Position: Lightweight static resources

Sexy types for embedded and systems programming

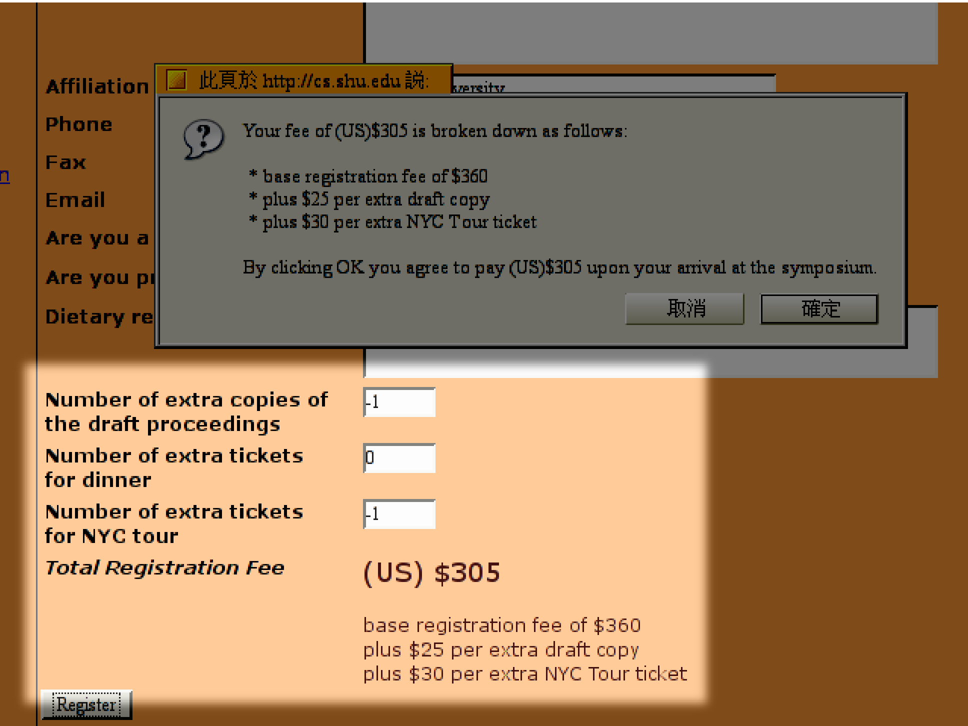
Oleg Kiselyov (FNMOC)
Chung-chieh Shan (Rutgers University)

TFP 2007

Types provide static assurances

"Well-typed programs don't go wrong."


SNOWMASS VILLAGE
ESTABLISHED 1967
ELEVATION 8368
POPULATION 1822
TOTAL 12,157
TOWN
MAPS


Affiliation
Phone
Fax
Email
Are you a
Are you pr
Dietary re
此頁於 http://cs.shu.edu 說:
Your fee of (US)$305 is broken down as follows:
* base registration fee of $360
* plus $25 per extra draft copy
* plus $30 per extra NYC Tour ticket
By clicking OK you agree to pay (US)$305 upon your arrival at the symposium.
取消
確定
Number of extra copies of the draft proceedings
-1
Number of extra tickets for dinner
0
Number of extra tickets for NYC tour
-1
Total Registration Fee
(US) $305
base registration fee of $360
plus $25 per extra draft copy
plus $30 per extra NYC Tour ticket
Register

@RISK: The Consensus Security Vulnerability Alert
Vol. 6 No. 14

"A really bad week." That's what the @RISK editor and Tippingpoint vulnerability researcher, Rohit Dhamankar wrote to us this morning. And the director of the Internet Storm Center, Johannes Ullrich readily agreed. Why?

Two zero-day vulnerabilities. Active exploits. No effective defenses. Windows had a zero-day that affects Vista as well as older versions. So important that Microsoft is issuing a special patch tomorrow and leaked it to a few folks today. The other zero-day hit CA's BrightStor. Holes in backup software may be more damaging than holes in operating systems because the vendors of backup software don't have the same level of automating patching that the operating system vendors have, and many users have *never* patched their backup software. And Lotus Domino users also had multiple vulnerabilities, some critical.

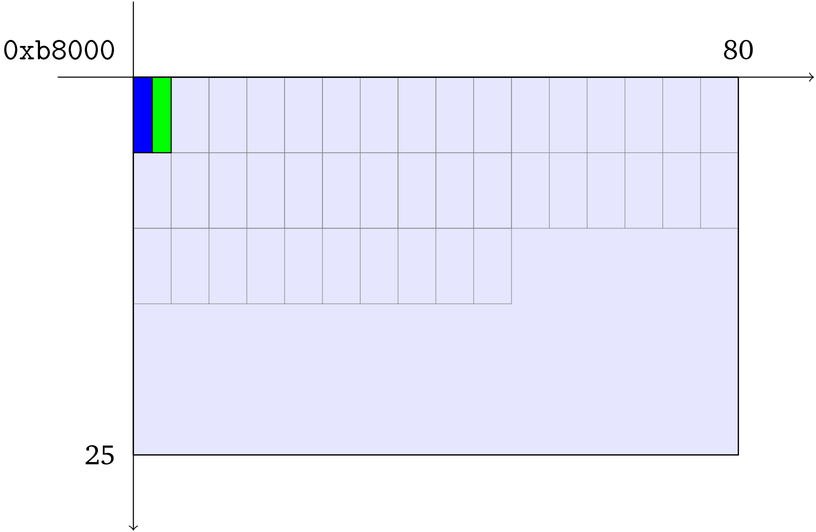
Alan

Types provide static assurances

"Well-typed programs don't go wrong."

- Express more safety properties in a general-purpose language (Haskell "sexy types")
- Showcase: embedded and systems programming
  - A safer and faster interface to raw hardware
  - Code generation
- Resource-aware programming
- Types are static capabilities

Video RAM example (Diatchki & Jones)



Video RAM example (Diatchki & Jones)

```
type Screen     = Array  25 (Array  80 ScreenChar)
type ScreenChar  = Pair (Stored Byte) (Stored Byte)
```

```
25 :: Nat
```

Video RAM example (Diatchki & Jones)

```
type ScreenT = Array N25 (Array N80 ScreenCharT)
type ScreenCharT = Pair AWord8 AWord8
```

```
N25 :: *
instance Nat N25
```

Video RAM example (Diatchki & Jones)

```
type ScreenT = Array N25 (Array N80 ScreenCharT)
type ScreenCharT = Pair AWord8 AWord8

area videoRAM = 0xb8000 :: Ref Screen
```

```
Screen :: Area

Ref :: Area -> *
```

Video RAM example (Diatchki & Jones)

```
type ScreenT = Array N25 (Array N80 ScreenCharT)
type ScreenCharT = Pair AWord8 AWord8

data ScreenAbs = ScreenAbs
instance Property ScreenAbs APInHeap HFalse
instance Property ScreenAbs APARef (ARef N8 ScreenT)
instance Property ScreenAbs APReadOnly HFalse
instance Property ScreenAbs APOverlayOK HTrue
instance Property ScreenAbs APFixedAddr HTrue
videoRAM = area_at ScreenAbs
           (nullPtr `plusPtr` 0xb8000)
```

Video RAM example (Diatchki & Jones)

```
type ScreenT = Array N25 (Array N80 ScreenCharT)
type ScreenCharT = Pair AWord8 AWord8

data ScreenAbs = ScreenAbs
instance Property ScreenAbs APInHeap HFalse
instance Property ScreenAbs APARef (ARef N8 ScreenT)
instance Property ScreenAbs APReadOnly HFalse
instance Property ScreenAbs APOverlayOK HTrue
instance Property ScreenAbs APFixedAddr HTrue
videoRAM = area_at ScreenAbs
           (nullPtr `plusPtr` 0xb8000)
```

- `:type videoRAM`
  `ARef N8 (AtArea ScreenAbs ScreenT)`

Video RAM example (Diatchki & Jones)

```
type ScreenT = Array N25 (Array N80 ScreenCharT)
type ScreenCharT = Pair AWord8 AWord8

data ScreenAbs = ScreenAbs
instance Property ScreenAbs APInHeap HFalse
instance Property ScreenAbs APARef (ARef N8 ScreenT)
instance Property ScreenAbs APReadOnly HFalse
instance Property ScreenAbs APOverlayOK HTrue
instance Property ScreenAbs APFixedAddr HTrue
videoRAM = area_at ScreenAbs
           (nullPtr `plusPtr` 0xb8000)
```

►
```
:type size_of (aref_area videoRAM) -- undefined
U (U (U (U (U (U (U (U (U (U (U
B1 B1) B1) B1) B1) B0) B1) B0) B0) B0) B0) B0
```

Video RAM example (Diatchki & Jones)

```
type ScreenT = Array N25 (Array N80 ScreenCharT)
type ScreenCharT = Pair AWord8 AWord8

data ScreenAbs = ScreenAbs
instance Property ScreenAbs APInHeap HFalse
instance Property ScreenAbs APARef (ARef N8 ScreenT)
instance Property ScreenAbs APReadOnly HFalse
instance Property ScreenAbs APOverlayOK HTrue
instance Property ScreenAbs APFixedAddr HTrue
videoRAM = area_at ScreenAbs
             (nullPtr `plusPtr` 0xb8000)
```

- `:type size_of (aref_area videoRAM) -- undefined`

```
size_of :: SizeOf area n => area -> n
size_of = undefined
```

Video RAM example (Diatchki & Jones)

```
attrAt i j = afst (videoRAM @@ i @@ j)
charAt i j = asnd (videoRAM @@ i @@ j)
```

Video RAM example (Diatchki & Jones)

```
attrAt i j = afst (videoRAM @@ i @@ j)
charAt i j = asnd (videoRAM @@ i @@ j)
```

- `:type attrAt`
  ```
  Ix N25 -> Ix N80 ->
  ARef (U B1 B0) (AtArea ScreenAbs AWord8)
  ```

- `:type charAt`
  ```
  Ix N25 -> Ix N80 ->
  ARef B1 (AtArea ScreenAbs AWord8)
  ```

Video RAM example (Diatchki & Jones)

```
attrAt i j = afst (videoRAM @@ i @@ j)
charAt i j = asnd (videoRAM @@ i @@ j)
```

- `:type attrAt`
  ```
  Ix N25 -> Ix N80 ->
  ARef (U B1 B0) (AtArea ScreenAbs AWord8)
  ```
- `:type charAt`
  ```
  Ix N25 -> Ix N80 ->
  ARef B1 (AtArea ScreenAbs AWord8)
  ```
- `:type (@@)`
  ```
  (INDEXABLE arr count base totalsize,
   GCD al n z, SizeOf base n) =>
  ARef al arr -> Ix count -> ARef z base
  ```

Types provide static assurances

"Well-typed programs don't go wrong."

- Express more safety properties in a general-purpose language (Haskell "sexy types")
- Showcase: embedded and systems programming
  - A safer and faster interface to raw hardware
  - Code generation
- Resource-aware programming
  - Custom kinds and predicates as type classes
  - Type computation using functional dependencies
  - Low notation overhead; "pay as you go"
- Types are static capabilities
  - Assure safety properties, not full correctness
  - Extend trust from small kernel to large sandbox

Custom kinds and predicates as type classes

- ```
  :type (@@)
  (INDEXABLE arr count base totalsize,
   GCD al n z, SizeOf base n) =>
  ARef al arr -> Ix count -> ARef z base
  ```
  ```

Custom kinds and predicates as type classes

- ```
  :type (@@)
  (INDEXABLE arr count base totalsize,
   GCD al n z, SizeOf base n) =>
  ARef al arr -> Ix count -> ARef z base

  class Nat0 a where toInt :: a -> Int
  class (Nat0 x, Nat0 y, Nat0 z) => GCD x y z
  ```
  ```

Type computation using functional dependencies

- ```
  :type (@@)
  (INDEXABLE arr count base totalsize,
   GCD al n z, SizeOf base n) =>
  ARef al arr -> Ix count -> ARef z base

  class Nat0 a where toInt :: a -> Int
  class (Nat0 x, Nat0 y, Nat0 z) => GCD x y z
                                   | x y -> z
  ```
- ```
  :type gcd (pred (pred nat8)) nat8 -- undefined
  U B1 B0
  ```

  Term notation for static computations: less scary?

Types are static capabilities: kernel

```
:type attrAt
Ix N25 -> Ix N80 ->
ARef (U B1 B0) (AtArea ScreenAbs AWord8)
```

Capabilities guard access to resources.

Types are static capabilities: kernel

```
:type attrAt
Ix N25 -> Ix N80 ->
ARef (U B1 B0) (AtArea ScreenAbs AWord8)
```

Capabilities guard access to resources.

- ```
  minBound :: Ix N8
  Ix 0
  ```
  ```

Types are static capabilities: kernel

```
:type attrAt
Ix N25 -> Ix N80 ->
ARef (U B1 B0) (AtArea ScreenAbs AWord8)
```

Capabilities guard access to resources.

- `minBound :: Ix N8`
  `Ix 0`
- `maxBound :: Ix N8`
  `Ix 7`

Types are static capabilities: kernel

```
:type attrAt
Ix N25 -> Ix N80 ->
ARef (U B1 B0) (AtArea ScreenAbs AWord8)
```

Capabilities guard access to resources.

- `minBound :: Ix N8`
  `Ix 0`
- `maxBound :: Ix N8`
  `Ix 7`
- `ixSucc (maxBound :: Ix N8)`
  `IxPlus 8 :: IxPlus`

Types are static capabilities: kernel

```
:type attrAt
Ix N25 -> Ix N80 ->
ARef (U B1 B0) (AtArea ScreenAbs AWord8)
```

Capabilities guard access to resources.

- `minBound :: Ix N8`
  `Ix 0`
- `maxBound :: Ix N8`
  `Ix 7`
- `ixSucc (maxBound :: Ix N8)`
  `IxPlus 8 :: IxPlus`
- `ixPred (maxBound :: Ix N8)`
  `IxMinus 6 :: IxMinus N8`

Types are static capabilities: kernel

```
:type attrAt
Ix N25 -> Ix N80 ->
ARef (U B1 B0) (AtArea ScreenAbs AWord8)
```

Capabilities guard access to resources.

- `minBound :: Ix N8`
  `Ix 0`
- `maxBound :: Ix N8`
  `Ix 7`
- `ixSucc (maxBound :: Ix N8)`
  `IxPlus 8 :: IxPlus`
- `ixPred (maxBound :: Ix N8)`
  `IxMinus 6 :: IxMinus N8`
- `(minBound :: Ix N8) <<= ixPred (maxBound :: Ix N8)`
  `Just (Ix 6) :: Maybe (Ix N8)`

Types are static capabilities: kernel

```
:type attrAt
Ix N25 -> Ix N80 ->
ARef (U B1 B0) (AtArea ScreenAbs AWord8)
```

Capabilities guard access to resources.

- `minBound :: Ix N8`
  `Ix 0`
- `maxBound :: Ix N8`
  `Ix 7`
- `ixSucc (maxBound :: Ix N8)`
  `IxPlus 8 :: IxPlus`
- `ixPred (maxBound :: Ix N8)`
  `IxMinus 6 :: IxMinus N8`
- `(minBound :: Ix N8) <<= ixPred (maxBound :: Ix N8)`
  `Just (Ix 6) :: Maybe (Ix N8)`
- `(minBound :: Ix N8) <<= ixPred (minBound :: Ix N8)`
  `Nothing :: Maybe (Ix N8)`

Types are static capabilities: sandbox

Indices are capabilities. Looping is not part of the trusted kernel!

```
forEachIx proc = loop minBound
 where
  loop ix = do
    proc ix
    maybe (return ())
          loop
          (ixSucc ix <<= maxBound `asTypeOf` ix)
```

The bound test often doubles as the loop termination criterion.

Types are static capabilities: sandbox

Indices are capabilities. Looping is not part of the trusted kernel!

```
forEachIx proc = loop minBound
 where
  loop ix = do
    proc ix
    maybe (return ())
          loop
          (ixSucc ix <<= maxBound `asTypeOf` ix)
```

The bound test often doubles as the loop termination criterion.
General recursion and nontermination are allowed.

Types are static capabilities: sandbox

Clear screen by writing words into video memory.

```
cls = forEachIx (\i -> write_area (vr @@ i) blank)
 where
  vr = as_area videoRAM
         (mk_array_t undefined
                     (undefined::BEA_Int16))
         nat0
  _ = size_of (aref_area videoRAM) `asTypeOf`
      size_of (aref_area vr)
```

Types are static capabilities: sandbox

Clear screen by writing words into video memory.

```
cls = forEachIx (\i -> write_area (vr @@ i) blank)
 where
  vr = as_area videoRAM
         (mk_array_t undefined
                     (undefined::BEA_Int16))
         nat0
  _ = size_of (aref_area videoRAM) `asTypeOf`
      size_of (aref_area vr)
```

ScreenAbs area declared with APReadOnly property HFalse.

Types are static capabilities: sandbox

Clear screen by writing words into video memory.

```
cls = forEachIx (\i -> write_area (vr @@ i) blank)
 where
  vr = as_area videoRAM
         (mk_array_t undefined
                     (undefined::BEA_Int16))
         nat0
  _ = size_of (aref_area videoRAM) `asTypeOf`
      size_of (aref_area vr)
```

`ScreenAbs` area declared with `APReadOnly` property `HFalse`.
`ScreenAbs` area declared with `APOverlayOK` property `HTrue`.

Types are static capabilities: sandbox

Clear screen by writing words into video memory.

```
cls = forEachIx (\i -> write_area (vr @@ i) blank)
 where
  vr = as_area videoRAM
         (mk_array_t undefined
                     (undefined::BEA_Int16))
         nat0
  _ = size_of (aref_area videoRAM) `asTypeOf`
      size_of (aref_area vr)
```

`ScreenAbs` area declared with `APReadOnly` property `HFalse`.
`ScreenAbs` area declared with `APOverlayOK` property `HTrue`.
Term notation triggers static relational computation of loop bounds.

Types are static capabilities: sandbox

Clear screen by writing words into video memory.

```
cls = forEachIx (\i -> write_area (vr @@ i) blank)
 where
  vr = as_area videoRAM
         (mk_array_t undefined
                     (undefined::BEA_Int16))
         nat0
  _ = size_of (aref_area videoRAM) `asTypeOf`
      size_of (aref_area vr)
```

`ScreenAbs` area declared with `APReadOnly` property `HFalse`.
`ScreenAbs` area declared with `APOverlayOK` property `HTrue`.
Term notation triggers static relational computation of loop bounds.
Replacing `nat0` by `nat1` reports misalignment.

Constraints over time

Types can express time and protocol constraints as a state machine.

- Same number of ticks consumed along every execution path
- Maximum number of ticks consumed in any execution path
- Protocol constraints
  - file open and close
  - lock acquire and release
  - interrupt disable and enable

A *parameterized monad* is a stateful notion of computation.
(Infer types like `VST IO N0 (U (U B1 B0) B1) Int`)

Constraints over time

Types can express time and protocol constraints as a state machine.

- Same number of ticks consumed along every execution path
- Maximum number of ticks consumed in any execution path
- Protocol constraints
  - file open and close
  - lock acquire and release
  - interrupt disable and enable

A *parameterized monad* is a stateful notion of computation.
(Infer types like `VST IO N0 (U (U B1 B0) B1) Int`)

Particularly useful with staging (program extraction).

Conclusion

Types provide static assurances

- Improve performance and reliability across *all* program runs
- Integrated assertion language with explicit *stage separation*

Lightweight approach: use a general-purpose language

- Practical experience for high-assurance low-level programming
- Our library statically assures control and data constraints

Long live low-level assurances in high-level languages!

`http://pobox.com/~oleg/ftp/Computation/resource-aware-prog/`

Programming Languages Meet Program Verification

`http://www.plpv.org/` (ICFP 2007, 5 October 2007)