Type soundness

If

- $\Gamma \vdash e : \tau$
- $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$
- Γ, ρ, **and** σ **are consistent,**

then

τ **predicts** v

Consistency: $\mathrm{dom}\,\Gamma = \mathrm{dom}\,\rho$, **and**
$\forall x \in \mathrm{dom}\,\Gamma : \Gamma(x)$ **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

Type	Produce Introduce	Consume Eliminate
`struct`	(definition form only)	dot notation e`.next`, e`->next`
pointer	`&`	`*`
function	(definition form only)	application

Types and their μScheme constructs

Type	Produce	Consume
	Introduce	Eliminate
record	constructor	accessor functions
	function	type predicate
function	`lambda`	application

Types and their ML constructs

Type	Produce Introduce	Consume Eliminate
arrow	Lambda (`fn`)	Application
constructed (algebraic)	Apply constructor	Pattern match
constructed (tuple)	$(e_1, \ldots, e_n)$	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

$$\frac{\tau \in \{\textsc{unit}, \textsc{int}, \textsc{bool}\}}{\tau \text{ is a type}} \qquad (\textsc{BaseTypes})$$

$$\frac{\tau \text{ is a type}}{\textsc{array}(\tau) \text{ is a type}} \qquad (\textsc{ArrayFormation})$$

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 "τ is a type"**

Type rules for variables

Look up the type of a variable:

$$\frac{x \in \mathrm{dom}\,\Gamma \qquad \Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textbf{(VAR)}$$

Types match in assignment (two τ's must be equal):

$$\frac{x \in \mathrm{dom}\ \Gamma \qquad \Gamma(x) = \tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \textsf{SET}(x, e) : \tau} \qquad \textbf{(SET)}$$

Understanding the SET rule

Types match in assignment (two τ's must be equal):

$$\frac{x \in \mathrm{dom}\ \Gamma \qquad \Gamma(x) = \tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{SET}(x, e) : \tau} \quad \textbf{(SET)}$$

Understanding the SET rule

Types match in assignment (two τ's must be equal):

$$\frac{x \in \mathrm{dom}\ \Gamma \qquad \Gamma(x) = \boxed{\tau} \qquad \Gamma \vdash e : \boxed{\tau}}{\Gamma \vdash \mathsf{SET}(x, e) : \boxed{\tau}} \quad \textbf{(SET)}$$

$$\frac{x \in \mathrm{dom}\ \Gamma \qquad \Gamma(x) = \boxed{\tau_x} \qquad \Gamma \vdash e : \boxed{\tau_e} \qquad \tau_x \equiv \tau_e}{\Gamma \vdash \mathsf{SET}(x, e) : \boxed{\tau_e}} \quad \textbf{(SET)}$$

Type rules for control

Boolean condition; matching branches

$$\frac{\Gamma \vdash e_1 : \textsf{BOOL} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textsf{IF}(e_1, e_2, e_3) : \tau} \quad \textsf{(IF)}$$

Product types: Both x and y

New abstract syntax: PAIR, FST, SND

$$\frac{\tau_1 \text{ and } \tau_2 \text{ are types}}{\tau_1 \times \tau_2 \text{ is a type}} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{PAIR}(e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{FST}(e) : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{SND}(e) : \tau_2}$$

Pair rules generalize to product types with many elements ("tuples," "structs," and "records")

Arrow types: Function from x to y

Syntax: `lambda`, **application**

Typed μScheme style:

$$\frac{\tau_1, \ldots, \tau_n \text{ and } \tau \text{ are types}}{(\tau_1 \cdots \tau_n \rightarrow \tau) \text{ is a type}} \quad \textbf{(ARROWFORMATION)}$$

ML style: each function takes a tuple:

$$\frac{\tau_1, \ldots, \tau_n \text{ and } \tau \text{ are types}}{\tau_1 \times \cdots \times \tau_n \rightarrow \tau \text{ is a type}} \quad \textbf{(MLARROWFORMATION)}$$

Arrow types: Function from x to y

Eliminate with application:

$$\frac{\Gamma \vdash e : (\tau_1 \cdots \tau_n \rightarrow \tau) \qquad \Gamma \vdash e_i : \tau_i, \quad 1 \leq i \leq n}{\Gamma \vdash \textsf{APPLY}(e, e_1, \ldots, e_n) : \tau}$$

Introduce with `lambda`:

$$\frac{\Gamma\{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau}{\Gamma \vdash \textsf{LAMBDA}(x_1 : \tau_1, \ldots, x_n : \tau_n, e) : (\tau_1 \cdots \tau_n \rightarrow \tau)}$$

Typical syntactic support for types

Explicit types on `lambda` and `define`:

- **For `lambda`, argument types:**

```
(lambda ([n : int] [m : int]) (+ (* n n) (* m m)))
```

- **For `define`, argument *and result* types:**

```
(define int max ([x : int] [y : int])
                (if (< x y) y x))
```

Abstract syntax:

```
datatype exp = ...
 | LAMBDA of (name * tyex) list * exp
    ...
datatype def = ...
 | DEFINE of name * tyex * ((name * tyex) list * exp)
    ...
```

Array types: Array of x

Formation:

$$\frac{\tau \text{ is a type}}{\textsf{ARRAY}(\tau) \text{ is a type}}$$

Introduction:

$$\frac{\Gamma \vdash e_1 : \textsf{INT} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \textsf{AMAKE}(e_1, e_2) : \textsf{ARRAY}(\tau)}$$

Array types continued

Elimination:

$$\frac{\Gamma \vdash e_1 : \textsf{ARRAY}(\tau) \qquad \Gamma \vdash e_2 : \textsf{INT}}{\Gamma \vdash \textsf{AAT}(e_1, e_2) : \tau}$$

$$\frac{\Gamma \vdash e_1 : \textsf{ARRAY}(\tau) \qquad \Gamma \vdash e_2 : \textsf{INT} \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textsf{APUT}(e_1, e_2, e_3) : \tau}$$

$$\frac{\Gamma \vdash e : \textsf{ARRAY}(\tau)}{\Gamma \vdash \textsf{ASIZE}(e) : \textsf{INT}}$$

References (similar to C/C++ pointers)

Your turn! Given

`ref` τ	$\texttt{REF}(\tau)$
`ref e`	$\texttt{REF-MAKE}(e)$
`!e`	$\texttt{REF-GET}(e)$
`e1 := e2`	$\texttt{REF-SET}(e1, e2)$

Write formation, introduction, and elimination rules.

Wait for it . . .

Reference Types

Formation:

$$\frac{\tau \text{ is a type}}{\textsf{REF}(\tau) \text{ is a type}}$$

Introduction:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textsf{REF-MAKE}(e) : \textsf{REF}(\tau)}$$

Elimination:

$$\frac{\Gamma \vdash e : \textsf{REF}(\tau)}{\Gamma \vdash \textsf{REF-GET}(e) : \tau}$$

$$\frac{\Gamma \vdash e_1 : \textsf{REF}(\tau) \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \textsf{REF-SET}(e_1, e_2) : \tau}$$

From rule to code

Arrow-introduction

$$\frac{\Gamma\{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau \qquad \tau_i \text{ is a type}, 1 \leq i \leq n}{\Gamma \vdash \texttt{LAMBDA}(x_1 : \tau_1, \ldots, x_n : \tau_n, e) : (\tau_1 \cdots \tau_n \rightarrow \tau)}$$

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
```