# Type Systems

Michael Hicks
University of Maryland

# What is a type?

- Types classify sets of expressions
  - The set of values an expression could evaluate to
  - We use metavariable ***t*** to designate a type
    - Examples include **`int`**, **`bool`**, **`string`**, and more.
- Expression ***e*** has type ***t*** if ***e*** will (always) evaluate to a value of type ***t***
  - { …, **`-1`**, **`0`**, **`1`**, … } are values of type **`int`**
  - **`34+17`** is an expression of type **`int`**, since it evaluates to **`51`**, which has type **`int`**

# Type Systems

- A type system is a series of rules that prove judgment $\vdash$ `e` : `t`, which says that `e` has type `t`
  - Rules of inference, as with operational semantics
- The process of applying these rules is called type checking
  - Or simply, typing
  - Type checking *aka* the program's static semantics
- Different languages have different type systems

# Type System for Micro-OCaml

```
Exprs   e ::= x | v | e + e | let x = e in e
          | eq0 e | if e then e else e
Values  v ::= n | true | false
Types   t ::= bool | int
```

# Typing Integers

- Integer constants have type **`int`**

$$\frac{}{\vdash n : \texttt{int}}$$

- Arithmetic expressions have type **`int`** too

$$\frac{\vdash e1 : \texttt{int} \qquad \vdash e2 : \texttt{int}}{\vdash e1 + e2 : \texttt{int}}$$

# Typing Booleans

- Boolean constants have type **`bool`**

$$\frac{}{\vdash \texttt{true} : \texttt{bool}} \qquad \frac{}{\vdash \texttt{false} : \texttt{bool}}$$

- Equality checking has type **`bool`** too
  - Assuming its target expression has type **`int`**

$$\frac{\vdash e : \texttt{int}}{\vdash \texttt{eq0}\ e : \texttt{bool}}$$

- Conditional: **`bool`** guard, same-typed branches

$$\frac{\vdash e1 : \texttt{bool} \quad \vdash e2 : t \quad \vdash e3 : t}{\vdash \texttt{if}\ e1\ \texttt{then}\ e2\ \texttt{else}\ e3 : t}$$

# Quiz 1

What is the type of the following program?

```
if eq0 1 then 1 else false
```

A. **`int`**

B. **`bool`**

C. *type error*

# Quiz 1

What is the type of the following program?

```
if eq0 1 then 1 else false
```

A. **`int`**

B. **`bool`**

C. *type error – both branches must have the same type*

# Variables and Binding

- What about the types of variables?
  - Taking inspiration from the environment-style operational semantics, what could you do?
- Change judgment to be G ⊢ **`e`** : **`t`** which says **`e`** *has type* **`t`** *under* type environment G
  - G is a map from variables **`x`** to types **`t`**
    - Analogous to map A, but maps vars to types, not values
- What would be the rules for **`let`**, and variables?

# Type Checking Variables and Binding

- Variable lookup

$$\frac{G(x) = t}{G \vdash x : t}$$

*analogous to*

$$\frac{A(x) = v}{A;\ x \Rightarrow v}$$

- Let binding

$$\frac{G \vdash e1 : t1 \qquad G, x:t1 \vdash e2 : t2}{G \vdash \texttt{let } x = e1 \texttt{ in } e2 : t2}$$

Extend type environment G with mapping from $x$ to $t1$

*analogous to*

$$\frac{A;\ e1 \Rightarrow v1 \qquad A, x:v1;\ e2 \Rightarrow v2}{A;\ \texttt{let } x = e1 \texttt{ in } e2 \Rightarrow v2}$$

# Quiz 2

What is a derivation of the following judgment?

- $\bullet \vdash \texttt{let x=3 in x+2 : int}$

(a)

$$\frac{\texttt{x:int 2:int}}{\texttt{3:int}\quad\texttt{x+2 : int}}{\texttt{let x=3 in x+2 : int}}$$

(c)

$$\frac{\texttt{x:int} \vdash \texttt{x:int}\quad\texttt{x:int} \vdash \texttt{2:int}}{\bullet \vdash \texttt{let x=3 in x+2 : int}}$$

(b)

$$\frac{\texttt{x:int} \vdash \texttt{x:int}\quad\texttt{x:int} \vdash \texttt{2 :int}}{\bullet \vdash \texttt{3:int}\quad\texttt{x:int} \vdash \texttt{x+2 : int}}{\bullet \vdash \texttt{let x=3 in x+2 : int}}$$

# Quiz 2

What is a derivation of the following judgment?

- $\bullet \vdash$ **let x=3 in x+2 : int**

(a)

$$\frac{\mathbf{x:int}\ \mathbf{2:int}}{\mathbf{3:int} \qquad \mathbf{x+2 : int}}$$
$$\mathbf{let\ x=3\ in\ x+2 : int}$$

(c)

$$\frac{\mathbf{x:int} \vdash \mathbf{x:int} \qquad \mathbf{x:int} \vdash \mathbf{2:int}}{\bullet \vdash \mathbf{let\ x=3\ in\ x+2 : int}}$$

**(b)**

$$\frac{\mathbf{x:int} \vdash \mathbf{x:int} \qquad \mathbf{x:int} \vdash \mathbf{2:int}}{\bullet \vdash \mathbf{3:int} \qquad \mathbf{x:int} \vdash \mathbf{x+2 : int}}$$
$$\bullet \vdash \mathbf{let\ x=3\ in\ x+2 : int}$$

# Implementing a Type Checker

- Just as it was easy to turn semantics rules into an interpreter, we can turn type rules into a type checker

```
let rec typeof env e =
  match e with
    Ident x -> lookup env x
  | Val (Int _) -> TInt
  | Val (Bool _) -> TBool
  | Plus (e1,e2) ->
     let TInt = typeof env e1 in
     let TInt = typeof env e2 in
     TInt
  | Let (x,e1,e2) ->
     let t1 = eval env e1 in
     let env' = extend env x t1 in
     let t2 = type env' e2 in t2
  |…
```

# Summary: Type Rules

$$\frac{}{G \vdash \texttt{true} : \texttt{bool}}$$

$$\frac{}{G \vdash \texttt{false} : \texttt{bool}}$$

$$\frac{G \vdash e : \texttt{int}}{G \vdash \texttt{eq0}\ e : \texttt{bool}}$$

$$\frac{G \vdash e1 : \texttt{bool} \quad G \vdash e2 : t \quad G \vdash e3 : t}{G \vdash \texttt{if}\ e1\ \texttt{then}\ e2\ \texttt{else}\ e3 : t}$$

# Summary: Type Rules

$$\frac{}{G \vdash n : \texttt{int}}$$

$$\frac{G \vdash e1 : \texttt{int} \qquad G \vdash e2 : \texttt{int}}{G \vdash e1 \texttt{ + } e2 : \texttt{int}}$$

$$\frac{G \vdash e1 : t1 \qquad G, x{:}t1 \vdash e2 : t2}{G \vdash \texttt{let } x \texttt{ = } e1 \texttt{ in } e2 : t2}$$

$$\frac{G(x) = t}{G \vdash x : t}$$

# Type Safety (aka Soundness)

- Type safe = “Well-typed programs never go wrong”
  - Robin Milner, 1978
- Program A;**e** is well typed *iff*
  - Exists G,**t** such that G ⊢ A and G ⊢ **e** : **t**
- Program A;**e** goes wrong (is *not* well defined) iff
  - There exists no **v** such that A;**e** ⇒ **v**
- All together: For all A,G,**e**,**t**, if G ⊢ A and G ⊢ **e** : **t** then there exists **v** such that A;**e** ⇒ **v** and G ⊢ **v** : **t**

# Proving Type Safety

For all A,G,**e**, **t**, if G ⊢ A and G ⊢ **e** : **t** then there exists **v** such that A;**e** ⇒ **v** and G ⊢ **v** : **t**

- Proof by induction on the **e**
- Case **v**:
  - A;**v** ⇒ **v** by the constant evaluation rule
  - Well-typed by assumption
- Case **x**:
  - G ⊢ **x** : **t** implies G(**x**) = **t** by the variables rule
  - G ⊢ A implies A(**x**) = **v** such that G ⊢ **v** : **t**
  - A;**x** ⇒ **v** by the variable evaluation rule

# Proving Type Safety

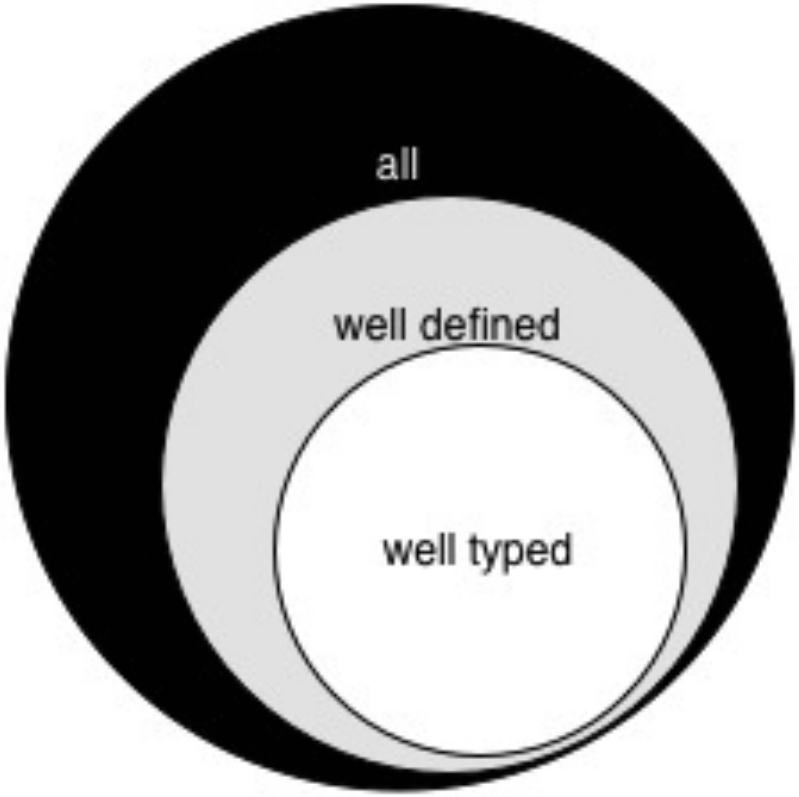
For all A,G,**e**, **t**, if G ⊢ A and G ⊢ **e** : **t** then there exists **v** such that A; **e** ⇒ **v** and G ⊢ **v** : **t**

- Case **let x = e1 in e2**:
  - G ⊢ **let x = e1 in e2** : **t** implies (by let rule)
    1. G ⊢ **e1** : **t1**
    2. G,**x**:**t1** ⊢ **e2** : **t**
  - G ⊢ A and (1) implies, by induction, that A;**e1** ⇒ **v** and G ⊢ **v** : **t**
  - G,**x**:**t1** ⊢ A,**x**:**v** by the previous, which along with (2) implies, by induction, that A,**x**:**v**;**e2** ⇒ **v2** and G,**x**:**t1** ⊢ **v2** : **t**
  - Hence, by the evaluation rule for let, we have A,**let x = e1 in e2** ⇒ **v2**
    - And G ⊢ **v2** : **t** since G ⊢ **v** : **t** implies ⊢ **v** : **t** for all **v**, **t**

# Type Safety, Small-step Semantics

- Safety: Each evaluation step is well typed
  - For all A,G,**e**,**t**, if G ⊢ A and G ⊢ **e** : **t** then exists **e′**,A',G' s.t. A;**e** → A';**e′** and G' ⊢ A' and G' ⊢ **e′** : **t**
- Progress: A well-typed expression that is not already a value can take a step
  - For all A,G,**e**,**t**, if G ⊢ A and G ⊢ **e** : **t** then either **e** is a value **v** or exists **e′**, A', s.t. A;**e** → A';**e′**
- Preservation: A well-typed expression that takes a step is also well-typed
  - For all A,G,A', **e**,**e′**,**t**, if G ⊢ A and G ⊢ **e** : **t** and A;**e** → A';**e′** then exists G' s.t. G' ⊢ A' and G' ⊢ **e′** : **t**

# Type Safety is Often Conservative




I.e., some well-defined programs are *not* well typed

A precise type system rules out few well-defined programs

http://www.pl-enthusiast.net/2014/08/05/type-safety/

# Well-defined but not Well-typed

- The expression **`4+false`** is undefined
  - Micro-OCaml's type system does not typecheck this expression, ensuring it is never executed
    - Good!

- But the following expressions are well-defined, but still rejected
  - **`if true then 0 else 4+false`**
    - Always evaluates to 0
  - **`let x = e in if x ≤ abs x then 0 else 4+false`**
    - Evaluates to `0` for all (**`e : int`**)

# Static vs. Dynamic Type Systems

- (Micro-)OCaml, Java, etc. are statically typed
  - Expressions are given one of various different types at compile time, e.g., `int`, `float`, `bool`, etc.
    - Or else they are rejected
- Ruby, Python, etc. are dynamically typed
  - Can view all expressions as having a single type `Dyn`
    - The language is uni-typed
  - *All* operations are permitted on values of this type
    - E.g., in Ruby, all objects accept any method call
  - But: Some operations result in a run-time exception
    - Nevertheless, such behavior is well defined

# Dynamic Type Checking

- The run-time checks performed by dynamic languages often called dynamic type checking
- The type is checked when needed, at run-time
  - Values keep tag, set when the value is created, indicating its type (e.g., what class it has)
- Disallowed operations cause an exception
  - Type errors may be latent in code for a long time

# Soundness and Completeness

- Type safety is a soundness property
  - That a term type checks implies its execution will be well-defined
- Static type systems are rarely complete
  - That a term is well-defined *does not* imply that it will type check
    - `if true then 0 else 4+"hi"`
- Dynamic type systems are often complete
  - *All* expressions are well defined, and all type check
  - `4+"hi"` well-defined: it gives a run-time exception

# Devil's Bargain?

- Dynamic typing is sound and complete
  - That seems good …
- But it trades compile-time errors for (well-defined) run-time exceptions!
- Can't we build a better static type system?
  - I.e., that that aims to eliminate all language-level run-time errors and is also complete?
- Yes, we can build fancier, more precise static type systems, but never a perfect one
  - To do so would be undecidable!

# Perfect Type System? Impossible

- No type system can do all of following
  - (1) always terminate, (2) be sound, (3) be complete
  - While trying to eliminate all run-time exceptions, e.g.,
    - Using an int as a function
    - Accessing an array out of bounds
    - Dividing by zero, …
- Doing so would be undecidable
  - by reduction to the halting problem
  - Eg., `while (…) {…} arr[-1] = 1;`
    - *Error tantamount to proving that the while loop terminates*

# Type Safe?

- Java, Haskell, OCaml: **Yes** (arguably).
  - The languages' type systems restrict programs to those that are defined
    - Caveats: Foreign function interfaces to type-unsafe C, bugs in the language design, bugs in the implementation, etc.

- C, C++: **No**.
  - The languages' type systems do not prevent undefined behavior
    - Unsafe casts (int to pointer), out-of-bounds array accesses, dangling pointer dereferences, etc.

# What's Bad about Being Undefined?

- Well, undefined behavior is unconstrained
  - Depends on the compiler/interpreter's treatment
- Undefined behavior in C/C++ is traditionally a source of severe security vulnerabilities
  - These are bugs that have security consequences
- Stack smashing exploits out-of-bounds array accesses to inject code into a running program
  - Write outside the bounds of an array (undefined!)
  - thereby corrupting the return address
  - to point to code the attacker provides
  - to gain control of the attacked machine

# Fancy Types

- Lots of ideas over the last few decades aimed at improving the precision of type systems
  - So they can rule out more run-time errors
- Generic types (parametric polymorphism)
  - for containers and generic operations on them
- Subtyping
  - for interchanging objects with related shapes
- Dependent types can include *data in types*
  - Instead of **`int list`**, we could have **`int` *`n`* `list`** for a list of ***`n`*** elements. Hence **`hd`** has type **`int` *`n`* `list`** where ***`n`***>0.

# Type Systems with Fancy Types

- OCaml's type system has types for
  - generics (polymorphism), objects, curried functions, …
  - all unsupported by C
- Haskell's type system has types for
  - Type classes (qualified types), effect-isolating monads, higher-rank polymorphism, …
  - All unsupported by OCaml
- More precision ensures more run-time errors prevented, with less contorted programs: Good!
  - But now the programmer must understand (and sometimes do) more ..

# Static vs. Dynamic: Age-old Debate

- Static vs. dynamic typing is too coarse a question
  - Better question: *What* should we enforce statically?
    - E.g., OCaml checks array bounds, division-by-zero, at run-time
  - Legitimate trade-offs

- Idea: Flexible languages allowing best-of-both-worlds?
  - Use static types in some parts of the program, but dynamic checking in other parts?
    - Called gradual typing: an idea still under active research
  - Would programmers use such flexibility well? Who decides?

# Type Inference

- Type inference is a part of (static) type checking
  - Reduces the programmer's effort
- Static types are explicit (*aka* manifest) or inferred
  - Manifest – specified in text (at variable declaration)
    - C, C++, Java, C#
  - Inferred – compiler determines type based on usage
    - OCaml, C# and Go (limited)
- Fancier type systems may require explicit types
  - Haskell considers adding a type signature your function to be good style, even when not required

# Type Safety and other properties

- Can prove interesting properties about programs in your language by
  - Augmenting the operational semantics so that violations of the property yield an undefined program
    - May add “ghost data” to characterize property violation
  - Defining a type system that is type safe
    - Thus, the property is always satisfied
    - And no need for ghost data in real implementation
  - Example: Tainting
- Or: Use typing as foundation of stronger property
  - Example: Noninterference

# Further Reading

- Practical Foundations of Programming Languages (Harper), Chap 6 and beyond
  - https://www.cs.cmu.edu/~rwh/pfpl/2nded.pdf
- Types and Programming Languages (Pierce), draft, Chap 3 (and the rest of the book)
  - http://ropas.snu.ac.kr/~kwang/520/pierce_book.pdf
- What is type safety (blog post)?
  - http://www.pl-enthusiast.net/2014/08/05/type-safety/