LECTURE 06

Theory and Design of PL (CS 538)

February 10, 2020

NEWS

HW1 WRAPUP

- Writing a puzzle solver
  - Manipulate lists
  - Write some recursive functions
  - Use higher-order functions
- Bigger picture
  - Decompose problem into small functions
  - Start with simple version, optimize

HW1: COMMENTS/QUESTIONS?

HW2 OUT AFTER CLASS

- Programming: purely functional data structures
- Written: types and type systems
- Due two weeks from now. Start early!

MORE PATTERN MATCHING

TAKING DATA APART

- Does two things simultaneously
  1. Does a case analysis (e.g., empty list or not?)
  2. Introduces new variables referring to parts of data
- Haskell defines *patterns*, which we can match against
  - Pattern: `42`, `'a'`, `[]`, `(x:xs)`, `(x, y)`
  - Not pattern: `i < 0`, `b == False`
- Function definitions, let-bindings, where-clauses,...

MORE EXAMPLES

- Haskell patterns are surprisingly flexible

```
foo :: (Bool, (Int, String)) -> String
foo (b, (i, c)) = ... b ... i ... c ...

-- SAME AS:
-- foo p = ... (fst p) ... (fst . snd $ p) ... (snd . snd $ p)

bar :: (Int, String) -> String
bar (1, str) = str ++ " one!"
bar (2, str) = str ++ " two!"
bar (_, str) = str ++ " something!"

-- BUT NOT:
-- baz :: Int -> String
-- baz (i < 0) = ...
```

CASE EXPRESSIONS

- Pattern match in other places with case expression

```haskell
-- General form
case expression of pattern1 -> expression1'
                   pattern2 -> expression2'
                   pattern3 -> expression3'
                   ...
-- Can put cases on following line, but must align starts

-- Example
foo :: [a] -> String
foo list = "Got an: " ++ (case list of
                              []     -> "empty"
                              (x:xs) -> "nonempty") ++ " list!"
```

ASIDE: INDENTATION

> *Code that is part of some expression should be indented further in than the beginning of that expression, even if the expression is not the first element of line.*

- Grouped expressions must be aligned exactly
- Let-bindings, where-clauses, case, guards, …
- Can ignore indentation if using `;` and `{ ... }`
- See more examples here

SPECIFYING WELL-BEHAVED PROGRAMS

WHAT DO WE WANT?

- A condition that can be checked *statically*
  - Verify correctness without running program
- Rule out classes of buggy programs
  - Prevent as many bugs as possible
- Condition should be *compositional*
  - Check on subprograms to check larger program
  - Necessary for checking big programs

GRAMMAR IS NOT ENOUGH

- Question: Should the following syntax be valid?

```
(foo 0) + 1
```

- No? `(foo 0)` is not a numeric expression
  - But want to be able to sum up two applications!
- Yes? Suppose grammar lets us sum up expressions
  - But then what to do if `foo` returns a boolean?

TYPE SYSTEMS

BRIEF HISTORY

- From *type theory* by Bertrand Russell (1900s)
  - Trying to fix paradoxes in foundations of math
  - "Is there a set containing all sets?"
- *Simple type theory* developed by Carnap, Ramsey, Quine, Tarski (1920-1930s)
  - This will be our focus
- Many fancier type theories developed later
  - We mostly won't talk about them

TYPES CLASSIFY PROGRAMS

- Simple idea: each *program* e has a *type* t
- Types describe what kind of program e is
- Some programs do not have a type
- All programs have *at most one* type

BASE TYPES

For our purposes: booleans and integers

```
base-ty = "bool" | "int"
```

FUNCTION TYPES

- Each function goes from input type to output type
- Note: input and output can themselves be functions!

```
ty = base-ty | ty "->" ty
```

- This is the full grammar of simple types. Examples:
  - `true` has type `bool`
  - `42` has type `int`
  - `plusOne = λx. x + 1` has type `int -> int`

TYPING CONTEXT

- Will need to type *open* terms with free variables
  - Type depends on types of free variables
- Track these types in a *typing context* G
  - *Bindings*: $(x : t)$ means variable x has type t
  - A typing context G is a list of bindings
- Examples:
  - Empty context: $G = \cdot$
  - Two bindings: $G = x : \mathrm{bool}, y : \mathrm{int}$

TYPING JUDGMENT

- Putting it all together:

$$G \vdash e : t$$

- Read: program e has type t in context G
  - Boolean constants: $\vdash \text{true} : \text{bool}$
  - Open terms: $x : \text{int} \vdash x + 1 : \text{int}$

HOW DO WE ASSIGN TYPES?

TYPES OF PROGRAMS FROM TYPES OF SUBPROGRAMS

- We have a set of *typing rules*, with form:
  - If: subprograms each have certain types
  - Then: whole program has some type
- Type of program doesn't depend on surroundings!

EXAMPLE

PROPERTIES OF TYPE SYSTEMS

BROKEN PROGRAMS

- "Well-typed programs should not go wrong"
- Many different choices for what "go wrong" means
- Simplest: a program "goes wrong" if it gets stuck
  - Bug: program that hasn't finished but can't step
  - Example: program `true + 1` is stuck

TYPE SAFETY

- Main *soundness* property of type systems
- If program e has type t, then it never gets stuck
- **Well-typed programs can't have this kind of bug!**
- Typically proved via *progress* and *preservation*

PROGRESS PROPERTY

- If a closed program e is well-typed, then either:
  - It is a value v (finished computing successfully)
  - It can step to some other program: $\mathrm{e} \rightarrow \mathrm{e}'$
- It can't be stuck!

PRESERVATION PROPERTY

- Type should be preserved as a program steps
  - If: a closed program e has type t and it steps to e'
  - Then: e' is a closed program with type t
- Well-typed term can only step to well-typed term

LIMITATIONS OF TYPE SYSTEMS

WELL-TYPED PROGRAMS CAN HAVE BUGS

- Plenty of ways to write buggy well-typed programs
- For example: this program has type $\mathtt{int} \to \mathtt{int}$

```
plusOne = λx. x + 2
```

  - Probably not what we wanted, though. Oops!

SOME CORRECT PROGRAMS ARE NOT WELL-TYPED

- This program not well-typed, but doesn't get stuck:

```
(if true then 0 else false) + 1
```

  - "Type systems are sound but not complete"
  - "Type systems are a *conservative* analysis"
  - From complexity theory, this is not surprising!
  - Usually soundness or completeness, not both

TERMINATION

- A well-typed program in our system could loop
- Soundness just guarantees that program can step, doesn't guarantee it will ever finish
- Fancier type systems can guarantee termination