CSEP505: Programming Languages

Lecture 3: semantics via translation, equivalence & properties, lambda introduction

Dan Grossman

Spring 2006

Course notices

- Final exam moved to Tuesday of “finals week”
- Homework 2 due in 2 weeks
  - Updated this afternoon (no changes)
- For homework 5, will probably have only one weekend
  - *Potentially* worth half of other homeworks
- And about homework 1…

HW1 Post-mortem

- Moving forward, Caml programming will be
  - Somewhat easier on an *absolute* scale
  - Much, much easier on a *relative* scale
  - Aiming for < 10 hours week

- We thought problem 1 was doable given…
  - An almost line-by-line English explanation
  - A reference implementation

  …but we overreached a bit

  (it happens – we just met you!)

  …especially because it was “not like stuff in class”

HW2 Primer

- Problem 1:
  - Extend IMP with saveheap, restoreheap
  - Requires 10-ish changes to the large-step interpreter we went through line-by-line
  - Minor Caml novelty: mutually recursive types
- Problem 2:
  - 3 semantics for a little Logo language
  - Intellectually transfer ideas from IMP
  - A lot of skeleton provided (more added today)

In total, much less code than homework 1

HW2 Primer cont'd

```
e ::= home | forward f | turn f | for i lst
lst ::= [] | e::lst
```

- Semantics of a move list is a "places-visited" list type `(float*float) list`
- A program state is a move list, two coordinates, and a current direction
- Given a list, "do the first thing then the rest"
- As usual, loops are the hardest case.

This is all in the assignment (with Logo description separated out at your suggestion)

One comment from Ben

- Most common “syntax thing” from your emails:
  - Top-level binding (a “statement” sort of)

    ```
    let p = e (* optional ;; at end *)
    ```

    Adds binding for rest of file

  - Local binding

    ```
    let p = e1 in e2 (*e2 can be a let…in*)
    ```

    Adds binding for `e2`

Where are we

Definition by interpretation

- We have *abstract syntax* and two *interpreters* for our *source language* IMP
- Our metalanguage is Caml

Now definition by translation

- Abstract syntax and source language still IMP
- Metalanguage still Caml
- *Target language* now “Caml with just functions strings, ints, and conditionals” tricky stuff?

Remember IMP?

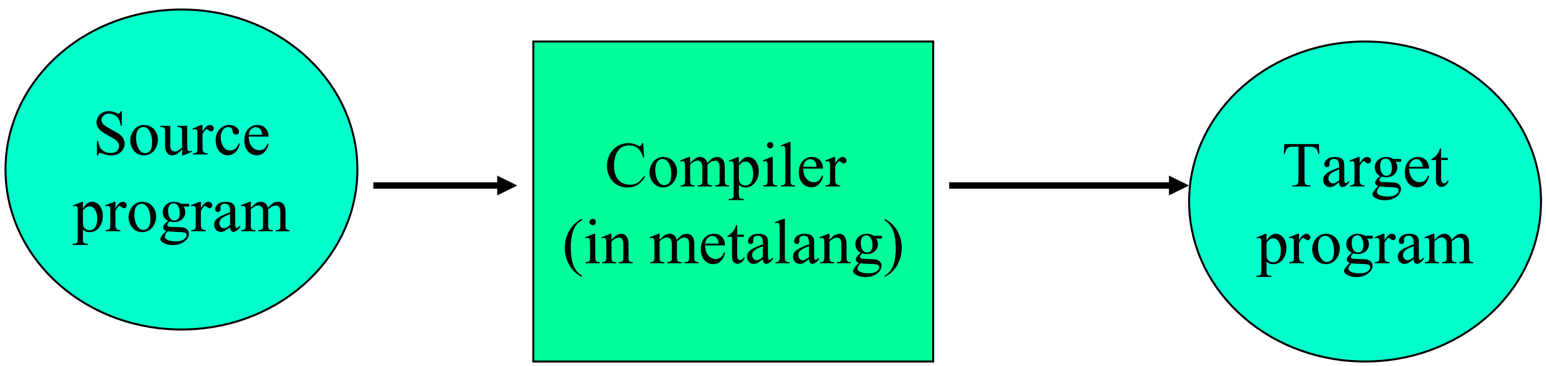
```
type exp = Int of int | Var of string
         | Plus of exp * exp | Times of exp * exp
type stmt = Skip | Assign of string * exp
          | Seq of stmt * stmt
          | If of exp * stmt * stmt
          | While of exp * stmt
```

- `interp_e_large : heap->exp->int`
- `interp_s_large : heap->stmt->heap`
- `interp_e_small : heap->exp->exp`
- `interp_s_small : heap->stmt->heap*stmt`

Small vs. large again

- Small is really inefficient (descends and rebuilds AST at every tiny step)
- But as a definition, it gives a trace of program states (pairs of `heap*stmt`)
  - Can talk about them e.g., “no state has x>17…”
- Theorem: Total equivalence: `interp_prog_large` returns `i` for `s` if & only if `interp_prog_small` does
- With the theorem, we can choose whatever semantics is most convenient

In pictures and equations



- If the target language has a semantics, then:
  compiler + targetSemantics = sourceSemantics

Deep vs. shallow

- Meta and target can be the same language
  - Unusual for a “real” compiler
  - Makes example harder to follow ☹
- Our target will be a subset of Caml
  - After translation, you could (in theory) “unload” the AST definition
  - This is a “deep embedding”
    - An IMP while loop becomes a function
    - Not a piece of data that says “I’m a while loop”

Goals

- **`xlate_e:`**

  `exp -> ((string->int)->int)`

  - "given an exp, produce a function that given a function from strings to ints returns an int"
  - (**`string->int`** acts like a heap)
  - An expression "is" a function from heaps to ints
- **`xlate_s`**:

  `stmt->((string->int)->(string->int))`

  - A statement "is" a function from heaps to heaps

Expression translation

```
xlate_e: exp -> ((string->int)->int)
```

```
let rec xlate_e (e:exp) =
  match e with
   Int i        ->  (fun h -> i)
  |Var str      ->  (fun h -> h str)
  |Plus(e1,e2) ->  let f1 = xlate_e e1 in
                    let f2 = xlate_e e2 in
                    (fun h -> (f1 h) + (f2 h))
  |Times(e1,e2) -> let f1 = xlate_e e1 in
                    let f2 = xlate_e e2 in
                    (fun h -> (f1 h) * (f2 h))
```

What just happened

```
(* an example *)
let e = Plus(Int 3, Times(Var “x”, Int 4))
let f = xlate_e e (* compile *)
(* the value bound to f is a function whose body
   does not use any IMP abstract syntax! *)
let ans = f (fun s -> 0)(* run w/ empty heap *)
```

- Our target sublanguage:
  - Functions (including `+` and `*`, not `interp_e`)
  - Strings and integers
  - Variables bound to things in our sublanguage
  - (later: if-then-else)
- Note: No lookup until “run-time” (of course)

Wrong

- This produces a program not in our sublanguage:

```
let rec xlate_e (e:exp) =
  match e with
   Int i        ->  (fun h -> i)
  |Var str      ->  (fun h -> h str)
  |Plus(e1,e2) ->  (fun h -> (xlate_e e1 h) +
                              (xlate_e e2 h))
  |Times(e1,e2) -> (fun h -> (xlate_e e1 h) *
                              (xlate_e e2 h))
```

- Caml evaluates function bodies when called (like YFL)
- Waits until run-time to translate Plus and Times children!

Statements, part 1

```
xlate_s:
   stmt->((string->int)->(string->int))
```

```
let rec xlate_s (s:stmt) =
 match s with
  Skip          -> (fun h -> h)
 |Assign(str,e) ->
   let f = xlate_e e in
   (fun h -> let i = f h in
             (fun s -> if s=str then i else h s))
 |Seq(s1,s2) ->
    let f2 = xlate_s s2 in (* order irrelevant! *)
    let f1 = xlate_s s1 in
    (fun h -> f2 (f1 h)) (* order relevant *)
 | …
```

Statements, part 2

```
xlate_s:
   stmt->((string->int)->(string->int))
```

```
let rec xlate_s (s:stmt) =
 match s with …
 |If(e,s1,s2) ->
   let f1 = xlate_s s1 in
   let f2 = xlate_s s2 in
   let f  = xlate_e e  in
   (fun h -> if (f h) <> 0 then f1 h else f2 h)
 |While(e,s1) ->
   let f1 = xlate_s s1 in
   let f  = xlate_e e  in
   (*???*)
```

- Why is translation of while tricky???

Statements, part 3

```
xlate_s:
   stmt->((string->int)->(string->int))
```

```
let rec xlate_s (s:stmt) =
 match s with
 …
 |While(e,s1) ->
   let f1 = xlate_s s1 in
   let f  = xlate_e e  in
   let rec loop h = (* ah, recursion! *)
     if f h <> 0
     then loop (f1 h)
     else h
   in loop
```

- Target language *must* have some recursion/loop!

Finishing the story

- Have **`xlate_e`** and **`xlate_s`**
- A "program" is just a statement
- An initial heap is (say) one that maps everything to 0

```
let interp_prog s =
  ((xlate_s s) (fun str -> 0)) "ans"
```

Fancy words: We have defined a "denotational semantics" (but target was not math)

Summary

- Three semantics for IMP
  - Theorem: they are all equivalent
- Avoided (for now?)
  - Inference rules (for “real” operational semantics)
  - Recursive-function theory (for “real” denotational semantics)
- Inference rules useful for reading PL research papers (so we will probably do it)
- If we assume Caml already has a semantics, then using it as a metalanguage and target language makes sense for IMP
- Loops and recursion are deeply connected!

Digression: Packet filters

- If you’re not a language semanticist, is this useful?

Almost everything I know about packet filters:

- Some bits come in off the wire
- Some applications want the “packet" and some do not (e.g., port number)
- For safety, only the O/S can access the wire
- For extensibility, only apps can accept/reject packets

Conventional solution goes to user-space for every packet and app that wants (any) packets.

Faster solution: Run app-written filters in kernel-space

Language-based approaches

1. Interpret a language
   - \+ clean operational semantics, portable
   - \- *may* be slow, unusual interface
2. Translate (JIT) a language into C/assembly
   - \+ clean denotational semantics, existing optimizers,
   - \- upfront (pre-1st-packet) cost, unusual interface
3. Require a conservative subset of C/assembly
   - \+ normal interface
   - \- too conservative without help
   - related to type systems (we'll get there!)

More generally…

Packet filters move the code to the data rather than data to the code

- General reasons: performance, security, other?
- Other examples:
  - Query languages
  - Active networks

Where are we

- Learned primary ways to define languages
  - Mathier metalanguage to come
- Next: Notions of equivalence
  - A core CS concept; PL knowledge is key
  - Two programs, two expressions, two semantics
  - Induction and language properties (informally)
- Then:
  - Lambda calculus (to study scope and functions)

(Code) equivalence motivation

"Is this code equivalent to that code" is hugely important

- Code maintenance
- Compatibility (backward, forward, …)
- Program verification (e.g., verify simpler approach)
- Program optimization (manual, compiler, …)
- Abstraction and strong interfaces
  - Equivalence *easier* to show if context is *more* restricted

But what does equivalence mean?

Equivalence: what

Equivalence depends on what is observable.

Some standard ones:

- Partial I/O equivalence (if terminates, same **`ans`**)
  - IMP: **`while 1 skip`** equivalent to all **`s`**
  - Not transitive
- Total I/O equivalence (partial + same termination)
- Total heap equivalence ((almost) all variables end with the same value)
- … + complexity bounds (is $O(2^n)$ equivalent to $O(n)$)
- Syntactic equivalence (w/ renaming) – uninteresting?

Equivalence: what cont'd

Being too coarse about what is observable has security implications (covert channels)

- The clock?
- Power consumed?
- Cache state?

But the compiler/programmer needs freedom too

- Make code usually run faster
- Leave local variables in a different state
- Assume all calls obey an API protocol

(Code) Equivalence: where

Equivalence depends on context.

- Program equivalence
  - **`interp_prog s1`** vs. **`interp_prog s2`**
- Contextual equivalence
  - Can replace **`s1`** with **`s2`** anywhere in any program
  - Can formalize this notion in a metalanguage: “a context is a program with a hole”…

Formal context definition

IMP expression contexts

(ignore statements just to keep example small)

```
C ::= [.] | C + e | e + C | C * e | e * C
```

Different definitions could allow the hole in fewer places

Recursive metafunction: `ctxt->exp->exp` "fills the hole"

```
        [.][e]  =  e
    (C + e')[e] =  C[e] + e'
    (e' + C)[e] =  e' + C[e]
    (C * e')[e] =  C[e] * e'
    (e' * C)[e] =  e' * C[e]
```

(Could have used Caml for our metalanguage.)

Stating equivalences

Precise definitions lead to precise thinking

Theorem: For all heaps ***h*** and expressions **e**,
`interp_e_large h (Plus(e,e))` evaluates to ***i*** if & only if **`interp_e_large h (Times(e,2))`** does

Proof: Each way "easy" with **`interp_e_large`** & math

Lifting to contexts

But *this* theorem is much more useful:

Theorem: For all heaps ***h***, expressions ***e***, and contexts ***C***,

let **`e1=C[Plus(e,e)]`** & **`e2=C[Times(e,2)]`**. Then

`interp_e_large h e1` evaluates to ***i*** if & only if

`interp_e_large h e2` does.

Proof: By *induction on height of context*.

Base: *C*=[.] (uses previous theorem)

Inductive: (All "trivial induction".)

Moral

- I fully admit this theorem is “obvious” and “pedantic”
- But that let’s us focus on the structure of what we did
  - Recursive metafunction for contexts
  - Inductive proof of property defined in terms of the metafunction
  - Much more general than first theorem

Another example

- Informally, “IMP’s sequence operator is associative”
- Formally,

Another example

- Informally, "IMP's sequence operator is associative"
- Formally,

For all heaps ***h***, statements ***s1***, ***s2***, ***s3***, and contexts ***C***,

1. **`interp_s_large h Seq(s1,Seq(s2,s3))`**
   can return **`h'`** if & only if

   `interp_s_large h Seq(Seq(s1,s2),s3)`
   can
2. **`interp_s_large h Seq(s1,Seq(s2,s3))`**
   can diverge (not terminate) if & only if
   `interp_s_large h Seq(Seq(s1,s2),s3)`
   can

(Second part unnecessary given (total) determinism.)

Where are we

- Code equivalence
  - What is observable
  - What context is code in
  - Semi-silly examples, stated and proven precisely
- Now: language equivalence (briefly)
- Then: Code equivalences in (pure) Caml
  - Will revisit with our next “tiny” language
    (“the lambda calculus”)

Language equivalence

- Previously claimed: "IMP small-step and large-step are equivalent *(for all programs)*"
- Another example: "Semantics with extra rules"
  - (PL-geek term: admissable rules)
  - Example: `Times(e,0)-> 0`
    - Only an example because e terminates and has no effect!
- Another example: Language backward compatibility
  - "Java 1.5 and 1.4 are equivalent *on 1.4 programs*"
  - Probably not true (even ignoring new keywords)

Old news: sugar

- Non-trivial equivalences arise in real languages
- We have already seen “syntactic sugar”:

```
if e1 then e2 else e3   match e1 with
                           true  -> e2
                         | false -> e3
let rec f x y = e       let rec f x = fun y->e
e1 && e2                if e1 then e2 else false
```

- Last one exists in most languages

Function equivalences

There are 3 equivalences related to functions

1. Systematic renaming (“alpha”-equivalence)

   `fun x -> e1` is equivalent to **`fun y -> e2`**
   where **`e2`** is like **`e1`** with all **`x`** replaced by **`y`**

Actually, this claim is subtly wrong; we will fix it soon

This (fixed) claim is true in many languages.

It’s a Good Thing (local renamings stay local!)

Function equivalences

There are 3 equivalences related to functions

2. Function application (“beta”-equivalence)

   `(fun x -> e1) e2` is equivalent to **`e3`** where **`e3`** is like **`e1`** with all **`x`** replaced by **`e2`**

Actually, this claim is also wrong; we will fix it soon

Also wrong unless **`e2`** is “pure” (and terminates!)

Claim: **`e3`** could be faster or slower (why?)

Function equivalences

There are 3 equivalences related to functions

3. Unnecessary function wrapping (“eta”-equivalence)

   `(fun x -> e1 x)` is equivalent to **`e1`** provided **`e1`** is pure, terminates, and does not use **`x`**

Example:

`map f lst` vs. **`map (fun x -> f x) lst`**

Claim: All 3 caveats are necessary. (Why?)

Where are we

- To talk about functions more precisely, we need to define them as carefully as we did IMP's constructs

- Let's first try to add functions and local variables to IMP "on the cheap"
  - It won't work

- Then we'll back up and define a language with *nothing* but functions
  - And we'll be able to encode everything else

Worth a try…

```
type exp  = … (* no change *)
type stmt = … | Call of string * exp
(*prog now has a list of named 1-arg functions*)
type funs = (string*(string*stmt)) list
type prog = funs * stmt

let rec interp_s (fs:funs) (h:heap) (s:stmt) =
 match s with
   …
  | Call(str,e) ->
    let (arg,body) = List.assoc str fs in
    (* str(e) becomes arg:=e; body *)
    interp_s fs h (Seq(Assign(arg,e),body))
```

- A definition yes, but one we want?

The “wrong” definition

- The previous slide makes function call assign to a global variable
  - So choice of argument name matters
  - And affects caller
- Example (with IMP concrete syntax):

```
[ (fun f x = 37)] x := 2; f(3); ans := x
```

- We could try “making up a new variable” every time…

2nd wrong try

```
(* return some string not used in h or s *)
let fresh h s = …

let rec interp_s (fs:funs) (h:heap) (s:stmt) =
 match s with
   …
   | Call(str,e) ->
     let (arg,body) = List.assoc str fs in
     let y = fresh h s in
   (* str(e) becomes y:=arg; arg:=e; body; arg:=y
      where y is "fresh" *)
     interp_s fs h (Seq(Assign(y,Var arg),
                    Seq(Assign(arg,e),
                    Seq(body,
                        Assign(arg,Var y)))))
```

Did that work?

```
(* str(e) becomes y:=arg; arg:=e; body; arg:=y
     where y is "fresh" *)
```

- “fresh” is pretty sloppy and unrealistic
- Not an elegant model of a key PL feature
- Wrong:
  - In functional or OOP: variables in **`body`** should be looked up based on where **`body`** came from
  - Even in C: If **`body`** calls a function that accesses a global variable named **`arg`**

Let's give up

- You cannot properly model local scope with a global heap of integers
  - Functions are not syntactic sugar for assignments
- So let's build a model of this key concept
  - Or just borrow one from 1930s logic
- And for now, drop mutation, conditionals, and loops
  - We won't need them!
- The Lambda calculus in BNF

Expressions: $e ::= x \mid \lambda x.\ e \mid e\ e$

Values: $v ::= \lambda x.\ e$

That's all of it!

Expressions: $e ::= x \mid \lambda x.\ e \mid e\ e$

Values: $v ::= \lambda x.\ e$

A program is an e. To call a function:

substitute the argument for the bound variable

That's the key operation we were missing

Example substitutions:

$$(\lambda x.\ x)\ (\lambda y.\ y) \rightarrow \lambda y.\ y$$

$$(\lambda x.\ \lambda y.\ y\ x)\ (\lambda z.\ z) \rightarrow \lambda y.\ y\ (\lambda z.\ z)$$

$$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \rightarrow (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$$

Why substitution

- After substitution, the bound variable is *gone*, so clearly its name did not matter
  - That was our problem before
- Given substitution (correct definition is subtle; we'll come back to it), we can define a little programming language
  - That turns out to be Turing-complete

Full large-step interpreter

```
(* return some string not used in h or s *)
type exp = Var of string
         | Lam of string*exp
         | Apply of exp * exp
exception BadExp
let subst e1_in e2_for x = … (*to be discussed*)
let rec interp_large e =
  match e with
   Var _ -> raise BadExp(*should have gone away*)
  | Lam _ -> e (*functions are values*)
  | Apply(e1,e2) ->
    let v1 = interp_large e1 in
    let v2 = interp_large e2 in
    match v1 with
      Lam(x,e3) -> interp_large (subst e3 v2 x)
     | _ -> failwith "impossible" (* why? *)
```

Another interpreter

```
(* return some string not used in h or s *)
type exp = Var of string
         | Lam of string*exp
         | Apply of exp * exp
exception BadExp
let subst e1_in e2_for x = … (*to be discussed*)
let rec interp_large2 e =
  match e with
    Var _ -> raise BadExp(*should have gone away*)
  | Lam _ -> e (*functions are values*)
  | Apply(e1,e2) ->
     let v1 = interp_large2 e1 in
     (* we used to evaluate e2 to v2 here *)
     match v1 with
       Lam(x,e3) -> interp_large2 (subst e3 e2 x)
      | _ -> failwith "impossible" (* why? *)
```

What have we done

- Gave syntax and two large-step semantics to the untyped lambda calculus
  - First was “call by value”
  - Second was “call by name”
- Real implementations don’t use substitution; they do something *equivalent*
- Amazing (?) fact:
  - If call-by-value terminates, then call-by-name terminates
  - (They might both not terminate)

What will we do

- Go back to math metalanguage
  - Notes on concrete syntax (relates to Caml)
  - Define semantics with inference rules
- Do small-step
  - CBV and CBN
  - Caml and/or inference rules
- Lambda encodings (show our language is mighty)
- Revisit functional equivalences
- Define substitution (logically out of order)