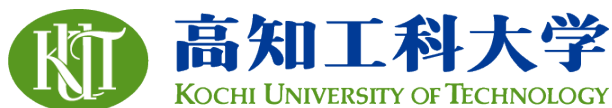


Collecting Type Information Using Unit Tests for Customizing JavaScript VMs

work-in-progress report on eJS project

Tomoharu Ugawa¹ Hideya Iwasaki² Takafumi Kataoka¹

¹ Kochi University of Technology



² The University of Electro-Communications



eJS: JavaScript VM for IoT

- Goal: make IoT programming easier for many people
- Why JavaScript
 - JavaScript is popular
 - suitable for prototyping
- Challenge
 - Reduce VM footprint

JavaScript engines for IoT

- Duktape
- Espruino
- JerryScript
- MuJS
- v7/mJS

JavaScript engines for IoT

- DukTape
- Espruino
- JerryScript
- MuJS
- v7/mJS
- QuickJS

QuickJS Javascript Engine

News

- 2019-07-09:
 - First public release

Introduction

QuickJS is a small and embeddable Javascript engine. It supports the [ES2019 specification](#) including modules, asynchronous generators and proxies.

JavaScript engines for IoT

- DukTape
- Espruino
- JerryScript
- MuJS
- v7/mJS
- QuickJS

QuickJS Javascript Engine

News

- 2019-07-09:
 - First public release

Introduction

QuickJS is a small and embeddable Javascript engine. It supports the [ES2019 specification](#) including modules, asynchronous generators and proxies.

- support all features of JS, or
- support features selected by VM developer

Specialization

Assumptions

- Applications on an embedded system are fixed
- Each application uses a subset of JavaScript features

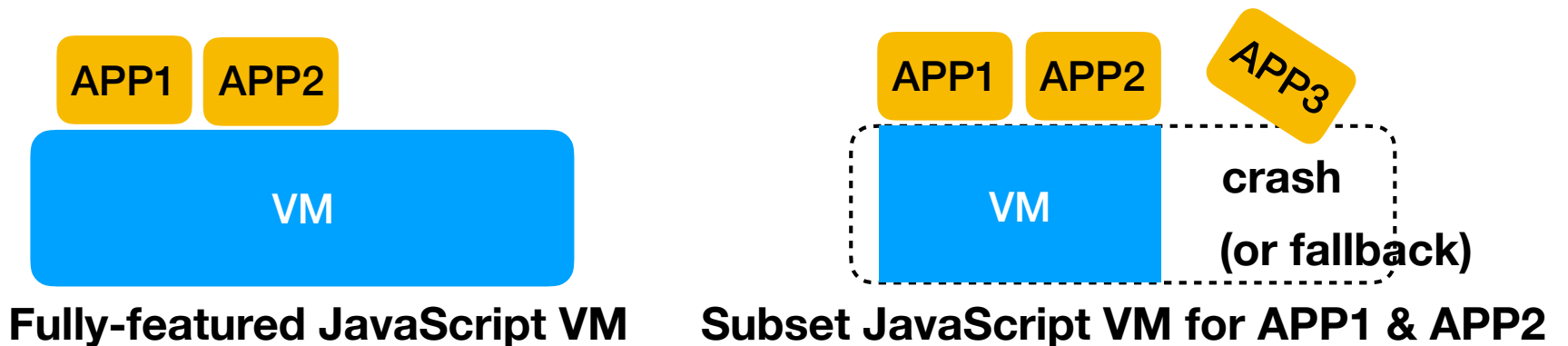
Specialization

Assumptions

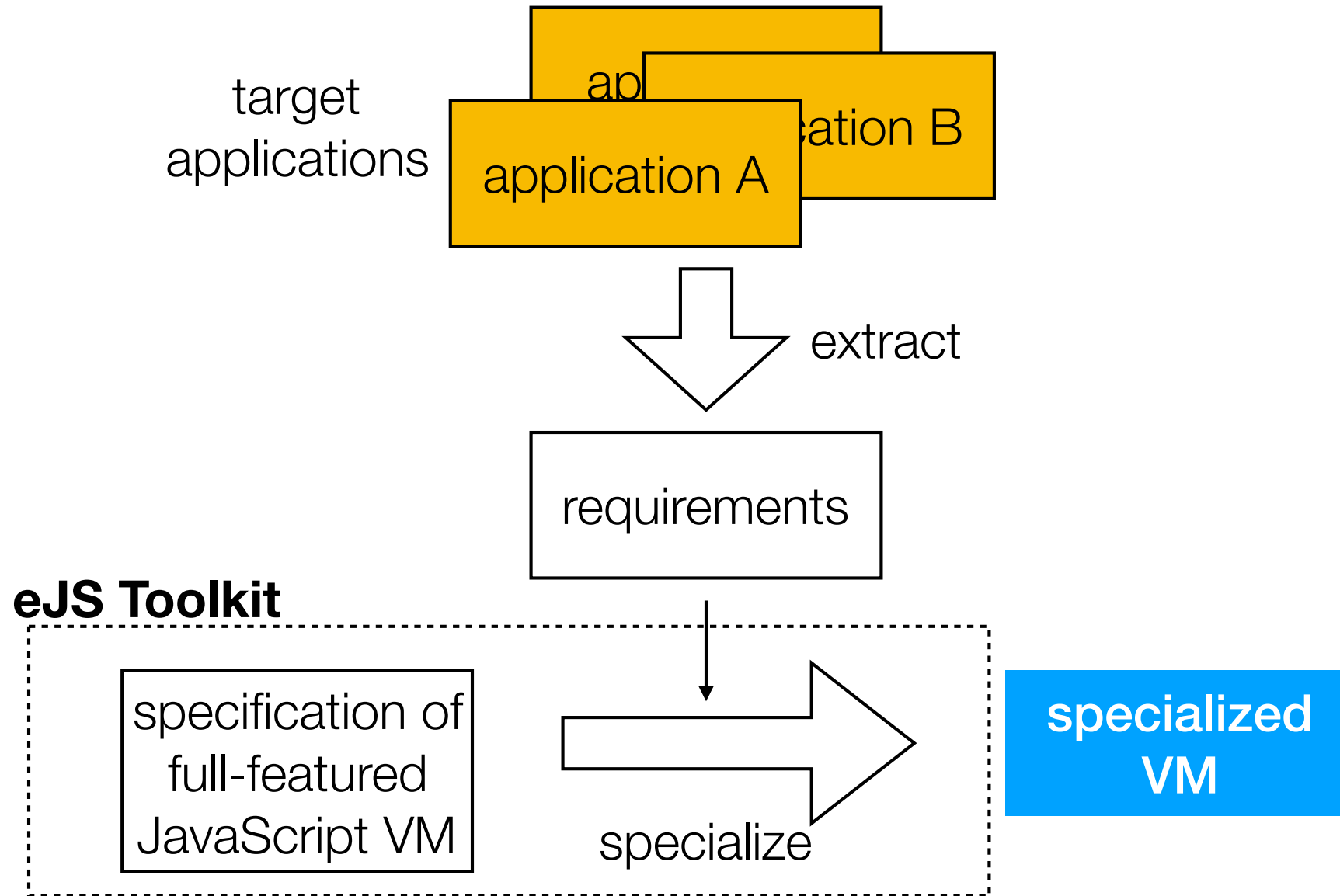
- Applications on an embedded system are fixed
- Each application uses a subset of JavaScript features

Our approach

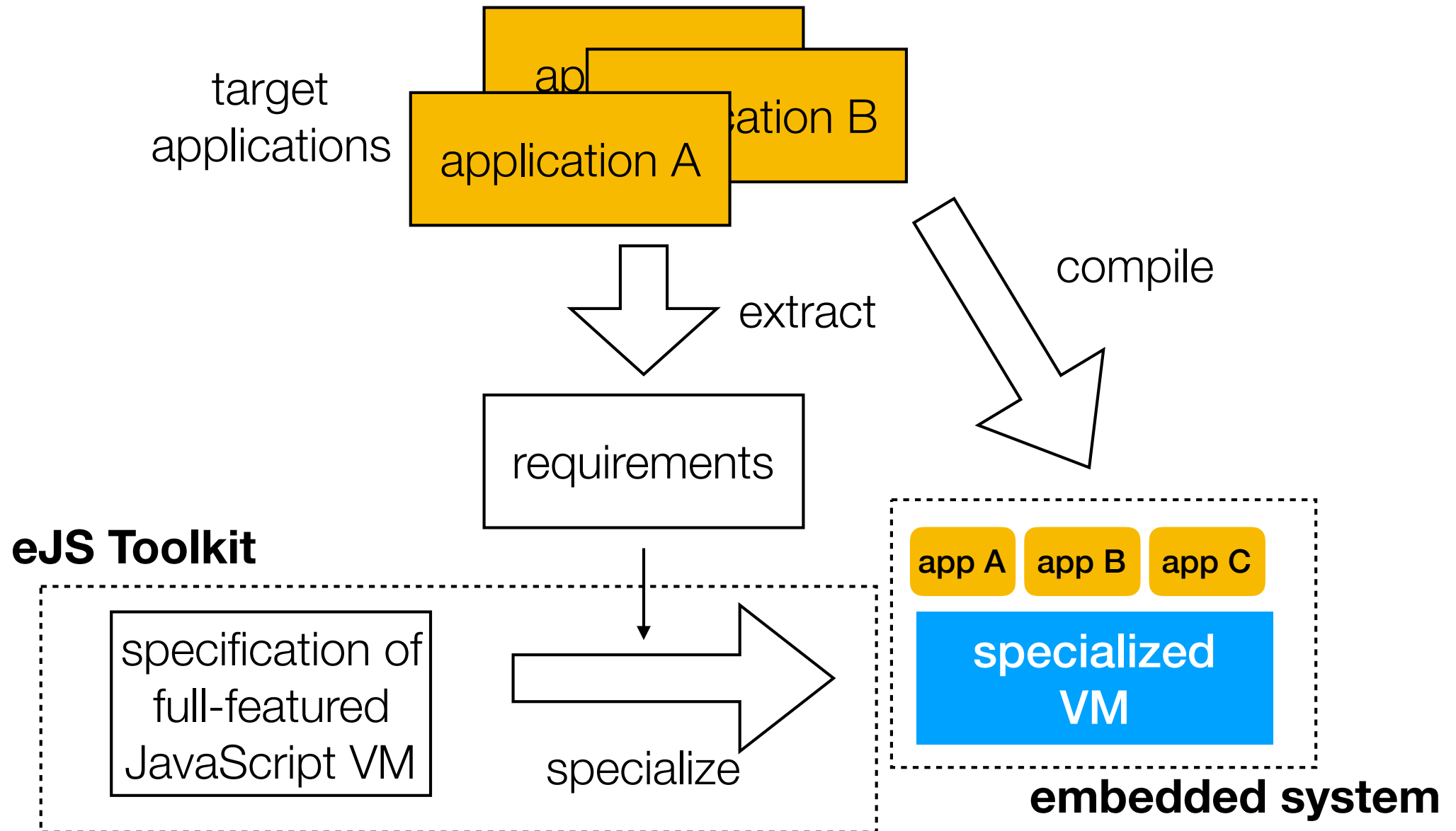
- Generate a specialized VM for each set of applications
- Give up supporting other applications



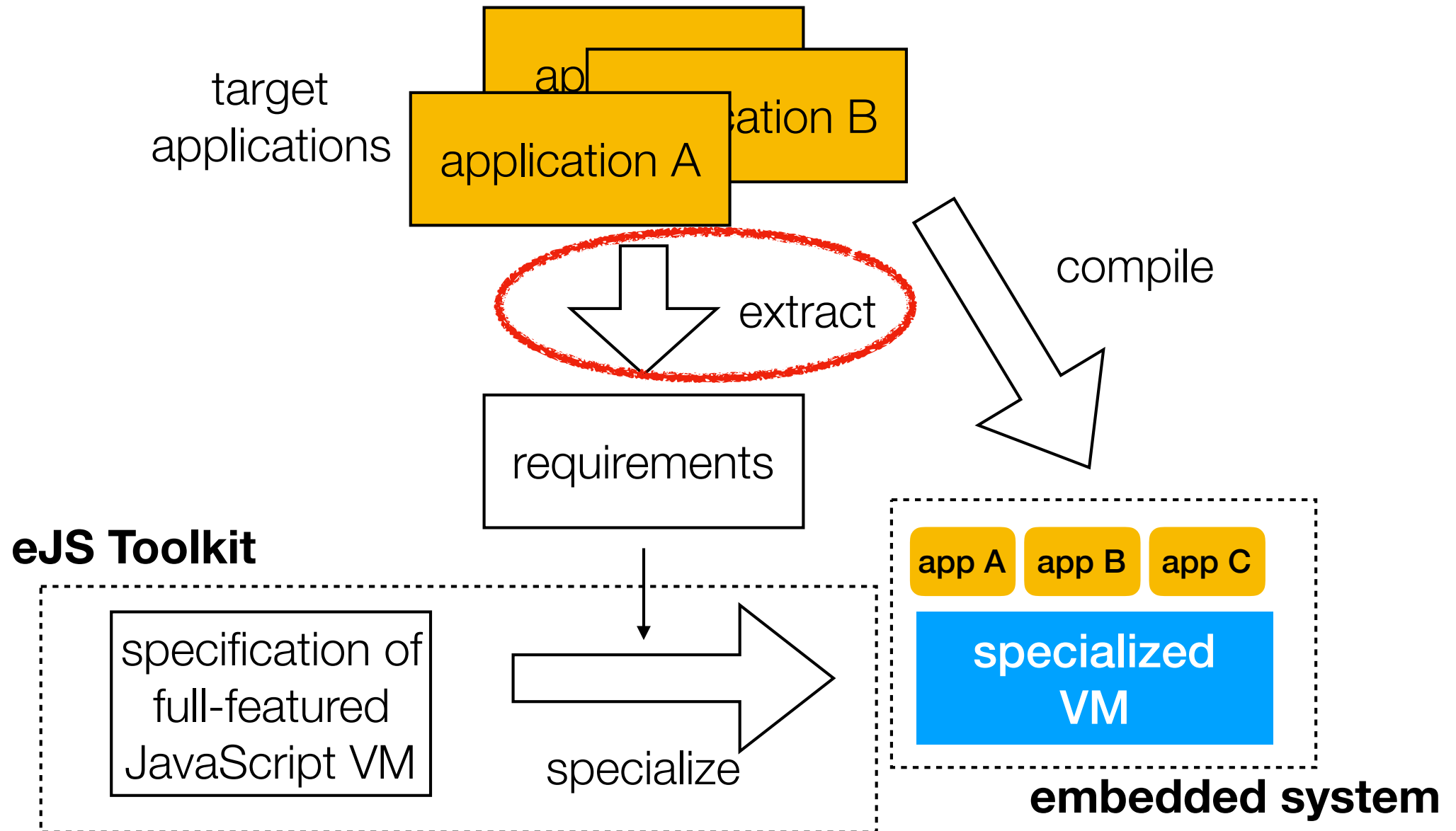
Overview of eJS



Overview of eJS

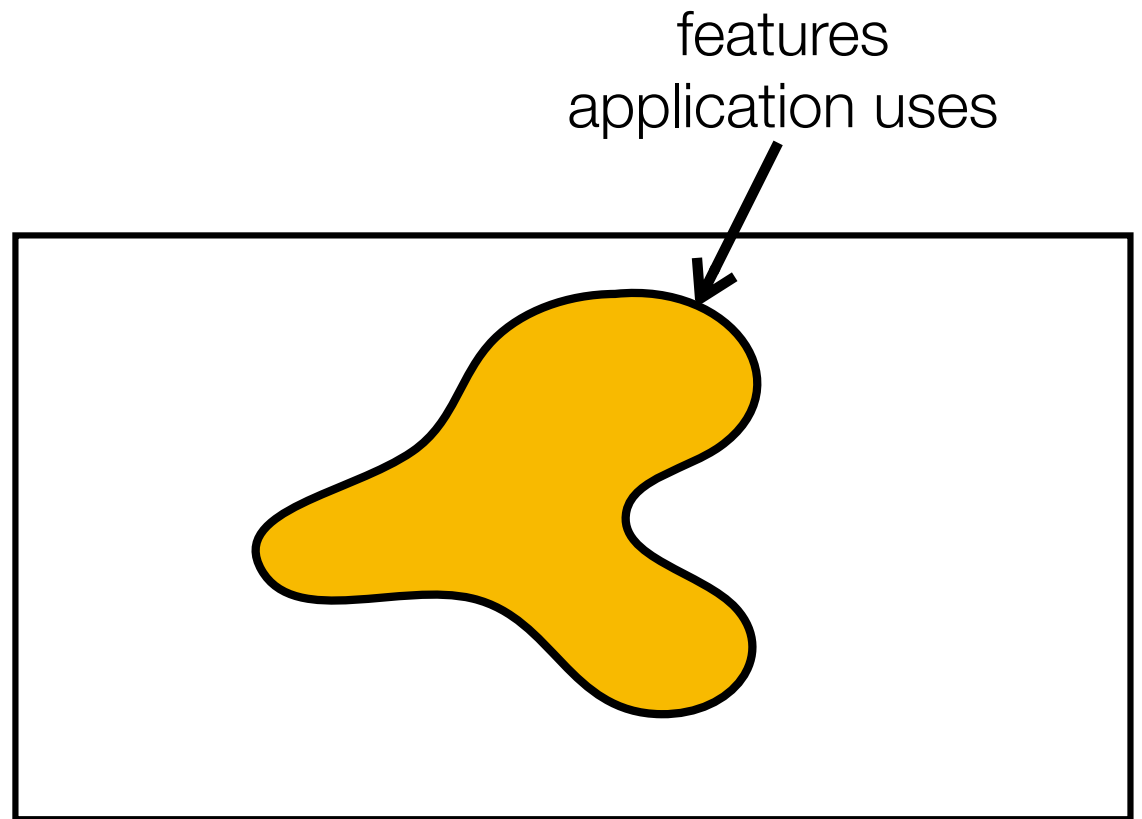


Overview of eJS



Accuracy of Requirements

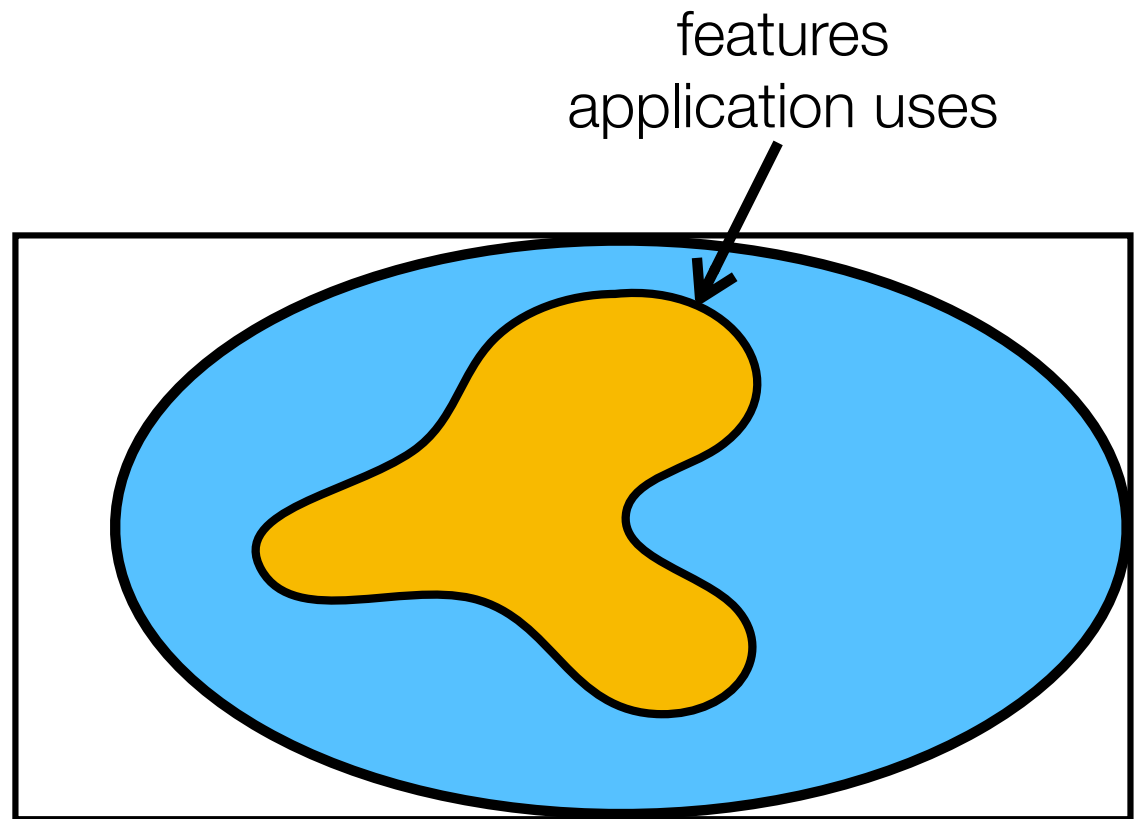
Extract accurate requirements of applications



Accuracy of Requirements

Extract accurate requirements of applications

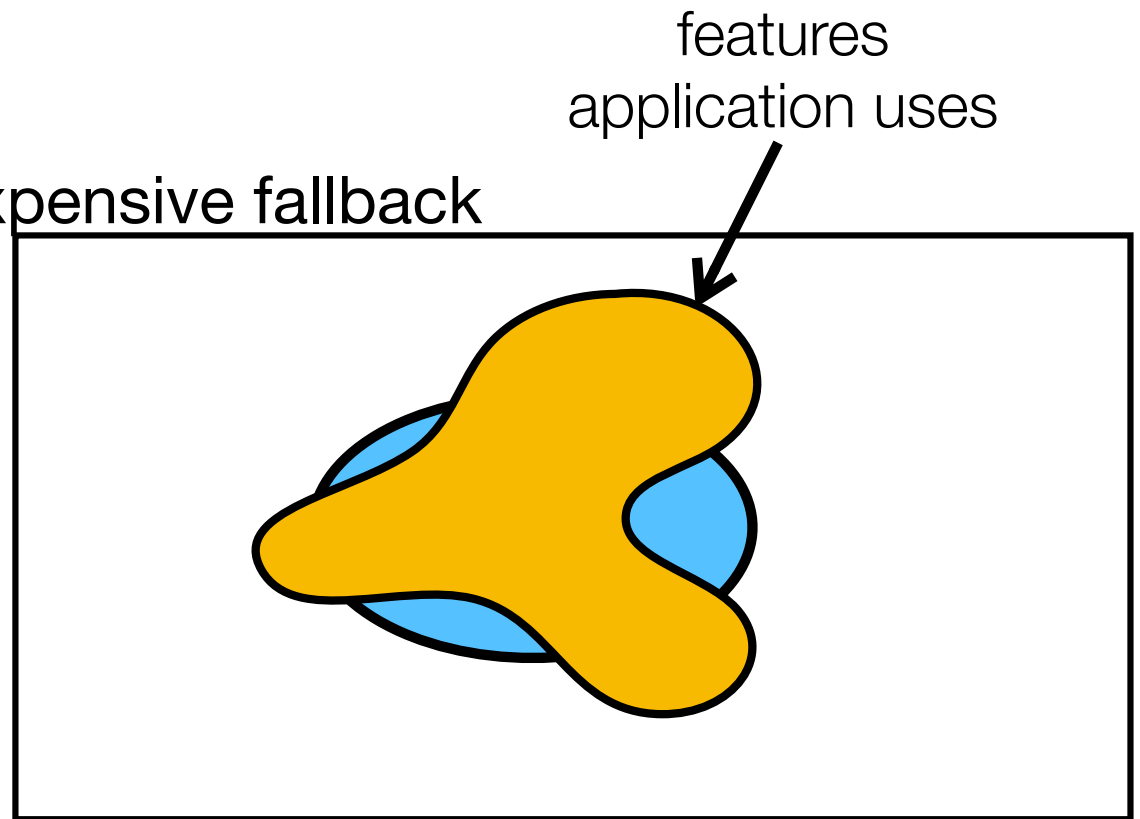
- too conservative
→ large VM is generated



Accuracy of Requirements

Extract accurate requirements of applications

- too conservative
→ large VM is generated
- missing features
→ VM may crash or expensive fallback



Operator Overloading

- $\text{Number} + \text{Number} = \text{Number}$
- $\text{Number} + \text{String} = \text{String}$
- $\text{Number} + \text{Boolean} = \text{Number}$

Operator Overloading

- Number + Number = Number
- Number + String = String
- Number + Boolean = Number

```
switch(type(v1)) {  
  case NUM:  
    switch (type(v2)) {  
      case NUM:  
        dst = NUM(val(v1) + val(v2));  
        break;  
      case STR:  
        v1 = ToString(v1);  
        dst = concat(v1, v2);  
        break;  
      ...  
    }  
  case STR:  
    ...  
}
```

ADD instruction

Size Reduction by Specialization

- Exclude code for unused operations
- Simplify dispatching code

```
switch(type(v1)) {  
  case NUM:  
    switch (type(v2)) {  
      case NUM:  
        dst = Num(val(v1) + val(v2));  
        break;  
      case STR:  
        v1 = toStr(v1);  
        dst = concat(v1, v2);  
        break;  
      ...  
    }  
  case STR:  
    ...  
}
```

```
switch(type(v1)) {  
  case NUM:  
    dst = NUM(val(v1) + val(v2));  
    break;  
  case STR:  
    dst = concat(v1, v2);  
    break;  
}
```

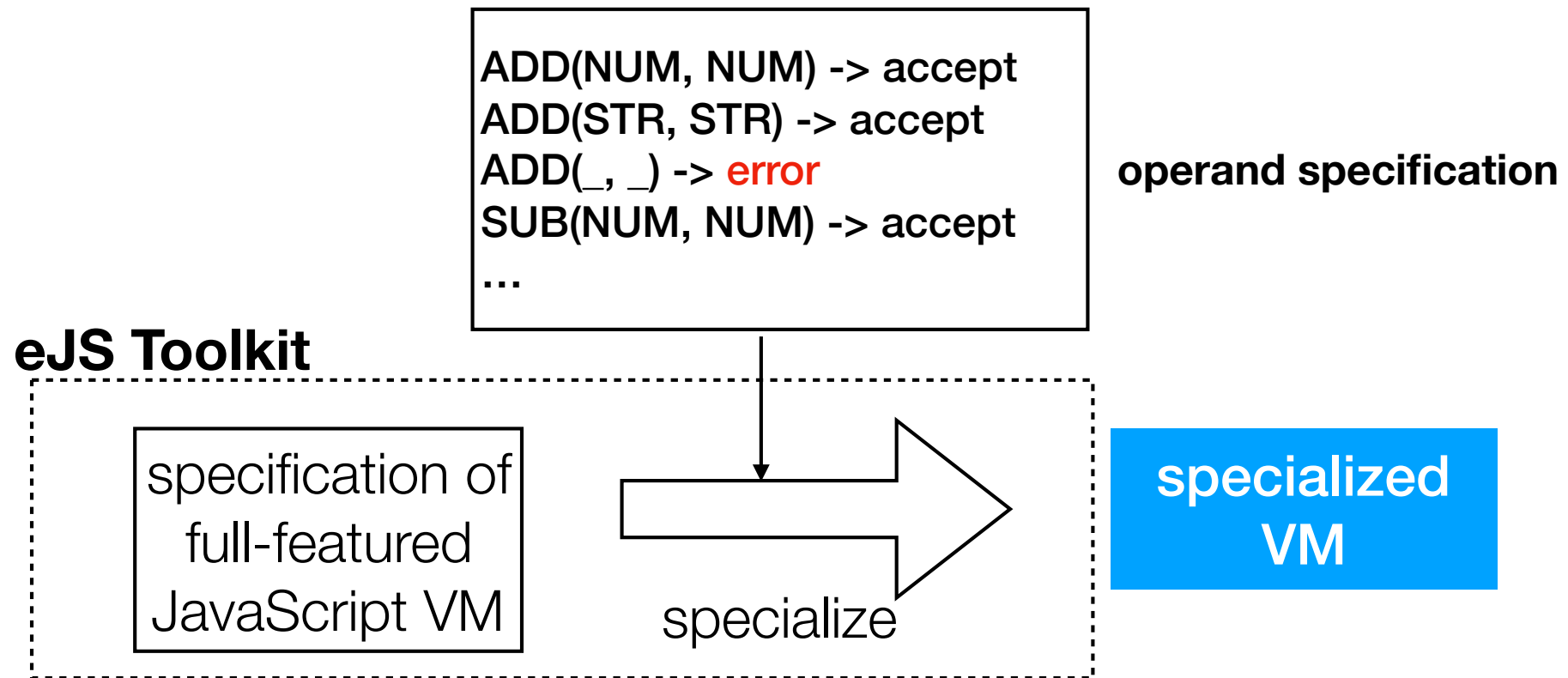
Specialized Interpreter
(Only supports NUM+NUM & STR+STR)

Code for unused operation (NUM + STR)

Application's Requirement = Type Information

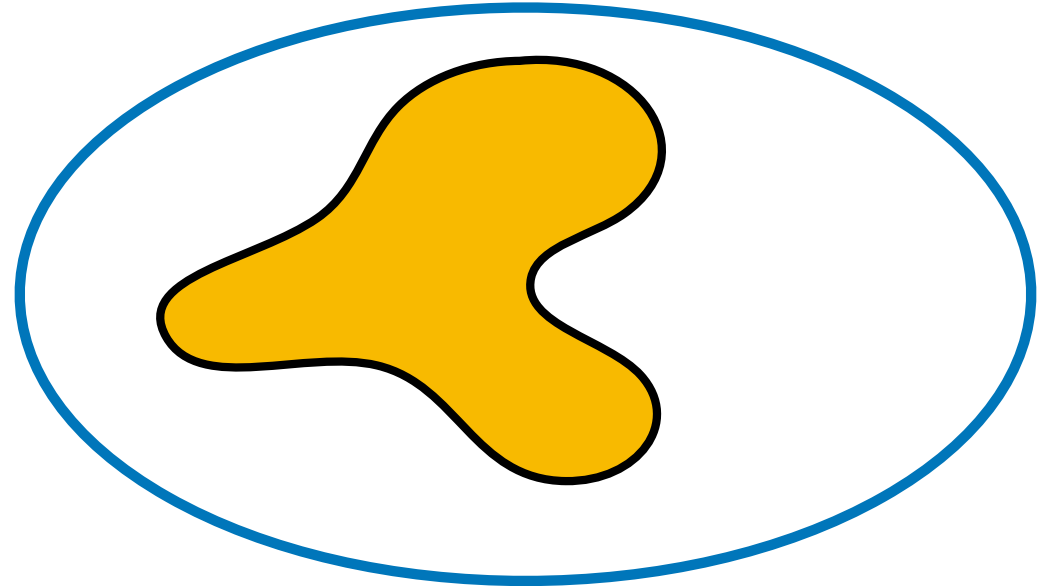
Operand Specification

- combinations of operand datatypes of each instruction

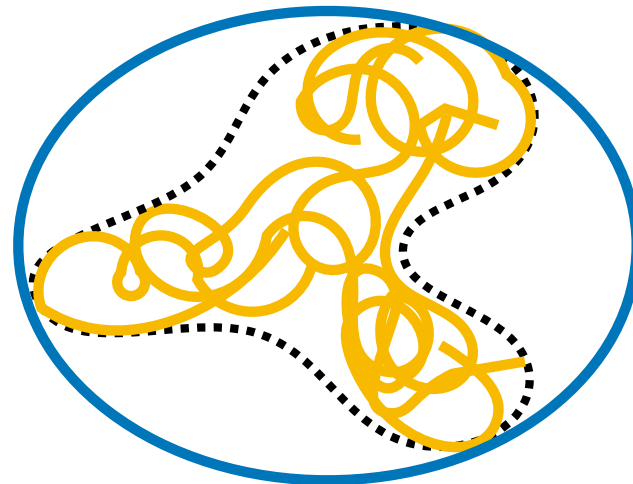


Collecting Type Information

- Type inference

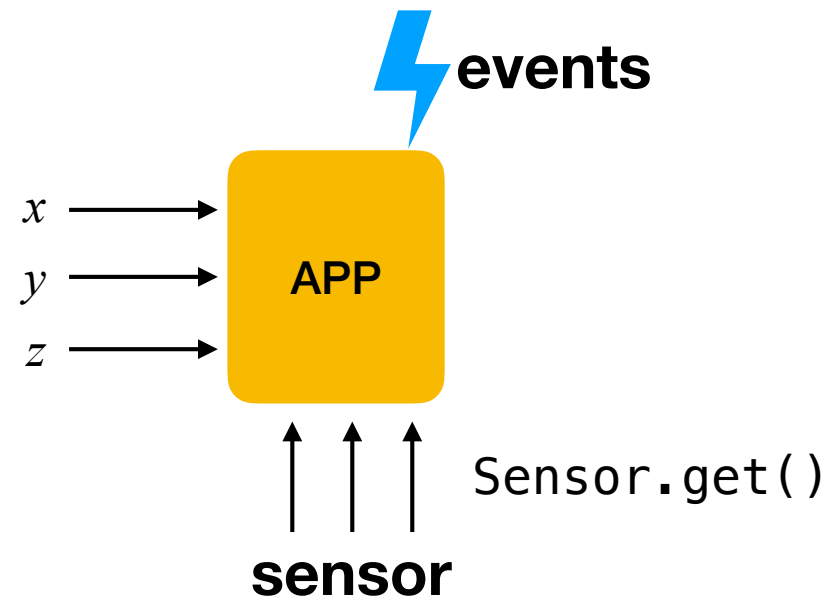


- Profiling



Problem

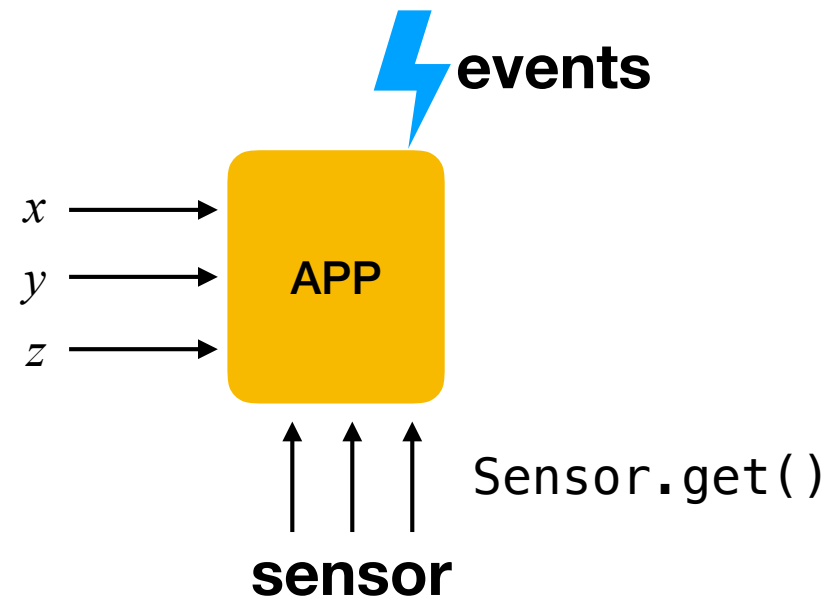
- Input to application
 - parameters, sensor responses, events, ...
 - large space to be explored



Problem

- Input to application
 - parameters, sensor responses, events, ...
 - large space to be explored

- Execution environment
 - real hardware is needed to execute application



Solution: Unit tests

```
function getTemp() {  
  var s = Sensor.get();  
  c = Number("0x" + s);  
  f = (c * 9 / 5) + 32;  
  return f;  
}
```

application in question

"00" → 32

"1E" → 86

```
describe("test", function() {  
  it("getTemp", function() {  
    spyOn("Sensor", "get")  
      .and.returnValue("0");  
    var ret = getTemp();  
    expect(ret).toBe(32);  
    success += (ret === 32);  
  });  
  it("string", function() {  
    spyOn("Sensor", "get")  
      .and.returnValue("1E");  
    var ret = getTemp();  
    expect(ret).toBe(86);  
    success += (ret === 86);  
  });  
});
```

test cases using

 **Jasmine** unit test framework

Profiling Framework



profiling
VM

Profiling Framework



application


```
c = Number("0x" + s);
```



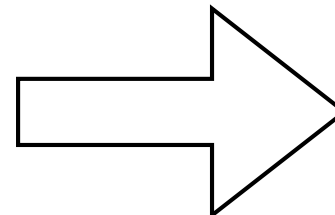
testcase

```
success += (ret === 32);
```



pseudo-
 Jasmine

compile
with log flag



compile

```
...  
string_log r3 "0x"  
add_log    r1 r3 r1  
...  
fixnum     r4 32  
eq          r3 r3 r4  
add        r2 r2 r3  
...
```

record operand datatypes
of each instruction
if log flag is set

profiling
VM

Experiment

JavaScript programs

- morse (113 LOC)

test LOC	coverag
----------	---------

78	99.8%
----	-------

- ported from C

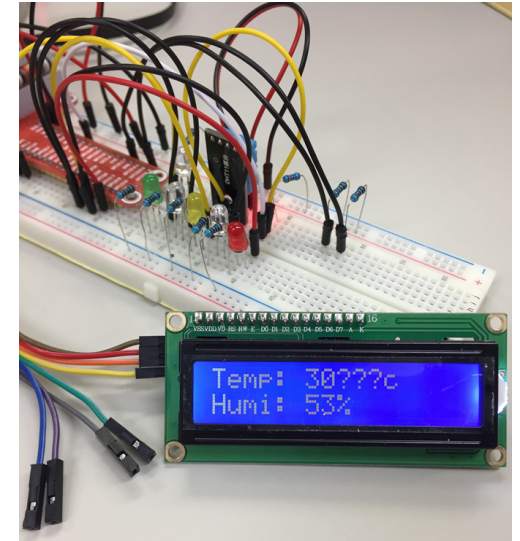
- humidity and temperature meter (HT) (459 LOC)

- main

- sensor — ported from Python

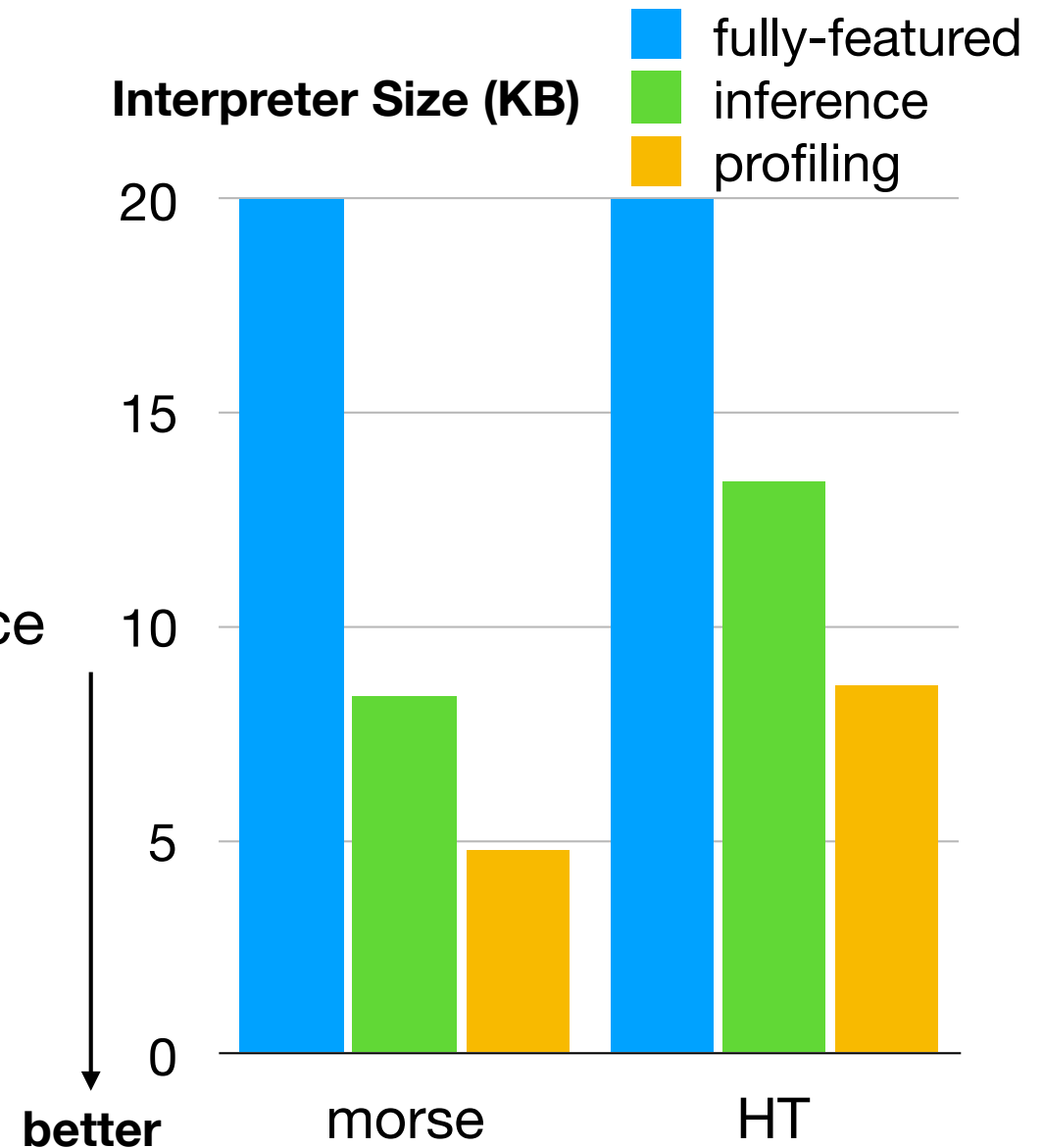
- LCD — ported from C

module	test LOC	coverage
main	113	87.4%
sensor	170	66.0%
LCD	99	77.1%
total	382	77.4%



Result

- Generated operand specifications were correct
- VM size
 - fully-featured VM
 - conservative type inference
 - abstract interpretation of bytecode
 - profiling



Advantage & Disadvantage

- Advantage
 - free — users do not need to create extra stuff
 - practical — we can do on desktop computer
 - no overestimation
- Disadvantage
 - unsound
 - accuracy depends on quality of test cases

Conclusion and Future Work

- Framework to collect application requirements
- Observe executions of unit tests
- Future Work
 - Combination with type inference
 - Measure quality of test suite — coverage?