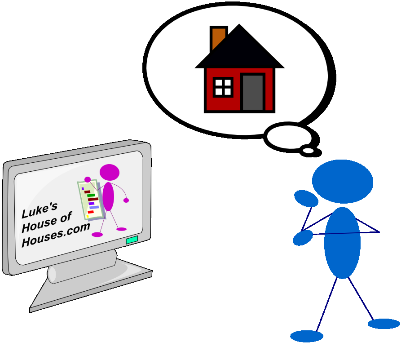
A Monadic Analysis of Information-Flow Security with Mutable State

Enforcing Secrecy with Types

Aleksey Kliger
joint work with
Karl Crary and Frank Pfenning

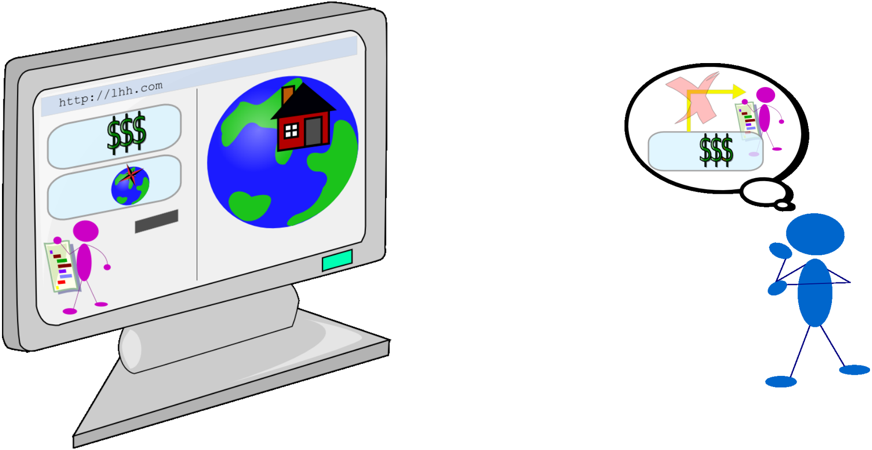
October 21, 2005 / Student Seminar Series

Helen shops for a house

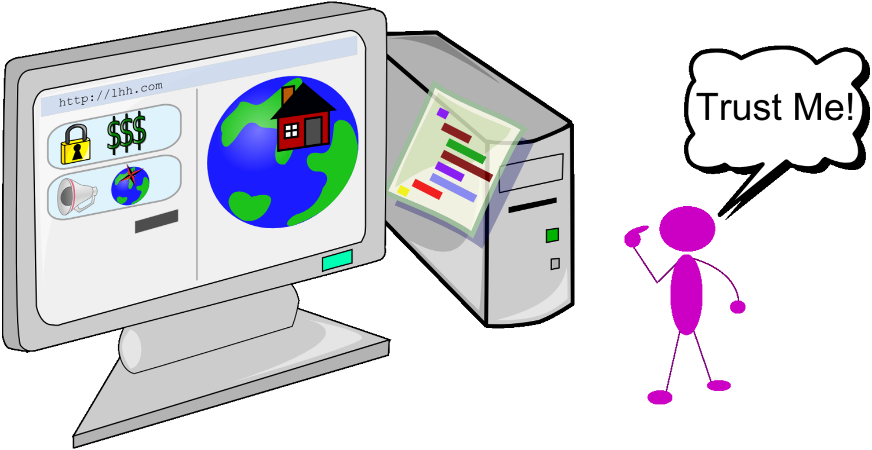


LHH.com

LHH.com



LHH.com



Running untrusted code

Helen wants

A guarantee that Luke cannot learn income

Luke wants

Reasonable burden of proof

Running untrusted code

Language-based information flow security.

Type system tracks flow of information.
Well typed programs do not leak secrets. □

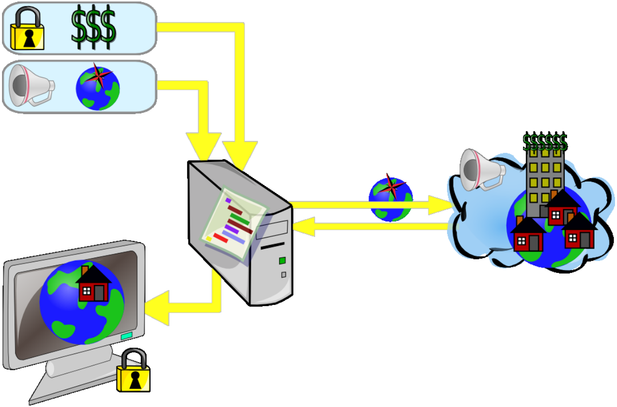
Helen wants

A guarantee that Luke cannot learn income

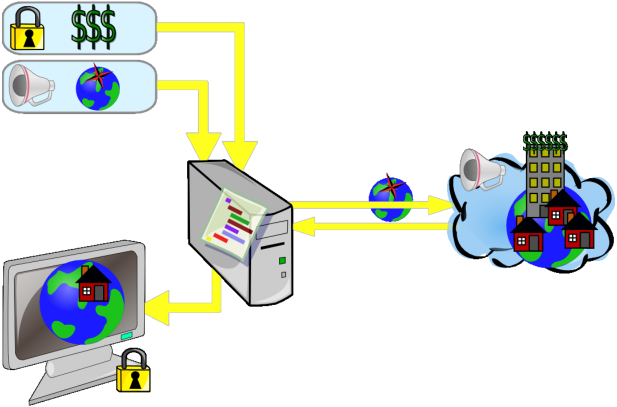
Luke wants

Reasonable burden of proof

Information flow



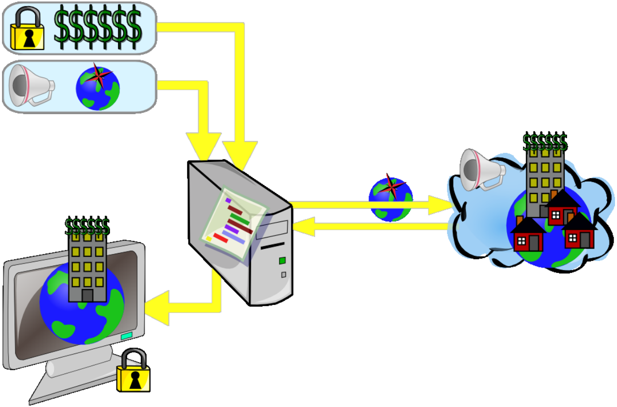
Information flow



Luke thinks

Someone wants a house in Oakland

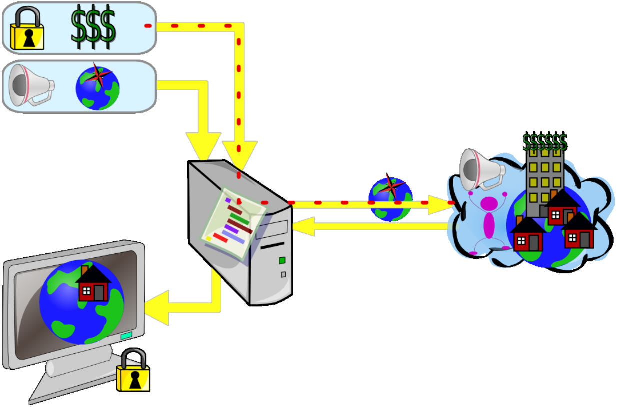
Information flow



Luke thinks

Someone wants a house in Oakland

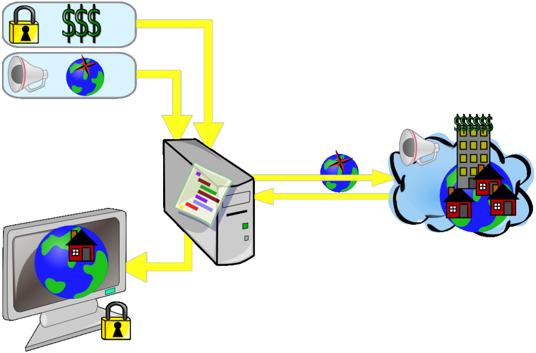
Information flow



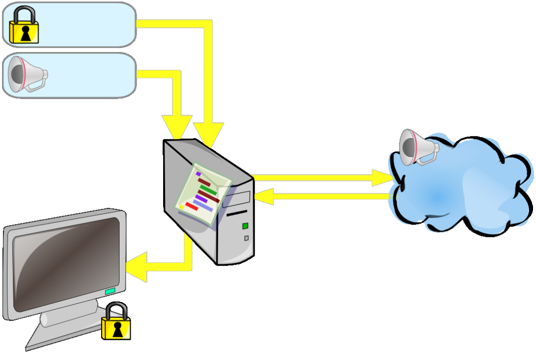
Luke thinks

Someone wants a cheap house in Oakland

Abstracting away



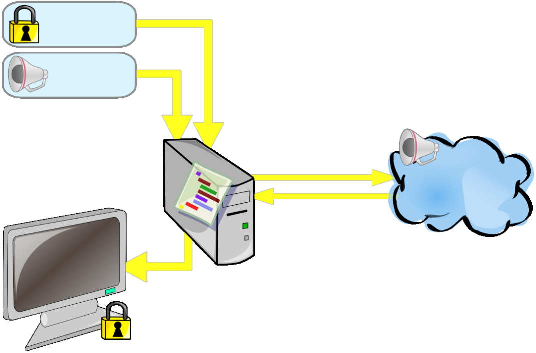
Abstracting away



Situation

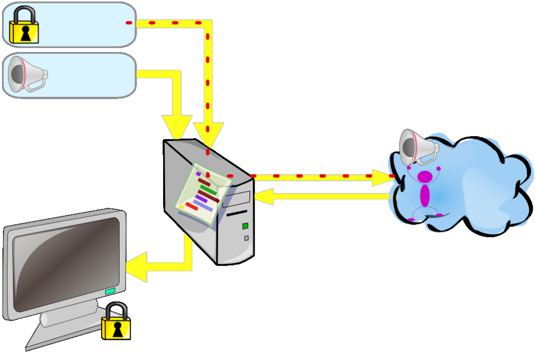
- Untrusted program
- Private data 🔒
- Public data 📢
- Must access both

Abstracting away



Guarantee

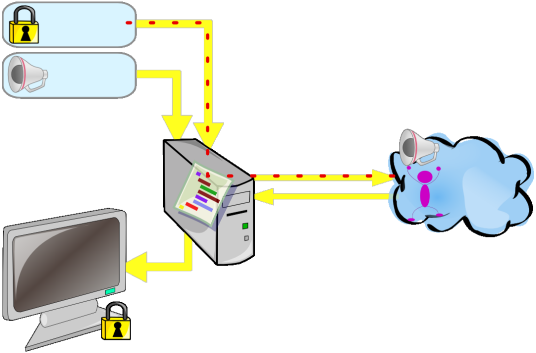
Abstracting away



Guarantee

No information flow of *high*-security data to *low*-security observer.

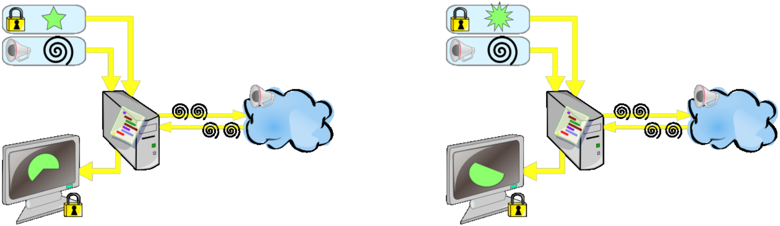
Abstracting away



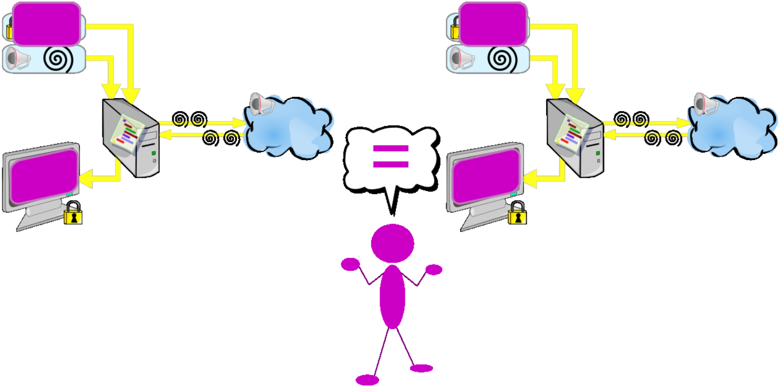
Guarantee

The high-security data does not interfere with the low-security behavior of the program.

Non-interference



Non-interference



Outline

Direct information flow

First try at LHH.com

Example

```
fun processForm () {
  houses📢 :=
    fetchHouses (zipcodeField);
  priceRange🔒 :=
    calcPriceRange (incomeField);
  showJustAffordable (houses,
                      priceRange);
}
```

Direct information flow

First try at LHH.com

Example

```
fun processForm () {
  houses📢 :=
    fetchHouses (zipcodeField);  In: 📢,📢, Out: 📢,📢. ✔
  priceRange🔒 :=
    calcPriceRange (incomeField);
  showJustAffordable (houses,
                      priceRange);
}
```

Direct information flow

First try at LHH.com

Example

```
fun processForm () {
  houses📢 :=
    fetchHouses (zipcodeField);
  priceRange🔒 :=
    calcPriceRange (incomeField);  In: 🔒, Out: 🔒. ✔
  showJustAffordable (houses,
                      priceRange);
}
```

Direct information flow

First try at LHH.com

Example

```
fun processForm () {
  houses📢 :=
    fetchHouses (zipcodeField);
  priceRange🔒 :=
    calcPriceRange (incomeField);
  showJustAffordable (houses,
                      priceRange);  In: 🔒 Out: 🔒.
}
```

Direct information flow

First try at LHH.com

Example

```
fun processForm () {
  houses📢 :=
    fetchHouses (zipcodeField);
  priceRange🔒 :=
    calcPriceRange (incomeField);
  showJustAffordable (houses,
                      priceRange);  In: 📢, 🔒 Out: 🔒. ?
}
```

Direct information flow

First try at LHH.com

Example

```
fun processForm () {
  houses📢
    fetchHo
  priceRan
    calcPriceRange (incomeField);
  showJustAffordable (houses,
                      priceRange);   In: 🔒 Out: 🔒. ✔
}
```

To Combine Inputs: Maximum

$$\max(📢, 🔒) = 🔒$$

Direct information flow

Another try at LHH.com

Example (What if `houses` is secret)

```
fun processForm () {
  houses🔒 :=
    fetchHouses (zipcodeField);
  priceRange🔒 :=
    calcPriceRange (incomeField);
  showJustAffordable (houses,
                      priceRange);
}
```

Direct information flow

Another try at LHH.com

Example (What if `houses` is secret)

```
fun processForm () {
  houses🔒 :=
    fetchHouses (zipcodeField);  In: 📢,📢, Out: 🔒,📢. ?
  priceRange🔒 :=
    calcPriceRange (incomeField);
  showJustAffordable (houses,
                      priceRange);
}
```

Direct information flow

Another try at LHH.com

Example (What if `houses` is secret)

```
fun processForm () {
  houses🔒 :=
    fetchHouses (zipcodeField);   In: 📢, Out: 📢. ✔
  priceRang🔒 :=
    calcPric
  showJustA

}
```

To Combine Outputs: Minimum

$$\min(🔒, 📢) = 📢$$

Direct information flow

Another try at LHH.com

Example (What if `houses` is secret)

```
fun processForm () {
  houses🔒 :=
    fetchHouses (zipcodeField);
  priceRange🔒 :=
    calcPriceRange (incomeField);  In: 🔒, Out: 🔒. ✔
  showJustAffordable (houses,
                      priceRange);
}
```

Direct information flow

Another try at LHH.com

Example (What if `houses` is secret)

```
fun processForm () {
  houses🔒 :=
    fetchHouses (zipcodeField);
  priceRange🔒 :=
    calcPriceRange (incomeField);
  showJustAffordable (houses,
                      priceRange);   In: 🔒, 🔒 Out: 🔒. ✔
}
```

Indirect information flow

Example (LHH.com "I'm Feeling Lucky")

1. Fetch a random house in the zipcode
2. If not in the price range, go back to (1)
3. Otherwise show the house

Indirect information flow

Example (Simplified)

```
randHouse📢 :=
  fetchRandom (zipcodeField);
if (not (isInPriceRange
          (randHouse,
           priceRange🔒)))
  then
    randHouse📢 :=
        fetchRandom
          (zipCodeField);
showHouse (randHouse)
```

Interaction

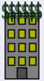
Indirect information flow

Example (Simplified)

```
randHouse📢 :=
  fetchRandom (zipcodeField);
if (not (isInPriceRange
          (randHouse,
           priceRange🔒)))
  then
    randHouse📢 :=
        fetchRandom
          (zipCodeField);
showHouse (randHouse)
```

Interaction

⇒ “fetchRandom 15213”

⇐ 

Indirect information flow

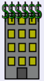
Example (Simplified)

```
randHouse📢 :=
  fetchRandom (zipcodeField);
if (not (isInPriceRange
          (randHouse,
           priceRange🔒)))
  then
    randHouse📢 :=
        fetchRandom
          (zipCodeField);
showHouse (randHouse)
```

Interaction

⇒ “fetchRandom 15213”

⇐



Indirect information flow

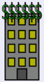
Example (Simplified)

```
randHouse📢 :=
  fetchRandom (zipcodeField);
if (not (isInPriceRange
           (randHouse,
            priceRange🔒)))
  then
    randHouse📢 :=
        fetchRandom
          (zipCodeField);
showHouse (randHouse)
```

Interaction

⇒ “fetchRandom 15213”

⇐



⇒ “fetchRandom 15213”

Luke: $$$

Indirect information flow

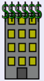
Example (Simplified)

```
randHouse📢 :=
  fetchRandom (zipcodeField);
if (not (isInPriceRange
            (randHouse,
             priceRange🔒)))
  then
    randHouse📢 :=
        fetchRandom
          (zipCodeField);
showHouse (randHouse)
```

Interaction

⇒ "fetchRandom 15213"

⇐



Indirect information flow

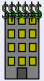
Example (Simplified)

```
randHouse📢 :=
  fetchRandom (zipcodeField);
if (not (isInPriceRange
            (randHouse,
             priceRange🔒)))
  then
    randHouse📢 :=
        fetchRandom
           (zipCodeField);
showHouse (randHouse)
```

Interaction

⇒ "fetchRandom 15213"

⇐



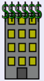
Indirect information flow

Example (Simplified)

```
randHouse🔊 :=
  fetchRandom (zipcodeField);
if (not (isInPriceRange
           (randHouse,
            priceRange🔒)))
  then
    randHouse🔊 :=
        fetchRandom
           (zipCodeField);
showHouse (randHouse)
```

Interaction

⇒ “fetchRandom 15213”

⇐ 

⇒ nothing

Luke: $$$$$$

Indirect information flow, cont'd

Example (Simplified)

```
if (not (isInPriceRange
            (randHouse,
             priceRange🔒)))
  then
    randHouse📢 :=
        fetchRandom
           (zipCodeField);
```

Principle

Security level of a conditional is an *implicit input* to the branches.

Example

In: 📢, 📢, 🔒 *Out:* 📢, 📢 ✗

Outline

Why Monads?

Problems with traditional languages

1. Every expression may have an *effect* (I/O, memory read/write) and this is not reflected in the types
2. Complicated control flow — indirect information flow

A monadic language for security

Definition (Monad)

A value of type $\bigcirc A$ is a suspended computation, that *each time it is forced to execute* will

- return a (possibly different) result of type A
- potentially produce some effects

Note

And expressions of all other types are *pure* (do not have effects).

A monadic language for security

Definition (Monad)

A value of type $\bigcirc A$ is a suspended computation, that *each time it is forced to execute* will

- return a (possibly different) result of type A
- potentially produce some effects

Forcing suspended computations

No way to force $\bigcirc A$, to get A. Only build up larger composite computations.

Runtime forces $\texttt{main} : \bigcirc()$.

Writing programs with monads

Values of $\bigcirc A$

```
do
  stmt_1
  stmt_2
  ...
  expr
```

Example (increment)

$inc : (\mathbf{Ptr}\ Int,\ Int) \rightarrow \bigcirc Int$

```
fun inc (ptr, amt) =
do
  ...
```

Writing programs with monads

Statements

`var ← expr`

Expressions

do `stmts`

deref `expr`

Example (increment)

$inc : (\textbf{Ptr}\ Int,\ Int) \rightarrow \bigcirc Int$

```
fun inc (ptr, amt) =
do
  oldVal ← deref ptr
  ...
```

Writing programs with monads

Statements

```
var ← expr
expr
```

Expressions

do `stmts`

deref `expr`

`expr := expr`

Example (increment)

$inc : (\mathbf{Ptr}\ Int,\ Int) \rightarrow \bigcirc Int$

fun *inc* `(ptr, amt) =`

do

```
  oldVal ← deref ptr
  ptr := oldVal + amt
  ...
```

Writing programs with monads

Statements

```
var ← expr
expr
```

Expressions

```
do stmts
deref expr
expr := expr
pure expr
```

Example (increment)

```
inc : (Ptr Int, Int)
        → ◯Int
fun inc (ptr, amt) =
do
   oldVal ← deref ptr
   ptr := oldVal + amt
   pure oldVal
```

Writing programs with monads

Statements

```
var ← expr
expr
```

Expressions

```
do stmts
deref expr
expr := expr
pure expr
```

```
var
if (expr)
   then expr
   else expr
func (args)
```

Example (increment)

```
inc : (Ptr Int, Int)
        → ◯Int
fun inc (ptr, amt) =
do
   oldVal ← deref ptr
   ptr := oldVal + amt
   pure oldVal
```

Control flow and monads

Example

```
repeatUntil :
    $(A \to \mathbf{Bool},\ \bigcirc A) \to \bigcirc A$
fun repeatUntil
      (test, comp) =
do
  x ← comp
  if (test (x))
    then pure x
    else repeatUntil
            (test, comp)
```

Control flow and monads

Example

```
repeatUntil :
    $(A \to \mathbf{Bool},\ \bigcirc A) \to \bigcirc A$
fun repeatUntil
     (test, comp) =
do
  x ← comp
  if (test (x))
    then pure x
    else repeatUntil
           (test, comp)
```

Example (Sample Output)

```
State = {ptr ↦ 0}

repeatUntil (above100,
               inc (ptr, 1))
```

Control flow and monads

Example

```
repeatUntil :
    (A → Bool,  ○A) → ○A
fun repeatUntil
     (test, comp) =
do
  x ← comp
  if (test (x))
    then pure x
    else repeatUntil
           (test, comp)
```

Example (Sample Output)

```
State = {ptr ↦ 0}

do
  x ← inc (ptr, 1)
  if (above100 (x))
    then pure x
    else repeatUntil
            (above100,
             inc(ptr, 1))
```

Control flow and monads

Example

```
repeatUntil :
     $(A \to \mathbf{Bool},\ \bigcirc A) \to \bigcirc A$
fun repeatUntil
      (test, comp) =
do
  x ← comp
  if (test (x))
    then pure x
    else repeatUntil
             (test, comp)
```

Example (Sample Output)

$\text{State} = \{\texttt{ptr} \mapsto 0\}$

```
do
⇒ x ← inc (ptr, 1)
  if (above100 (x))
    then pure x
    else repeatUntil
             (above100,
              inc(ptr, 1))
```

Control flow and monads

Example

```
repeatUntil :
      (A → Bool, ◯A) → ◯A
fun repeatUntil
      (test, comp) =
do
  x ← comp
  if (test (x))
    then pure x
    else repeatUntil
           (test, comp)
```

Example (Sample Output)

```
State = {ptr ↦ 1}

do
   x ← inc (ptr, 1)
⇒ if (above100 (0))
     then pure 0
     else repeatUntil
            (above100,
             inc(ptr, 1))
```

Control flow and monads

Example

```
repeatUntil :
    (A → Bool,  ◯A) → ◯A
fun repeatUntil
     (test, comp) =
do
  x ← comp
  if (test (x))
    then pure x
    else repeatUntil
           (test, comp)
```

Example (Sample Output)

State = {ptr ↦ 1}

```
if (above100 (0))
  then pure 0
  else repeatUntil
        (above100,
         inc (ptr, 1))
```

Control flow and monads

Example

```
repeatUntil :
    $(A \to \mathbf{Bool},\ \bigcirc A) \to \bigcirc A$
fun repeatUntil
     (test, comp) =
do
  x ← comp
  if (test (x))
    then pure x
    else repeatUntil
           (test, comp)
```

Example (Sample Output)

State $= \{\texttt{ptr} \mapsto 1\}$

```
repeatUntil
  (above100,
   inc (ptr, 1))
```

Control flow and monads

Example

```
$repeatUntil$ :
    $(A \to \mathbf{Bool},\ \bigcirc A) \to \bigcirc A$
fun $repeatUntil$
      (test, comp) =
do
  x ← comp
  if ($test$ (x))
    then pure x
    else $repeatUntil$
           (test, comp)
```

Example (Sample Output)

$\text{State} = \{\texttt{ptr} \mapsto 1\}$

```
$repeatUntil$
  ($above100$,
   $inc$ (ptr, 1))
```

Monadic version of LHH.com

Original version

```
fun processForm () {
  houses📢 := fetchHouses (zipcodeField);
  priceRange🔒 := calcPriceRange (incomeField);
  showJustAffordable (houses, priceRange);
}
```

Monadic version of LHH.com

Monadic Version

```
fun processForm () =
do
  houses📢 := fetchHouses (zipcodeField)
  priceRange🔒 := calcPriceRange (incomeField)
  showJustAffordable (houses, priceRange)
```

Monadic version of LHH.com

Types

$\mathit{fetchHouses} : \mathbf{Ptr}\ \mathit{ZipCode} \to \bigcirc\ \mathbf{List}\ \mathit{House}$

Monadic Version

```
fun processForm () =
do
  houses📢 := fetchHouses (zipcodeField)
  priceRange🔒 := calcPriceRange (incomeField)
  showJustAffordable (houses, priceRange)
```

Monadic version of LHH.com

Types

Monadic Version

```
fun processForm () =
do
  h ← fetchHouses (zipcodeField)
  houses📢 := h
  priceRange🔒 := calcPriceRange (incomeField)
  showJustAffordable (houses, priceRange)
```

Monadic version of LHH.com

Types

$calcPriceRange : Int \longrightarrow Range$

Monadic Version

```
fun processForm () =
do
  h ← fetchHouses (zipcodeField)
  houses📢 := h
  priceRange🔒 := calcPriceRange (incomeField)
  showJustAffordable (houses, priceRange)
```

Monadic version of LHH.com

Types

Monadic Version

```
fun processForm () =
do
  h ← fetchHouses (zipcodeField)
  houses📢 := h
  i ← deref incomeField
  priceRange🔒 := calcPriceRange (i)
  showJustAffordable (houses, priceRange)
```

Monadic version of LHH.com

Types

$showJustAffordable : (\textbf{Ptr List}\ House, \textbf{Ptr}\ Range) \to \bigcirc()$

Monadic Version

```
fun processForm () =
do
  h ← fetchHouses (zipcodeField)
  houses🔊 := h
  i ← deref incomeField
  priceRange🔒 := calcPriceRange (i)
  showJustAffordable (houses, priceRange)
```

Idiomatic monadic version of LHH.com

Types

$\mathit{showJustAffordable} : (\mathbf{List}\ \mathit{House}, \mathit{Range}) \rightarrow \bigcirc()$

Simplified

```
fun processForm () =
do
  h ← fetchHouses (zipcodeField)
  i ← deref incomeField
  showJustAffordable (h, calcPriceRange (i))
```

Idiomatic monadic version of LHH.com

Simplified

```
fun processForm () =
do
  h ← fetchHouses (zipcodeField)
  do
    i ← deref incomeField
    showJustAffordable (h, calcPriceRange (i))
```

So what about security?

Idea

Track security levels of input and output in the monad: $\bigcirc_{\text{📢},\text{🔒}} A$

LHH.com with security monads

Types

Code unchanged

```
fun processForm () =
do
  h ← fetchHouses (zipcodeField)
  do
    i ← deref incomeField
    showJustAffordable (h, calcPriceRange (i))
```

LHH.com with security monads

Types

fetchHouses : **Ptr**$_{📢}$ *ZipCode* → $\bigcirc_{📢,📢}$ **List** *House*

Code unchanged

```
fun processForm () =
do
  h ← fetchHouses (zipcodeField)
  do
    i ← deref incomeField
    showJustAffordable (h, calcPriceRange (i))
```

LHH.com with security monads

Types

deref `incomeField` : $\bigcirc_{\text{🔒},\bullet}$ *Int*

Code unchanged

```
fun processForm () =
do
  h ← fetchHouses (zipcodeField)
  do
    i ← deref incomeField
    showJustAffordable (h, calcPriceRange (i))
```

LHH.com with security monads

Types

$showJustAffordable : (\textbf{List}\ House, Range) \rightarrow \bigcirc_{\bullet,\text{🔒}}()$

Code unchanged

```
fun processForm () =
do
  h ← fetchHouses (zipcodeField)
  do
    i ← deref incomeField
    showJustAffordable (h, calcPriceRange (i))
```

Composing suspensions

Portion of `processForm`

do

  `i` ← **deref** `incomeField` : $\bigcirc_{🔒,\bullet}\ \mathit{Int}$

  showJustAffordable

    `(h,` *calcPriceRange* `(i))` : $\bigcirc_{\bullet,🔒}\ ()$

Composing suspensions

Portion of `processForm`

do

  `i` ← **deref** `incomeField` : $\bigcirc_{🔒,\bullet}$ *Int*

  showJustAffordable

    `(h,` *calcPriceRange* `(i))` : $\bigcirc_{\bullet,🔒}$ ()

: $\bigcirc_{🔒,🔒}$ ()

Composing suspensions, cont'd

`processForm`

do

h ← *fetchHouses*

(zipcodeField) : $\bigcirc_{\text{📢},\text{📢}}$ **List** *House*

do . . . : $\bigcirc_{\text{🔒},\text{🔒}}$ ()

Composing suspensions, cont'd

Compose Inputs

max(📢, 🔒) = 🔒

Compose Outputs

min(📢, 🔒) = 📢

processForm

do
  h ← *fetchHouses*
      `(zipcodeField)` : $\bigcirc_{\text{📢},\text{📢}}$ **List** *House*
  do . . . : $\bigcirc_{\text{🔒},\text{🔒}}$ ()
: $\bigcirc_{\text{🔒},\text{📢}}$ ()

What are the types telling us?

Suppose `showJustAffordable` had type

$(\textbf{List}\ \mathit{House}, \mathit{Range}) \to \bigcirc_{\bullet,\text{🔒}} \mathit{Int}$

Consider

```
do
  nHouses ← do
              i ← deref incomeField
              showJustAffordable (...)
              : ○_{🔒,🔒} Int
  sendL (nHouses)      : ○_{•,📢} ()
```

() is not informative

Definition

A type A is not informative at low-security
if no computation with only low input could make use of it

Theorem

Non-interference is preserved if for any A that is not informative at low-security, we promote $\bigcirc_{\text{🔒},\text{🔒}} A$ to $\bigcirc_{\bullet,\text{🔒}} A$

`processForm` is well-typed

Promote the inner do-block

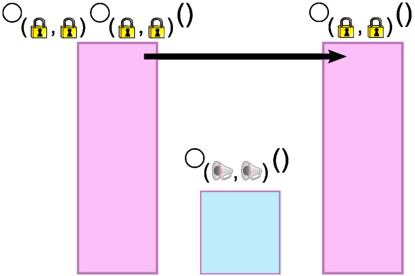
```
fun processForm () =
do
  h ← fetchHouses (zipcodeField) : ◯_{📢,📢} List House
  do
    i ← deref incomeField
    showJustAffordable (...)
  : ◯_{●,🔒} ()
: ◯_{📢,📢} ()
```

Other non-informative types?

Not informative for low security

- $A \longrightarrow B$ not informative, whenever B isn't
- $\mathbf{Ptr}_{\text{🔒}} A$ is not informative
- $\bigcirc_{\text{🔒},\text{🔒}} A$ is not informative, whenever A isn't

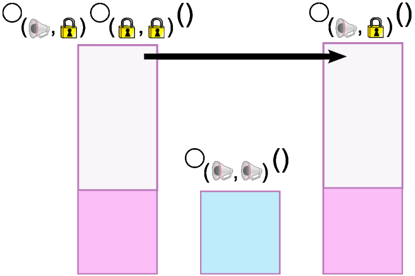
Secure computation continuation passing



Example

```
do
    cont ← highComp
    lowComp
    cont
```

Secure computation continuation passing

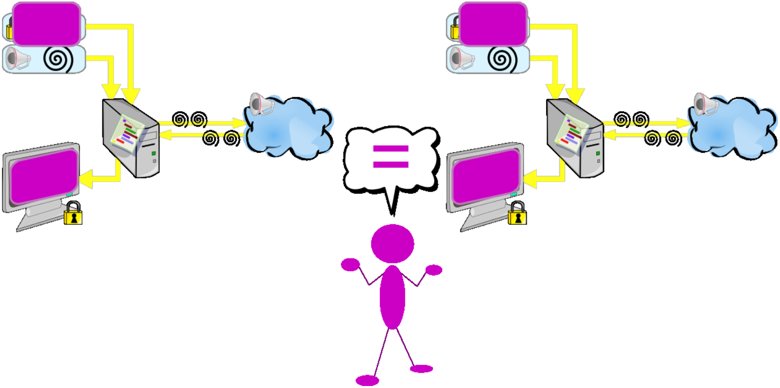


Example

```
do
    cont ← highComp
    lowComp
    cont
```

Is it still secure?

Yes



Outline

Related work

Monads in language design

Foundations

- Semantics [Moggi 1989]

Have been used for:

- I/O in Haskell [Peyton-Jones, *et al.* 1993]
- Parsing Combinators [Wadler 1992]
- Composable Transactional Memory [Peyton-Jones, *et al.* 2005]
- Probabilistic Computation [Park *et al.* 2005]
- and much more...

Related work
Language-based security

Non-interference:

- [Volpano, *et al.* 1996]
- SLam [Heintze, *et al.* 1998], DCC [Abadi *et al.* 1999]
- CoreML^2 [Pottier, *et al.* 2003]
- and many others ... [Sabelfeld, *et al.* 2003]

Extensions:

- Concurrency [Honda, *et al.* 2002]
- Timing channels [Agat 2000]
- Robust declassification [Zdancewic, *et al.* 2001], [Myers, *et al.* 2004]
- and many others...

Our contributions

- Monads for tracking information flow
- Informativeness

Monads for tracking information flow

Isolate security concerns: simplify reasoning
Only monads and channels are tagged

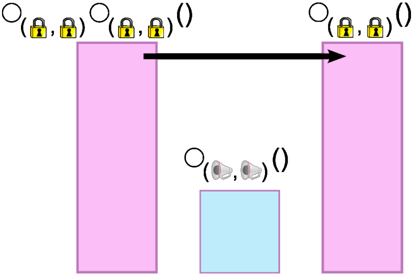
Example

$\mathit{fetchHouses} : \mathbf{Ptr}_{\text{📢}}\ \mathit{ZipCode} \to \bigcirc_{\text{📢},\text{📢}} \mathbf{List}\ \mathit{House}$

$\mathit{calcPriceRange} : \mathit{Int} \to \mathit{Range}$

Informativeness

Allow high-security computations to pass temporary results through low-security computations

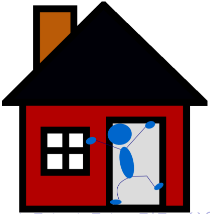


Future work

Push out	Concurrency (*e.g.* transactional memory), robust declassification
Push down	Security-Typed Assembly Language

Thanks

Questions?



The three monad laws

Program equivalences

Within each security level, we obey the monad laws
(Upcalls are an additional relationship between monad families)

1.
```
do x ← pure expr
   func (x)
```
2.
```
do x ← comp
   pure x
```
3.
```
do y ← do
         x ← comp1
         comp2 (x)
   comp3 (y)
```

1.
```
do
   func (expr)
```
2.
```
do
   comp
```
3.
```
do
   x ← comp1
   y ← comp2 (x)
   comp3 (y)
```

Differences in the paper

Additions in the paper

- Full lattice of security levels
- A proof of non-interference
- Discussion of allocation
- Encoding of a prior work

Stylistic differences

- No $\bullet$ for "no interesting reads/writes," use $\bot, \top$
- "A informative only above 🔒"
  vs. "A not informative for 📢"
- Pfenning-style expressions *vs.* **do**-notation