Generating a Minimal JavaScript VM Specialized for Target Applications

work-in-progress project *eJS*

Tomoharu Ugawa (Kochi University of Technology)
Hideya Iwasaki (The University of Electro-Communications)

Background

- Goal: Make programming of “Internet of Things” easier
- Use JavaScript
  - One of the most popular language
  - Suitable for rapid prototyping
  - Matches event-driven programming style of embedded systems
- Challenge: memory limitation
  - Reduce VM image size & heap size

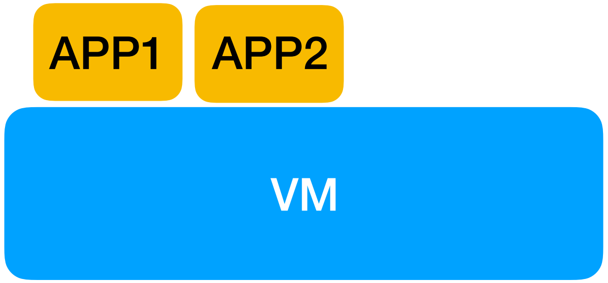
Specialization

Key observation

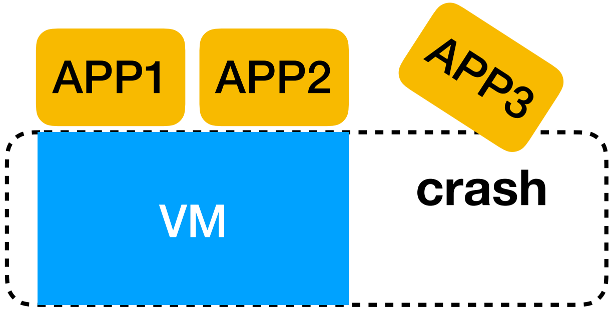
- Applications on a particular embedded system are fixed
- Each application uses a subset of JavaScript features

Our approach

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



Full-set JavaScript VM



Subset JavaScript VM for APP1 & APP2

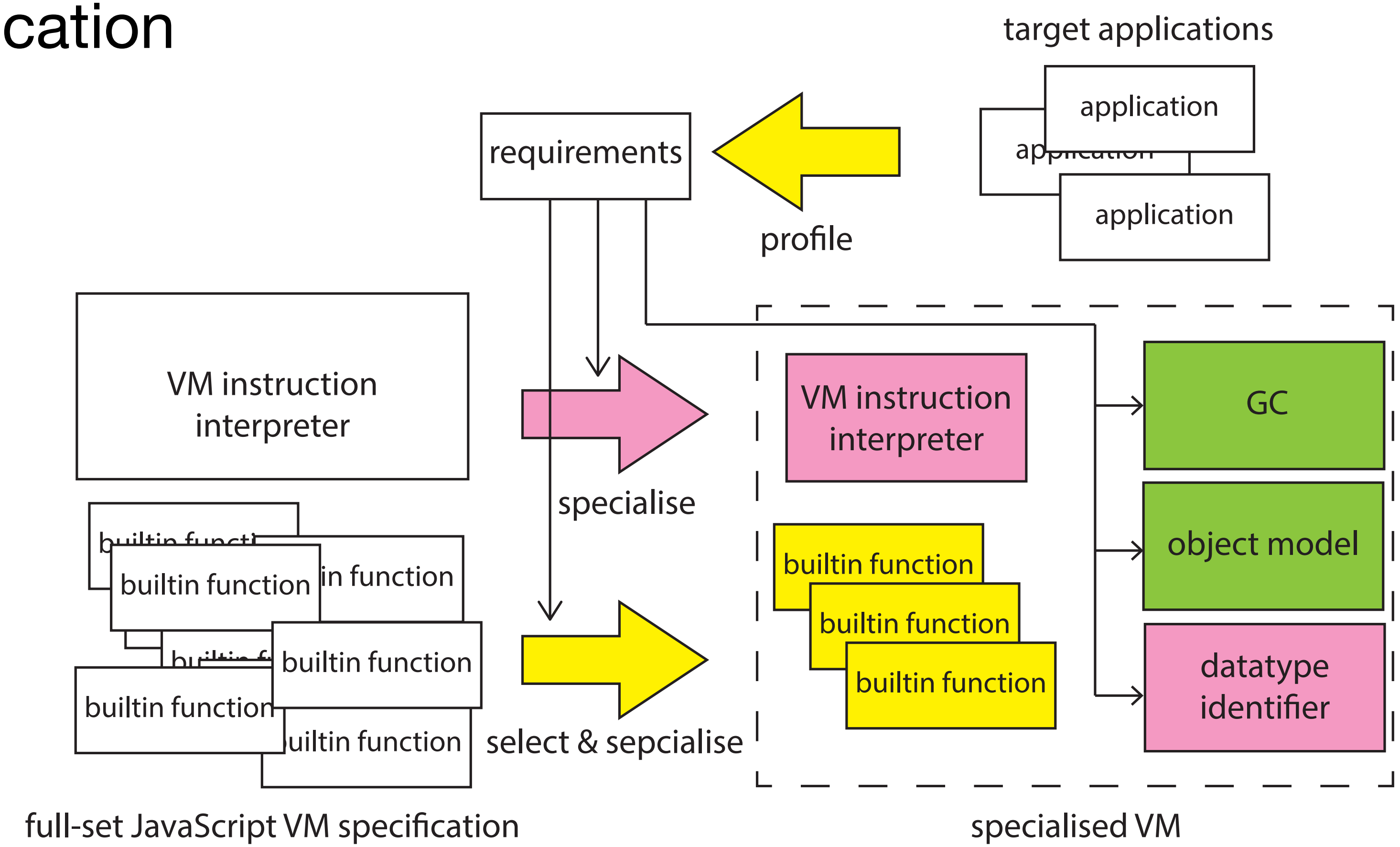
How do we specialize?

- Collect applications requirements **on going**
- Customize VM code related to datatype-based dispatch
  - VM instruction interpreter **done**
  - Built-in functions **on going**
  - Type conversion internal functions **on going**
- Customize object representation **future work**

Overview of *eJSTK*

1. Collect requirements of target applications

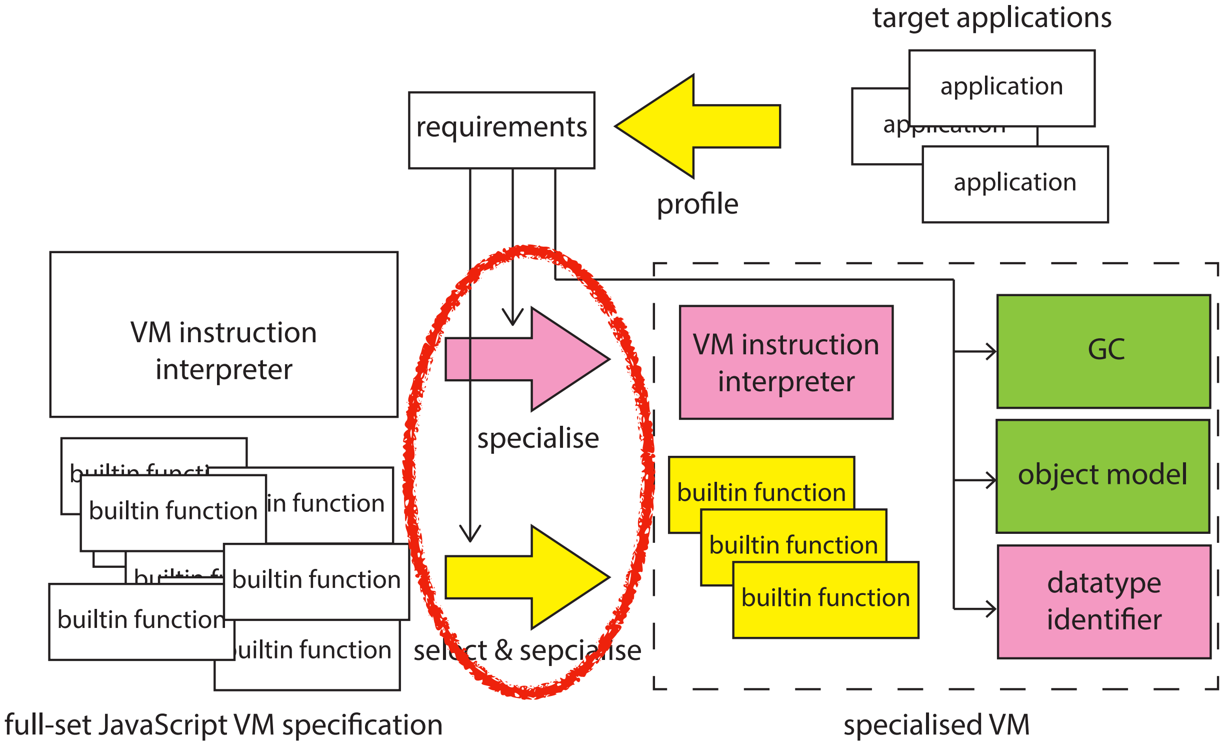
2. Generate specialized VM source code from the full-set VM specification



done

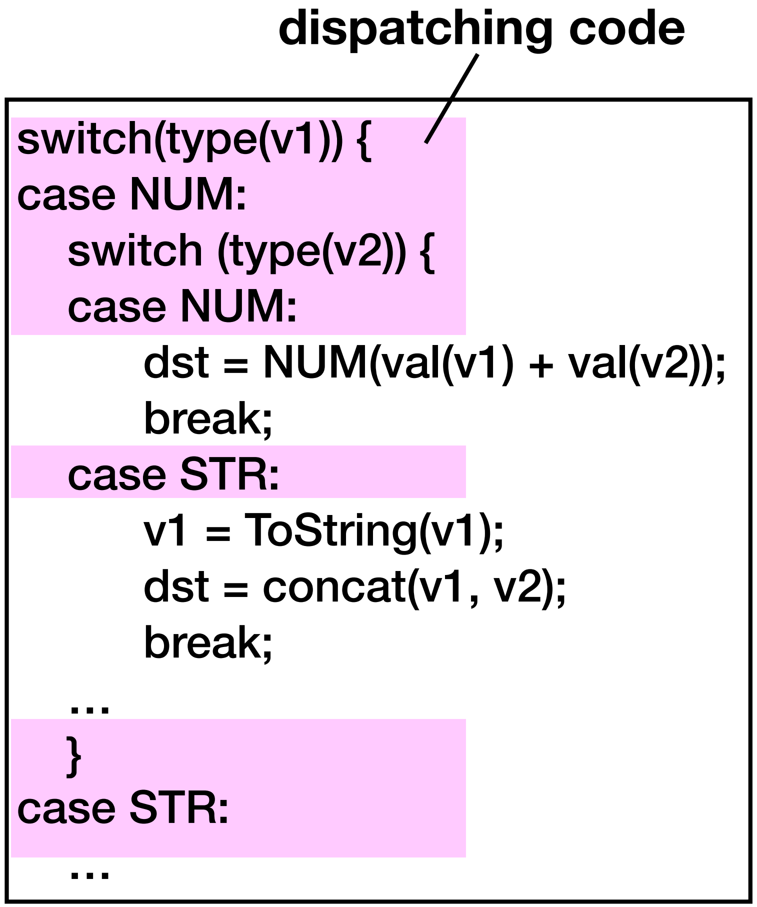
on going

1. VM code related to datatype-based dispatching



Datatype-based Dispatching Code in VM Instruction Interpreter

- Operator overloading
    - Number + Number = Number
    - Number + String = String



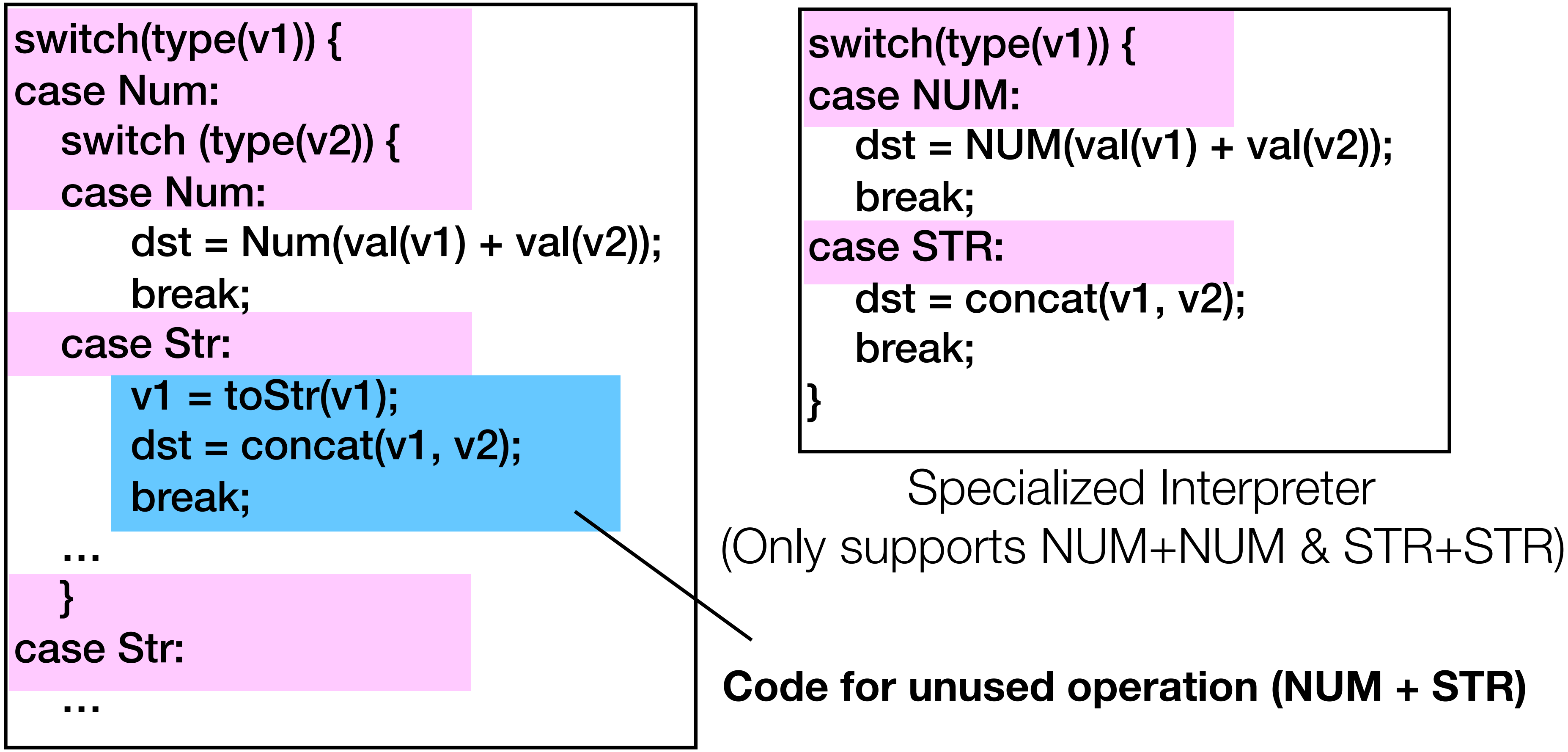
dispatching code

```
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



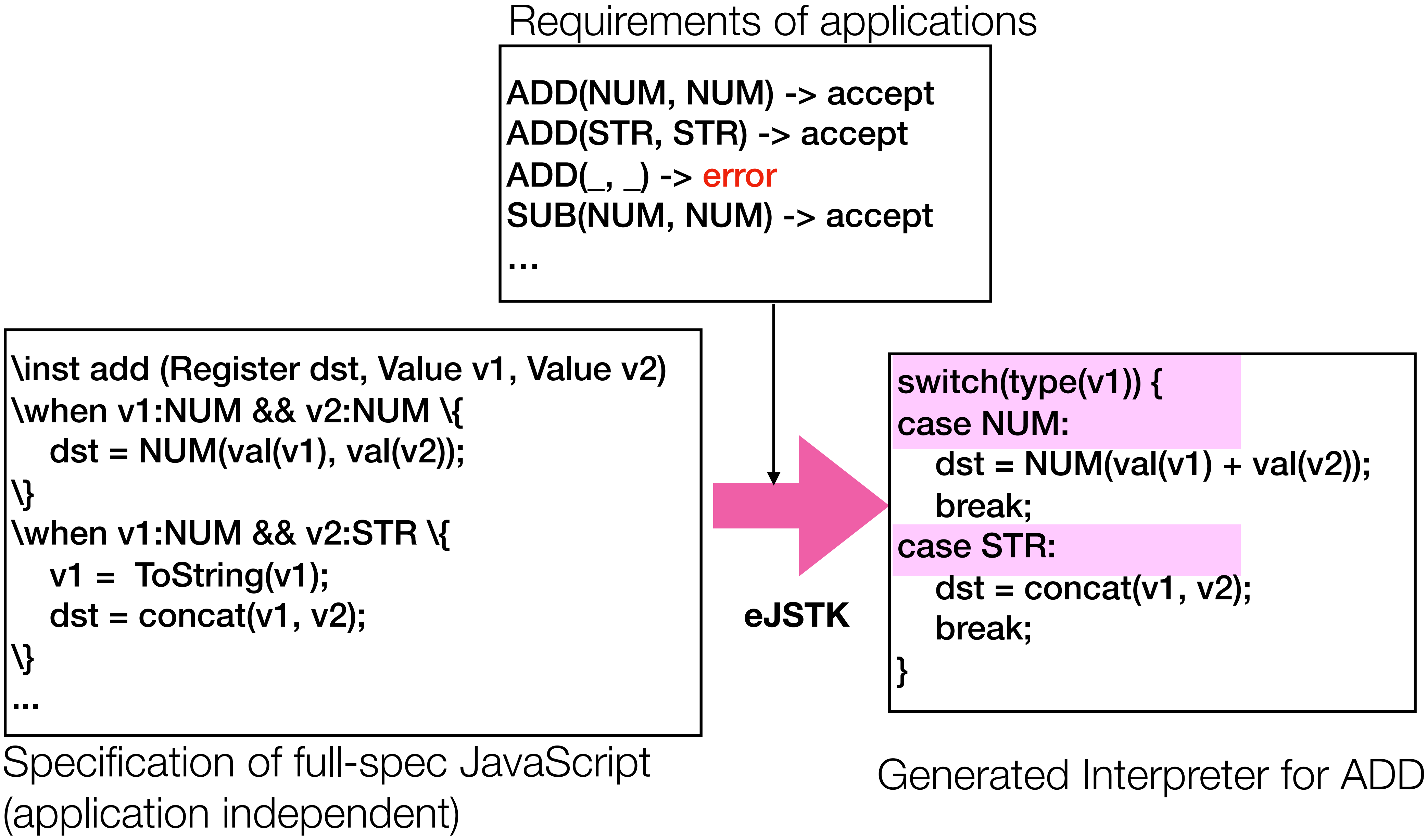
```
switch(type(v1)) {
case Num:
    switch (type(v2)) {
    case Num:
        dst = Num(val(v1) + val(v2));
        break;
    case Str:
        v1 = toStr(v1);
        dst = concat(v1, v2);
        break;
    …
    }
case Str:
    …
```

```
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)

Specialized Interpreter Generator



Requirements of applications

```
ADD(NUM, NUM) -> accept
ADD(STR, STR) -> accept
ADD(_, _) -> error
SUB(NUM, NUM) -> accept
…
```

```
\inst add (Register dst, Value v1, Value v2)
\when v1:NUM && v2:NUM \{
    dst = NUM(val(v1), val(v2));
\}
\when v1:NUM && v2:STR \{
    v1 =  ToString(v1);
    dst = concat(v1, v2);
\}
...
```

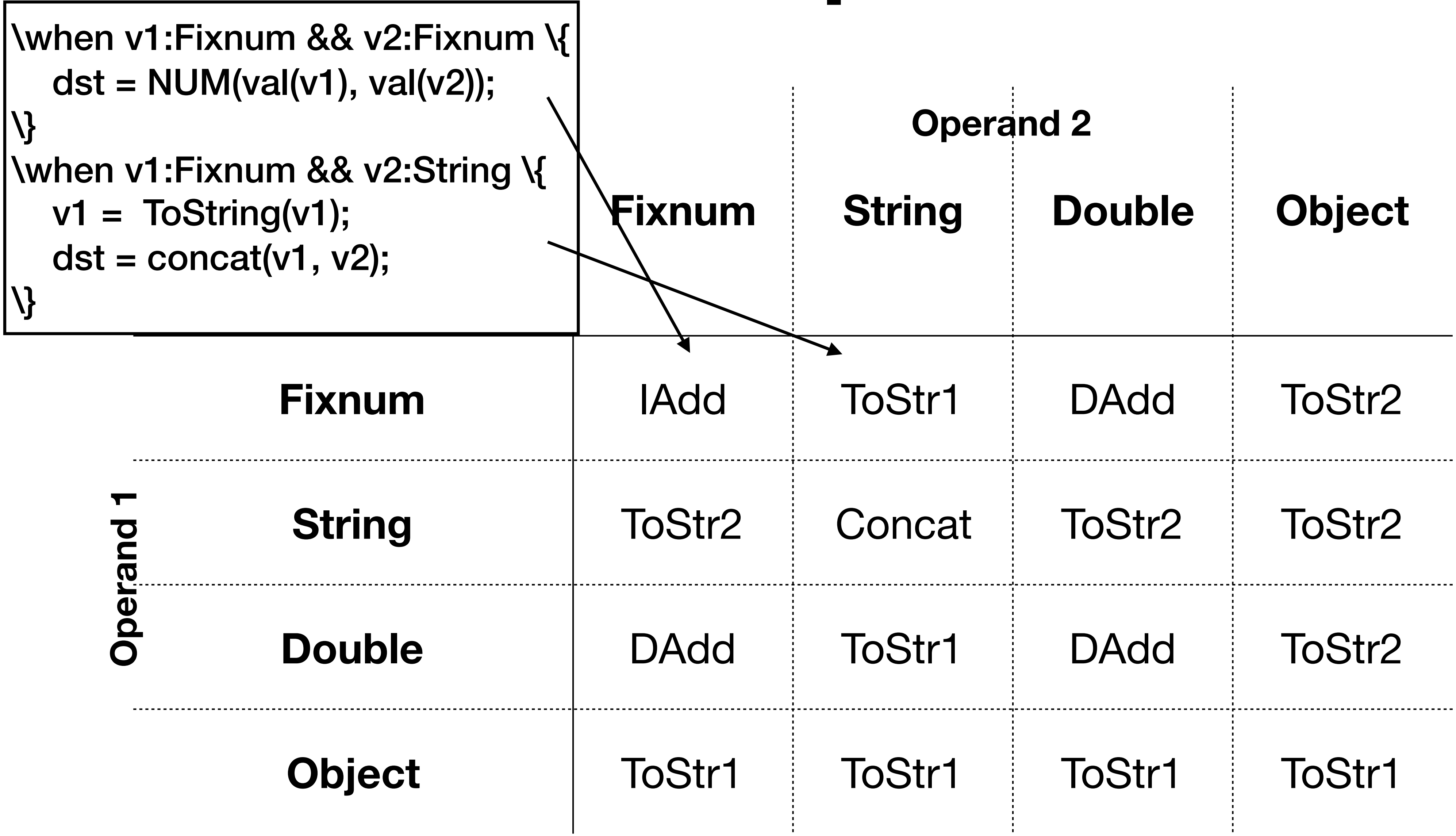
Specification of full-spec JavaScript
(application independent)

eJSTK

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

Generated Interpreter for ADD

Example



```
\when v1:Fixnum && v2:Fixnum \{
    dst = NUM(val(v1), val(v2));
\}
\when v1:Fixnum && v2:String \{
    v1 =  ToString(v1);
    dst = concat(v1, v2);
\}
```

		Operand 2			
		Fixnum	**String**	**Double**	**Object**
Operand 1	**Fixnum**	IAdd	ToStr1	DAdd	ToStr2
	String	ToStr2	Concat	ToStr2	ToStr2
	Double	DAdd	ToStr1	DAdd	ToStr2
	Object	ToStr1	ToStr1	ToStr1	ToStr1

Dispatch Table

Apply Requirements

ADD(Fix, Fix) -> accept

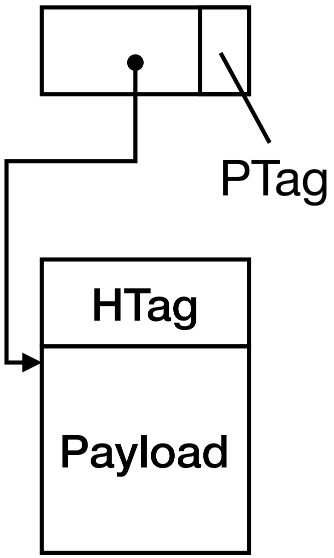
ADD(Fix, Str) -> accept

ADD(Fix, Dbl) -> accept

ADD(Fix, Obj) -> error

…

			Operand 2		
		Fixnum	**String**	**Double**	**Object**
Operand 1	**Fixnum**	IAdd	ToStr1	DAdd	
	String	ToStr2	Concat		ToStr2
	Double	DAdd		DAdd	
	Object		ToStr1		

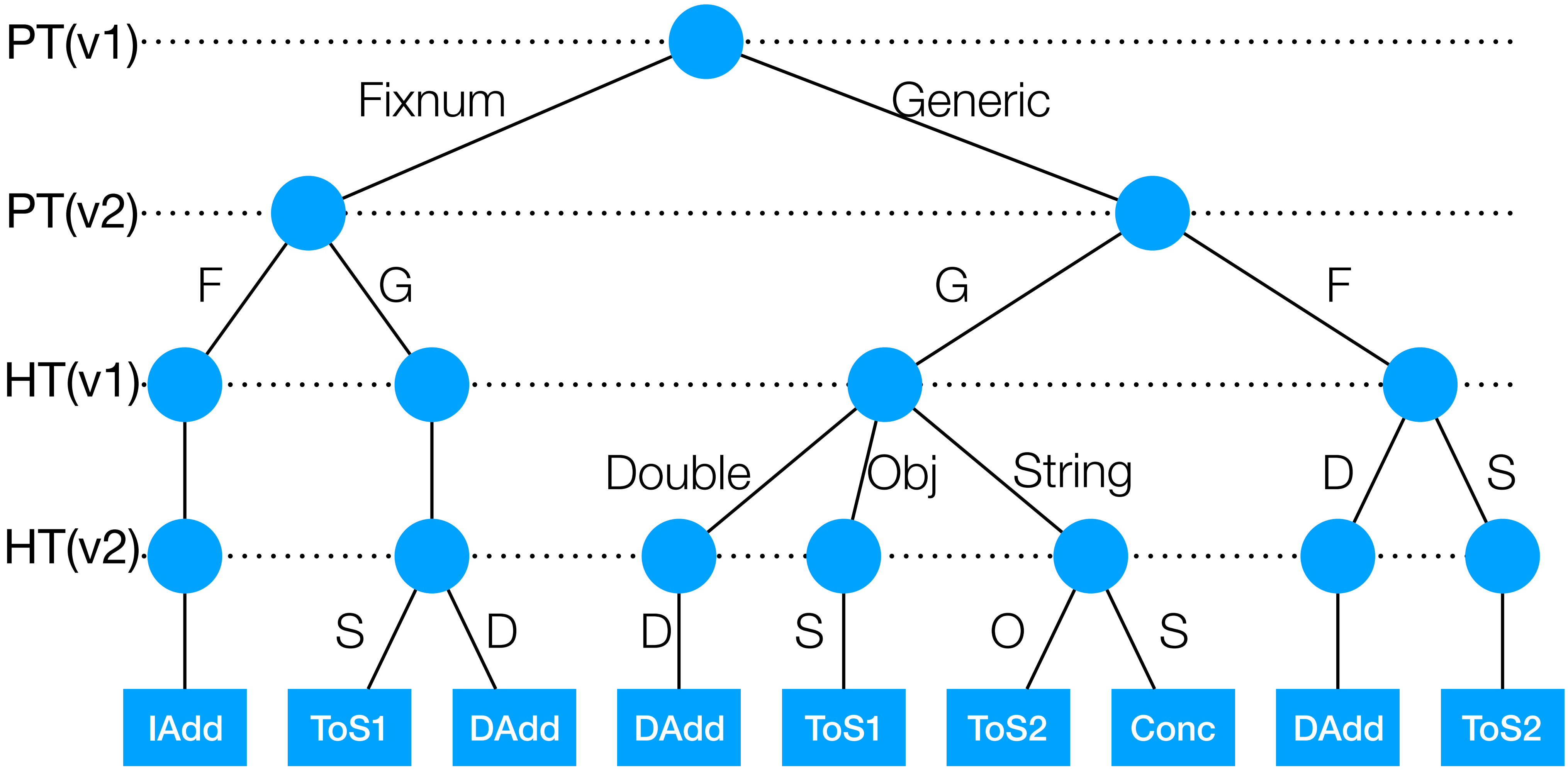


Pointer Tagging

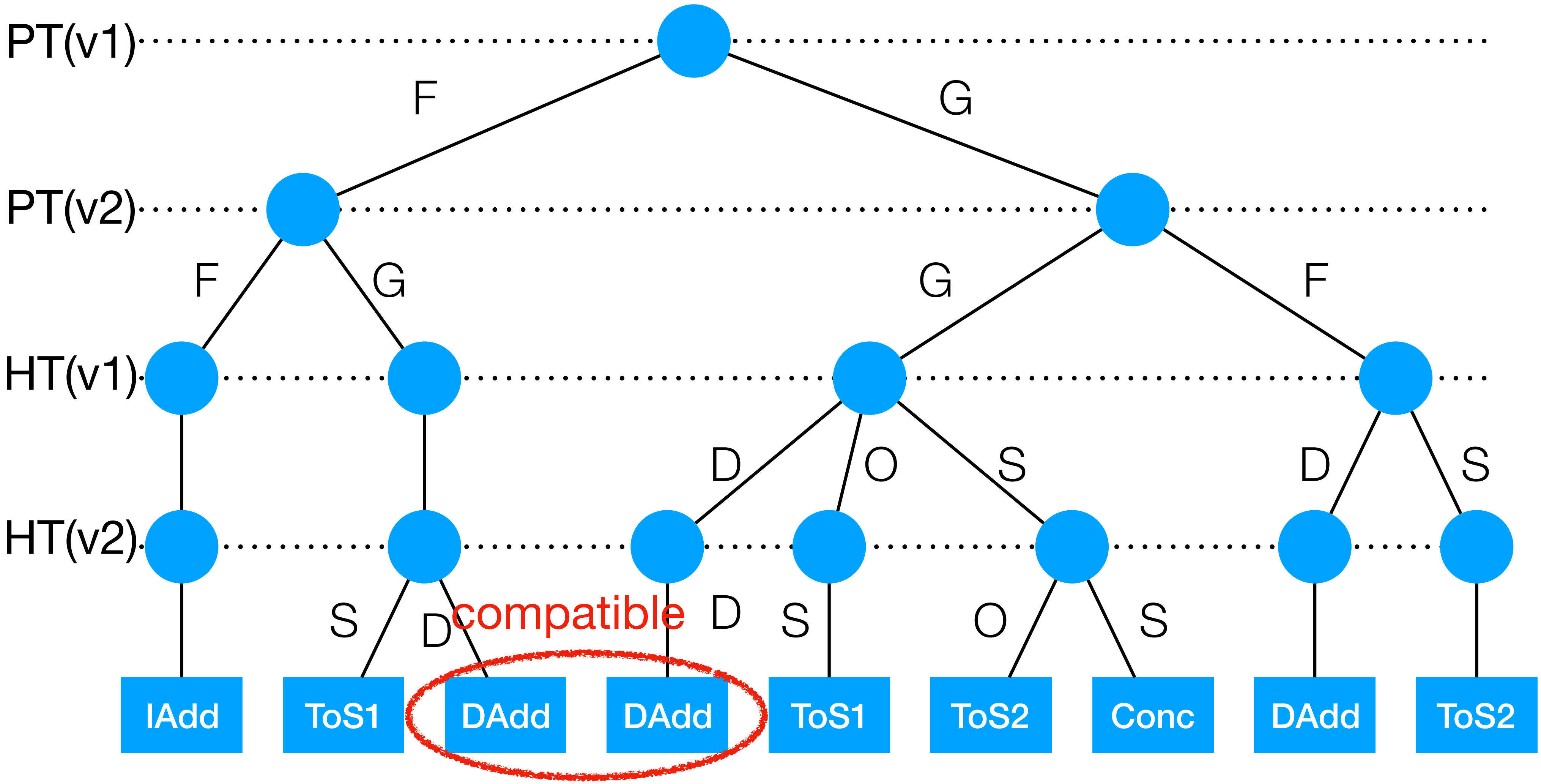
			Operand 2			
		PTag	Fixnum	Generic	Generic	Generic
		HTag	-	String	Double	Object
	PTag	HTag				
Operand 1	Fixnum	-	IAdd	ToStr1	DAdd	
	Generic	String	ToStr2	Concat		ToStr2
	Generic	Double	DAdd		DAdd	
	Generic	Object		ToStr1		

Step 1: Construct Decision Tree

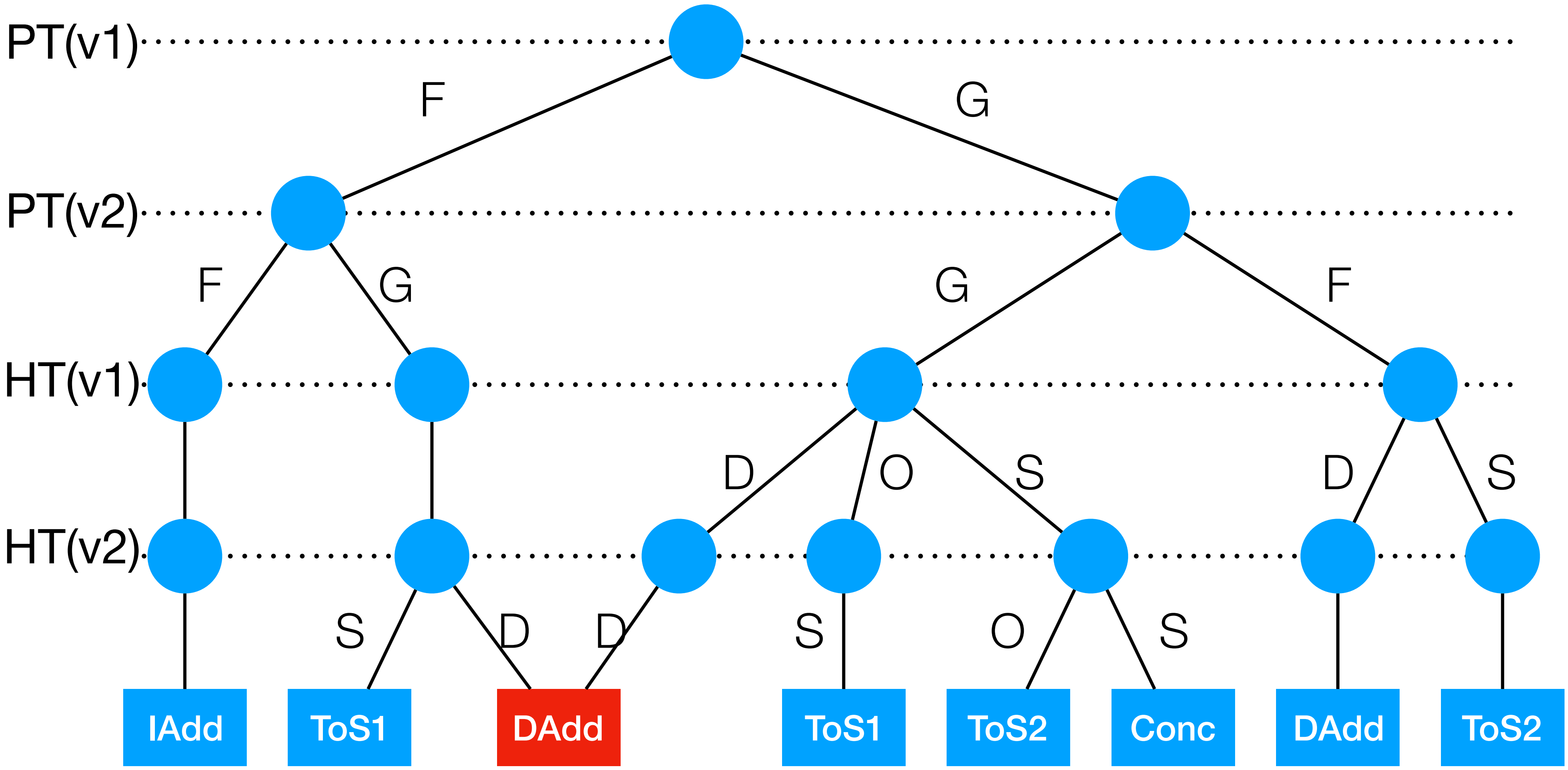
Unlabeled edge means wildcard



Step 2: Combine Subgraphs

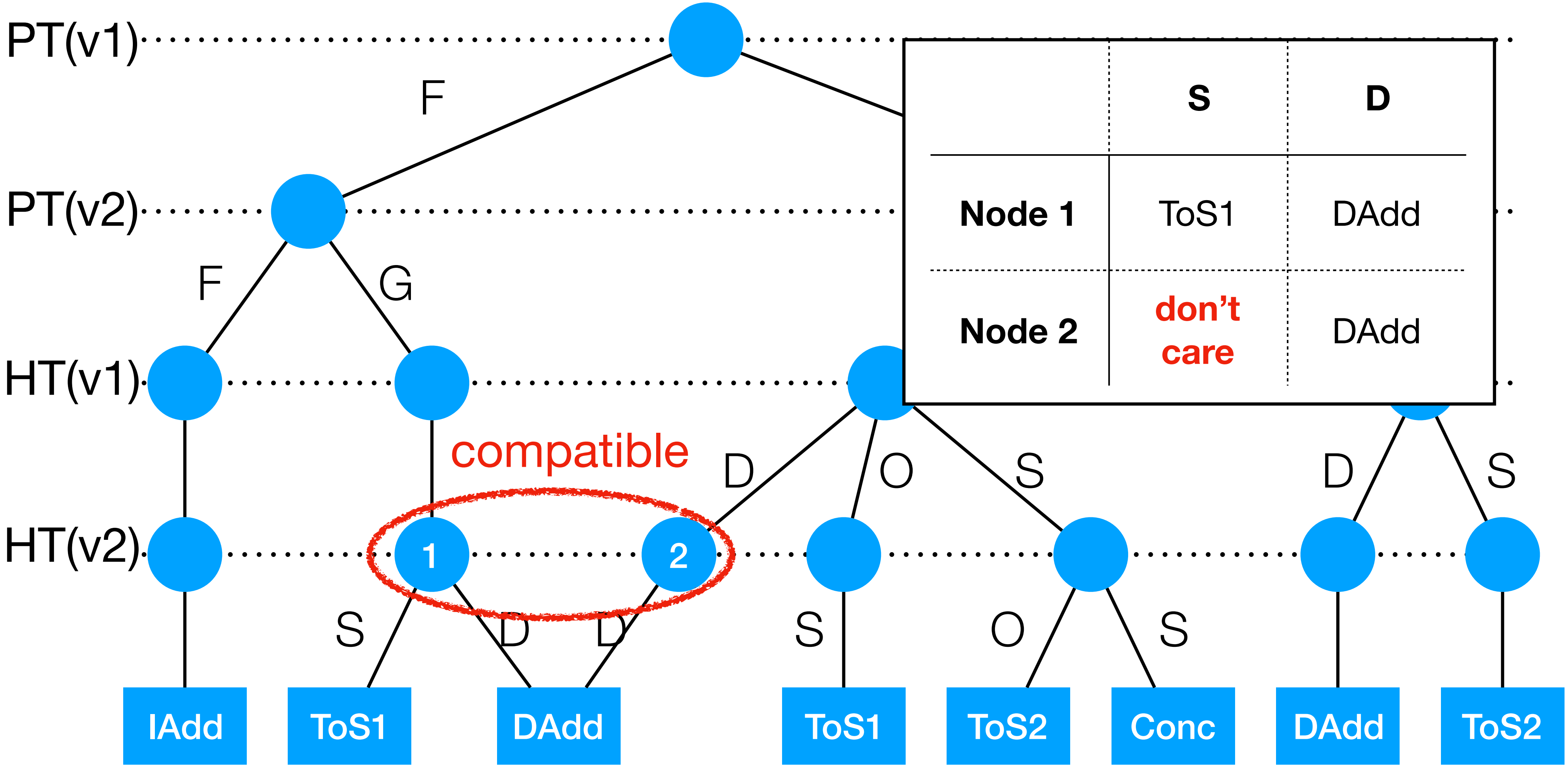


Step 2: Combine Subgraphs



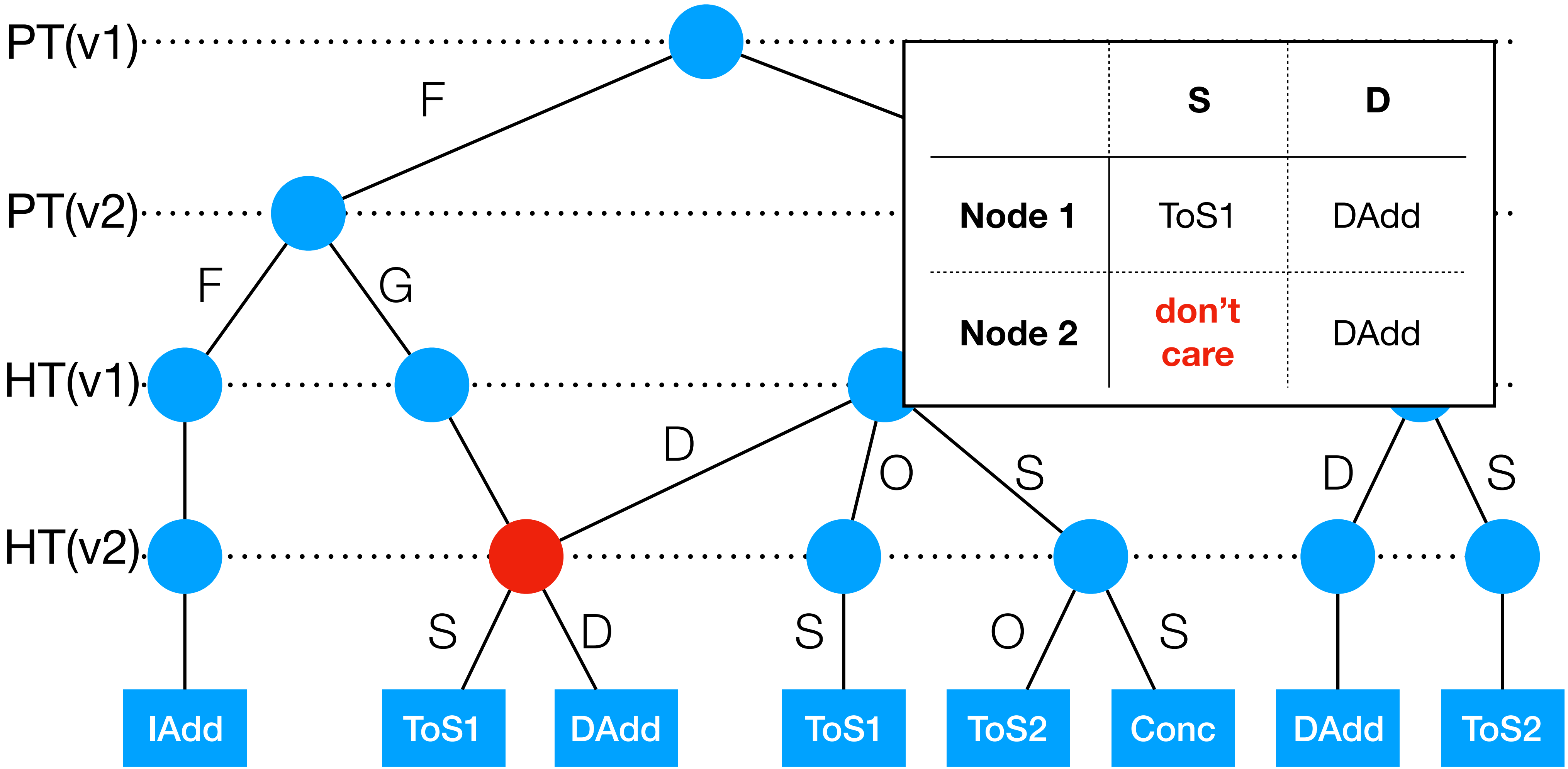
Step 2: Combine Subgraphs

Key idea: Leverage “Don’t care”



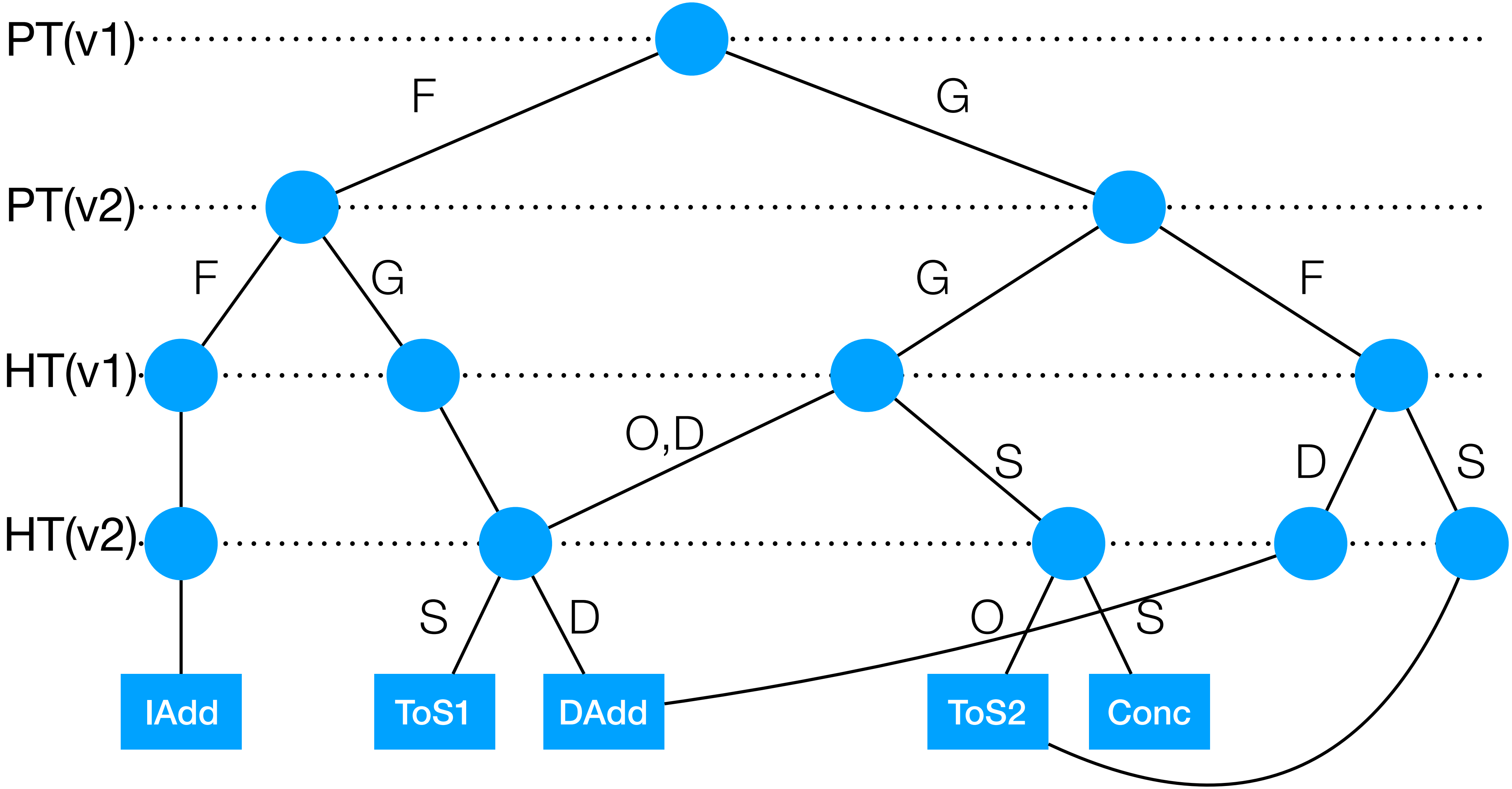
	S	**D**
Node 1	ToS1	DAdd
Node 2	**don’t care**	DAdd

Step 2: Combine Subgraphs

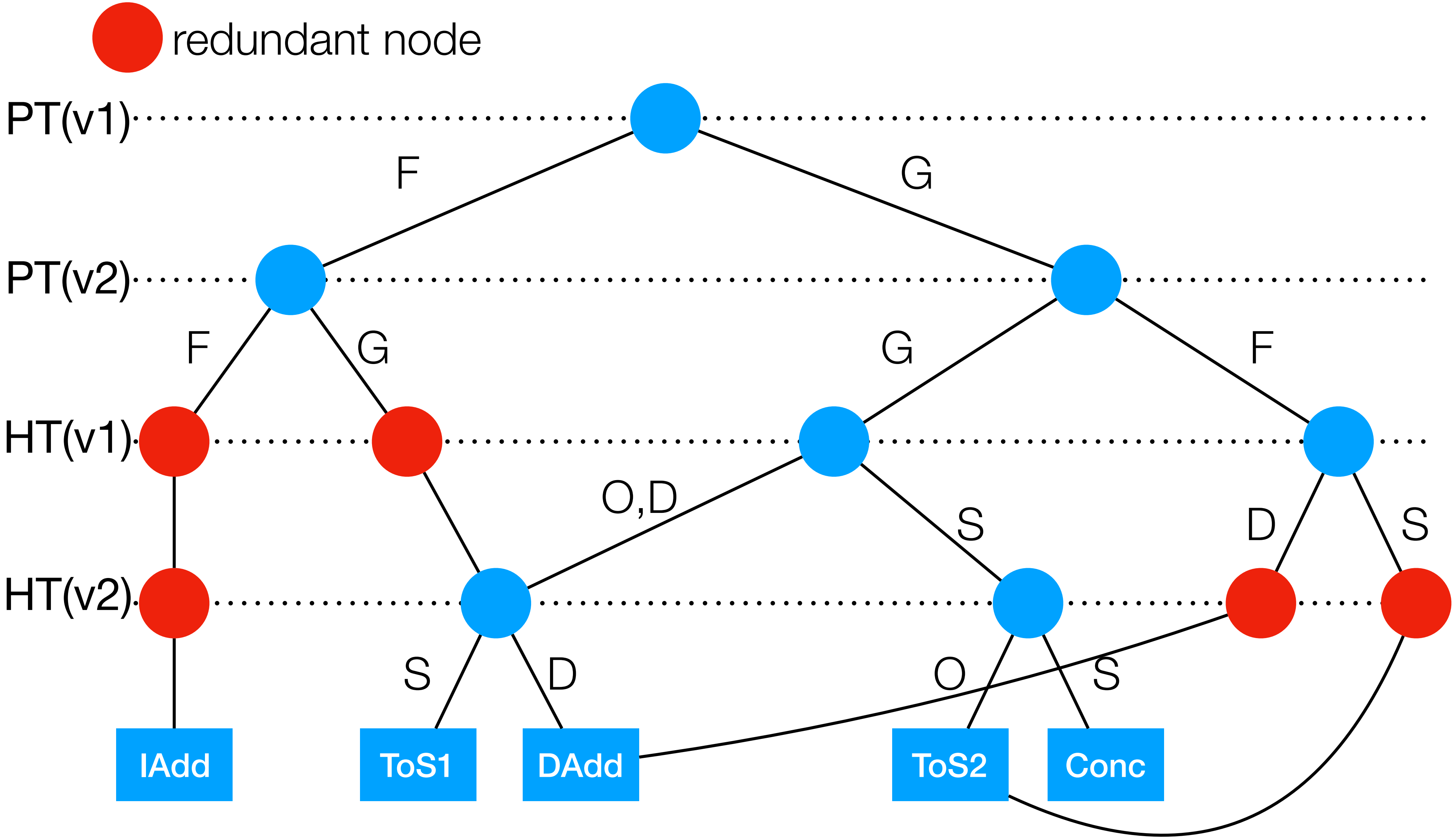


	S	**D**
Node 1	ToS1	DAdd
Node 2	**don’t care**	DAdd

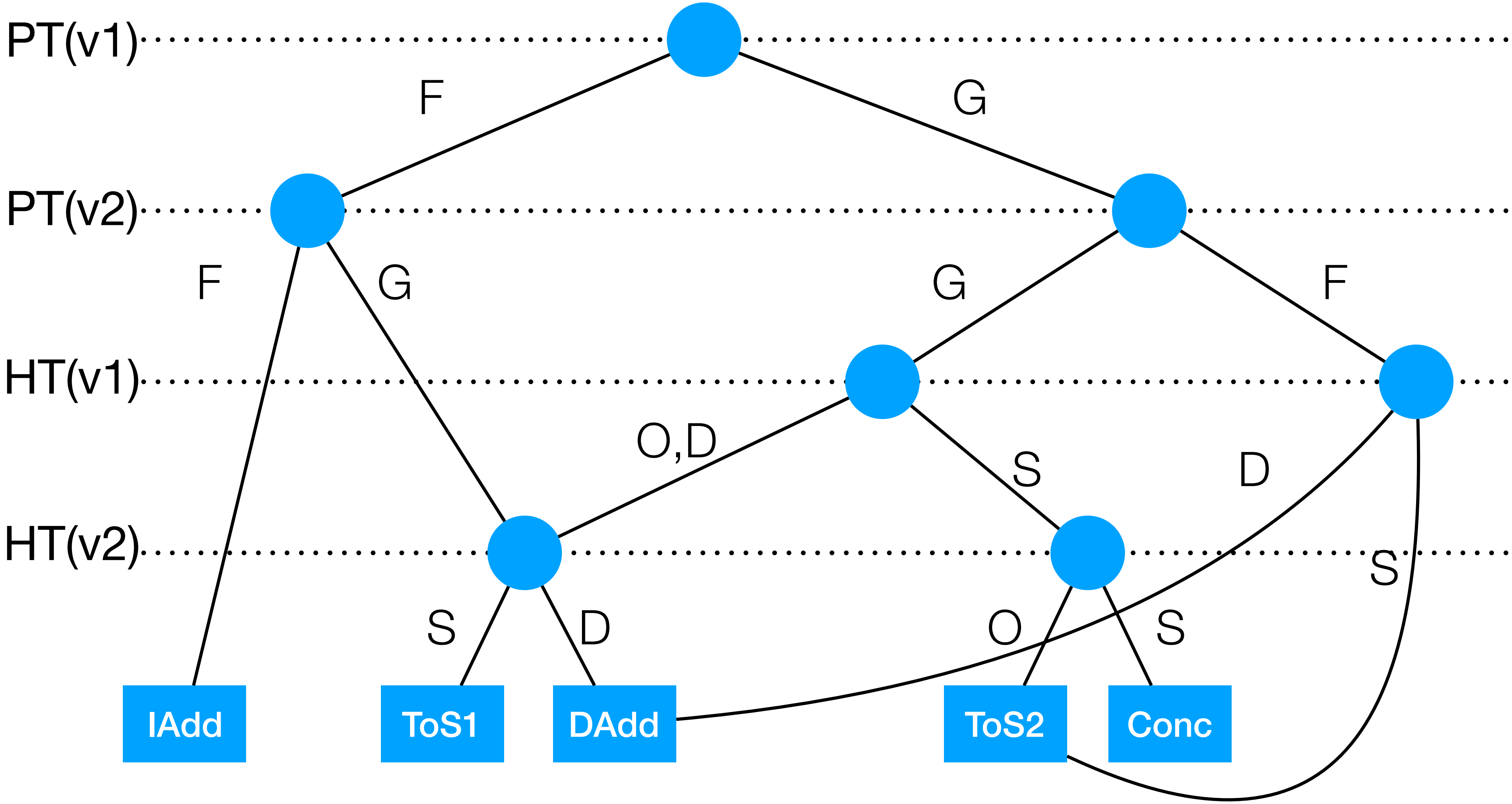
Step 2: Combine Subgraphs



Step 3: Shortcut Redundant Nodes



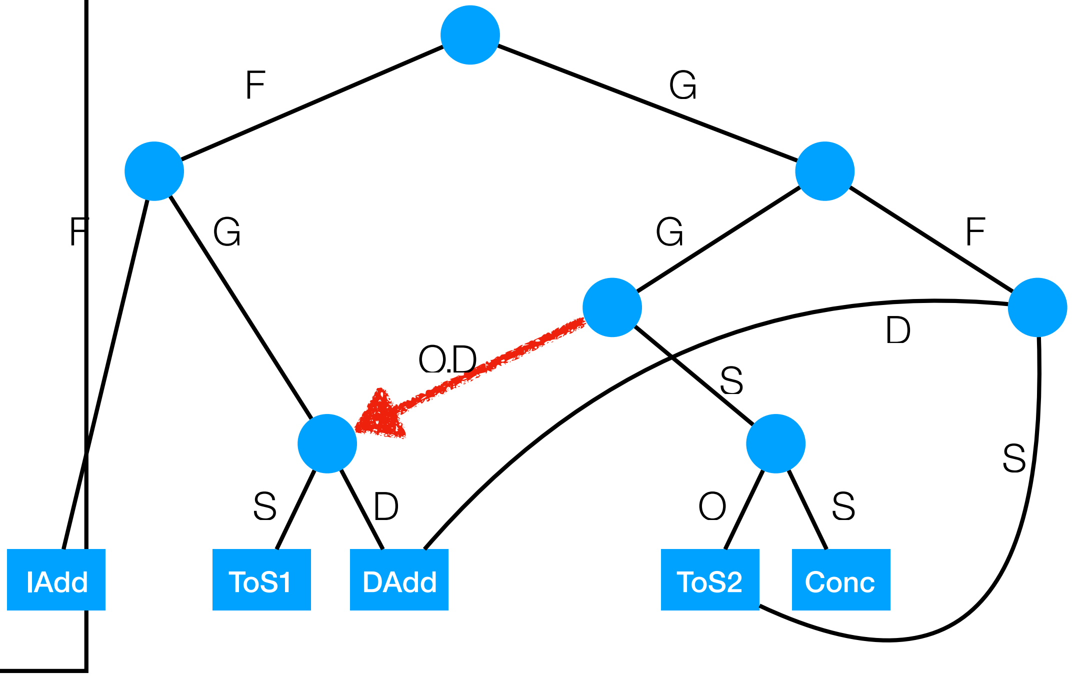
Step 3: Shortcut Redundant Nodes



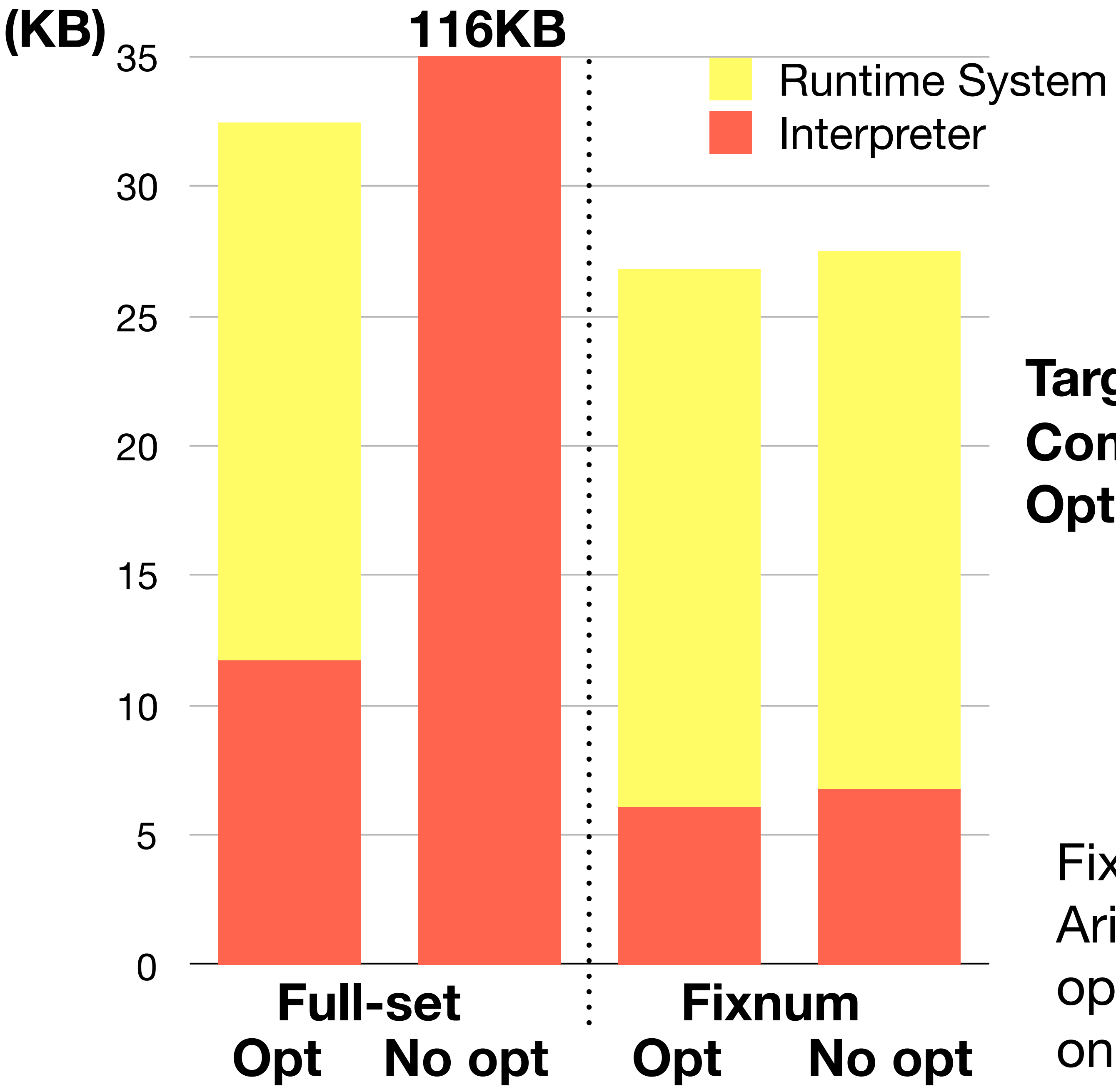
Step 4: Translate to `switch-case`

```
switch (PT(v1)) {
case F:
    switch (PT(v2)) {
    case F: IAdd; break;
    case G:
        L1: switch (HT(v2)) {
        case S: ToS1; break;
        case D: DAdd; break;} break; }
case G:
    switch (PT(v2)) {
    case G:
        switch (HT(v1)) {
        case O: case D: goto L1;
        case S:
            switch (HT(v2)) {
            ...
```

Straightforwardly translate to nested switch-case statement



Sizes of Generated VM

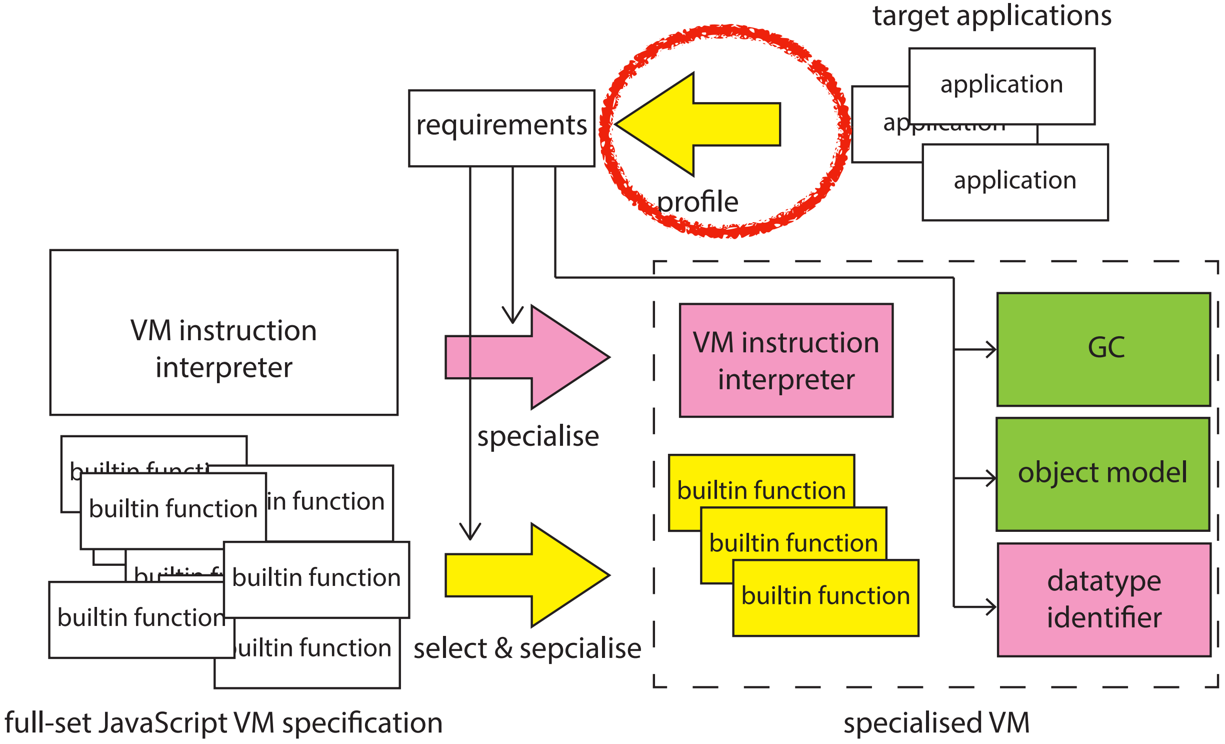


Target: ARM
Compiler: GCC 7.3.0
Option: -Os

Fixnum:
Arithmetic/logical operations supports only fixnums

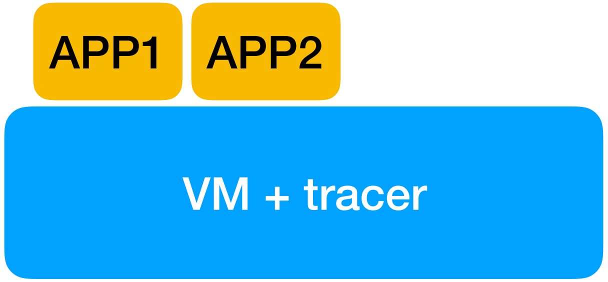
2. Collect requirements of applications

on going



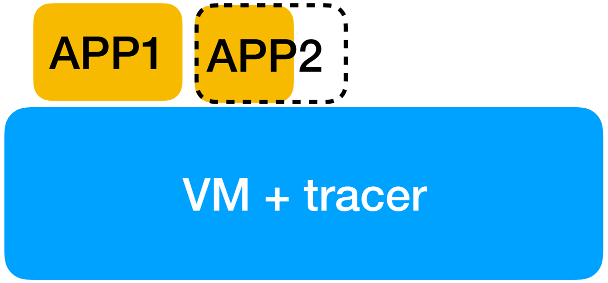
Collect Requirements

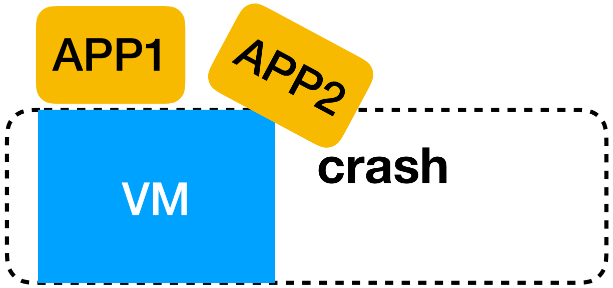
- Collect applications' requirements from test runs
  - Execute apps on full-set JavaScript VM with tracer
- High code coverage in test runs is required
  - VM will crash if collected requirements are insufficient



Collect Requirements

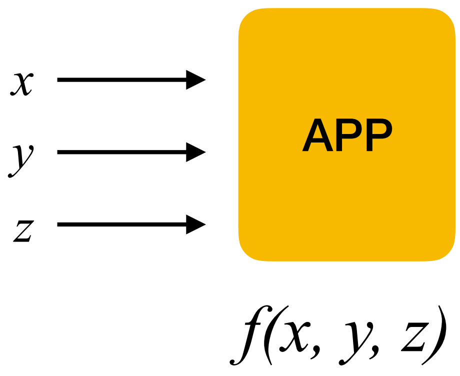
- Collect applications’ requirements by tracing test runs
  - Execute apps on full-spec JavaScript VM with tracer
- High code coverage in test runs is required
  - VM will crash if collected requirements are insufficient

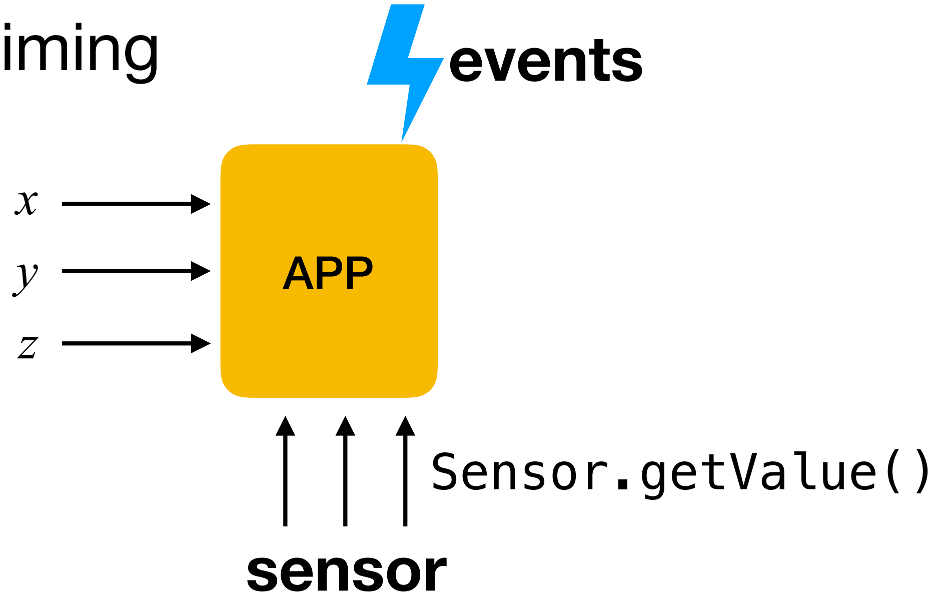




Challenge: Input Generation

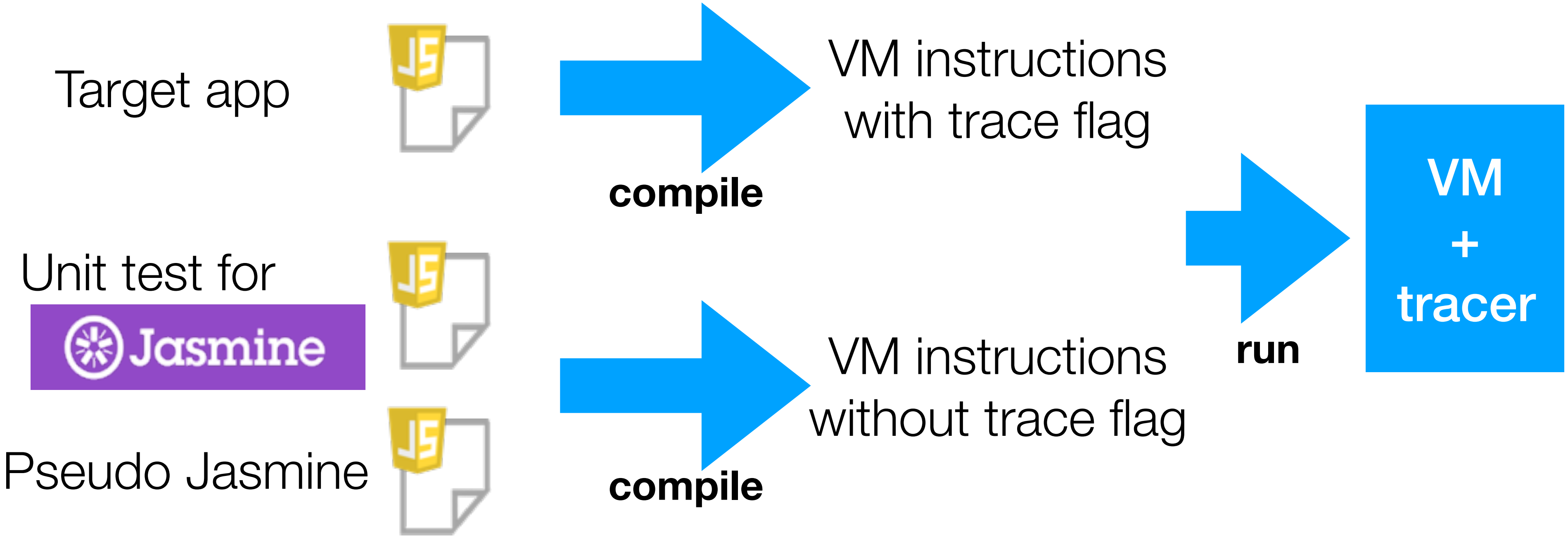
- Parameters
- Polling sensor device using built-in functions
  - Large space to be explored
- Events
  - App's behavior depends on timing





Piggy-back Unit Tests

Assumption:
Application developers write appropriate unit tests



Target app

Unit test for Jasmine

Pseudo Jasmine

compile

VM instructions with trace flag

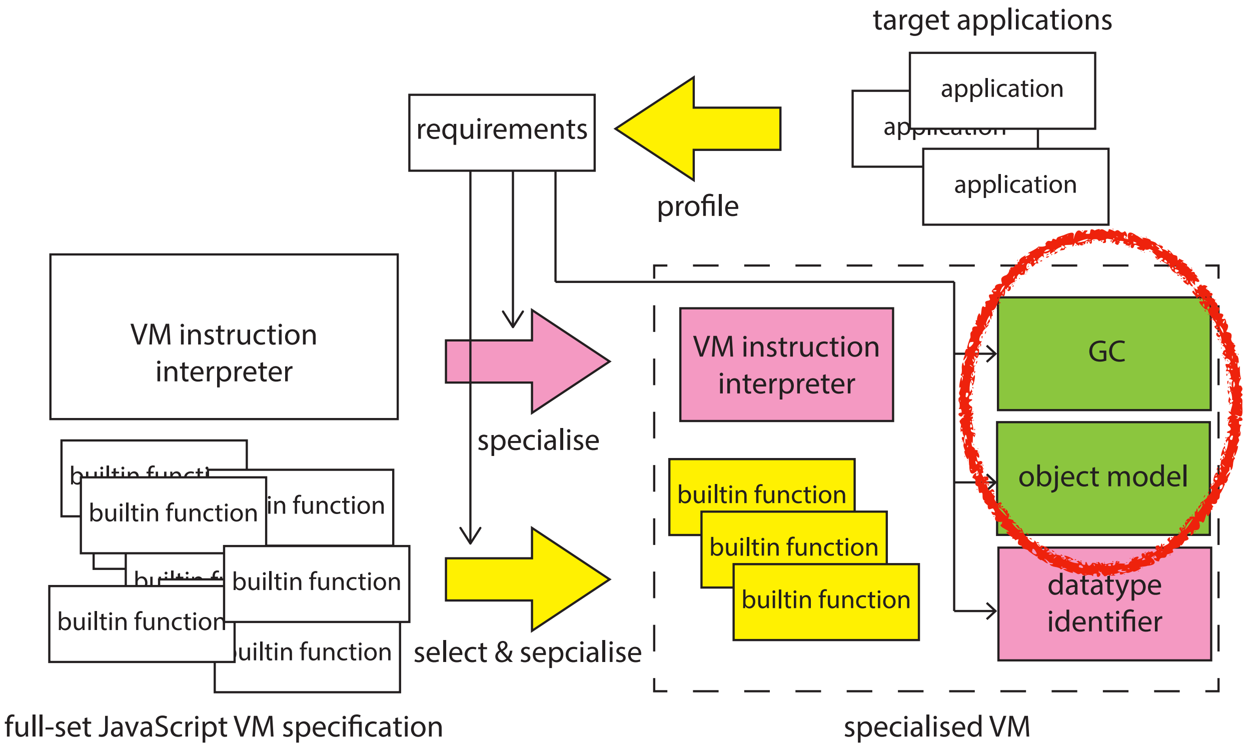
compile

VM instructions without trace flag

run

VM + tracer

3. Object representation future work



Conclusion

- eJSTK:
  Framework for generating customized JavaScript VM for selected set of applications
- Collect applications’ requirements from execution trace using unit tests
- Generate datatype-baed dispatching code
- Customize object representation