Lightweight Static Guarantees

Oleg Kiselyov
FNMOC
oleg@pobox.com

Chung-chieh Shan
Rutgers University
ccshan@cs.rutgers.edu

Safe and Efficient

We have identified a disciplined programming style that uses existing type systems in practical, mature languages to statically verify safety properties and make software more reliable and efficient. The technique uses *existing* facilities to express in types a wide range of safety properties:

- never dereferencing a null pointer;
- always sanitizing user input;
- accessing an array using only indices that are in bounds;
- performing modular arithmetic in cryptography using a consistent modulus;
- releasing all acquired resources such as locks, exactly once.

The resulting program is not just more reliable but also more efficient, because fewer run-time checks are needed.

We welcome suggestions of new areas and examples

1 Main Idea

Enforcing invariants by using an abstract data type

Simple and old basic idea:

- The value of an ADT represents a *capability*
- The small code that builds such values is a trusted kernel
- The type checker enforces authorization of capability-based operations
- The type checker extends trust from the kernel to the rest of the program
- With parametric polymorphism: using types as proxies for values
  e.g., statically unknown array size

Types provide static assurances

We state safety properties in the type language, and the type checker enforces them during compilation

2 Simplest case: Assure the safety of partial operations

Allowed values are known at compile time

- dereferencing only non-null pointers;
- dividing by only non-zero denominators;
- taking the head or tail of only non-empty linked lists;
- indexing into a static buffer with only in-range indices;
- executing SQL commands with no unsanitized input.

In this case, we can implement our approach already in C++ and Java.

Example

Binary search in a statically allocated array
Array indices are *always* in-bounds, no array bound check

```
class index                              // Security kernel
{ int v;
  index();
  index(int i): v(i) {}
public:
  index(const index &i): v(i.v) {}
  int as_int() const {return v;}

  static const int MIN = 0, MAX = 255;
  static const index min() {return index(MIN);}
  static const index max() {return index(MAX);}

  // bounded increment: not past h
  bool incr(const index h)
  { if(v >= h.v) return false;
    v++; return true;}

  // bounded decrement: not past l
  bool decr(const index l)
  { if(v <= l.v) return false;
    v--; return true; }

  index middle(const index other) const
  { return index(v + (other.v-v)/2); }
};

class fixed_array                    // Security kernel
{ int array[index::MAX - index::MIN + 1];
public:
  fixed_array()
  { assert(sizeof(array) > 0);
    for(int i=0; i<sizeof(array)/sizeof(array[0]); i++)
```

```
        array[i] = 10*i+1;
    }
    int operator [] (const index i) const ...
    int& operator [] (const index i)
    { return array[i.as_int() - index::MIN]; }
};

int search(const fixed_array &array, const int target) {
    index low = index::min(), high = index::max();
    for (;;) {
        index mid = low.middle(high);
        const int v = array[mid];
        if (v == target) return mid.as_int();
        if (v < target)
            if (mid.incr(high)) low  = mid; else return -1;
        else
            if (mid.decr(low )) high = mid; else return -1;
    }
}
```

Generated assembly code shows no overhead

Safe and Efficient

3 Safe indexing into a dynamically allocated array

Array indices are *always* in-bounds, no array bound check

More complex case: need more advanced type systems

at least Java 5 generics

more generally, higher-rank types of OCaml, Scala, or Haskell

```
bsearch cmp (key, arr) =
  brand arr (\arrb -> bsearch' cmp (key, arrb))
```

```
bsearch' cmp (key,(arr,lo,hi)) = look lo hi
  where
  look lo hi =
      index_cmp lo hi Nothing $
        \lo' hi' ->
              let m = bmiddle lo' hi'
                  x = arr !. m
              in case cmp (key,x) of
                       LT -> look lo (bpred m)
                       EQ -> Just (unbi m, x)
                       GT -> look (bsucc m) hi
```

Trusted Kernel

```
newtype BArray s i a = BArray (Array i a)
newtype Integral i => BIndex s i   = BIndex i
newtype Integral i => BIndexL s i  = BIndexL i
newtype Integral i => BIndexH s i  = BIndexH i

brand:: (Ix i, Integral i) =>
  Array i e
  -> (forall s. (BArray s i e,BIndexL s i,BIndexH s i)->w)
  -> w
brand (a::Array i e) k =
    let (l,h) = bounds a
    in k ((BArray a)::BArray () i e, BIndexL l, BIndexH h)

bmiddle:: Integral i =>
  BIndex s i -> BIndex s i -> BIndex s i
bmiddle (BIndex i1) (BIndex i2) =
  BIndex ((i1 + i2) `div` 2)

bsucc:: Integral i => BIndex s i -> BIndexL s i
```

```
bsucc (BIndex i) = (BIndexL (succ i))

(!.):: (Ix i) => BArray s i e -> BIndex s i -> e
(BArray a) !. (BIndex i) = unsafeAt a i
```

the value of the type `BIndex s i`: assuredly `low <= i <= high`,
type `s` represents `(low,high)`

the value of the type `BIndexL s i`: assuredly `low <= i`,
type `s` represents `(low,high)`

More complex example: KMP string search
indices are stored in a mutable array. See the paper

4 Better type systems – more assurances

Accessing 'raw metal' safely

Assure accessing a memory region through a pointer respects properties such as region's size, alignment, endianness, and write permissions—even when allowing pointer arithmetic and casts

```
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
  B1 B1) B1) B1) B1) B0) B1) B0) B0) B0) B0) B0

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

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)
```

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

5 Conclusions

Safe and Efficient

Types provide static assurances

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

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

In an industrial setting, we are applying this approach to our own programs, such as Web application servers

Motivating the use of logic to reason about programs

We welcome suggestions of new areas and examples

References

[1] Oleg Kiselyov and Chung-chieh Shan. 2006. Lightweight static capabilities. In *Programming languages meet program verification*, ed. Aaron Stump and Hongwei Xi, 28–39. Electronic Notes in Theoretical Computer Science, Amsterdam: Elsevier Science. `http://okmij.org/ftp/Computation/lightweight-dependent-typing.html#lightweight-static-capabilities`

[2] Oleg Kiselyov and Chung-chieh Shan. 2007. Lightweight static resources: sexy types for embedded and systems programming. In *Trends in Functional Programming*. `http://okmij.org/ftp/Computation/resource-aware-prog/tfp.pdf`