```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Introduction to Section 4: Remaining ML Topics

Remaining Topics

- Type Inference
- Mutual Recursion
- Module System
- Equivalence

- No homework assignment focused on this material
  - But some will be on the Part A exam

Next section:

Start using Racket for more programming-languages concepts

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

What is Type Inference?

Type-checking

- (Static) type-checking can reject a program before it runs to prevent the possibility of some errors
  - A feature of statically typed languages

- Dynamically typed languages do little (none?) such checking
  - So might try to treat a number as a function at run-time

- Will study relative advantages after some Racket
  - Racket, Ruby (and Python, Javascript, …) dynamically typed

- ML (and Java, C#, Scala, C, C++) is statically typed
  - Every binding has one type, determined "at compile-time"

Implicitly typed

- ML is statically typed
- ML is implicitly typed: rarely need to write down types

```
fun f x = (* infer val f : int -> int *)
    if x > 3
    then 42
    else x * 2

fun g x = (* report type error *)
    if x > 3
    then true
    else x * 2
```

- Statically typed: Much more like Java than Javascript!

Type inference

- Type inference problem: Give every binding/expression a type such that type-checking succeeds
  - Fail if and only if no solution exists

- In principle, could be a pass before the type-checker
  - But often implemented together

- Type inference can be easy, difficult, or *impossible*
  - Easy: Accept all programs
  - Easy: Reject all programs
  - Subtle, elegant, and *not magic*: ML

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

ML Type Inference

Overview

- Will describe ML type inference via several examples
  - General algorithm is a slightly more advanced topic
  - Supporting nested functions also a bit more advanced
- Enough to help you “do type inference in your head”
  - And appreciate it is not magic

Key steps

- Determine types of bindings in order
  - (Except for mutual recursion)
  - So you cannot use later bindings: will not type-check
- For each **`val`** or **`fun`** binding:
  - Analyze definition for all necessary facts (constraints)
  - Example: If see **`x > 0`**, then **`x`** must have type **`int`**
  - Type error if no way for all facts to hold (over-constrained)
- Afterward, use type variables (e.g., **`'a`**) for any unconstrained types
  - Example: An unused argument can have any type
- (Finally, enforce the *value restriction*, discussed later)

Very simple example

Next segments will go much more step-by-step

- Like the automated algorithm does

```
val x = 42 (* val x : int *)

fun f (y, z, w) =
    if y (* y must be bool *)
    then z + x (* z must be int *)
    else 0 (* both branches have same type *)
(* f must return an int
   f must take a bool * int * ANYTHING
   so val f : bool * int * 'a -> int
 *)
```

Relation to Polymorphism

- Central feature of ML type inference: it can infer types with type variables
  - Great for code reuse and understanding functions

- But remember there are two orthogonal concepts
  - Languages can have type inference without type variables
  - Languages can have type variables without type inference

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Type Inference Examples

Key Idea

- Collect all the facts needed for type-checking

- These facts constrain the type of the function

- This segment:
  - Two examples without type variables
  - And one example that does not type-check

- See the code file and/or the reading notes

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Polymorphic Examples

Key Idea

- Collect all the facts needed for type-checking

- These facts constrain the type of the function

- This segment:
  - Examples with type variables
  - Happens when constraints do not require particular types (but some types may still need to be the same as each other)

- See the code file and/or the reading notes

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Optional: The Value Restriction and Other Type-Inference Challenges

Two more topics

- ML type-inference story so far is too lenient
  - Value restriction limits where polymorphic types can occur
  - See why and then what

- ML is in a "sweet spot"
  - Type inference more difficult without polymorphism
  - Type inference more difficult with subtyping

Important to "finish the story" but these topics are:

- A bit more advanced
- A bit less elegant
- Will not be on the exam

The Problem

As presented so far, the ML type system is *unsound*!

- Allows putting a value of type **`t1`** (e.g., **`int`**) where we expect a value of type **`t2`** (e.g., **`string`**)

A combination of polymorphism and mutation is to blame:

```
val r = ref NONE (* val r : 'a option ref *)
val _ = r := SOME "hi"
val i = 1 + valOf (!r)
```

- Assignment type-checks because (infix) **`:=`** has type **`'a ref * 'a -> unit`**, so instantiate with **`string`**
- Dereference type-checks because **`!`** has type **`'a ref -> 'a`**, so instantiate with **`int`**

What to do

To restore soundness, need a stricter type system that rejects at least one of these three lines

```
val r = ref NONE (* val r : 'a option ref *)

val _ = r := SOME "hi"

val i = 1 + valOf (!r)
```

- And cannot make special rules for reference types because type-checker cannot know the definition of all type synonyms
  - Module system coming up

```
type 'a foo = 'a ref
val f = ref (* val f : 'a -> 'a foo *)
val r = f NONE
```

The fix

```
val r = ref NONE (* val r : ?.X1 option ref *)

val _ = r := SOME "hi"

val i = 1 + valOf (!r)
```

- Value restriction: a variable-binding can have a polymorphic type only if the expression is a variable or value
  - Function calls like **`ref NONE`** are neither
- Else get a warning and unconstrained types are filled in with dummy types (basically unusable)
- Not obvious this suffices to make type system sound, but it does

The downside

As we saw previously, the value restriction can cause problems when it is unnecessary because we are not using mutation

```
val pairWithOne = List.map (fn x => (x,1))
(* does not get type 'a list -> ('a*int) list *)
```

The type-checker does not know **`List.map`** is not making a mutable reference

Saw workarounds in previous segment on partial application

– Common one: wrap in a function binding

```
fun pairWithOne xs = List.map (fn x => (x,1)) xs
(* 'a list -> ('a*int) list *)
```

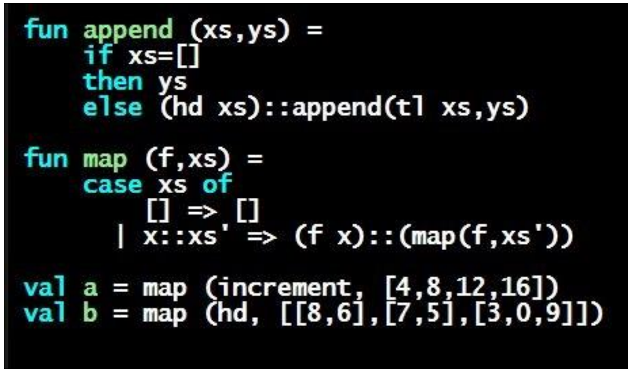
A local optimum

- Despite the value restriction, ML type inference is elegant and fairly easy to understand

- More difficult *without* polymorphism
  - What type should length-of-list have?

- More difficult *with* subtyping
  - Suppose pairs are supertypes of wider tuples
  - Then **`val (y,z) = x`** constrains **`x`** to have at least two fields, not exactly two fields
  - Depending on details, languages can support this, but types often more difficult to infer and understand

  - Will study subtyping later, but not with type inference



Programming Languages

Dan Grossman

Mutual Recursion

Mutual Recursion

- Allow **`f`** to call **`g`** and **`g`** to call **`f`**

- Useful? Yes.
  - Idiom we will show: implementing state machines

- The problem: ML's bindings-in-order rule for environments
  - Fix #1: Special new language construct
  - Fix #2: Workaround using higher-order functions

New language features

- Mutually recursive functions (the `and` keyword)

```
fun f1 p1 = e1
and f2 p2 = e2
and f3 p3 = e3
```

- Similarly, mutually recursive datatype bindings

```
datatype t1 = …
and t2 = …
and t3 = …
```

- Everything in “mutual recursion bundle” type-checked together and can refer to each other

State-machine example

- Each “state of the computation” is a function
  - “State transition” is “call another function” with “rest of input”
  - Generalizes to any finite-state-machine example

```
fun state1 input_left = …

and state2 input_left = …

and …
```

Work-around

- Suppose we did not have support for mutually recursive functions
  - Or could not put functions next to each other
- Can have the “later” function pass itself to the “earlier” one
  - Yet another higher-order function idiom

```
fun earlier (f,x) = … f y …

… (* no need to be nearby *)

fun later x = … earlier(later,y) …
```

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Modules for Namespace Management

Modules

For larger programs, one “top-level” sequence of bindings is poor

- Especially because a binding can use *all* earlier (non-shadowed) bindings

So ML has *structures* to define *modules*

```
structure MyModule = struct bindings end
```

Inside a module, can use earlier bindings as usual

- Can have any kind of binding (val, datatype, exception, ...)

Outside a module, refer to earlier modules’ bindings via **`ModuleName.bindingName`**

- Just like **`List.foldl`** and **`String.toUpper`**; now you can define your own modules

Example

```
structure MyMathLib =
struct

fun fact x =
    if x=0
    then 1
    else x * fact(x-1)

val half_pi = Math.pi / 2

fun doubler x = x * 2

end
```

Namespace management

- *So far*, this is just *namespace management*
  - Giving a hierarchy to names to avoid shadowing
  - Allows different modules to reuse names, e.g., **`map`**
  - Very important, but not very interesting

Optional: Open

- Can use **`open ModuleName`** to get "direct" access to a module's bindings
  - Never necessary; just a convenience; often bad style
  - Often better to create local val-bindings for just the bindings you use a lot, e.g., **`val map = List.map`**
    - But doesn't work for patterns
    - And **`open`** can be useful, e.g., for testing code

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Signatures and Hiding Things

Signatures

- A *signature* is a type for a module
  - What bindings does it have and what are their types
- Can define a signature and ascribe it to modules – example:

```
signature MATHLIB =
sig
val fact : int -> int
val half_pi : real
val doubler : int -> int
end

structure MyMathLib :> MATHLIB =
struct
fun fact x = …
val half_pi = Math.pi / 2.0
fun doubler x = x * 2
end
```

In general

- Signatures

```
signature SIGNAME =
sig types-for-bindings end
```

  - Can include variables, types, datatypes, and exceptions defined in module

- Ascribing a signature to a module

```
structure MyModule :> SIGNAME =
struct bindings end
```

  - Module will not type-check unless it matches the signature, meaning it has all the bindings at the right types
  - Note: SML has other forms of ascription; we will stick with these [opaque signatures]

Hiding things

Real value of signatures is to to *hide* bindings and type definitions

- So far, just documenting and checking the types

Hiding implementation details is the most important strategy for writing correct, robust, reusable software

So first remind ourselves that functions already do well for some forms of hiding…

Hiding with functions

These three functions are totally equivalent: no client can tell which we are using (so we can change our choice later):

```
fun double x = x*2
fun double x = x+x
val y = 2
fun double x = x*y
```

Defining helper functions locally is also powerful

- Can change/remove functions later and know it affects no other code

Would be convenient to have “private” top-level functions too

- So two functions could easily share a helper function
- ML does this via signatures that omit bindings…

Example

Outside the module, **`MyMathLib.doubler`** is simply unbound

- So cannot be used [directly]
- Fairly powerful, very simple idea

```
signature MATHLIB =
sig
val fact : int -> int
val half_pi : real
end

structure MyMathLib :> MATHLIB =
struct
fun fact x = …
val half_pi = Math.pi / 2.0
fun doubler x = x * 2
end
```

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

A Module Example

A larger example [mostly see the code]

Now consider a module that defines an Abstract Data Type (ADT)

- A type of data and operations on it

Our example: rational numbers supporting `add` and `toString`

```
structure Rational1 =
struct
datatype rational = Whole of int | Frac of int*int
exception BadFrac

(*internal functions gcd and reduce not on slide*)

fun make_frac (x,y) = …
fun add (r1,r2) = …
fun toString r = …
end
```

Library spec and invariants

Properties [externally visible guarantees, up to library writer]

- Disallow denominators of 0
- Return strings in reduced form (“4” not “4/1”, “3/2” not “9/6”)
- No infinite loops or exceptions

Invariants [part of the implementation, not the module’s spec]

- All denominators are greater than 0
- All **`rational`** values returned from functions are reduced

More on invariants

Our code maintains the invariants and relies on them

Maintain:

- **`make_frac`** disallows 0 denominator, removes negative denominator, and reduces result
- **`add`** assumes invariants on inputs, calls **`reduce`** if needed

Rely:

- **`gcd`** does not work with negative arguments, but no denominator can be negative
- **`add`** uses math properties to avoid calling **`reduce`**
- **`toString`** assumes its argument is already reduced

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Signatures for Our Example

A first signature

With what we know so far, this signature makes sense:

– **gcd** and **reduce** not visible outside the module

```
signature RATIONAL_A =
sig
datatype rational = Whole of int | Frac of int*int
exception BadFrac
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end

structure Rational1 :> RATIONAL_A = …
```

The problem

By revealing the datatype definition, we let clients violate our invariants by directly creating values of type **`Rational1.rational`**

- At best a comment saying “must use **`Rational1.make_frac`**”

```
signature RATIONAL_A =
sig
datatype rational = Whole of int | Frac of int*int
…
```

Any of these would lead to exceptions, infinite loops, or wrong results, which is why the module’s code would never return them

- **`Rational1.Frac(1,0)`**
- **`Rational1.Frac(3,~2)`**
- **`Rational1.Frac(9,6)`**

So hide more

Key idea: An ADT must hide the concrete type definition so clients cannot create invariant-violating values of the type directly

Alas, this attempt doesn't work because the signature now uses a type **`rational`** that is not known to exist:

```
signature RATIONAL_WRONG =
sig
exception BadFrac
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end

structure Rational1 :> RATIONAL_WRONG = …
```

Abstract types

So ML has a feature for exactly this situation:

In a signature:

`type foo`

means the type exists, but clients do not know its definition

```
signature RATIONAL_B =
sig
type rational
exception BadFrac
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end

structure Rational1 :> RATIONAL_B = …
```

This works! (And is a Really Big Deal)

```
signature RATIONAL_B =
sig
type rational
exception BadFrac
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end
```

Nothing a client can do to violate invariants and properties:

- Only way to make first rational is **`Rational1.make_frac`**
- After that can use only **`Rational1.make_frac`**, **`Rational1.add`**, and **`Rational1.toString`**
- Hides constructors and patterns – don't even know whether or not **`Rational1.rational`** is a datatype
- But clients can still pass around fractions in any way

Two key restrictions

So we have two powerful ways to use signatures for hiding:

1. Deny bindings exist (val-bindings, fun-bindings, constructors)

2. Make types abstract (so clients cannot create values of them or access their pieces directly)

(Later we will see a signature can also make a binding's type more specific than it is within the module, but this is less important)

A cute twist

In our example, exposing the **`Whole`** constructor is no problem

In SML we can expose it as a function since the datatype binding in the module does create such a function

- Still hiding the rest of the datatype
- Still does not allow using **`Whole`** as a pattern

```
signature RATIONAL_C =
sig
type rational
exception BadFrac
val Whole : int -> rational
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end
```

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Signature Matching

Signature matching

Have so far relied on an informal notion of, “does a module type-check given a signature?” As usual, there are precise rules…

`structure Foo :> BAR` is allowed if:

- Every non-abstract type in **`BAR`** is provided in **`Foo`**, as specified
- Every abstract type in **`BAR`** is provided in **`Foo`** in some way
  - Can be a datatype or a type synonym
- Every val-binding in **`BAR`** is provided in **`Foo`**, possibly with a *more general* and/or *less abstract* internal type
  - Discussed “more general types” earlier in course
  - Will see example soon
- Every exception in **`BAR`** is provided in **`Foo`**

Of course **`Foo`** can have more bindings (implicit in above rules)

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

An Equivalent Structure

Equivalent implementations

A key purpose of abstraction is to allow *different implementations* to be *equivalent*

- *No* client can tell which you are using
- So can improve/replace/choose implementations later
- Easier to do if you *start* with more abstract signatures (reveal only what you must)

Now:

Another structure that can also have signature **`RATIONAL_A`**, **`RATIONAL_B`**, or **`RATIONAL_C`**

- But only *equivalent* under **`RATIONAL_B`** or **`RATIONAL_C`**
  (ignoring overflow)

Next:

A third equivalent structure implemented very differently

Equivalent implementations

Example (see code file):

- **`structure Rational2`** does not keep rationals in reduced form, instead reducing them "at last moment" in **`toString`**
  - Also make **`gcd`** and **`reduce`** local functions
- Not equivalent under **`RATIONAL_A`**
  - **`Rational1.toString(Rational1.Frac(9,6)) = "9/6"`**
  - **`Rational2.toString(Rational2.Frac(9,6)) = "3/2"`**
- Equivalent under **`RATIONAL_B`** or **`RATIONAL_C`**
  - Different invariants, but same properties
  - Essential that type **`rational`** is abstract

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Another Equivalent Structure

More interesting example

Given a signature with an abstract type, different structures can:

- Have that signature
- But implement the abstract type differently

Such structures might or might not be equivalent

Example (see code):

- **`type rational = int * int`**
- Does *not* have signature **`RATIONAL_A`**
- *Equivalent* to both previous examples under **`RATIONAL_B`** or **`RATIONAL_C`**

More interesting example

```
structure Rational3 =
struct
type rational = int * int
exception BadFrac

fun make_frac (x,y) = …
fun Whole i = (i,1) (* needed for RATIONAL_C *)
fun add ((a,b)(c,d)) = (a*d+b*c,b*d)
fun toString r = … (* reduce at last minute *)
end
```

Some interesting details

- Internally **`make_frac`** has type **`int * int -> int * int`**, but externally **`int * int -> rational`**
  - Client cannot tell if we return argument unchanged
  - Could give type **`rational -> rational`** in signature, but this is awful: makes entire module unusable – why?

- Internally **`Whole`** has type **`'a -> 'a * int`** but externally **`int -> rational`**
  - This matches because we can specialize **`'a`** to **`int`** and then abstract **`int * int`** to **`rational`**
  - **`Whole`** cannot have types **`'a -> int * int`** or **`'a -> rational`** (must specialize all **`'a`** uses)
  - Type-checker figures all this out for us

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Different Modules Define Different Types

Can't mix-and-match module bindings

Modules with the *same signatures* still define *different types*

So things like this do not type-check:

- **`Rational1.toString(Rational2.make_frac(9,6))`**
- **`Rational3.toString(Rational2.make_frac(9,6))`**

This is a crucial feature for type system and module properties:

- Different modules have different internal invariants!
- In fact, they have different type definitions
  - **`Rational1.rational`** looks like **`Rational2.rational`**, but clients and the type-checker do not know that
  - **`Rational3.rational`** is **`int*int`** not a datatype!

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Equivalent Functions

Last Topic of Section

More careful look at what "two pieces of code are equivalent" means

- Fundamental software-engineering idea

- Made easier with
  - Abstraction (hiding things)
  - Fewer side effects

Not about any "new ways to code something up"

Equivalence

Must reason about "are these equivalent" *all the time*

- The more precisely you think about it the better

- *Code maintenance:* Can I simplify this code?

- *Backward compatibility:* Can I add new features without changing how any old features work?

- *Optimization:* Can I make this code faster?

- *Abstraction:* Can an external client tell I made this change?

To focus discussion: When can we say two functions are equivalent, even without looking at all calls to them?

- May not know all the calls (e.g., we are editing a library)

A definition

Two functions are equivalent if they have the same “observable behavior” no matter how they are used anywhere in any program

Given equivalent arguments, they:

- Produce equivalent results
- Have the same (non-)termination behavior
- Mutate (non-local) memory in the same way
- Do the same input/output
- Raise the same exceptions

Notice it is much easier to be equivalent if:

- There are fewer possible arguments, e.g., with a type system and abstraction
- We avoid *side-effects*: mutation, input/output, and exceptions

Example

Since looking up variables in ML has no side effects, these two functions are equivalent:

```
fun f x = x + x
```

=

```
val y = 2
fun f x = y * x
```

But these next two are not equivalent in general: it depends on what is passed for `f`

– Are equivalent *if* argument for `f` has no side-effects

```
fun g (f,x) =
    (f x) + (f x)
```

≠

```
val y = 2
fun g (f,x) =
    y * (f x)
```

– Example: `g (fn i => (print "hi" ; i), 7)`

– Great reason for "pure" functional programming

Another example

These are equivalent *only if* functions bound to **`g`** and **`h`** do not raise exceptions or have side effects (printing, updating state, etc.)

– Again: pure functions make more things equivalent

```
fun f x =
    let
      val y = g x
      val z = h x
    in
      (y,z)
    end
```

$\neq$

```
fun f x =
    let
      val z = h x
      val y = g x
    in
      (y,z)
    end
```

– Example: **`g`** divides by **`0`** and **`h`** mutates a top-level reference

– Example: **`g`** writes to a reference that **`h`** reads from

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Standard Equivalences

Syntactic sugar

Using or not using syntactic sugar is always equivalent

- By definition, else not syntactic sugar

Example:

```
fun f x =
    x andalso g x
```

$=$

```
fun f x =
    if x
    then g x
    else false
```

But be careful about evaluation order

```
fun f x =
    x andalso g x
```

$\neq$

```
fun f x =
    if g x
    then x
    else false
```

Standard equivalences

Three general equivalences that always work for functions

– In any (?) decent language

1. Consistently rename bound variables and uses

```
val y = 14
fun f x = x+y+x
```

=

```
val y = 14
fun f z = z+y+z
```

But notice you can't use a variable name already used in the function body to refer to something else

```
val y = 14
fun f x = x+y+x
```

≠

```
val y = 14
fun f y = y+y+y
```

```
fun f x =
  let val y = 3
  in x+y end
```

≠

```
fun f y =
  let val y = 3
  in y+y end
```

Standard equivalences

Three general equivalences that always work for functions

- In (any?) decent language

2. Use a helper function or do not

```
val y = 14
fun g z = (z+y+z)+z
```

$=$

```
val y = 14
fun f x = x+y+x
fun g z = (f z)+z
```

But notice you need to be careful about environments

```
val y = 14
val y = 7
fun g z = (z+y+z)+z
```

$\neq$

```
val y = 14
fun f x = x+y+x
val y = 7
fun g z = (f z)+z
```

Standard equivalences

Three general equivalences that always work for functions

– In (any?) decent language

3. Unnecessary function wrapping

```
fun f x = x+x
fun g y = f y
```

$=$

```
fun f x = x+x
val g = f
```

But notice that if you compute the function to call and *that computation* has side-effects, you have to be careful

```
fun f x = x+x
fun h () = (print "hi";
            f)
fun g y = (h()) y
```

$\neq$

```
fun f x = x+x
fun h () = (print "hi";
            f)
val g = (h())
```

One more

If we ignore types, then ML let-bindings can be syntactic sugar for calling an anonymous function:

```
let val x = e1
in e2 end
```

```
(fn x => e2) e1
```

- These both evaluate **`e1`** to **`v1`**, then evaluate **`e2`** in an environment extended to map **`x`** to **`v1`**
- So *exactly* the same evaluation of expressions and result

But in ML, there is a type-system difference:

- **`x`** on the left can have a polymorphic type, but not on the right
- Can always go from right to left
- If **`x`** need not be polymorphic, can go from left to right

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Equivalence Versus Performance

What about performance?

According to our definition of equivalence, these two functions are equivalent, but we learned one is awful

– (Actually we studied this before pattern-matching)

```
fun max xs =
  case xs of
    [] => raise Empty
  | x::[] => x
  | x::xs' =>
      if x > max xs'
      then x
      else max xs'
```

```
fun max xs =
  case xs of
    [] => raise Empty
  | x::[] => x
  | x::xs' =>
     let
       val y = max xs'
     in
       if x > y
       then x
       else y
     end
```

Different definitions for different jobs

- PL Equivalence: given same inputs, same outputs and effects
  - Good: Lets us replace bad **`max`** with good **`max`**
  - Bad: Ignores performance in the extreme
- Asymptotic equivalence: Ignore constant factors
  - Good: Focus on the algorithm and efficiency for large inputs
  - Bad: Ignores “four times faster”
- Systems equivalence: Account for constant overheads, performance tune
  - Good: Faster means different and better
  - Bad: Beware overtuning on “wrong” (e.g., small) inputs; definition does not let you “swap in a different algorithm”

Claim: Computer scientists implicitly (?) use all three every (?) day