Formal Semantics and Types

Adapted from CMSC 330 @UMD

Special Thanks to Mike Hicks

Formal Semantics of a Prog. Lang.

- Mathematical description of the meaning of programs written in that language
  - What a program computes, and what it does

- Three main approaches to formal semantics
  - Denotational
  - Operational
  - Axiomatic

Styles of Semantics

- Denotational semantics: translate programs into math!
  - Usually: convert programs into functions mapping inputs to outputs
  - Analogous to compilation
- Operational semantics: define how programs execute
  - Often on an abstract machine (mathematical model of computer)
  - Analogous to interpretation
- Axiomatic semantics
  - Describe programs as predicate transformers, i.e. for converting initial assumptions into guaranteed properties after execution
    - Preconditions: assumed properties of initial states
    - Postcondition: guaranteed properties of final states
  - Logical rules describe how to systematically build up these transformers from programs

This Course: Operational Semantics

- We will show how an operational semantics may be defined for a simple language

- Approach: use rules to define a judgment

$$e \Rightarrow v$$

  - Says "e *evaluates to* v"
  - e: expression in the language
  - v: value that results from evaluating e

Expression Grammar

$$e ::= x \mid n \mid e + e \mid \texttt{let } x \texttt{ = } e \texttt{ in } e$$

- ***e***, ***x***, ***n*** are *meta-variables* that stand for categories of syntax
  - ***x*** is any identifier (like **`z`**, **`y`**, **`foo`**)
  - ***n*** is any numeral (like **`1`**, **`0`**, **`10`**, **`-25`**)
  - ***e*** is any expression (here defined, recursively!)

- *Concrete syntax* of actual expressions in **black**
  - Such as **`let`**, **`+`**, **`z`**, **`foo`**, **`in`**, …

- ::= and | are *meta-syntax* used to define the syntax of a language (part of "Backus-Naur form," or BNF)

Expression Grammar

$$e ::= x \mid n \mid e + e \mid \texttt{let } x \texttt{ = } e \texttt{ in } e$$

- Examples
  - **`1`** is a numeral *n* which is an expression *e*
  - **`1+z`** is an expression *e* because
    - **`1`** is an expression *e*,
    - **`z`** is an identifier *x*, which is an expression *e*, and
    - *e* + *e* is an expression *e*
  - **`let z = 1 in 1+z`** is an expression *e* because
    - **`z`** is an identifier *x*,
    - **`1`** is an expression *e* ,
    - **`1+z`** is an expression *e*, and
    - **`let`** *x* **`=`** *e* **`in`** *e* is an expression *e*

Values

- An expression's final result is a value. What can values be?

$$v ::= n$$

- Just numerals for now
  - In terms of an interpreter's representation:

    ```
    type value = int
    ```

  - In a full language, values v will also include booleans (**`true`**, **`false`**), strings, functions, …

Defining the Semantics

- Use rules to define judgment $\boldsymbol{e} \Rightarrow \boldsymbol{v}$
- These rules will allow us to show things like
  - **`1+3`** $\Rightarrow$ **`4`**
    - **`1+3`** is an expression $\boldsymbol{e}$, and **`4`** is a value $\boldsymbol{v}$
    - This judgment claims that **`1+3`** evaluates to **`4`**
    - We use rules to prove it to be true
  - **`let foo=1+2 in foo+5`** $\Rightarrow$ **`8`**
  - **`let f=1+2 in let z=1 in f+z`** $\Rightarrow$ **`4`**

Rules as English Text

No rule for ***x***

- Suppose ***e*** is a numeral ***n***
  - Then ***e*** evaluates to itself, i.e., ***n*** ⇒ ***n***
- Suppose ***e*** is an addition expression ***e1*** **+** ***e2***
  - If ***e1*** evaluates to ***n1***, i.e., ***e1*** ⇒ ***n1***
  - If ***e2*** evaluates to ***n2***, i.e., ***e2*** ⇒ ***n2***
  - Then ***e*** evaluates to ***n3***, where ***n3*** is the sum of ***n1*** and ***n2***
  - I.e., ***e1*** **+** ***e2*** ⇒ ***n3***
- Suppose ***e*** is a let expression `let` ***x*** `=` ***e1*** `in` ***e2***
  - If ***e1*** evaluates to ***v***, i.e., ***e1*** ⇒ ***v1***
  - If ***e2{v1/x}*** evaluates to ***v2***, i.e., ***e2{v1/x}*** ⇒ ***v2***
    - Here, ***e2{v1/x}*** means "the expression after substituting occurrences of ***x*** in ***e2*** with ***v1***"
  - Then ***e*** evaluates to ***v2***, i.e., `let` ***x*** `=` ***e1*** `in` ***e2*** ⇒ ***v2***

Rules of Inference

- We can use a more compact notation for the rules we just presented: rules of inference
  - Has the following format

$$\frac{H_1 \quad \ldots \quad H_n}{C}$$

  - Says: if the conditions $H_1 \quad \ldots \quad H_n$ ("hypotheses") are true, then the condition C ("conclusion") is true
  - If n=0 (no hypotheses) then the conclusion automatically holds; this is called an axiom
- We will use inference rules to speak about evaluation

Rules of Inference: Num and Sum

- Suppose ***e*** is a numeral ***n***
  - Then ***e*** evaluates to itself, i.e., ***n*** ⇒ ***n***

$$\frac{}{n \Rightarrow n}$$

- Suppose ***e*** is an addition expression ***e1*** + ***e2***
  - If ***e1*** evaluates to ***n1***, i.e., ***e1*** ⇒ ***n1***
  - If ***e2*** evaluates to ***n2***, i.e., ***e2*** ⇒ ***n2***
  - Then ***e*** evaluates to ***n3***, where ***n3*** is the sum of ***n1*** and ***n2***
  - I.e., ***e1*** + ***e2*** ⇒ ***n3***

$$\frac{e1 \Rightarrow n1 \qquad e2 \Rightarrow n2 \qquad n3 \text{ is } n1+n2}{e1 + e2 \Rightarrow n3}$$

Rules of Inference: Let

- Suppose e is a let expression **`let`** x **`=`** $e1$ **`in`** $e2$
  - If $e1$ evaluates to v, i.e., $e1 \Rightarrow v1$
  - If $e2\{v1/x\}$ evaluates to $v2$, i.e., $e2\{v1/x\} \Rightarrow v2$
  - Then e evaluates to $v2$, i.e., **`let`** x **`=`** $e1$ **`in`** $e2 \Rightarrow v2$

$$\frac{e1 \Rightarrow v1 \qquad e2\{v1/x\} \Rightarrow v2}{\texttt{let}\ x\ \texttt{=}\ e1\ \texttt{in}\ e2 \Rightarrow v2}$$

Derivations

- When we apply rules to an expression in succession, we produce a derivation
  - It's a kind of tree, rooted at the conclusion

- Produce a derivation by goal-directed search
  - Pick a rule that could prove the goal
  - Then repeatedly apply rules on the corresponding hypotheses
    - Goal: Show that **`let x = 4 in x+3`** $\Rightarrow$ **`7`**

Derivations

$$\frac{}{n \Rightarrow n}$$

$$\frac{e1 \Rightarrow n1 \qquad e2 \Rightarrow n2 \qquad n3 \text{ is } n1+n2}{e1 + e2 \Rightarrow n3}$$

$$\frac{e1 \Rightarrow v1 \qquad e2\{v1/x\} \Rightarrow v2}{\texttt{let } x \texttt{ = } e1 \texttt{ in } e2 \Rightarrow v2}$$

Goal: show that

`let x = 4 in x+3` $\Rightarrow$ `7`

$$\frac{4 \Rightarrow 4 \qquad \dfrac{4 \Rightarrow 4 \qquad 3 \Rightarrow 3 \qquad 7 \text{ is } 4+3}{4+3 \Rightarrow 7}}{\texttt{let x = 4 in x+3} \Rightarrow 7}$$

Quiz 1

What is derivation of the following judgment?

$$2 + (3 + 8) \Rightarrow 13$$

(a)

```
2 ⇒ 2      3 + 8 ⇒ 11
-------------------------
2 + (3 + 8) ⇒ 13
```

(b)

```
3 ⇒ 3     8 ⇒ 8
--------------
3 + 8 ⇒ 11              2 ⇒ 2
------------------------------
2 + (3 + 8) ⇒ 13
```

(c)

```
              8 ⇒ 8
              3 ⇒ 3
              11 is 3+8
              ----------
2 ⇒ 2       3 + 8 ⇒ 11    13 is 2+11
------------------------------------
2 + (3 + 8) ⇒ 13
```

Quiz 1

What is derivation of the following judgment?

$$2 + (3 + 8) \Rightarrow 13$$

(a)

```
2 ⇒ 2      3 + 8 ⇒ 11
-------------------------
2 + (3 + 8) ⇒ 13
```

(b)

```
3 ⇒ 3     8 ⇒ 8
--------------
3 + 8 ⇒ 11                2 ⇒ 2
------------------------------
2 + (3 + 8) ⇒ 13
```

(c)

```
          8 ⇒ 8
          3 ⇒ 3
          11 is 3+8
          ----------
2 ⇒ 2     3 + 8 ⇒ 11    13 is 2+11
------------------------------
2 + (3 + 8) ⇒ 13
```

Semantics Defines Program Meaning

- $e \Rightarrow v$ holds if and only if a *proof* can be built
  - Proofs are derivations: axioms at the top, then rules whose hypotheses have been proved to the bottom
  - No proof means $e \not\Rightarrow v$
- Proofs can be constructed bottom-up
  - In a goal-directed fashion
- Thus, function eval $e = \{v \mid e \Rightarrow v\}$
  - Determinism of semantics implies at most one element for any e
- So: Expression e *means* v

Environment-style Semantics

- The previous semantics uses substitution to handle variables
  - As we evaluate, we replace all occurrences of a variable ***x*** with values it is bound to
- An alternative semantics, closer to a real implementation, is to use an environment
  - As we evaluate, we maintain an explicit map from variables to values, and look up variables as we see them

Environments

- Mathematically, an environment is a partial function from identifiers to values
  - If A is an environment, and ***x*** is an identifier, then A(***x***) can either be …
  - … a value (intuition: the variable has been declared)
  - … or undefined (intuition: variable has not been declared)
- An environment can also be thought of as a table
  - If A is

Id	**Val**
`x`	`0`
`y`	`2`

  - then A(**`x`**) is **`0`**, A(**`y`**) is **`2`**, and A(**`z`**) is undefined

Notation, Operations on Environments

- • is the empty environment (undefined for all ids)
- x:v is the environment that maps x to v and is undefined for all other ids
- If A and A' are environments then A, A' is the environment defined as follows

$$(A, A')(x) = \begin{cases} A'(x) & \text{if } A'(x) \text{ defined} \\ A(x) & \text{if } A'(x) \text{ undefined but } A(x) \text{ defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

- So: A' *shadows* definitions in A
- For brevity, can write •, A as just A

Semantics with Environments

- The environment semantics changes the judgment

$$e \Rightarrow v$$

to be

$$A;\ e \Rightarrow v$$

where A is an environment
  - Idea: A is used to give values to the identifiers in e
  - A can be thought of as containing declarations made up to e
- Previous rules can be modified by
  - Inserting A everywhere in the judgments
  - Adding a rule to look up variables x in A
  - Modifying the rule for `let` to add x to A

Quiz 2

What is a derivation of the following judgment?

```
•; let x=3 in x+2 ⇒ 5
```

```
(a)
         x ⇒ 3    2 ⇒ 2   5 is 3+2
         --------------------
3 ⇒ 3     x+2 ⇒ 5
------------------------
let x=3 in x+2 ⇒ 5
```

```
(c)
x:2; x⇒3   x:2; 2⇒2   5 is 3+2
-------------------------------
•; let x=3 in x+2 ⇒ 5
```

```
(b)          x:3; x ⇒ 3   x:3; 2 ⇒ 2    5 is 3+2
             -----------------------------------
•;3 ⇒ 3      x:3; x+2 ⇒ 5
--------------------------
•; let x=3 in x+2 ⇒ 5
```

Quiz 2

What is a derivation of the following judgment?

```
•; let x=3 in x+2 ⇒ 5
```

```
(a)
          x ⇒ 3    2 ⇒ 2   5 is 3+2
          --------------------
3 ⇒ 3      x+2 ⇒ 5
------------------------
let x=3 in x+2 ⇒ 5
```

```
(c)
x:2; x⇒3   x:2; 2⇒2   5 is 3+2
-------------------------------
•; let x=3 in x+2 ⇒ 5
```

```
(b)           x:3; x ⇒ 3   x:3; 2 ⇒ 2    5 is 3+2
             ------------------------------
•;3 ⇒ 3       x:3; x+2 ⇒ 5
-------------------------
•; let x=3 in x+2 ⇒ 5
```

Adding Conditionals to the Language

e ::= x | v | e `+` e | `let` x `=` e `in` e
| `eq0` e | `if` e `then` e `else` e

v ::= n | `true` | `false`

Rules for Eq0 and Booleans

$$\frac{}{A;\ \texttt{true} \Rightarrow \texttt{true}}$$

$$\frac{A;\ e \Rightarrow 0}{A;\ \texttt{eq0}\ e \Rightarrow \texttt{true}}$$

$$\frac{}{A;\ \texttt{false} \Rightarrow \texttt{false}}$$

$$\frac{A;\ e \Rightarrow v \qquad v \neq 0}{A;\ \texttt{eq0}\ e \Rightarrow \texttt{false}}$$

- Booleans evaluate to themselves
  - A; **false** ⇒ **false**
- **eq0** tests for 0
  - A; **eq0 0** ⇒ **true**
  - A; **eq0 3+4** ⇒ **false**

Rules for Conditionals

$$\frac{A;\ \mathit{e1} \Rightarrow \mathtt{true} \qquad A;\ \mathit{e2} \Rightarrow \mathit{v}}{A;\ \mathtt{if}\ \mathit{e1}\ \mathtt{then}\ \mathit{e2}\ \mathtt{else}\ \mathit{e3} \Rightarrow \mathit{v}}$$

$$\frac{A;\ \mathit{e1} \Rightarrow \mathtt{false} \qquad A;\ \mathit{e3} \Rightarrow \mathit{v}}{A;\ \mathtt{if}\ \mathit{e1}\ \mathtt{then}\ \mathit{e2}\ \mathtt{else}\ \mathit{e3} \Rightarrow \mathit{v}}$$

- Notice that only one branch is evaluated
  - A; **`if eq0 0 then 3 else 4`** ⇒ **`3`**
  - A; **`if eq0 1 then 3 else 4`** ⇒ **`4`**

Quiz 3

What is the derivation of the following judgment?

```
•; if eq0 3-2 then 5 else 10 ⇒ 10
```

```
(a)
•; 3 ⇒ 3    •; 2 ⇒ 2   3-2 is 1
----------------------------
•; eq0 3-2  ⇒  false         •; 10 ⇒ 10
----------------------------------------
•; if eq0 3-2 then 5 else 10  ⇒  10
```

```
(b)
3 ⇒ 3   2 ⇒ 2
3-2 is 1
---------------
eq0 3-2 ⇒ false          10 ⇒ 10
-------------------------------
if eq0 3-2 then 5 else 10 ⇒ 10
```

```
(c)
•; 3 ⇒ 3
•; 2 ⇒ 2
3-2 is 1
----------
•; 3-2 ⇒ 1     1 ≠ 0
-------------------
•; eq0 3-2 ⇒ false      •; 10 ⇒ 10
-----------------------------------
•; if eq0 3-2 then 5 else 10 ⇒ 10
```

Quiz 3

What is the derivation of the following judgment?

$\bullet$; `if eq0 3-2 then 5 else 10` $\Rightarrow$ `10`

(a)

```
•; 3 ⇒ 3    •; 2 ⇒ 2   3-2 is 1
--------------------------
•; eq0 3-2  ⇒  false         •; 10 ⇒ 10
-------------------------------------
•; if eq0 3-2 then 5 else 10  ⇒  10
```

(b)

```
3 ⇒ 3   2 ⇒ 2
3-2 is 1
----------------
eq0 3-2 ⇒ false          10 ⇒ 10
-------------------------------
if eq0 3-2 then 5 else 10 ⇒ 10
```

(c)

```
•; 3 ⇒ 3
•; 2 ⇒ 2
3-2 is 1
----------
•; 3-2 ⇒ 1     1 ≠ 0
-------------------
•; eq0 3-2 ⇒ false      •; 10 ⇒ 10
-----------------------------------
•; if eq0 3-2 then 5 else 10 ⇒ 10
```

Quick Look: Type Checking

- Inference rules can also be used to specify a program's static semantics
  - I.e., the rules for type checking
- We won't cover this in depth in this course, but here is a flavor.

- Types t ::= **`bool`** | **`int`**
- Judgment $\vdash e : t$ says e has type t
  - We define inference rules for this judgment, just as with the operational semantics

Some Type Checking Rules

- 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`**

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

- Conditionals

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

Type Systems

- A type system is a series of rules that ascribe types to expressions
  - The rules prove statements ***`e : t`***
- 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 Safety

- Well-typed
  - A well-typed program passes the language's type system
- Going wrong
  - The language definition deems the program nonsensical
    - "Colorless green ideas sleep furiously"
    - If the program were to be run, anything could happen
    - char buf[4]; buf[4] = 'x'; // undefined!
- Type safe = "Well-typed programs never go wrong"
  - Robin Milner, 1978
  - In other words: Well-typed $\Longrightarrow$ well-defined

Type Safe?

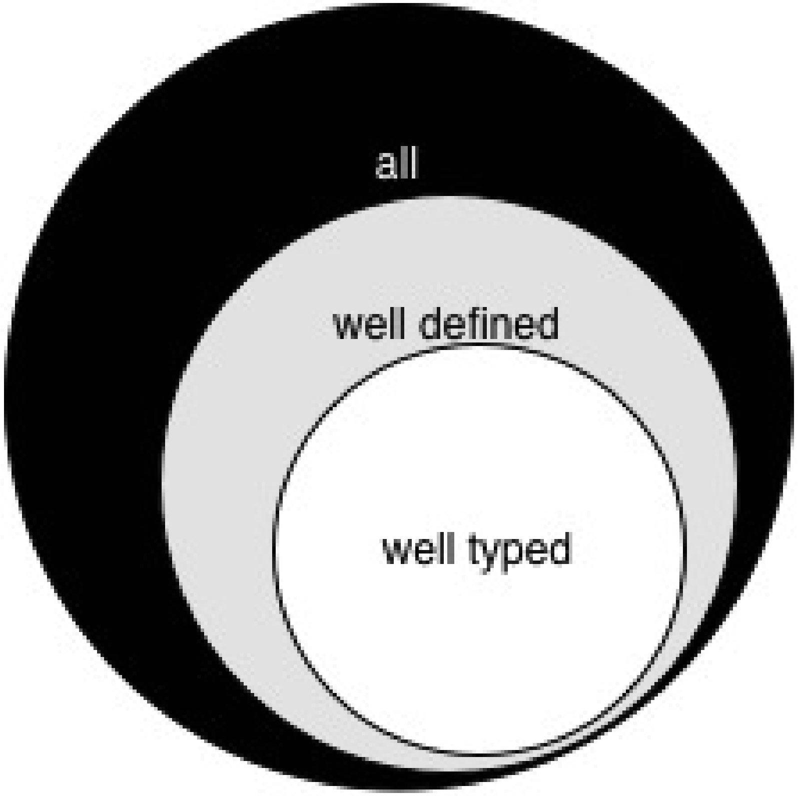
- Java, Haskell, OCaml: **Yes** (arguably).
  - The languages' type systems restrict programs to those that are defined
    - Caveats: Foreign function interfaces to type-unsafe C, bugs in the language design, bugs in the implementation, etc.

- 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

Type Safety is Often Conservative



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

Static vs. Dynamic Type Systems

- OCaml, Java, Haskell, 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 of an expression checked when needed
  - Values keep tag, set when the value is created, indicating its type (e.g., what class it has)
- Disallowed operations cause run-time exception
  - Type errors may be latent in code for a long time

Quiz 1

- When is the type of a variable determined in a dynamically typed language?

- A. When the program is compiled
- B. At run-time, when that variable is first assigned to
- C. At run-time, when the variable is last assigned to
- D. At run-time, when the variable is used

Quiz 1

- When is the type of a variable determined in a dynamically typed language?

- A. When the program is compiled
- B. At run-time, when that variable is first assigned to
- C. At run-time, when the variable is last assigned to
- D. At run-time, when the variable is used

Quiz 2

- When is the type of a variable determined in a statically typed language?

- A. When the program is compiled
- B. At run-time, when that variable is first assigned to
- C. At run-time, when the variable is last assigned to
- D. At run-time, when the variable is used

Quiz 2

- When is the type of a variable determined in a statically typed language?

- A. When the program is compiled
- B. At run-time, when that variable is first assigned to
- C. At run-time, when the variable is last assigned to
- D. At run-time, when the variable is used

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 more precise static type systems, but never a perfect one
  - To do so would be undecidable!

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 ..

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 Checking and 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