CSC 520, Spring 2020

# Principles of Programming Languages

*Michelle Strout*

THE UNIVERSITY OF ARIZONA

Version: 1

April 1, 2020

# Plan

- **Announcements**
  - HW7 is due ~~Friday~~ Wednesday April 8th
  - Will have a zoom waiting room in office hours

- **Last time**
  - Type Systems
  - A type system for two types: typing rules and how to implement them

- **Today**
  - What is type soundness?
  - Formation, Introduction, and Elimination Rules
  - Type checking with type constructors

# 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:** $\operatorname{dom} \Gamma = \operatorname{dom} \rho$, **and** $\forall x \in \operatorname{dom} \Gamma : \Gamma(x)$ **predicts** $\sigma(\rho(x))$.

**Sample predictions:** `int` **predicts** `7`, `bool` **predicts** `#t`

# Programmers understanding language design

- Questions about types never seen before (aka new types)
  - What types can I make?
  - What syntax goes with 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<br>Introduce | Consume<br>Eliminate |
| --- | --- | --- |
| `struct` | (definition form only) | dot notation<br>*e* `.next`, *e* `->next` |
| pointer | `&` | `*` |
| function | (definition form only) | application |

# Types and their μScheme constructs

| Type | Produce<br>Introduce | Consume<br>Eliminate |
| --- | --- | --- |
| record | constructor<br>function | accessor functions<br>type predicate |
| function | `lambda` | application |

# Types and their ML constructs

THE UNIVERSITY OF ARIZONA
Computer Science

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

# Functions

- Here is an example of how to determine types for "introducing" a value and "eliminating" a value

- Introduction

```
Gamma{x->tau1} |- e : tau2
------------------------------------------
Gamma |- fn x : tau1 => e : tau1 -> tau2
```

- Elimination

```
Gamma |- e : tau1 -> tau2     Gamma |- e1 : tau1
-------------------------------------------------
Gamma |- e e1 : tau2
```

# Where we've been and where we're going



- **New watershed in the homework**
  - You've been developing and polishing programming skills: recursion, higher-order functions, using types to your advantage
  - Now shifting to doing real programming-languages stuff like type systems
  - You've seen everything needed to implement a basic type checker and now want to learn how type constructors
  - What's next? More sophisticated type systems with an infinite number of types
- **Questions to consider about monomorphic and polymorphic type systems**
  - What is and is not a good type (classifier for terms)?
  - How shall we represent types?

# Monomorphic vs. Polymorphic Types

- **Monomorphic types have no type parameters**
  - `int`
  - `bool`
  - `int -> bool`
  - `int * int`

- **Polymorphic Types have type parameters**
  - `'a list`
  - `'a list -> 'a list`
  - `('a * int)`

# Design and Implementation of Monomorphic Languages

- **Mechanisms**
  - Every new type require special syntax (eg. structs, pointers arrays)
  - Implementation is a straightforward application of what you already know

- **Language designer's process when adding new kinds of types**
  - What new types do I have (formation rules)?
  - What new syntax to create new values with that type (introduction rules)?
  - What new syntax do I have to observe terms of a type (elimination rules)?

- **Q: What if I add lists to a language? How many types?**

# Type formation rules for type expressions



- **Types that classify terms**
  - `int`
  - `bool`
  - `int -> bool`
  - `int * int`
- **Type constructors, don't classify by self**
  - `list (but "int list" is a type)`
  - `array (but "char array" is a type)`
  - `records/structs`
- **Nonsense types, don't mean anything**
  - `int int`
  - `bool*array`

# What's a good type?

The University of Arizona
Computer Science

## Type formation rules for Typed Impcore

$$\frac{\tau \in \{\mathsf{UNIT}, \mathsf{INT}, \mathsf{BOOL}\}}{\tau \text{ is a type}} \quad (\textsc{BaseTypes})$$

$$\frac{\tau \text{ is a type}}{\mathsf{ARRAY}(\tau) \text{ is a type}} \quad (\textsc{ArrayFormation})$$

# Type rules for variables

**Lookup the type of a variable:**

$$\frac{x \in \operatorname{dom}\Gamma \qquad \Gamma(x) = \tau}{\Gamma \vdash x : \tau} \quad (\textsc{Var})$$

**Types match in assignment:**

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

# Type rules for control

## Boolean condition; matching branches

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

# 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

THE UNIVERSITY OF ARIZONA
Computer Science

**Syntax:** `lambda`, **application**

**Use a tuple to represent a multi-argument function:**

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

Arrow types: Function from x to y

**Eliminate with application:**

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

**Introduce with** `lambda`**:**

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

# 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, Q: what is different from before?**

```
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

THE UNIVERSITY OF ARIZONA
Computer Science

**Elimination:**

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

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

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

# References (similar to C/C++ pointers)

**Given**

| | |
|---|---|
| `ref` $\tau$ | REF$(\tau)$ |
| `ref e` | REF-MAKE$(e)$ |
| `*e` | REF-GET$(e)$ |
| `e1 := e2` | REF-SET$(e1, e2)$ |

**Write formation, introduction, and elimination rules.**

# Rules for references

- **Formation**

```
tau is a type
-------------------
REF(tau) is a type
```

- **Introduction**

```
Gamma |- ???
-------------------
Gamma |- REF-MAKE(e) : REF(tau)
```

- **Elimination**

```
Gamma |- e : REF(tau)
----------------------
Gamma |- REF-GET(???) : ???
```

```
Gamma |- e1 : REF(tau)      Gamma |- e2 : ???
----------------------------------------------
Gamma |- REF-SET(???) : tau
```

# From rule to code

THE UNIVERSITY OF ARIZONA
Computer Science

## Arrow-introduction

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

```
(* 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 funtype (formaltypes, bodytype)
  end
```