A Framework for Constructing JavaScript Virtual Machines with Customized Datatype Representations

Takafumi Kataoka[1], Tomoharu Ugawa[1], Hideya Iwasaki[2]

[1] Kochi University of Technology

[2] The University of Electro-Communications

Background

- JavaScript (JS) offers high productivity
- JS matches event-driven programming style of embedded systems

```
onSensor(function(data){
    currentValue = data;
});
onRequest(function(){
    send(currentValue);
});
```

We are interested in development of embedded systems using JS

Goal

Challenges in embedded system development in JS

- Limited resources
  - Memory
  - CPU speed

Goal

- Construct a VM that is small but runs fast

Characteristics of Embedded Applications

- A particular application hardly uses all the features offered by JS
- We focus on the datatypes that an application uses

Operator overloading in JS

	Example	**Some application**
int + int -> int	1 + 2 -> 3	Use
str + str -> str	‘a’ + ‘b’ -> ‘ab’	Use
int + str -> str	1 + ‘a’ -> ’1a'	Not use
int + bool -> int	1 + true -> 2	Not use
...	...	Not use

Approach

- Automatically generate a customized VM for each application
  - Each VM instruction can only process specific combinations of operand types

Example of ADD instruction

full-set

```
switch(type(op1)){
case INT:
  switch(type(op2)){
  case INT:
    // add two integers
  case STR:
    // convert op1 into string
    // and concatenate two strings
   ...}
case STR: ...}
```

Specialized in INT+INT and STR+STR

```
switch(type(op1)){
case INT:
  // add two integers
default:
  // concatenate
  // two strings
}
```

Proposal: eJSTK

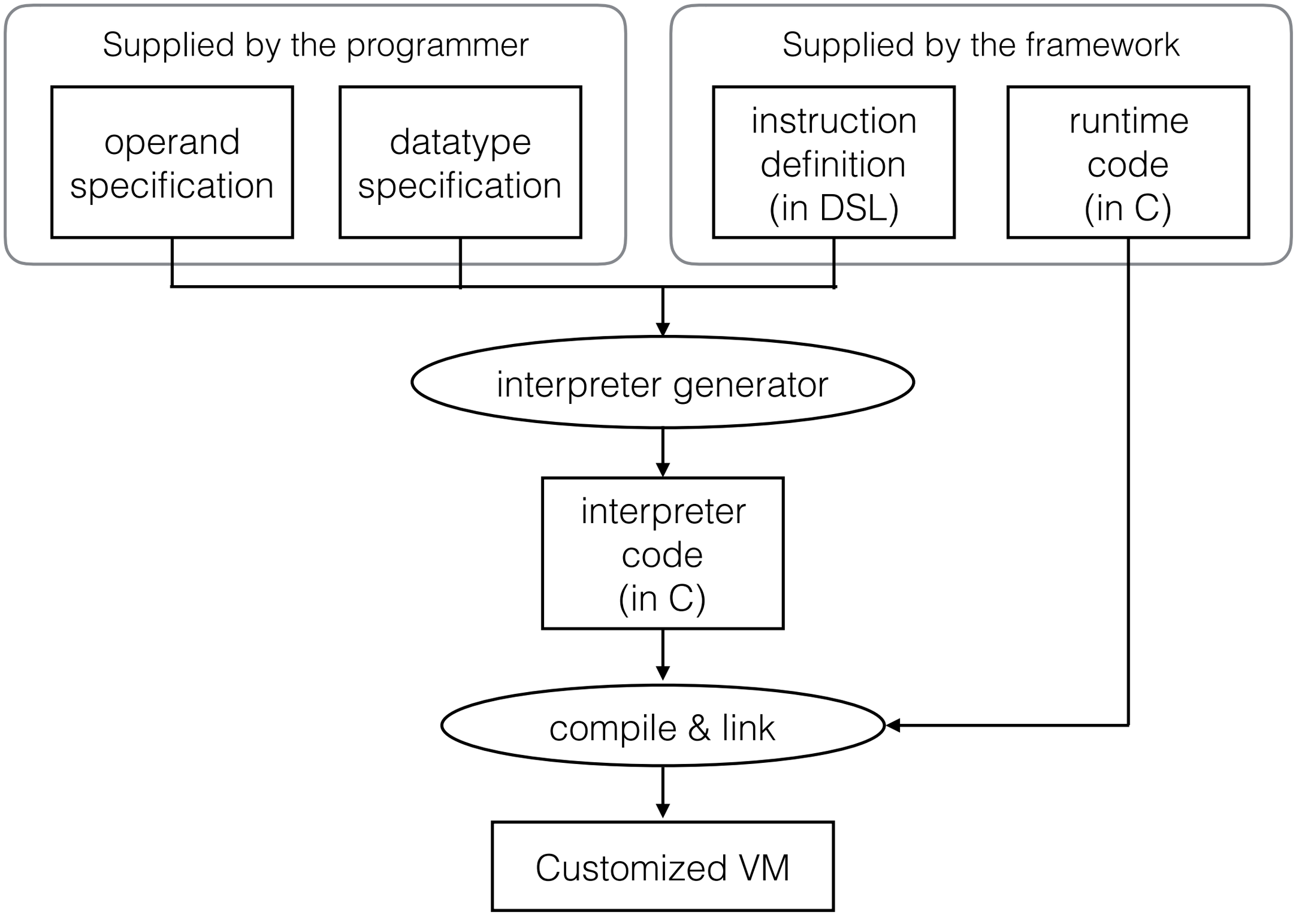
A framework to generate a “tailor-made” VM that is suitable for a particular application

eJSTK (embedded JavaScript Tool Kit)

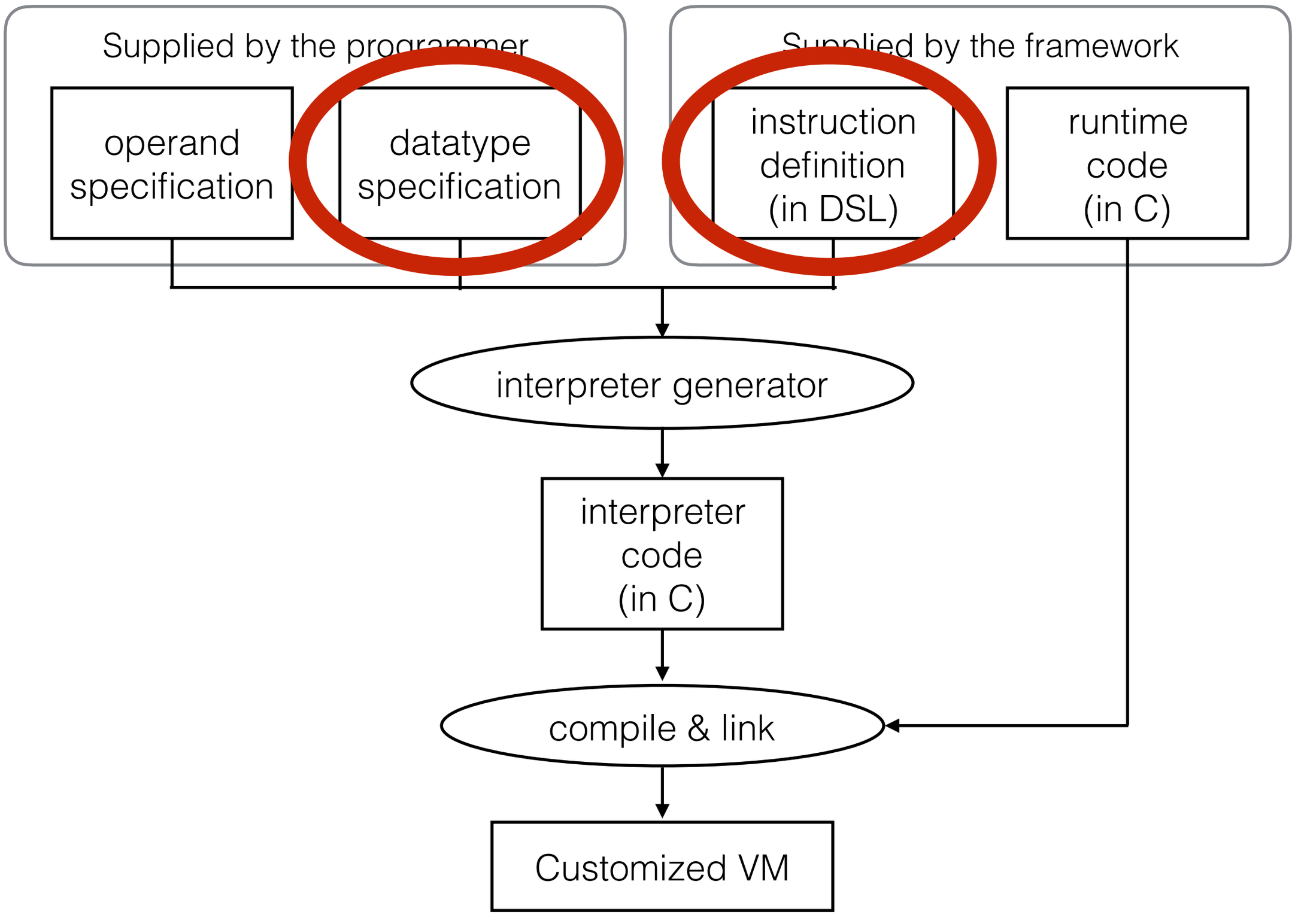
Features of the generated VMs

- Code for impossible datatype combinations of operands is eliminated
- Cost for type-based dispatching is reduced

Overview of eJSTK

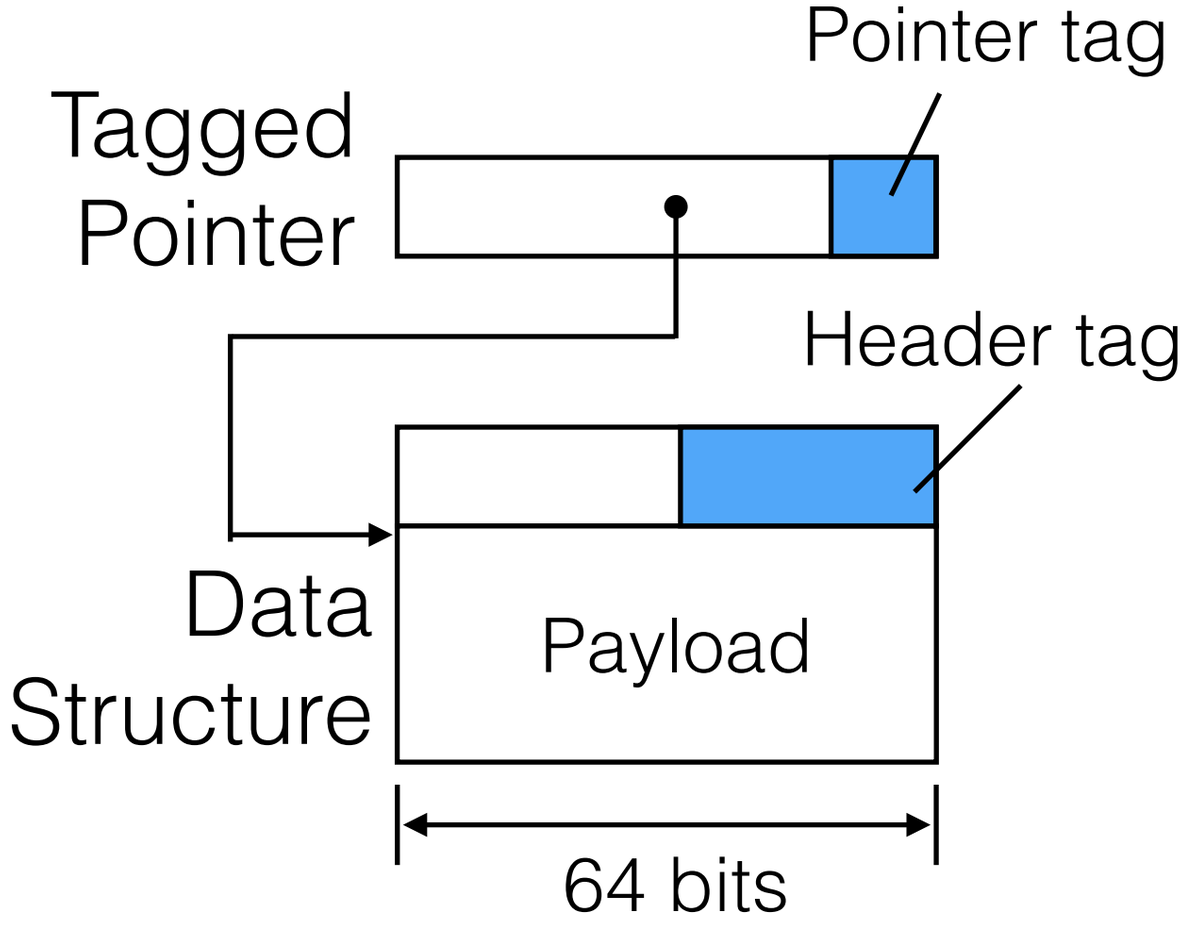


Overview of eJSTK



Datatype Specification

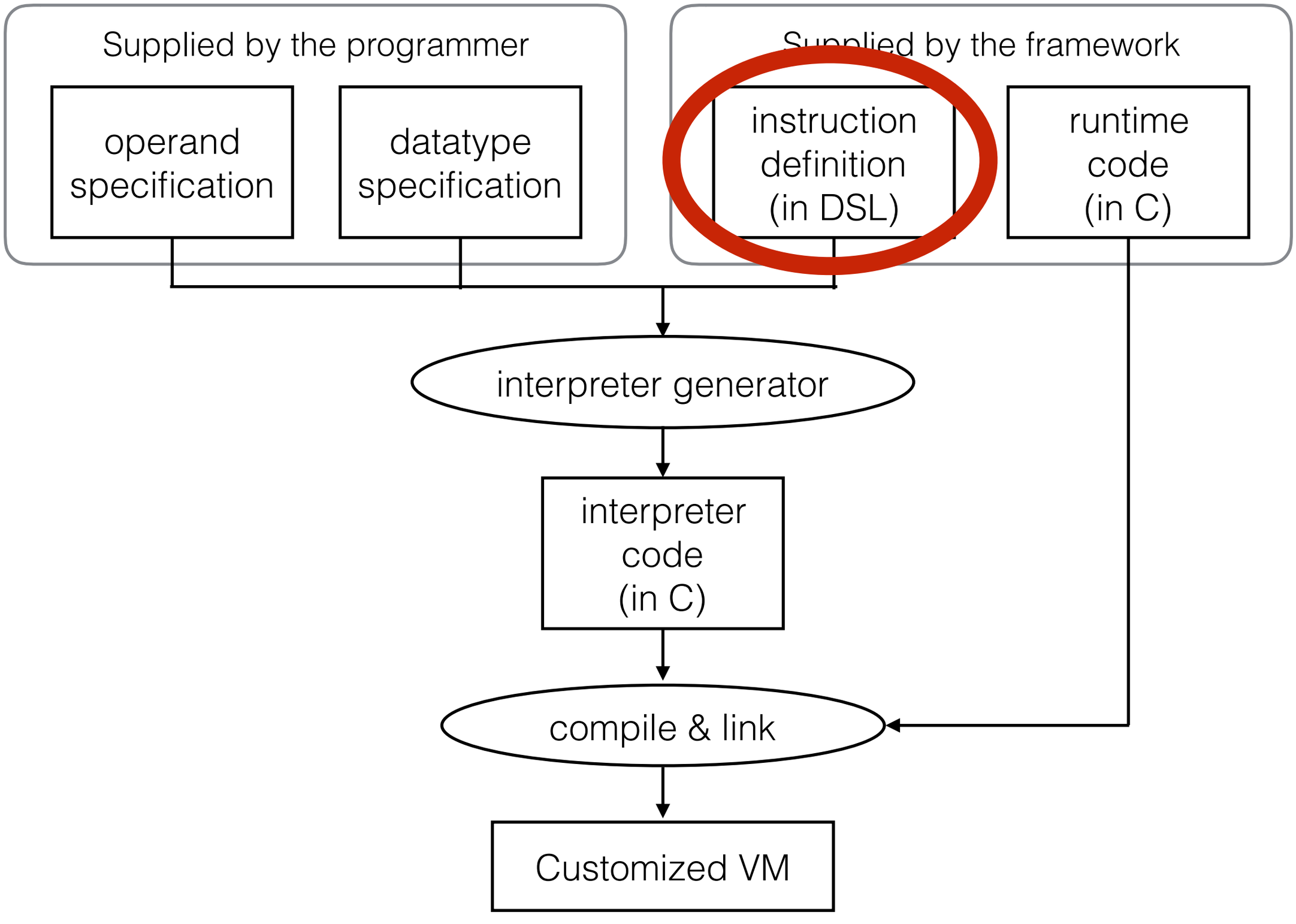
- VM identifies datatype by pointer-tag an header-tag
- Datatype specification specifies assignment of pointer- and header-tags to datatypes
  - Datatypes with unique pointer-tag are identified quickly



Datatype Specification

	Pointer tag	Header tag
int	P_INT	-
string	P_STR	-
array	P_GEN	H_ARY
simple_object	P_GEN	H_OBJ
function	P_GEN	H_FUN
...	...	...

Overview of eJSTK



DSL for Instruction Definition

- Describe dispatch rules based on operand datatypes
- Names of datatypes are independent of concrete representation of datatypes

```
// op1 + op2
\inst add (Register dst, Value op1, Value op2)
\when op1:int && op2:int \{
  // C code fragment for operands of (INT,INT)
\}
\when op1:string && op2:string \{
  // C code fragment for operands of (STR,STR)
\}
\when op1:int && op2:string \{
  // C code fragment for operands of (INT,STR)
\}
...
```

DSL for Instruction Definition

- Describe dispatch rules based on operand datatypes
- Names of datatypes are independent of concrete representation of datatypes

```
// op1 + op2
\inst add (Register dst, Value op1, Value op2)
\when op1:int && op2:int \{
  // C code fragment for operands of (INT,INT)
\}
\when op1:string && op2:string \{
  // C code fragment for operands of (STR,STR)
\}
\when op1:int && op2:string \{
  // C code fragment for operands of (INT,STR)
\}
...
```

Example of Generated Code for ADD

`PTR_TAG(x)`: pointer-tag of x
`HDR_TAG(x)`: header-tag of x

	Pointer tag	Header tag
int	P_INT	-
string	P_STR	-
array	P_GEN	H_ARY
simple_object	P_GEN	H_OBJ
...	...	...

```
// op1 + op2
switch(PTR_TAG(op1)){
case P_INT:
  switch(PTR_TAG(op2)){
  case P_INT:
    // C code fragment for operands of (INT,INT)
  case P_STR:
    // C code fragment for operands of (INT,STR)
  case P_GEN:
    switch(HDR_TAG(op2)){
    case H_ARY:
      // C code fragment for operands of (INT,ARY)
    case H_OBJ:
      // C code fragment for operands of (INT,OBJ)
...
```

Example of Generated Code for ADD

PTR_TAG(x): pointer-tag of x
HDR_TAG(x): header-tag of x

	Pointer tag	Header tag
int	P_INT	-
string	P_STR	-
array	P_GEN	H_ARY
simple_object	P_GEN	H_OBJ
...	...	...

Type-based dispatch

```
// op1 + op2
switch(PTR_TAG(op1)){
case P_INT:
  switch(PTR_TAG(op2)){
  case P_INT:
    // C code fragment for
  case P_STR:
    // C code fragment for operands of (INT,STR)
  case P_GEN:
    switch(HDR_TAG(op2)){
    case H_ARY:
      // C code fragment for operands of (INT,ARY)
    case H_OBJ:
      // C code fragment for operands of (INT,OBJ)
...
```

Example of Generated Code for ADD

PTR_TAG(x): pointer-tag of x
HDR_TAG(x): header-tag of x

	Pointer tag	Header tag
int	P_INT	-
string	P_STR	-
array	P_GEN	H_ARY
simple_object	P_GEN	H_OBJ
...	...	...

C code fragment (copied as-is from instruction definition)

```
{
  case P_INT:
    // C code fragment for operands of (INT,INT)
  case P_STR:
    // C code fragment for operands of (INT,STR)
  case P_GEN:
    switch(HDR_TAG(op2)){
    case H_ARY:
      // C code fragment for operands of (INT,ARY)
    case H_OBJ:
      // C code fragment for operands of (INT,OBJ)
...
```

Example of Generated Code for ADD

`PTR_TAG(x)`: pointer-tag of x
`HDR_TAG(x)`: header-tag of x

	Pointer tag	Header tag
int	P_INT	-
string	P_STR	-
array	**P_GEN**	**H_ARY**
simple_object	**P_GEN**	**H_OBJ**
...	...	...

```
// op1 + op2
switch(PTR_TAG(op1)){
case P_INT:
  switch(PTR_TAG(op2)){
    // C code fragment for operands of (INT,INT)
    // C code fragment for operands of (INT,STR)
  case P_GEN:
    switch(HDR_TAG(op2)){
    case H_ARY:
      // C code fragment for operands of (INT,ARY)
    case H_OBJ:
      // C code fragment for operands of (INT,OBJ)
...
```

dispatch based on header-tag

VM Customization

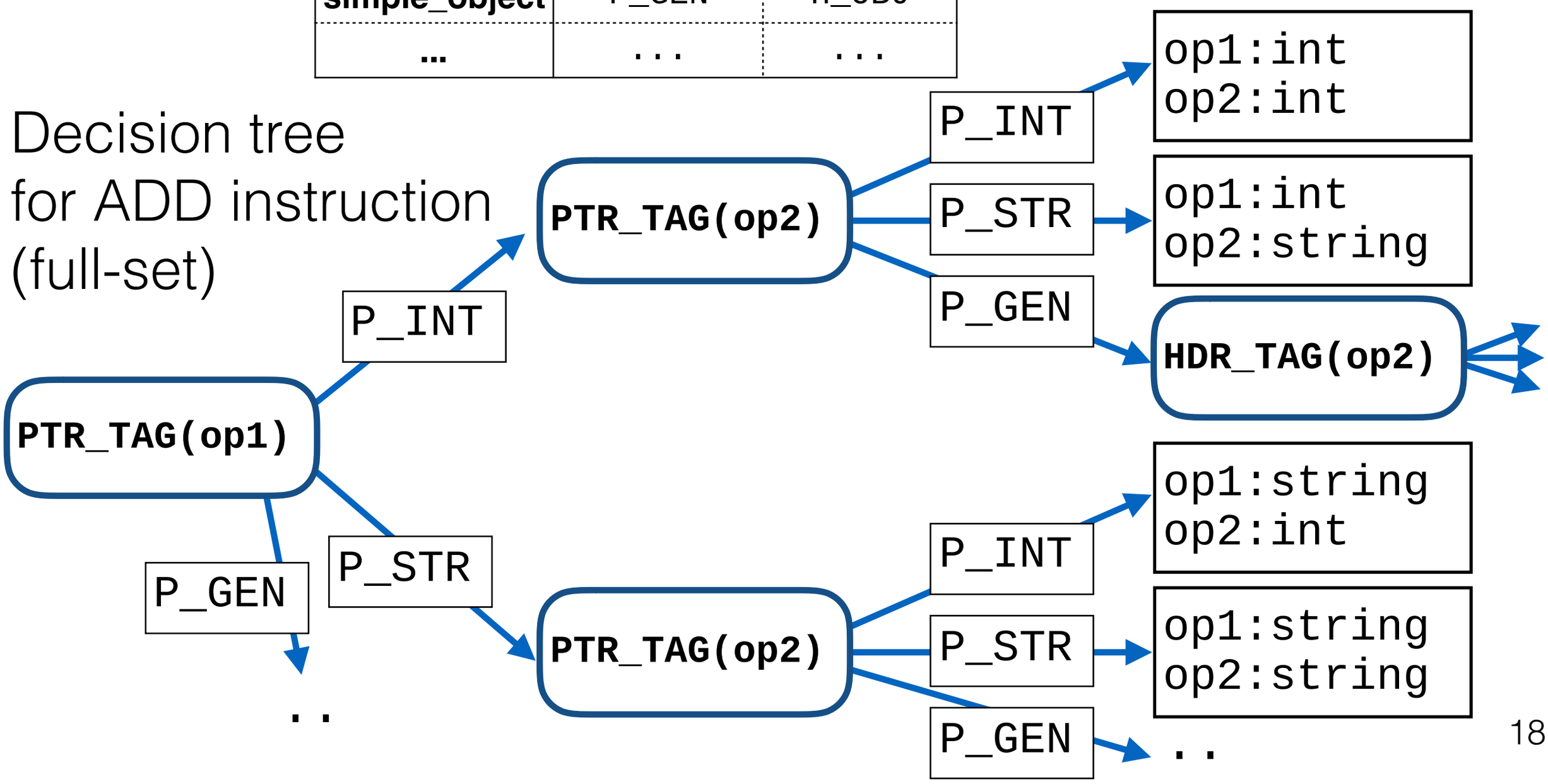
Supported customization

- Limiting operand datatypes of VM instructions
- Changing datatype representation

Limiting Operand Datatypes

	Pointer tag	Header tag
int	P_INT	-
string	P_STR	-
array	P_GEN	H_ARY
simple_object	P_GEN	H_OBJ
...	...	...

Decision tree for ADD instruction (full-set)



Generated Code for ADD instruction

`PTR_TAG(x)`: pointer-tag of x
`HDR_TAG(x)`: header-tag of x

	Pointer tag	Header tag
int	P_INT	-
string	P_STR	-
array	P_GEN	H_ARY
simple_object	P_GEN	H_OBJ
...	...	...

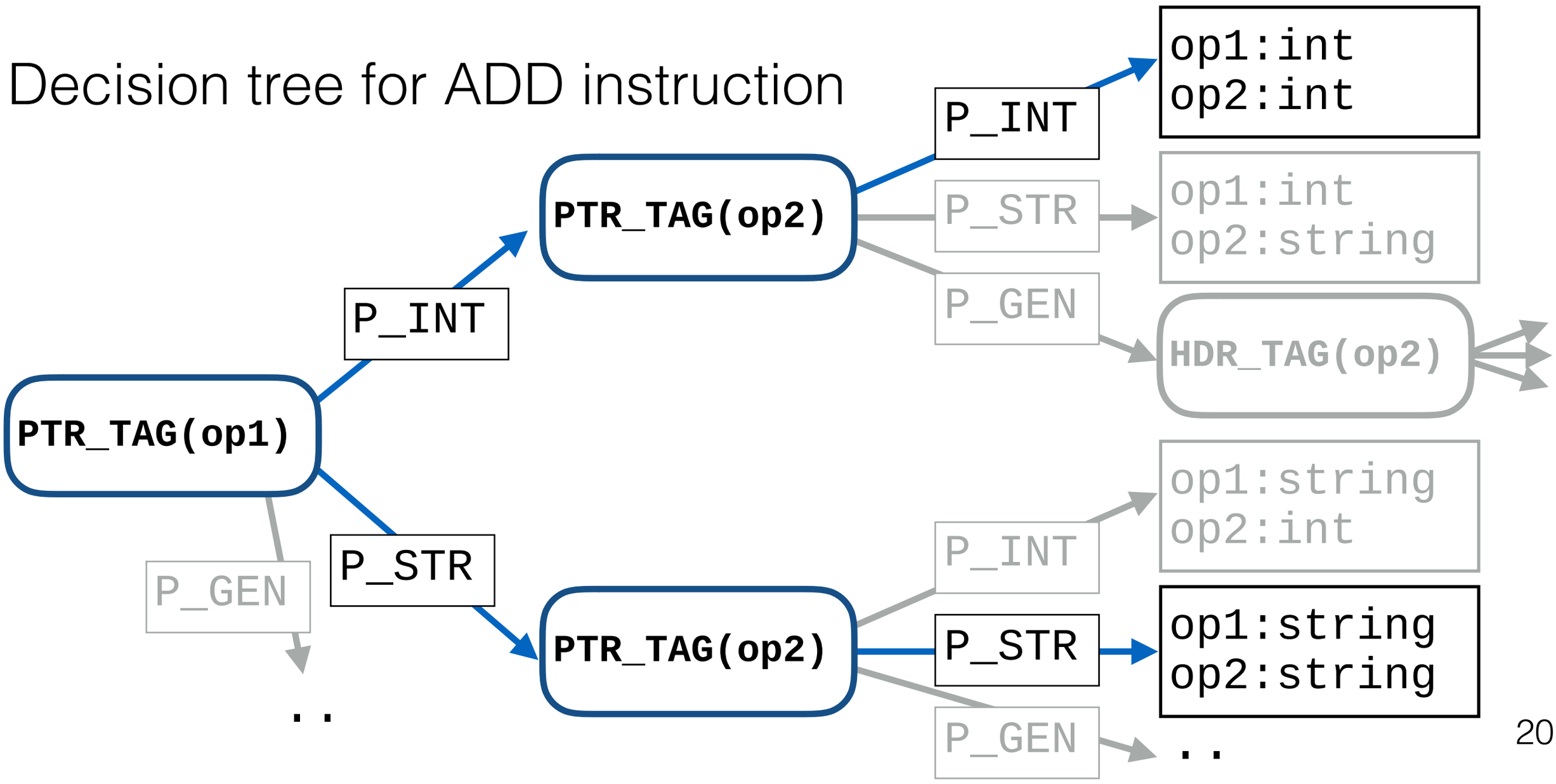
```
// op1 + op2
switch(PTR_TAG(op1)){
case P_INT:
  switch(PTR_TAG(op2)){
  case P_INT:
    // C code fragment for operands of (INT,INT)
  case P_STR:
    // C code fragment for operands of (INT,STR)
  case P_GEN:
    switch(HDR_TAG(op2)){
    case H_ARY:
      // C code fragment for operands of (INT,ARY)
    case H_OBJ:
      // C code fragment for operands of (INT,OBJ)
...
```

Limiting Operand Datatypes

Example: ADD instruction specialized in (INT, INT) and (STR, STR)

- eJSTK generates simple dispatch code w.r.t. the limitation

Decision tree for ADD instruction

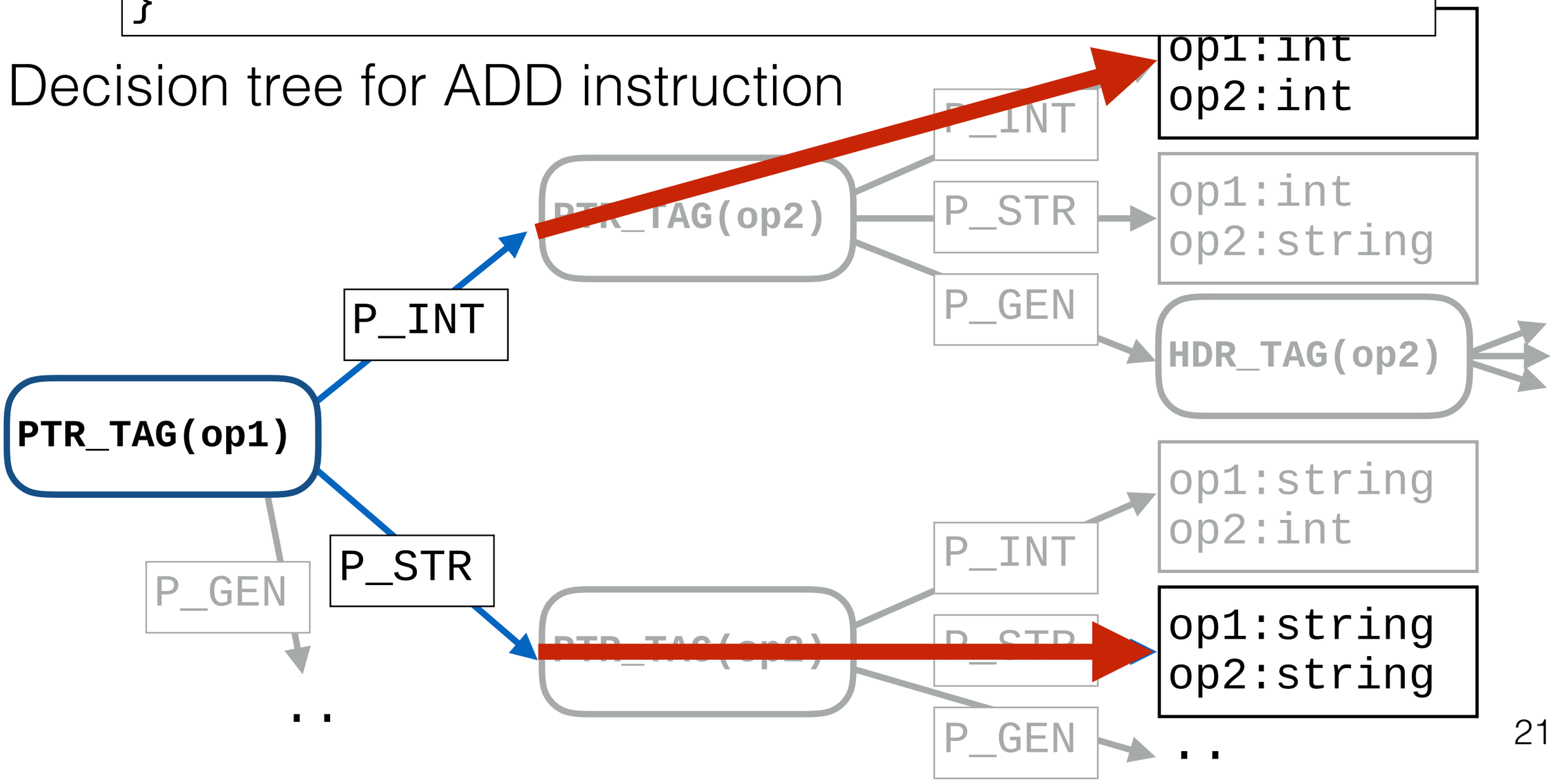


Limiting Operand Datatypes

```
// op1 + op2
switch(PTR_TAG(op1)){
case P_INT:
  // C code fragment for operands of (INT,INT)
default:
  // C code fragment for operands of (STR,STR)
}
```

- E… …R)

Decision tree for ADD instruction



Limiting Operand Datatypes

- Ex

```
// op1 + op2
switch(PTR_TAG(op1)){
case P_INT:
  // C code fragment for operands of (INT,INT)
default:
  // C code fragment for operands of (STR,STR)
}
```

Decision tree for ADD instruction

```
// op1 + op2
\inst add (Register dst, Value op1, Value op2)
\when op1:int && op2:int \{
  // C code fragment for operands of (INT,INT)
\}
\when op1:string && op2:string \{
  // C code fragment for operands of (STR,STR)
\}
\when op1:int && op2:string \{
  // C code fragment for operands of (INT,STR)
\}
...
```

Evaluation

- What we evaluated
  - Effects of limiting operand datatypes
  - Effects of changing datatype representations
- Baseline: full-set VM with default datatype representation, manually tuned by us

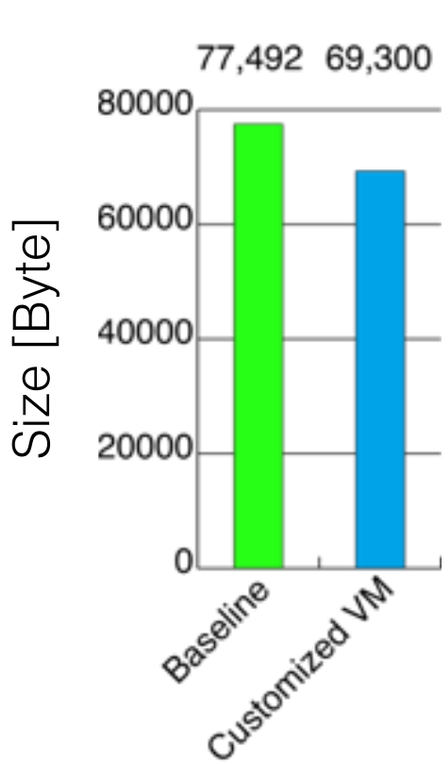
Environment

- Hardware：Raspberry Pi 3
  - CPU：BCM2837 64bit (1.2GHz)
    - OS: Raspbian GNU/Linux 8
- Compiler：GCC 4.9.2
  - option：-Os （optimize for code size）
- Benchmark programs：selected from SunSpider

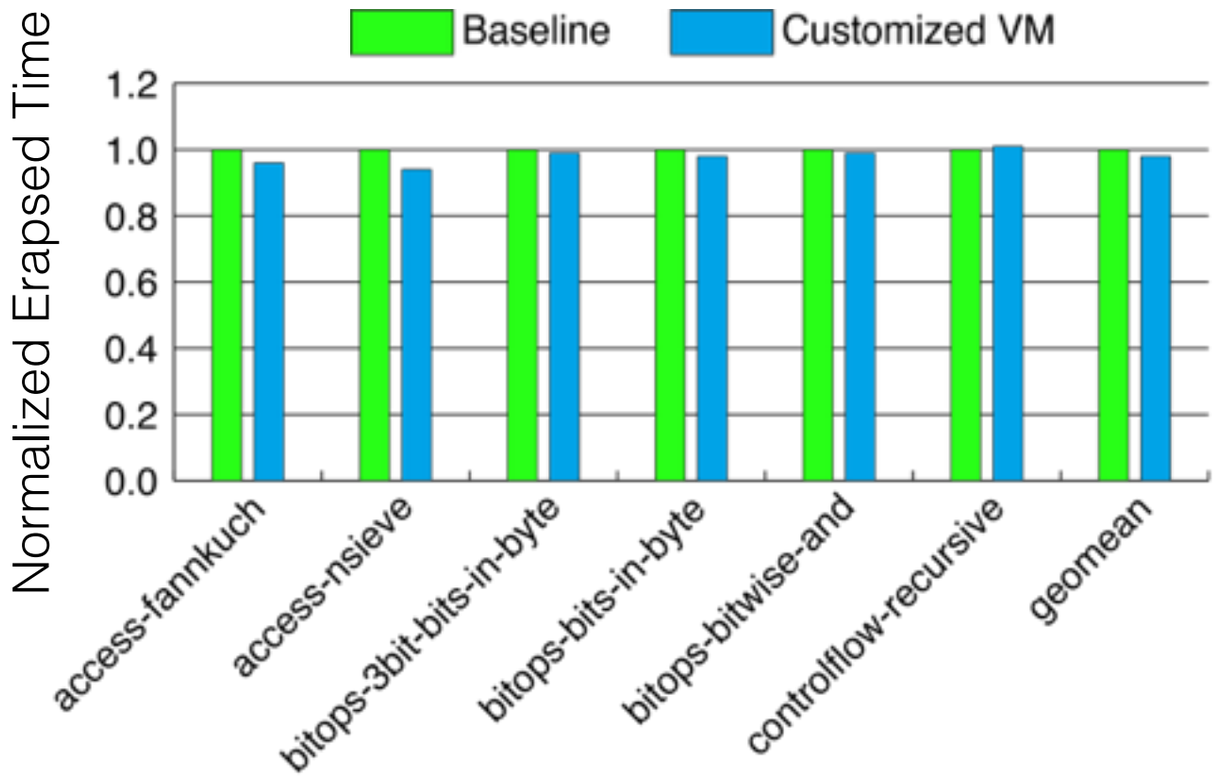
Effects of Limiting Operand Datatypes

Customized VM: Operators accept limited datatypes

- ADD: (INT, INT) and (STR, STR)
- others: (INT, INT)



Size of the VMs



Elapsed time of benchmarks for the VMs

- 10% smaller in VM size
- Execution times did not change significantly

Effects of Changing Datatype Representation

Customized VM: Represent “array” type with pointer tag

Datatype Specification

Default (baseline)

	Pointer tag	Header tag
int	P_INT	-
string	P_STR	-
array	P_GEN	H_ARY
simple_object	P_GEN	H_OBJ
...	...	...

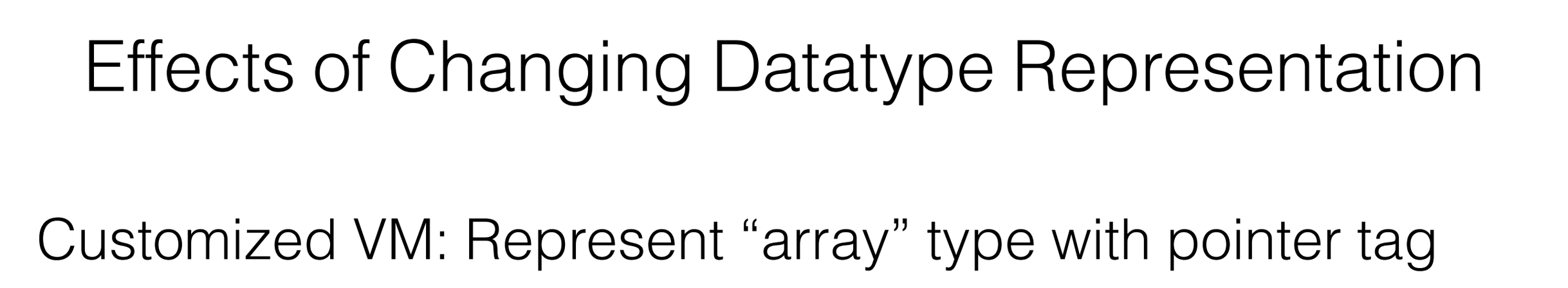
Customized

	Pointer tag	Header tag
int	P_INT	-
string	P_STR	-
array	P_ARY	-
simple_object	P_GEN	H_OBJ
...	...	...

Effects of Changing Datatype Representation

Customized VM: Represent “array” type with pointer tag

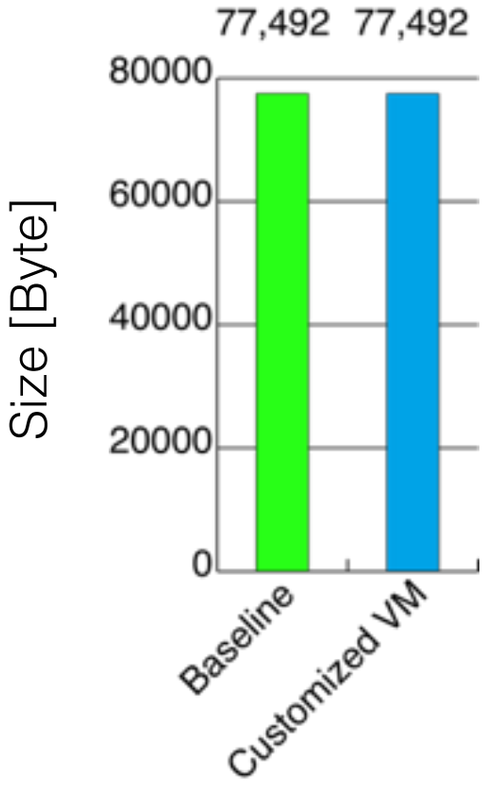
- The number of header-tag accesses was dramatically reduced for property access intensive benchmarks



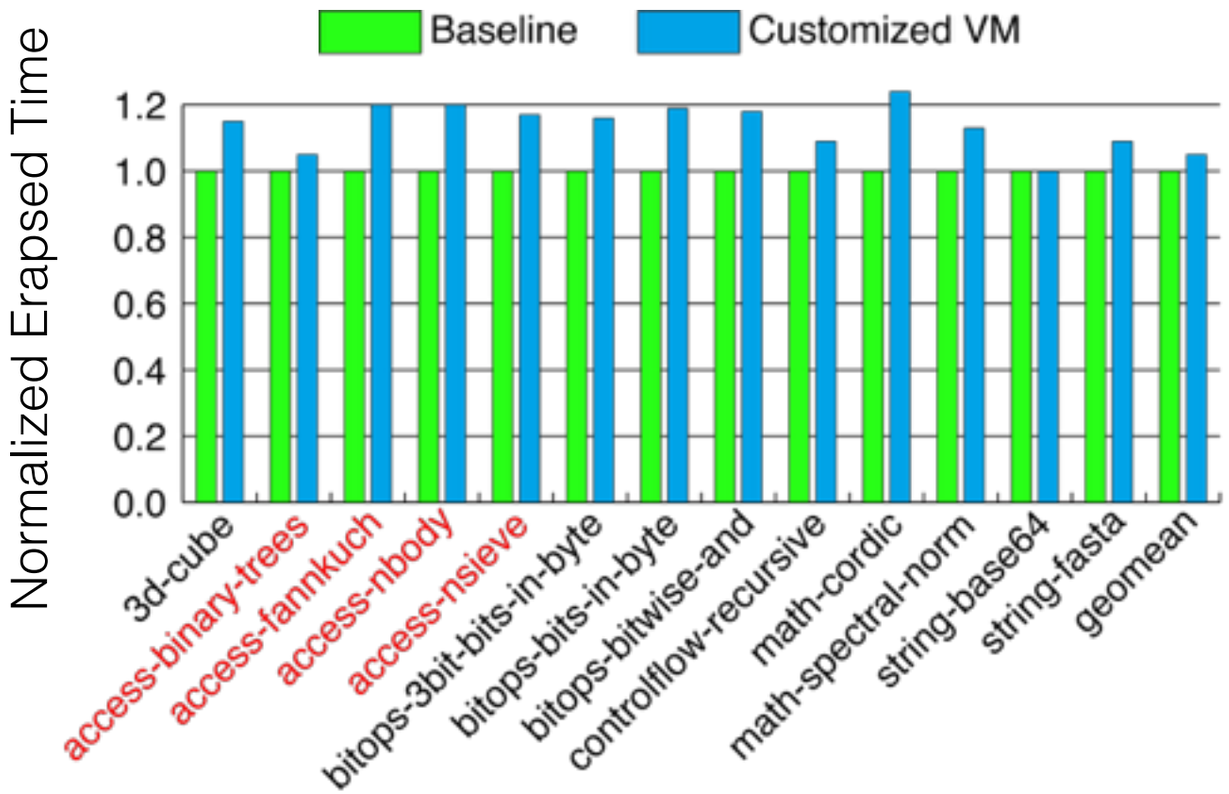
×10^5 number of header-tag accesses in access-fannkuch

Effects on Performance

- Customized VM was slower
  - Header-tag access was not expensive
  - Effects of manual tuning was more significant
- VM size did not increase



Size of the VMs



Elapsed time of benchmarks for the VMs

Related Work

- JavaScript for embedded systems (JerryScript, mJS)
  - They are optimized according to general characteristics of general applications
  - VM generated by eJSTK is optimized according to specific characteristics of a target application
- Vmgen [M.A.Ertl et al.,2002]
  - It is aimed at instruction-centric customization and generates stack-based VM
  - eJSTK is aimed at datatype-centric customization

Conclusion

- We generated eJSTK, a framework for generating JavaScript VM customized for each application
  - Generates small VM
  - Execution time did not improve by changing datatype representation although accesses to header-tag wad reduced

eJSTK in github: https://github.com/plasklab/ejstk