Calculi and Models for Security

Chiara Bodei, Massimo Bartoletti,
Pierpaolo Degano, Gian-Luigi Ferrari,
and Roberto Zunino

Dipartimento di Informatica,
Università di Pisa, Italy

Pisa, 17-28 Settembre 2007

Outline

The road to λ^{req}:

- pure, untyped λ calculus
- semantics
  - equational theory
  - operational semantics: lazy vs eager
- type systems
  - type checking / type inference
- impure λ calculi: adding side effects
  - operational semantics
  - type and effect systems

The λ calculus

A language to express functions

$f(x) = x + 5$ vs $f = ???$

Notation

$$f = \lambda x.\ x + 5$$

Example 1:

$$g(h) = \bar{g} \text{ where } \bar{g}(x) = h(h(x))$$

vs

$$g = \lambda h.\ \lambda x.\ h(h(x))$$

Example 2:

$$\text{compose} = \lambda f.\ \lambda g.\ \lambda x.\ f(g(x))$$

The λ calculus

The core of all functional programming languages

- Lisp, Haskell, OCaml (and other MLs), Clean, . . .

Sometimes a fragment of λ calculus appears in imperative languages

- Python, ($\sim$ Java), Javascript, . . .

Usual Conventions

"Curried functions"

$$\lambda x\ y.\ \cdots \quad \text{vs} \quad \lambda x.\ \lambda y.\ \cdots$$

Set isomorphism

$$(B \times C) \to A = A^{B \times C}$$
$$\approx$$
$$B \to (C \to A) = (A^C)^B$$

Syntax and Other Conventions

Syntax

$$
\begin{array}{rcll}
e & := & x & \text{variable} \\
 & | & e\ e & \text{application} \\
 & | & \lambda x.\ e & \text{abstraction}
\end{array}
$$

Application notation

$$
\begin{array}{rcl}
f(x) & \text{vs} & f\ x \\
f(g(x)) & \text{vs} & f(g\ x) \\
f(g)(x) & \text{vs} & f\ g\ x
\end{array}
$$

Other Conventions

Application associates to the left

$$f\ x\ y = (f\ x)\ y \neq f\ (x\ y)$$

In set notation, function arrow associates to the right

$$A \to B \to C = A \to (B \to C) \neq (A \to B) \to C$$

This provides a consistent notation

$$\begin{array}{lcr} x & \in & A \\ y & \in & B \\ f & \in & A \to B \to C \\ f\ x & \in & B \to C \\ f\ x\ y & \in & C \end{array}$$

Towards a semantics

What equations should λ terms satisfy?

$$\lambda x.\ x + 6 = \lambda y.\ y + 6$$

Renaming of λ-bound variables (α-conversion)

$$\lambda x.\ e = \lambda y.\ e\{x \mapsto y\} \text{ if } y \notin \mathsf{free}(e)$$

where free means “variables not bound by a λ” and

Example:

$$\lambda x.\ \lambda y.\ x + y \neq \lambda y.\ \lambda y.\ y + y$$

Also, substitution must respect bound variables...

Beta reduction

Usual function definition expansion:

$$(\lambda x.\ x + 4)\ 6 = 6 + 4$$

More generally

$$(\lambda x.\ e_1)\ e_2 = e_1\{x \mapsto e_2\}$$

The above β reduction is the most important equation of the λ calculus

Other Classic Equations

η reduction

$$\lambda x.\ e\ x = e \text{ if } x \notin \mathsf{free}(e)$$

extensionality

$$e_1 = e_2 \text{ if } e_1\ x = e_2\ x \text{ where } x \notin \mathsf{free}(e_1, e_2)$$

The above are actually equivalent axioms

Operational Semantics

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

Usually numbers, booleans, etc. are values too.

Semantics for (closed) λ terms, as a reduction

$$e \longrightarrow \cdots \longrightarrow v$$

If e is a variable, it is not closed
If e is an abstraction, then $e = v$
If e is an application $e_0\ e_1$, ???

Operational Semantics

If e is an application $e_0\ e_1$, we want to apply the β reduction.

$$\frac{e_0 \longrightarrow e_0'}{e_0\ e_1 \longrightarrow e_0'\ e_1}$$

If e_0 terminates, it will reduce to a value, and we can apply β.

Operational Semantics

Two possible β rules:

lazy (Haskell) “just do it”

$$(\lambda x.e_0)\ e_1 \longrightarrow e_0\{x \mapsto e_1\}$$

eager (OCaml) “evaluate the argument before”

$$\frac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'} \qquad \frac{}{(\lambda x.e)\ v \longrightarrow e\{x \mapsto v\}}$$

Considerations: semantics adequacy, performance

Computational Power

The untyped λ calculus is Turing-equivalent

Recursion can be epressed through a fixed point combinator, which is expressible in the language

$$\Theta = (\lambda w.\ \lambda f.\ f(wwf))(\lambda w.\ \lambda f.\ f(wwf))$$

satisfies $\Theta\ f = f(\Theta\ f)$

Example:

$$\mathsf{fact} = \Theta\ (\lambda r.\ \lambda n.\ \cdots(*\ n\ (r\ (-\ n\ 1))\cdots)$$

Types

Type Systems

Simply typed λ calculus

e	$:=$	$\cdots$	
	$\mid$	$\mathsf{true} \mid \mathsf{false}$	booleans
	$\mid$	$0 \mid 1 \mid \cdots$	integers
	$\mid$	$*$	dummy value
	$\mid$	$\mathtt{if}\, b\, \mathtt{then}\, e\, \mathtt{else}\, e$	conditional

Types as approximations of the dynamic behaviour:

τ	$:=$	$\mathsf{unit} \mid \mathsf{int} \mid \mathsf{bool} \mid \cdots$	base types
	$\mid$	$\tau \to \tau$	functional types

Example: $\lambda x.\ x + 5 : \mathsf{int} \to \mathsf{int}$

Types

Below, Γ is a type environment:

$$\Gamma : \text{Variable} \to \text{Type}$$
$$\Gamma = x : \tau; y : \tau'; \cdots$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma \vdash * : \mathsf{unit}}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash e' : \tau}{\Gamma \vdash \mathtt{if}\ b\ \mathtt{then}\ e\ \mathtt{else}\ e' : \tau}$$

Types

$$\frac{\Gamma; x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : \tau \to \tau'}$$

$$\frac{\Gamma \vdash e : \tau \to \tau' \quad \Gamma \vdash e' : \tau}{\Gamma \vdash ee' : \tau'}$$

These rules are actually isomorphic to the rules of intuitionistic logic (by the Curry-Howard isomorphism)

Break?

Computational Power

The simply typed λ calculus is not Turing-equivalent!

Only terminating programs are typeable (Θ is not)

We introduce recursion in the calculus: new β

$$(\lambda_z x.\ e_1)\ e_2 = e_1\{x \mapsto e_2, z \mapsto \lambda_z x.\ e_1\}$$

Lazy and eager semantics are updated accordingly.

Types, with recursion

New rule for the abstraction

$$\frac{\Gamma; x : \tau; z : \tau \to \tau' \vdash e : \tau'}{\Gamma \vdash \lambda_z x.e : \tau \to \tau'}$$

Classical Facts About Types

- type safety
  typeable expressions never go wrong at run time
  (e.g. $(\lambda f.\ f\ 1)4$ is not typeable)
- subject reduction
  types are preserved by reductions
  $e : \tau \wedge e \rightarrow e' \implies e' : \tau$
- type checking algorithms
- type inference algorithms e.g. Hindley-Milner
- more complex types
  polynomial – sum, product –, type recursion
  polymorphic – $(\lambda x.\ x) : \forall \tau.\ \tau \rightarrow \tau$

Impure λ calculi

Events as side effects

Side effect: input, output, communication...
Events

$$\begin{array}{rcll} e & := & \cdots & \\ & | & \alpha & \text{event} \end{array}$$

Intuition: the computation generates a sequence of events (a history η)

$$\begin{array}{rcll} \eta & := & \varepsilon & \text{empty history} \\ & | & \alpha\eta & \text{event} \end{array}$$

So, $\eta \in \{\alpha_1, \alpha_2\}^*$.

Semantics

Reduction has now the form

$$\eta, e \longrightarrow \eta', e'$$

Initially, $\eta = \varepsilon$
Note that $\eta' = \eta\eta''$ for some η''

Here's eager semantics

$$\frac{\eta, e_0 \longrightarrow \eta', e_0'}{\eta, e_0\ e_1 \longrightarrow \eta', e_0'\ e_1} \qquad \frac{\eta, e \longrightarrow \eta', e'}{\eta, v\ e \longrightarrow \eta', v\ e'}$$

$$\frac{}{\eta, \alpha \longrightarrow \eta\alpha, *} \qquad \frac{}{\eta, (\lambda x.e)\ v \longrightarrow \eta, e\{x \mapsto v\}}$$

Note the order of the events (function, argument, body)

Example

We write $e; e'$ for $(\lambda x.\ e')e$ where x is fresh

$$\begin{aligned}\mathsf{fib} = \lambda_f n.\ \ & \texttt{if}\ n = 0\ \texttt{then}\ 0 \\ & \texttt{else if}\ n = 1\ \texttt{then}\ \alpha; 1 \\ & \texttt{else} + (f\ (-\ n\ 1))(f\ (-\ n\ 2))\end{aligned}$$

Returns the n^{th} Fibonacci number, performing the same number of α events

Semantics

Unfortunately, lazy semantics is less clear
Example:

$$(\lambda x.\ e)\ (\alpha; *)$$

How many α are performed?
Depends on how many times x is demanded by e

Semantics

Unfortunately, lazy semantics is less clear
Example:

$$(\lambda x.\ e)\ (\alpha; *)$$

How many α are performed?
Depends on how many times x is demanded by e

Actually, the $e; e'$ contraption does not work if β holds!

$$e; e' = (\lambda x.\ e')e = e'$$

Even with a real $;$ operator, we lose many properties

Better approach: Haskell's `Control.Monad.Writer`

Effects: history expressions

To build a type system for our impure calculus, we need to approximate sets of histories η.

We use history expressions to denote context free languages.

H	$::=$	
	ε	empty history
	α	event
	$H \cdot H'$	sequence
	$H + H'$	choice
	$\mu h.H$	recursion
	h	variable

Example: $H = \alpha_{\mathsf{init}} \cdot \mu h.\ (\varepsilon + \alpha_{(} \cdot h \cdot \alpha_{)})$

Effects: history expressions

Equations for history expressions

Some of them:

$$H + H' = H' + H$$

$$H + H = H$$

$$H \cdot (H' + H'') = H \cdot H' + H \cdot H''$$

$$\mu h.\, H \;=\; H\{h \mapsto \mu h.\, H\}$$

History expressions also have their semantics (CFL)

Types and Effect System

Typing judgement $\Gamma, H \vdash e : \tau$

$$\frac{}{\Gamma, \varepsilon \vdash x : \Gamma(x)} \qquad \frac{}{\Gamma, \varepsilon \vdash * : \mathsf{unit}} \qquad \frac{}{\Gamma, \alpha \vdash \alpha : \mathsf{unit}}$$

$$\frac{\Gamma, H \vdash e : \tau}{\Gamma, H + H' \vdash e : \tau}$$

$$\frac{\Gamma, H \vdash e : \tau \quad \Gamma, H \vdash e' : \tau}{\Gamma, H \vdash \mathtt{if}\ b\ \mathtt{then}\ e\ \mathtt{else}\ e' : \tau}$$

Type and Effect System

$$\frac{\Gamma; x : \tau; z : \tau \xrightarrow{H} \tau', H \vdash e : \tau'}{\Gamma, \varepsilon \vdash \lambda_z x.e : \tau \xrightarrow{H} \tau'}$$

$$\frac{\Gamma, H \vdash e : \tau \xrightarrow{H''} \tau' \quad \Gamma, H' \vdash e' : \tau}{\Gamma, H \cdot H' \cdot H'' \vdash e\ e' : \tau'}$$

Examples

- $e_1 = \texttt{if}\ b\ \texttt{then}\ \alpha\ \texttt{else}\ \alpha'$:unit
  $H_1 = \alpha + \alpha'$
- $e_2 = (\lambda x.\, \alpha')$:unit $\xrightarrow{\alpha'}$ unit
  $H_2 = \varepsilon$
- $e_3 = (\lambda x.\, \alpha')\alpha$: unit
  $H_3 = \alpha \cdot \alpha'$
- $e_4 = (\lambda_z x.\, \alpha; z\, x)$: unit $\xrightarrow{\mu h.\, \alpha \cdot h}$ unit
  $H_4 = \varepsilon$
- $e_5 = (\lambda_z x.\, \alpha; z\, x)*$: unit
  $H_5 = \mu h.\, \alpha \cdot h$

Correctness

Type safety: when $\emptyset, H \vdash e : \tau$

- the computation does not go wrong
- $\varepsilon, e \rightarrow^* \eta, e' \implies \eta \in [\![H]\!]$

So, types and effects correctly over-approximate run-time behaviour

Conclusions

- λ calculus
- pure vs impure
- a type system
- a type and effect system
- how to apply all this to security?
  - (tomorrow)