Summer School on
Language-Based Techniques for Integrating with the External World

Types for Safe C-Level Programming
Part 1: Quantified-Types Background

Dan Grossman
University of Washington
18 July 2007

C-level

- Most PL theory is done for *safe*, *high-level* languages
- A lot of software is written in C
- Jeff: how to interface with C
- Me: Adapt and extend our *theory* to make a safe C
  - Today: review the theory (useful crash course)
  - Next week: Some theory underlying *Cyclone*
    - Love to talk about the engineering off-line
    - Parametric polymorphism ("generics")
    - Existential types
    - Region-based memory management

How is C different?

A brief teaser before our PL theory tutorial…

- C has "left expressions" and "address-of" operator

```
{ int* y[7]; int x = 17; y[0] = &x; }
```

- C has explicit pointers, "unboxed" structures

`struct T` vs. `struct T *`

- C function pointers are not objects or closures

```
void apply_to_list(void (*f)(void*,int),
                   void*, IntList);
```

- C has manual memory management

low-level issues distinct from safety stuff like array-bounds

Lambda-calculus in 1 hour (or so)

- Syntax (abstract)
- Semantics (operational, small-step, call-by-value)
- Types (filter out "bad" programs)

All have inductive definitions using a mathematical *metalanguage*

Will likely speed through things (this is half a graduate course), but follow up with me and fellow students

Syntax

Syntax of an *untyped lambda-calculus*

Expressions: $e ::= x \mid \lambda x.\ e \mid e\ e \mid c \mid e + e$
Constants: $c ::= \ldots \mid -1 \mid 0 \mid 1 \mid \ldots$
Variables: $x ::= \ldots \mid x1 \mid x' \mid y \mid \ldots$
Values: $v ::= \lambda x.\ e \mid c$

Defines a set of trees (ASTs)

Conventions for writing these trees as strings:

- $\lambda x.\ e1\ e2$ is $\lambda x.\ (e1\ e2)$, not $(\lambda x.\ e1)\ e2$
- *e1 e2 e3* is *(e1 e2) e3*, not *e1 (e2 e3)*
- Use parentheses to disambiguate or clarify

Semantics

- One computation step rewrites the program to something "closer to the answer"

$$e \rightarrow e'$$

- Inference rules describe what steps are allowed

$$\frac{e1 \rightarrow e1'}{e1\ e2 \rightarrow e1'\ e2} \qquad \frac{e2 \rightarrow e2'}{v\ e2 \rightarrow v\ e2'} \qquad \frac{}{(\lambda x.\ e)\ v \rightarrow e\{v/x\}}$$

$$\frac{e1 \rightarrow e1'}{e1{+}e2 \rightarrow e1'{+}e2} \qquad \frac{e2 \rightarrow e2'}{v{+}e2 \rightarrow v{+}e2'} \qquad \frac{\text{“}c1{+}c2{=}c3\text{”}}{c1{+}c2 \rightarrow c3}$$

Notes

- These are rule schemas
  - Instantiate by replacing metavariables consistently
- A derivation tree justifies a step
  - A proof: “read from leaves to root”
  - An interpreter: “read from root to leaves”
- Proper definition of substitution requires care
- Program evaluation is then a sequence of steps

  e0 → e1→ e2 → …
- Evaluation can “stop” with a value (e.g., 17) or a “stuck state” (e.g., 17 `λx. x`)

More notes

- I chose left-to-right call-by-value
  - Easy to change by changing/adding rules
- I chose to keep evaluation-sequence deterministic
  - Also easy to change
  - I chose small-step operational
  - Could spend a year on other semantics
- This language is Turing-complete (even without constants and addition)
  - Therefore, infinite state-sequences exist

Adding pairs

e ::= … | (e,e) | e.1 | e.2
v ::= … | (v,v)

$$\frac{e1 \rightarrow e1'}{(e1,e2) \rightarrow (e1',e2)} \qquad \frac{e2 \rightarrow e2'}{(v,e2) \rightarrow (v,e2')} \qquad \frac{e \rightarrow e'}{e.1 \rightarrow e'.1} \qquad \frac{e \rightarrow e'}{e.2 \rightarrow e'.2}$$

$$\frac{}{(v1,v2).1 \rightarrow v1} \qquad \frac{}{(v1,v2).2 \rightarrow v2}$$

Adding mutation

Expressions: `e ::= … | ref e | e1 := e2 | !e | l`
Values: `v ::= … | l`
Heaps: `H ::= . | H,l→v`
States: `H,e`

Change *e → e'* to *H,e → H',e'*

Change rules to modify heap (or not). 2 examples:

$$\frac{H,e1 \rightarrow H',e1'}{H,e1\ e2 \rightarrow H',e1'\ e2} \qquad \frac{\text{“c1+c2=c3”}}{H,\ c1+c2 \rightarrow H,\ c3}$$

New rules

$$\frac{l \text{ not in } H}{H,\ \texttt{ref}\ v \rightarrow H,l\rightarrow v,\ l} \qquad \frac{}{H,\ !\ l \rightarrow H,\ H(l)}$$

$$\frac{}{H,\ l := v \rightarrow H,l\rightarrow v,\ 42}$$

$$\frac{H,e \rightarrow H',e'}{H,\ !\ e \rightarrow H',\ !\ e'} \qquad \frac{H,e \rightarrow H',e'}{H,\ \texttt{ref}\ e \rightarrow H',\ \texttt{ref}\ e'}$$

$$\frac{H,e \rightarrow H',e'}{H,\ e1 := e2 \rightarrow H',\ e1' := e2} \qquad \frac{H,e \rightarrow H',e'}{H,\ v := e2 \rightarrow H',\ v := e2'}$$

Toward evaluation contexts

For each step, *e → e'* or *H,e → H',e'*, we have a derivation tree (actually nonbranching) where:
- The top rule “does something interesting”
- The rest “get us to the right place”

After a step, the next “right place” could be deeper or shallower
- Shallower: (3+4)+5
- Deeper: (3+4)+((1+2)+(5+6))
- Deeper: (λx. (((x+x)+x)+x) 2

Evaluation contexts

A more concise metanotation exploits this "inductive" vs. "active" distinction

- For us, more convenient but unnecessary
- With control operators (e.g., continuations), really adds power

Evaluation contexts: "expressions with one hole where something interesting can happen", so for left-to-right lambda calculus:

```
E ::= [] | E e | v E | E+e | v+E
    | (E,e) | (v,E) | E.1 | E.2
    | ref E | E:=e | v:=E | !E
```

Exactly one case per inductive rule in our old way

The context rule

To finish our "convenient rearrangement":

- Define "filling a hole" metanotation (could formalize)
  $E[e]$: the expression from E with e in its hole
- A single context rule

$$\frac{H,e \rightarrow_p H',e'}{H,E[e] \rightarrow H',E[e']}$$

- Our other rules as "primitive reductions"
  $H,e \rightarrow_p H',e'$
- Now each step is one context rule (find right place) and one primitive reduction (do something)

Summary so far

- Programs as syntax trees
  – Add a heap to program state for mutation
- Semantics as sequence of tree rewrites
- Evaluations contexts separate out the "find the right place"

Next week we'll have two different kinds of primitive reductions (left vs. right) and two kinds of contexts (to control which can occur where)

Why types?

A type system classifies (source) programs

- Ones that do not type-check "not in the language"

Why might we want a smaller language?

1. Prohibit bad behaviors
   Example: never get to a state H;e where e is E[!42]
2. Enforce user-defined interfaces
   Example: `struct T; struct T* newT();` …
3. Simplify/optimize implementations
4. Other

Types

A 2nd judgment $\Gamma \vdash \mathtt{e1} : \tau$ gives types to expressions

– No derivation tree means "does not type-check"
– Use a context to give types to variables in scope

"Simply typed lambda calculus" a starting point

Types: $\tau ::= \mathtt{int} \mid \tau \rightarrow \tau \mid \tau * \tau \mid \mathtt{ref}\ \tau$

Contexts: $\Gamma ::= . \mid \Gamma, \mathtt{x} : \tau$

$$\frac{}{\Gamma \vdash \mathtt{c} : \mathtt{int}} \qquad \frac{\Gamma \vdash \mathtt{e1} : \mathtt{int} \quad \Gamma \vdash \mathtt{e2} : \mathtt{int}}{\Gamma \vdash \mathtt{e1+e2} : \mathtt{int}} \qquad \frac{}{\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})}$$

$$\frac{\Gamma, \mathtt{x} : \tau1 \vdash \mathtt{e} : \tau2}{\Gamma \vdash (\lambda \mathtt{x.e}) : \tau1 \rightarrow \tau2} \qquad \frac{\Gamma \vdash \mathtt{e1} : \tau1 \rightarrow \tau2 \quad \Gamma \vdash \mathtt{e2} : \tau1}{\Gamma \vdash \mathtt{e1\ e2} : \tau2}$$

Notes

- Our declarative rules "infer" types, but we could just as easily adjust the syntax to make the programmer tell us
- These rules look arbitrary but have deep logical connections
- With this simple system:
  – "does it type-check" is decidable (usually wanted)
  – "does an arbitrary e terminate" is undecidable
  – "does a well-typed e terminate" is "always yes" (!)
    • "fix" (pun intended) by adding explicit recursion

The rest of the rules

$$\frac{\Gamma \vdash e1 : \tau1 \quad \Gamma \vdash e2 : \tau2}{\Gamma \vdash (e1,e2) : \tau1 * \tau2} \qquad \frac{\Gamma \vdash e : \tau1 * \tau2}{\Gamma \vdash e.1 : \tau1} \qquad \frac{\Gamma \vdash e : \tau1 * \tau2}{\Gamma \vdash e.2 : \tau2}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{ref}\ e : \texttt{ref}\ \tau}$$

$$\frac{\Gamma \vdash e1 : \texttt{ref}\ \tau \quad \Gamma \vdash e2 : \tau}{\Gamma \vdash e1 := e2 : \texttt{int}} \qquad \frac{\Gamma \vdash e : \texttt{ref}\ \tau}{\Gamma \vdash\ !e : \tau}$$

Soundness

Reason we defined rules how we did:
If $. \vdash e : \tau$ and after some number of steps .;e becomes H';e', then either e' is a value v or there exists an H'';e'' such that H';e' → H'';e''

An infinite number of different type systems have this property for our language, but want to show at least ours is one of them

Also: we wrote the semantics, so we defined what the "bad" states are. Extreme example: every type system is sound if any H;e can step to H;42

Showing soundness

Soundness theorem is true, but how would we show it:

1. Extend our type system to program states (heaps and expressions with labels) *only for the proof*
2. Progress: Any well-typed program state has an expression that is a value or can take one step
3. Preservation: If a well-typed program state takes a step, the new state is well-typed

Perspective: "is well-typed" is just an induction hypothesis (preservation) with a property (progress) that describes what we want (e.g., don't do !42)

Motivating type variables

Common motivation: Our simple type system rejects too many programs, requiring code duplication

- If x is bound to `λy.y`, we can give x type `int→ int` or `(ref int)→(ref int)`, but not both
- Recover expressiveness of C casts

More powerful motivation: Abstraction restricts clients

- If f has type `∀α.∀β. ((α→β)*α) → β`, then if f returns a value that value comes from applying its first argument to its second
- The key theory underlying ADTs

Syntax

```
e ::= c | x | λx:τ. e | e e | Λα. e | e [τ]
v ::= λx:τ. e | c | Λα. e
τ ::= int | τ→ τ | α | ∀α. τ
Γ ::= . | Γ, x : τ | Γ, α
```

New:

- Type variables and *universal types*
- Contexts include "what type variables in scope"
- Explicit type *abstraction* and *instantiation*

Semantics

- Left-to-right small-step CBV needs only 1 new primitive reduction

E ::= … | E [τ]

$$\frac{}{(\Lambda\alpha.\ e)\ [\tau] \rightarrow e\{\tau/\alpha\}}$$

- But: must also define `e{τ/α}` (and `τ'{τ/α}`)
  - Much like `e{v/x}` (including capture issues)
  - Λ and ∀ are both bindings (can shadow)
- e.g., `(Λα. Λβ. λ x:α. λ f:α→β. f x) [int] [int] 3 (λ y:int.y+y)`

Typing

- Mostly just be picky: no *free type variables* ever
- Let $\Gamma \vdash \tau$ mean all free type variables are in Γ
  - Rules straightforward and important but boring
- 2 new rules (and 1 picky new premise on old rule)

$$\frac{\Gamma, \alpha \vdash e : \tau}{\Gamma \vdash (\Lambda\alpha.\ e) : \forall\alpha.\tau} \qquad \frac{\Gamma \vdash e : \forall\alpha.\tau1 \quad \Gamma \vdash \tau2}{\Gamma \vdash e\ [\tau2] : \tau1\{\tau2/\alpha\}}$$

- e.g.: $(\Lambda\alpha.\ \Lambda\beta.\ \lambda x{:}\alpha.\ \lambda f{:}\alpha\to\beta.\ f\ x)$ $[\text{int}]\ [\text{int}]\ 3\ (\lambda y{:}\text{int}.y+y)$

Beware mutation

Mutation and abstraction can be surprisingly difficult to reconcile:

Pseudocode example:

let x : ∀α. **ref (ref α)** = ref null
let sr : **string ref = ref "hello"**
(x [string]) := sr
!(x [int]) := 42
print_string (!sr) -- stuck!

Worth walking through on paper

- Can blame any line, presumably line 1 or line 3

The other quantifier

If I want to pass around ADTs, universal quantification is wrong!

Example, an int-set library via a record (like pairs with n fields and field names) of functions

- Want to hold implementation of set abstract with a type including:

```
{ new_set : ()          → α
  add_to  : (α * int) → ()
  union   : (α * α)   → α
  member  : (α * int) → bool }
```

- Clearly unimplementable with ∀α around it

Existentials

Extend our type language with $\exists\alpha.\tau$, and intuitively

```
∃α.{ new_set : ()        → α
     add_to  : (α * int) → ()
     union   : (α * α)   → α
     member  : (α * int) → bool }
```

seems right. But we need:

- New syntax, semantics, typing to make things of this type
- New syntax, semantics, typing to use things of this type

(Just like we did for universal types, but existentials are less well-known)

Making existentials

```
e ::= … | pack τ1,e as ∃α.τ2
E ::= … | pack τ1,E as ∃α.τ2
v ::= … | pack τ1,v as ∃α.τ2
```

(Only new primitive reduction is for using existentials)

$$\frac{\Gamma \vdash e : \tau2\{\tau1/\alpha\}}{\Gamma \vdash (\text{pack}\ \tau1, e\ \text{as}\ \exists\alpha.\tau2) : \exists\alpha.\tau2}$$

Intuition: Create abstraction by hiding a few τ as α, restricting what clients can do with "the package" …

Using existentials

```
e ::= … | unpack x,α=e1 in e2
E ::= … | unpack x,α=E in e2
```

New primitive reduction (intuition; just a let if you ignore the types, the point is stricter type-checking):

$$\frac{}{H;\ \text{unpack}\ x,\alpha = (\text{pack}\ \tau1, v\ \text{as}\ \exists\beta.\tau2)\ \text{in}\ e2 \ \to\ H;\ e2\{v/x\}\{\tau1/\alpha\}}$$

And the all-important typing rule (holds α abstract) :

$$\frac{\Gamma \vdash e1 : \exists\beta.\tau1 \quad \Gamma, \alpha, x{:}\tau1\{\alpha/\beta\} \vdash e2 : \tau \quad \Gamma \vdash \tau}{\Gamma \vdash \text{unpack}\ x,\alpha{=}e1\ \text{in}\ e2 : \tau}$$

Quantified types summary

- Type variables increase code reuse and let programmers define abstractions
- Universals are “generics”
- Existentials are “first-class ADTs”
  - May be new to many of you
  - May make more sense in Cyclone (next time)
  - *More important* in Cyclone
    - Use to encode things like objects and closures, given only code pointers