

# The SLam Calculus

Nevin Heintze and Jon Riecke

Presented by: Sam Tobin-Hochstadt

CSG 399

April 27, 2006

```
(print-to low-security-port
  (if (read high-security)
      #t #f))
```

# BAD!

```
(print-to low-security-port  
  (if (read high-security)  
      #t #f))
```

```
(print-to low-security-port
  (if (read high-security(H,H))
      #t(L,L) #f(L,L) ) )
```

# Rejected by typechecker

```
(print-to low-security-port
  (if (read high-security(H,H))
      #t(L,L) #f(L,L)))
```

**Fundamental Idea:**

**Fundamental Idea:**  
**Add security properties to types**

**Security properties:**

Who can read this data

Who can be influenced by this data

**(r,ir)**

**r**: readers

**ir**: indirect readers

readers can directly inspect an object

indirect readers can be given access to some information about an object

# Security Levels

We will just consider two

**L**

**H**

More complex: unix users and groups, roles, etc

# The Lambda Calculus

Functions: (lambda (x) e)

Variables: **x**

Function Application:  $(e_1 e_2)$

**That's it!**

**But not quite ...**

Pairs: (cons e<sub>1</sub> e<sub>2</sub>)

Projection: (proj<sub>i</sub> e)

Recursive Functions: ( **rec** **f** **e** )

Case: (case e of (inj<sub>1</sub> e<sub>1</sub>) | (inj<sub>2</sub> e<sub>2</sub>))

**Now we add types ...**

**Where do types go?**

Functions: (lambda (x : s) e)

Recursive Functions: (rec f : s e)

**What are types?**

Unit:  $t = ()$

Sum Type: | ( **s** + **s** )

Product Type: | ( s \* s )

Function Type: | ( **s** -> **s** )

**But what was s?**

# Security Properties

Security properties:  $\mathbf{k} = (r, ir)$

Security properties:  $\mathbf{k} = (r, ir)$

Types:  $\mathbf{s} = (t, k)$

**Now we use these in the grammar**

Basic values:

`bv = () | (inji v) | (cons v v) | (lambda (x : s) e)`

Labeled Values:  $v = bv_{(r,ir)}$

Labled Constructors:  $(inj_i e)_{(r, ir)}$

Labeled Destructors:  $(e\ e)_r$

# One More Construct

(protect<sub>ir</sub> e)

(protect<sub>ir</sub> e)

Increases the security level on e

# How Do We Check?

# Constructors:

`(cons e1 e2)k : (s1 * s2 , k)`

$(\text{cons } e_1 \ e_2)_k : (s_1 * s_2 , k)$

provided  $e_1 : s_1$  and  $e_2 : s_2$

$(inj_i e)_k : (s_1 + s_2, k)$

$(\text{inj}_i e)_k : (s_1 + s_2, k)$

provided  $e : s_i$

# **Destructors:**

$$(e_1 \ e_2)_{r^*} : s_2 \bullet ir$$

$$(e_1 \ e_2)_{r^*} : s_2 \bullet ir$$

provided  $e_1 : (s_1 \rightarrow s_2)$  ,  $(r , ir)$  and  $e_2 : s_1$

$$(e_1 \ e_2)_{r^*} : s_2 \bullet ir$$

provided  $e_1 : (s_1 \rightarrow s_2)$  ,  $(r \ , \ ir)$  and  $e_2 : s_1$   
and  $r \leq r^*$

**Wait, what was that •**

s • ir

**s • ir**

means increase the security of **s** to **ir**

**So checking a destructor involves 3 things:**

**So checking a destructor involves 3 things:**

That the basic types match

**So checking a destructor involves 3 things:**

That the basic types match

That the reader has sufficient access to read the value

## **So checking a destructor involves 3 things:**

That the basic types match

That the reader has sufficient access to read the value

That the result has the appropriate indirect security

**And that's it!**

**We can now prove non-interference:**

## **We can now prove non-interference:**

That is, the result of a (low-security) program does not depend on its high-security portions

**Also, we can prove an erasure property:**

**Also, we can prove an erasure property:**

We don't need to do any security checking at runtime

## **Some Weaknesses**

**Nontermination - not considered**

```
(let ([halt-if-true
      (lambda (x : bool(H,H))
        (if x ()(H,H)
            (halt-if-true x))))])
(halt-if-true secret-bool)
#t(L,L))
```

**Timing - not considered**

```
(let* ([t1 : int(L,L) (get-time)]
      [tmp (if secret-bool
                (long-comp)
                (short-comp))])
      [t2 : int(L,L) (get-time)])
  (> (- t2 t1) time-for-short-comp))
```

# Extending the system

# Mutation

# and Concurrency

**and Bears, Oh My!**

Everything is harder with concurrency

**Add reference cells, and spawn to the language**

(box e)

```
(set-box! e1 e2)
```

( unbox e )

( spawn<sub>ir</sub> e )

**Add reference types**

(refs)

**Add an effect system to the type system**

Add latent effects to function types

Basically, what will happen when this function runs

**Also take into account the context of actions**

`(set-box! e1 e2) : s`

in context `ir`

`(set-box! e1 e2) : s`

in context `ir`

if `e1 : (ref s)` and `e2 : s`

`(set-box! e1 e2) : s`

in context `ir`

if `e1 : (ref s)` and `e2 : s`

and `s • ir = s`

```
(let ([halt-if-true
      (lambda (x : bool(H,H))
        (if x ()(H,H)
            (halt-if-true x))))])
(halt-if-true secret-bool)
(set-box! y #t(L,L)))
```

```
(let ([halt-if-true
      (lambda (x : bool(H,H))
        (if x ()(H,H)
            (halt-if-true x))))])
(halt-if-true secret-bool)
(set-box! y #t(L,L)))
```

Only high-security readers can read  $y$

**Non-termination still not fixed**

**No non-interference result**

What is non-interference for parallel systems?

"Timing attacks" are part of the point

# Integrity checking

Adds creators and indirect creators

So now security properties are a 4-tuple

I'll spare you the details

But this system obeys non-interference

# Conclusions

We can apply standard PL techniques to design a secure language

And to prove theorems about it

**Questions?**