15-150 Fall 2013

Lecture 3

Stephen Brookes

Specifications and proofs

Last time

- Expression evaluation produces a value (if it terminates)
    - e =>* v means e *evaluates to* v
- Declarations produce value bindings
    - d =>* $[\![x_1{:}v_1, \ldots, x_k{:}v_k]\!]$
- Pattern matching succeeds or fails
    - *match*(p, v) *succeeds* with $[\![x_1{:}v_1, \ldots, x_k{:}v_k]\!]$
    - *match*(p, v) *fails*

TYPE SAFETY

Notation

- We write $[\![x_1{:}v_1, \ldots, x_k{:}v_k]\!]$
  for a collection of *value bindings*

- Conventions

$x, x_1, \ldots$	*variables*
$v, v_1, \ldots$	*(syntactic) values*
$e, e_1, \ldots$	*expressions*
$t, t_1, \ldots$	*types*

- Termination $\quad e \downarrow$ when $\exists v.\ e \Rightarrow^* v$

- Non-termination $\quad e \uparrow$

Basic properties

- Reflexive

$$e \Rightarrow^* e$$

- Transitive

If

$$e_1 \Rightarrow^* e_2 \text{ and } e_2 \Rightarrow^* e_3$$

then

$$e_1 \Rightarrow^* e_3$$

Substitution

- Given a collection of value bindings

  $$[\![\ x_1{:}v_1, \ldots, x_k{:}v_k\ ]\!]$$

  and an expression e

  we write

  $$[\![\ x_1{:}v_1, \ldots, x_k{:}v_k\ ]\!]\ e$$

  for the expression obtained by substituting

  $$v_1 \text{ for } x_1, \ldots, v_k \text{ for } x_k \quad \text{in } e$$

Examples

⟦ x:2 ⟧ (x + x)   is   (2 + 2)

⟦ x:2 ⟧ (fn y => x + y)   is   (fn y => 2 + y)

⟦ x:2 ⟧ (if x>0 then 1 else f(x-1))

is (if 2>0 then 1 else f(2-1))

Evaluation

(only the well-typed instances)

- **Arithmetic**

  + evaluates left-to-right

  If

  $e_1 =>^* v_1$ and $e_2 =>^* v_2$

  then

  $e_1 + e_2 =>^* v_1 + e_2$

  $=>^* v_1 + v_2$

- **Boolean**

  If $e =>^*$ **true**, then

  if e **then** e_1 **else** $e_2 =>^* e_1$

Evaluation

(only the well-typed instances)

- **Functions**

a function call evaluates its argument

If

$e_1 \Rightarrow^* (\text{fn } x \Rightarrow e)$ and $e_2 \Rightarrow^* v$

then

$$e_1\ e_2 \Rightarrow^* (\text{fn } x \Rightarrow e)\ e_2$$
$$\Rightarrow^* (\text{fn } x \Rightarrow e)\ v \Rightarrow^* [\![x{:}v]\!]e$$

- **Declarations**

In the scope of fun f(x) = e

$$f \Rightarrow^* (\text{fn } x \Rightarrow e)$$

Patterns

- If matching p against v *succeeds* with bindings $[\![x_1{:}v_1, \ldots, x_k{:}v_k]\!]$,

  $$(\text{fn } p \Rightarrow e)\ v \quad \Rightarrow^* \quad [\![x_1{:}v_1, \ldots, x_k{:}v_k]\!]e$$

- If matching p_1 against v *fails*, and matching p_2 against v *succeeds* with bindings $[\![x_1{:}v_1, \ldots, x_k{:}v_k]\!]$,

  $$(\text{fn } p_1 \Rightarrow e_1 \mid p_2 \Rightarrow e_2)\ v$$
  $$\Rightarrow^* \quad [\![x_1{:}v_1, \ldots, x_k{:}v_k]\!]e_2$$

Inversion

(only the well-typed instances)

- **Arithmetic**

+ evaluates its arguments

If

$$e_1 + e_2 \Rightarrow^* v$$

there are v_1, v_2 such that

$$e_1 \Rightarrow^* v_1$$

$$e_2 \Rightarrow^* v_2$$

$$v_1 + v_2 \Rightarrow^* v$$

Evaluation

- $=>^*$ is the *reflexive transitive closure* of => (one-step evaluation)
- $=>^+$ is the *transitive closure* of =>

e => e′	*one step*
e $=>^*$ e′	*finitely many steps*
e $=>^+$ e′	*at least one step*

Example

fun f(x:int):int = f x

$f\ 0 \quad \Rightarrow^{+} (\text{fn } x \Rightarrow f\ x)\ 0$

$\Rightarrow^{*} [\![x:0]\!]\ f\ x$

$\Rightarrow^{*}\ f\ 0$

hence $f\ 0 \Rightarrow^{+} f\ 0$

and $(f\ 0)\uparrow$

Summary

- Using => and =>* we can talk precisely about program behavior
- But sometimes we may want to *ignore* evaluation order...

For all e_1, e_2 : int and all v:int

if $e_1 + e_2 \Rightarrow^* v$ then $e_2 + e_1 \Rightarrow^* v$

evaluation order is different
but we only care about the final value

Example

fun addl(x:int, y:int):int = x + y
fun addr(x:int, y:int):int = y + x

- Let E be a well-typed expression of type int containing a call to addl
- Let E′ be obtained by changing to addr
- E′ is also well-typed with type int
- If E =>* 42 then also E′ =>* 42

addl and addr are *indistinguishable*

Equality
(extensional equivalence)

- For each type t there is a mathematical notion of *equality* for expressions of that type

$$e_1 =_{int} e_2 \Leftrightarrow \forall n.\ (e_1 \Rightarrow^* n \text{ iff } e_2 \Rightarrow^* n)$$

$$f_1 =_{int\text{->}int} f_2 \Leftrightarrow \quad (f_1\downarrow \text{ iff } f_2\downarrow) \text{ and}$$

$$\forall e_1,e_2{:}int.\ (e_1 =_{int} e_2 \text{ implies } f_1\ e_1 =_{int} f_2\ e_2)$$

$$\text{addl} =_{int\text{->}int} \text{addr}$$

Equations

(only the well-typed instances)

- **Arithmetic**

$$e + 0 = e$$

$$e_1 + e_2 = e_2 + e_1$$

$$e_1 + (e_2 + e_3) = (e_1 + e_2) + e_3$$

$$21 + 21 = 42$$

- **Boolean**

$$\textbf{if true then } e_1 \textbf{ else } e_2 = e_1$$

$$\textbf{if false then } e_1 \textbf{ else } e_2 = e_2$$

$$(0 < 1) = \textbf{true}$$

Equations
(only the well-typed instances)

- **Applications**

  (fn x => e) v = ⟦x:v⟧e

  only when argument is a value

- **Declarations**

  In the scope of   fun f(x) = e

  f = (fn x => e)

Equations

(only the well-typed instances)

- When e_1 and e_2 have type t -> t′

$$e_1 = e_2$$

if and only if

for all values v_1, v_2 of type t

$v_1 = v_2$ *implies* $e_1\ v_1 = e_2\ v_2$

extensionality

Compositionality

(only the well-typed instances)

- Substitution of equals
  - If $e_1 = e_2$ and $e_1' = e_2'$ then $(e_1\ e_1') = (e_2\ e_2')$
  - If $e_1 = e_2$ and $e_1' = e_2'$ then $(e_1 + e_1') = (e_2 + e_2')$

and so on

Useful facts

(when well-typed)

- If e => e′ then e = e′
- If e =>* v then e = v
- If e =>* v then (fn x => E) e = ⟦x:v⟧ E

Summary

- Can use precise math notation to talk about the *applicative behavior* of functional programs
- Equality is compositional
- Equality is defined in terms of evaluation
- e =>* v is consistent with ML

Specifications

For a function definition, specify (* as comments *)

- *name* and *type* of the function
- a ***requires***-condition
  - *assumptions* about the *argument*
- an ***ensures***-condition
  - *guarantees* about the *result* of function call, when the argument has required properties

Specifications

- Be clear and precise
- Use bound variables consistently
- Use $=>^*$ (evaluation) and $=$ (equality) accurately and consistently
- Don't leave any assumptions *hidden*

eval **spec**

```
fun eval ([ ]:int list) : int = 0
  |   eval (d::L) = d + 10 * (eval L);
```

```
(*  eval : int list -> int *)

(*  REQUIRES:                                *)
(*    every integer in L is a decimal digit  *)

(*  ENSURES:                                          *)
(*      eval(L) evaluates to a non-negative integer *)
```

decimal **spec**

```
fun decimal (n:int) : int list =
   if n<10 then [n]
            else (n mod 10) :: decimal (n div 10);
```

```
(*  decimal : int -> int list *)

(*  REQUIRES:   n >= 0                 *)

(*  ENSURES:                           *)
(*      decimal(n) evaluates to        *)
(*          a list L of decimal digits, *)
(*          such that eval(L) = n      *)
```

Does it work?

- A specification *asserts* a correctness property about the applicative behavior of a function
- How do we *prove* that a function *satisfies* a specification?
  - That it does *ensure* a correct result whenever applied to an argument that meets the *requir*ements
- It's not usually feasible or sufficient to do *testing*....

Proofs

- We give math-based proofs
  using *equational* or *evaluational* reasoning
- Use the *program structure* as a guide

program syntax	**reasoning**
if-then-else	boolean case analysis
case p of ...	case analysis
fun f(x) = ...f...	induction

Induction

- A family of proof techniques
  - simple (mathematical) induction
  - complete (strong) induction
  - structural induction
  - well-founded induction

Our plan

- Introduce induction
  - templates to help write accurately
  - learn when applicable
- Focus on *examples*
- Specifications will involve *equality* and *evaluation*

Simple induction

- To prove a property of the form

  P(n), for all non-negative integers n

- First, prove P(0). *base case*
- Then show that, for all k $\geq$ 0, P(k+1) follows logically from P(k).

inductive step

Why it works

- P(0) gets a direct proof
- P(1) follows from P(0)
- P(2) follows from P(1)
- Similarly, for each $n \geq 0$ we can show P(n)
  - for $k>0$, at the k^{th} step we've already shown P(k), so P(k+1) follows logically

Example

```
fun f(x:int):int =
      if x=0 then 1 else f(x-1) + 1
(* REQUIRES x≥0        *)
(* ENSURES f(x) = x+1 *)
```

- To prove:

For all values x:int
such that $x \geq 0$, $f(x) = x+1$

Proof by simple induction

- Let P(n) be f(n) = n+1
- Base case: we prove P(0), i.e. f(0) = 0+1

f 0 = (fn x => if x=0 then 1 else f(x-1)+1) 0
= ⟦x:0⟧(if x=0 then 1 else f(x-1)+1)
= **if** 0=0 **then** 1 **else** f(0-1) + 1
= **if true then** 1 **else** f(0-1) + 1
= 1

0+1 = 1

So f(0) = 0+1

Proof by simple induction

- Let P(n) be f(n) = n+1
- Inductive step:
  let k≥0, assume P(k), prove P(k+1).
  Let v be the numeral for k+1.

f(k+1) = **if** v=0 **then** 1 **else** f(v-1) + 1

= **if false then** 1 **else** f(v-1) + 1

= f(v-1) + 1

= f(k) + 1 since v=k+1

= (k + 1) + 1 by assumption P(k)

So P(k+1) follows from P(k)

Using simple induction

- Q: When can I use *simple* induction to prove a property of a recursive function f ?
- A: When there is a *non-negative* measure of *argument size* and f(x) only makes recursive calls of form f(y) with *size*(y) = *size*(x)-1

Example

```
fun eval ([ ]:int list) : int = 0
  |   eval (d::L) = d + 10 * (eval L);
```

(size = length of argument list, decreases by 1)

To prove:

For all values L:int list
there is an integer n such that
eval L =>* n

When it doesn't work

You cannot use
simple induction
for

```
fun decimal (n:int) : int list =
    if n<10 then [n]
            else (n mod 10) :: decimal (n div 10)
```

Why not?