Applying Language-based Static Verification in an ARM Operating System

Matthew Danish
Boston University
md@bu.edu

9 May 2013

What do we want?

Correctness

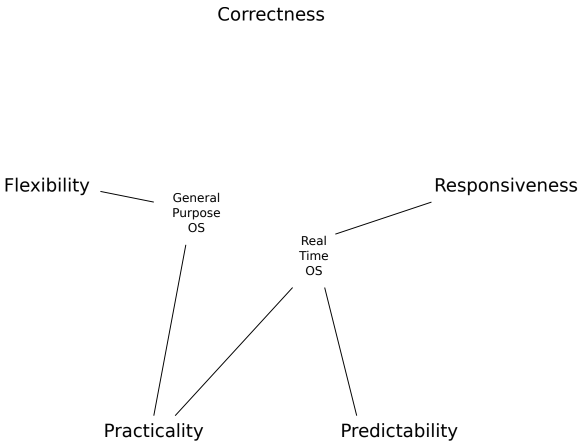
Flexibility

Responsiveness

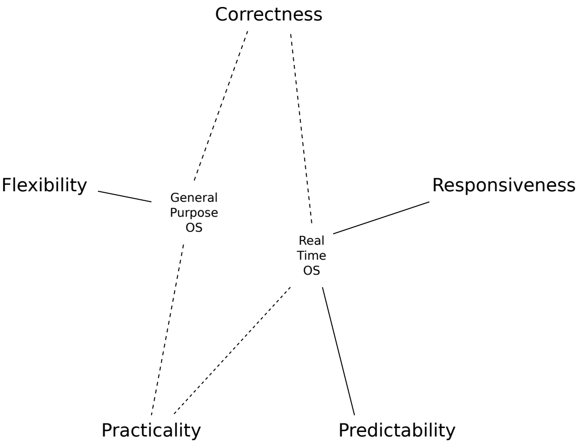
Practicality

Predictability

What do we want?



What do we want?

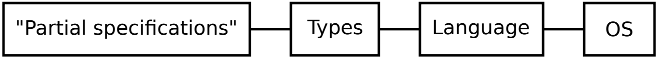


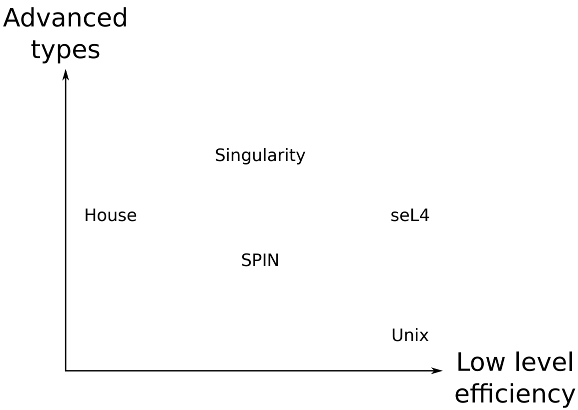
Programming Languages and Types

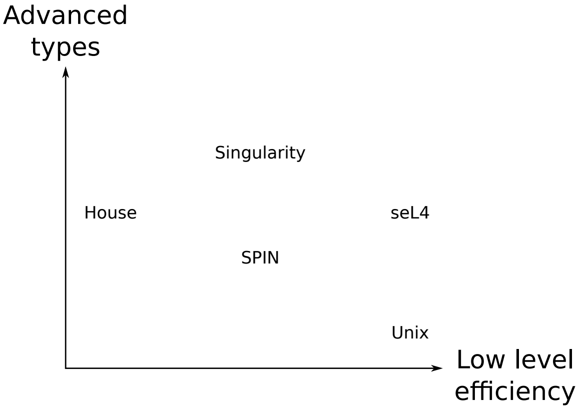
- Choosing or designing languages for systems
  - Unix: C
  - SPIN: Modula-3
  - Singularity: Sing#
  - seL4: Isabelle, Haskell and C
  - House: Haskell

Programming Languages and Types

- Choosing or designing languages for systems
  - Unix: C
  - SPIN: Modula-3
  - Singularity: Sing#
  - seL4: Isabelle, Haskell and C
  - House: Haskell




Advanced types
Singularity
House
seL4
SPIN
Unix
Low level efficiency



Can we have more advanced types and low level efficiency?

ATS

- ML-like, strong C integration, LF-style theorem proving
- Linear types (a.k.a. view types), dependent types
- Separation of proof-world and program-world,
  `(proof | program)`
- Practical, functional programming in system setting

Terrier OS

- ARM, TI OMAP4 MP-core, SMP, USB support
- Exploring advanced types in assisting OS development
- Compact and uncluttered design, with message-passing
- Work in progress

Challenges

- Bringing high level functional programming into OS
- Using advanced types to tackle common problems
- Interfacing with the low level code where needed
- Avoiding performance impacts

Functional programming

- Nested functions
- Tail recursion elimination
- Higher order functions
- Style

Resource management

- Linear reasoning: avoid memory leaks
- “Must be used once and exactly once”
- Typical pattern: allocate, transform, and release

```
let val (proof_var | pointer_var) = alloc ()
    val x = do_something (proof_var | pointer_var)
in free (proof_var | pointer_var)
```

Synchronization

- Linear reasoning for synchronization
- Ensure proper lock management
- Correct sequencing of steps

```
let val (outer | _) = outer_lock () in
  let val (inner | _) = inner_lock (outer | ) in
    ...
  let val (outer | _) = inner_unlock (inner | ) in
outer_unlock (outer | )
```

Safe use of pointers

- Concept: "value of type `t` is stored at address `l`"
- ATS "`@`-view": `type @ address`

```
fun alloc_pair(): [l: addr] ((int, int) @ l | ptr l)

fun free_pair {l: addr} (pf: (int, int) @ l | p: ptr l)
```

- Pointers: a "dependent type" i.e. a type indexed by a value
- In this case: the value is the address
- The "`@`-view" validates the pointer

Array bounds checking

- Integer constraint solver
- Automatic bounds checks
- `array`: dependent type indexed by length
- Array access must be within $0 \leq i < n$

```
fun f {n: int | n > 3}
      (A: array (int, n), len: int n): int =
  let val x = A[0] in
    if len > 4 then x + A[4] else x
```

Integer constraints

- Not just limited to arrays
- Example from scheduler
- "exists tick t such that $t > \mathit{now}$"

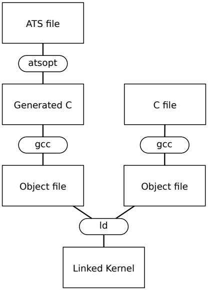
```
val [now: int] now: tick now = timer_32k_value()

fun is_earlier_than {n, m: nat}
    (tn: tick n, tm: tick m): bool (n < m)
...
val future: [t: int | t > now] tick t = ...
```

Avoiding overhead

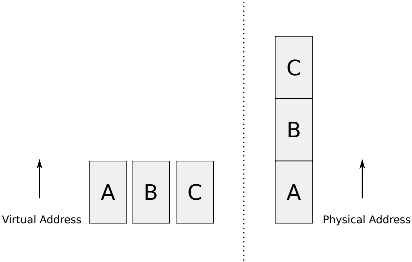
- Erasure of statics
- Flat types, C data representation
- Templates

ATS integration



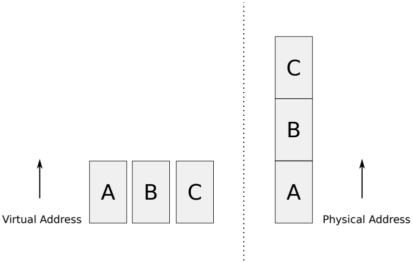
- ATS acts as preprocessor
- No run-time and minimal static support
- ATS in both kernel and program components

Protection



- Hardware memory protection optional
- Can rely on hardware protections when needed
- Or can switch to static verification when ready

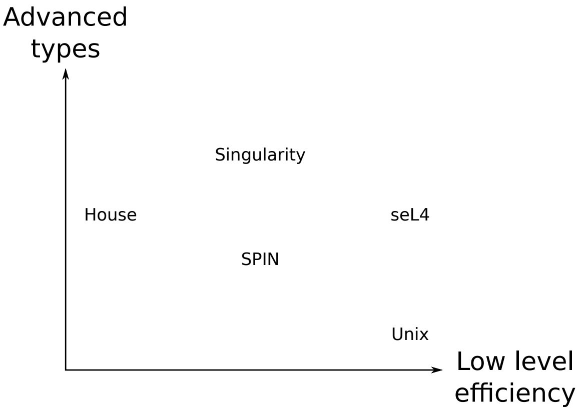
Protection

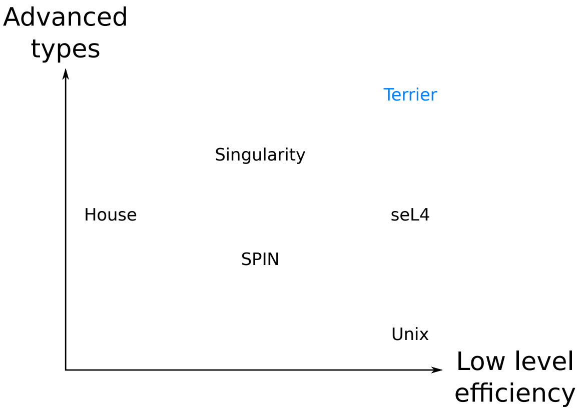


- All programs take advantage of ELF features for relocation
- Kernel has load-time linker which rewrites binary
- Can rewrite binaries into the two different memory models

Putting it together

- The role of type systems in OS development
- Application of advanced types for better assurance
- Incremental approach to verification
- Straightforward machine translation to C
- Depends on compiler and hardware correctness


Advanced types
Singularity
House
seL4
SPIN
Unix
Low level efficiency


Advanced types
Terrier
Singularity
House
seL4
SPIN
Unix
Low level efficiency

seL4 and Terrier

seL4

- Haskell prototype, Isabelle specification, refinement proof between specification and C
- Entire kernel, big effort
- Top-down

Terrier

- Written directly in C/ATS mix, ATS types
- Flexible, selective effort
- Bottom-up

Future work

- Writing more proofs
- Adding further hardware support
- Deploying on an experiment