<td>eliminate</td>
</tr>
<tr>
<td>function</td>
<td>function</td>
<td>type predicate</td>
</tr>
</tbody>
</table>
### Types and their ML constructs

<table>
<thead>
<tr>
<th>Type</th>
<th>Produce</th>
<th>Consume</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Produce</td>
<td>Consume</td>
</tr>
<tr>
<td>Young</td>
<td>Young</td>
<td>Young</td>
</tr>
<tr>
<td>Constructed (algebraic)</td>
<td>Constructed</td>
<td>Constructed</td>
</tr>
<tr>
<td>Constructed (tuple)</td>
<td>Constructed</td>
<td>Constructed</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arrow</th>
<th>Lambda (fn)</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructed (algebraic)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(e₁, ..., eₙ)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Pattern match             | Pattern match!| Pattern match!|
Examples: Well-formed types

These are types:

• int
• bool
• int * bool
• int * int -> int
Examples: Not yet types, or not types at all

These “types in waiting” don’t classify any terms
  • list (but int list is a type)
  • array (but char array is a type)
  • ref (but (int -> int) ref is a type)

These are utter nonsense
  • int int
  • bool * array
Type-formation rules

We need a way to classify type expressions into:

- types that classify terms
- type constructors that build types
- nonsense that doesn’t mean anything
Type constructors

Technical name for “types in waiting”

Given zero or more arguments, produce a type:
  - **Nullary** `int`, `bool`, `char` also called **base types**
  - **Unary** `list`, `array`, `ref`
  - **Binary (infix)** `->`

More complex type constructors:
  - `records/structs`
  - `function in C, uScheme, Impcore`
What’s a good type? (Type formation)

Type formation rules for Typed Impcore

\[
\tau \in \{\text{UNIT, INT, BOOL}\} \\
\tau \text{ is a type}
\]  

(BASETYPES)

\[
\tau \text{ is a type} \\
\text{ARRAY}(\tau) \text{ is a type}
\]

(ARRAYFORMATION)

Design idea: what values does it predict?
Type judgments for monomorphic system

Two judgments:

• The familiar *typing judgment* $\Gamma \vdash e : \tau$
• Today’s judgment “$\tau$ is a type”
Type rules for variables

Look up the type of a variable:

$x \in \text{dom} \Gamma \quad \Gamma(x) = \tau$

\[ \Gamma \vdash x : \tau \]  \hspace{1cm} (VAR)

Types match in assignment (two $\tau$’s must be equal):

$x \in \text{dom} \Gamma \quad \Gamma(x) = \tau \quad \Gamma \vdash e : \tau$

\[ \Gamma \vdash \text{SET}(x, e) : \tau \]  \hspace{1cm} (SET)
Understanding the \texttt{SET} rule

Types match in assignment (two $\tau$’s must be equal):

\[
\begin{align*}
x & \in \text{dom } \Gamma \\
\Gamma(x) & = \tau \\
\Gamma & \vdash e : \tau \\
\Gamma & \vdash \texttt{SET}(x, e) : \tau
\end{align*}
\] (\texttt{SET})
Understanding the SET rule

Types match in assignment (two $\tau$’s must be equal):

\[
x \in \operatorname{dom} \Gamma \quad \Gamma(x) = \boxed{\tau} \quad \Gamma \vdash e : \boxed{\tau}
\]

\[
\Gamma \vdash \text{SET}(x, e) : \boxed{\tau}
\]

(SET)

\[
x \in \operatorname{dom} \Gamma \quad \Gamma(x) = \boxed{\tau_x} \quad \Gamma \vdash e : \boxed{\tau_e} \quad \tau_x \equiv \tau_e
\]

\[
\Gamma \vdash \text{SET}(x, e) : \boxed{\tau_e}
\]

(SET)
Type rules for control

Boolean condition; matching branches

\[
\frac{\Gamma \vdash e_1 : \text{BOOL} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{IF}(e_1, e_2, e_3) : \tau}
\]
Product types: Both $x$ and $y$

**New abstract syntax:** PAIR, FST, SND

$\tau_1$ and $\tau_2$ are types

$$\underbrace{\quad \tau_1 \times \tau_2 \text{ is a type} \quad}_1$$

$\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2$

$$\underbrace{\quad \Gamma \vdash \text{PAIR}(e_1, e_2) : \tau_1 \times \tau_2 \quad}_2$$

$\Gamma \vdash e : \tau_1 \times \tau_2$

$$\underbrace{\quad \Gamma \vdash \text{FST}(e) : \tau_1 \quad}_3$$

$\Gamma \vdash e : \tau_1 \times \tau_2$

$$\underbrace{\quad \Gamma \vdash \text{SND}(e) : \tau_2 \quad}_4$$

Pair rules generalize to product types with many elements ("tuples," "structs," and "records")
Arrow types: Function from x to y

Syntax: \texttt{lambda}, application

Typed \(\mu\) Scheme style:

\[
\frac{\tau_1, \ldots, \tau_n \text{ and } \tau \text{ are types}}{(\tau_1 \cdots \tau_n \rightarrow \tau) \text{ is a type}} \quad \text{(\textsc{arrowformation})}
\]

ML style: each function takes a tuple:

\[
\frac{\tau_1, \ldots, \tau_n \text{ and } \tau \text{ are types}}{\tau_1 \times \cdots \times \tau_n \rightarrow \tau \text{ is a type}} \quad \text{(\textsc{mlarrowformation})}
\]
Arrow types: Function from x to y

Eliminate with application:

\[
\Gamma \vdash e : (\tau_1 \cdots \tau_n \rightarrow \tau) \\
\Gamma \vdash e_i : \tau_i, \ 1 \leq i \leq n \\
\Gamma \vdash \text{APPLY}(e, e_1, \ldots, e_n) : \tau
\]

Introduce with \texttt{lambda}:

\[
\Gamma \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau \\
\Gamma \vdash \text{LAMBDA}(x_1 : \tau_1, \ldots, x_n : \tau_n, e) : (\tau_1 \cdots \tau_n \rightarrow \tau)
\]
Typical syntactic support for types

Explicit types on lambda and define:

• For lambda, argument types:
  \[(\text{lambda} \ ([n : \text{int}] \ [m : \text{int}]) \ (+ \ (* \ n \ n) \ (* \ m \ m)))\]

• For define, argument and result types:
  \[(\text{define} \ \text{int} \ \text{max} \ ([x : \text{int}] \ [y : \text{int}])
    \quad (\text{if} \ (< \ x \ y) \ y \ x))\]

Abstract syntax:

```plaintext
datatype \text{exp} = \ldots
  | \text{LAMBDA} \ of \ (\text{name} * \text{tyex}) \ list * \text{exp}
\ldots

datatype \text{def} = \ldots
  | \text{DEFINE} \ of \ \text{name} * \text{tyex} * ((\text{name} * \text{tyex}) \ list * \text{exp})
\ldots
```
Array types: Array of x

Formation: \( \tau \) is a type
\[
\frac{}{\text{ARRAY}(\tau) \text{ is a type}}
\]

Introduction:
\[
\frac{\Gamma \vdash e_1 : \text{INT} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{AMAKE}(e_1, e_2) : \text{ARRAY}(\tau)}
\]
Array types continued

Elimination:

$$\Gamma \vdash e_1 : \text{ARRAY}(\tau) \quad \Gamma \vdash e_2 : \text{INT}$$

$$\Gamma \vdash \text{AAT}(e_1, e_2) : \tau$$

$$\Gamma \vdash e_1 : \text{ARRAY}(\tau) \quad \Gamma \vdash e_2 : \text{INT} \quad \Gamma \vdash e_3 : \tau$$

$$\Gamma \vdash \text{APUT}(e_1, e_2, e_3) : \tau$$

$$\Gamma \vdash e : \text{ARRAY}(\tau)$$

$$\Gamma \vdash \text{ASIZE}(e) : \text{INT}$$
References (similar to C/C++ pointers)

Your turn! Given

\[
\begin{align*}
\text{ref } \tau & \quad \text{REF}(\tau) \\
\text{ref } e & \quad \text{REF-MAKE}(e) \\
\text{!e} & \quad \text{REF-GET}(e) \\
e_1 := e_2 & \quad \text{REF-SET}(e_1, e_2)
\end{align*}
\]

Write formation, introduction, and elimination rules.
Wait for it . . .
Reference Types

Formation: \[ \tau \text{ is a type} \]
\[ \text{REF}(\tau) \text{ is a type} \]

Introduction: \[ \Gamma \vdash e : \tau \]
\[ \Gamma \vdash \text{REF-MAKE}(e) : \text{REF}(\tau) \]

Elimination: \[ \Gamma \vdash e : \text{REF}(\tau) \]
\[ \Gamma \vdash \text{REF-GET}(e) : \tau \]
\[ \Gamma \vdash e_1 : \text{REF}(\tau) \quad \Gamma \vdash e_2 : \tau \]
\[ \Gamma \vdash \text{REF-SET}(e_1, e_2) : \tau \]
From rule to code

Arrow-introduction

\[
\Gamma \{ x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n \} \vdash e : \tau \quad \tau_i \text{ is a type, } 1 \leq i \leq n \\
\Gamma \vdash \text{LAMBDA} (x_1 : \tau_1, \ldots, x_n : \tau_n, e) : (\tau_1 \cdots \tau_n \rightarrow \tau)
\]
Type-checking LAMBDA

datatype exp = LAMBDA of (name * tyex) list * exp
...
fun ty (Gamma, LAMBDA (formals, body)) = 
  let val Gamma’ = (* body gets new env *)
    foldl (fn ((x, ty), g) => bind (x, ty, g))
      Gamma formals
  val bodytype = ty (Gamma’, body)
  val formaltypes =
    map (fn (x, ty) => ty) formals
  in  FUNTY (formaltypes, bodytype)
  end