

Compiling for Multi-language Task Migration

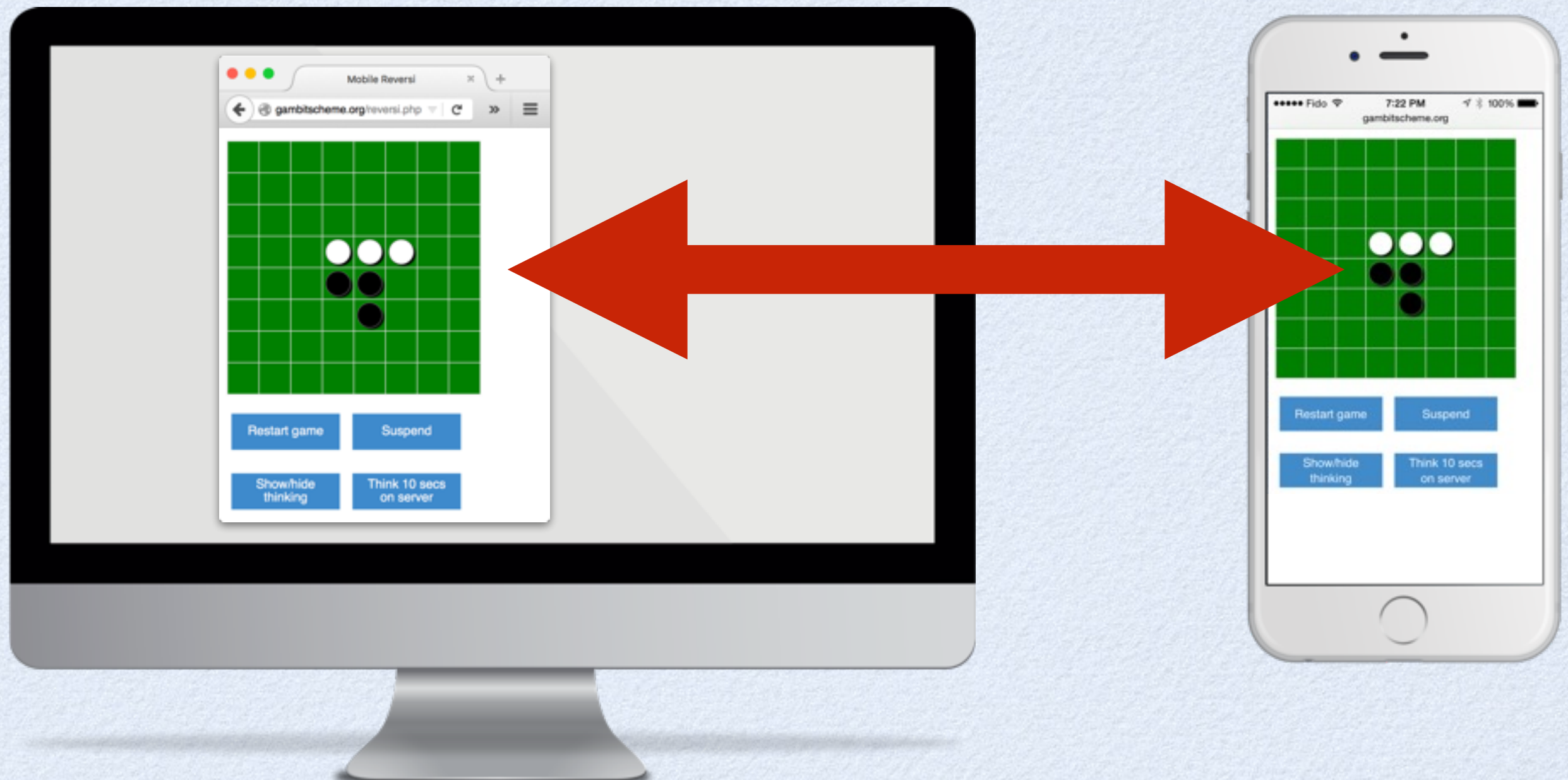
Marc Feeley

DLS 2015

Université 
de Montréal

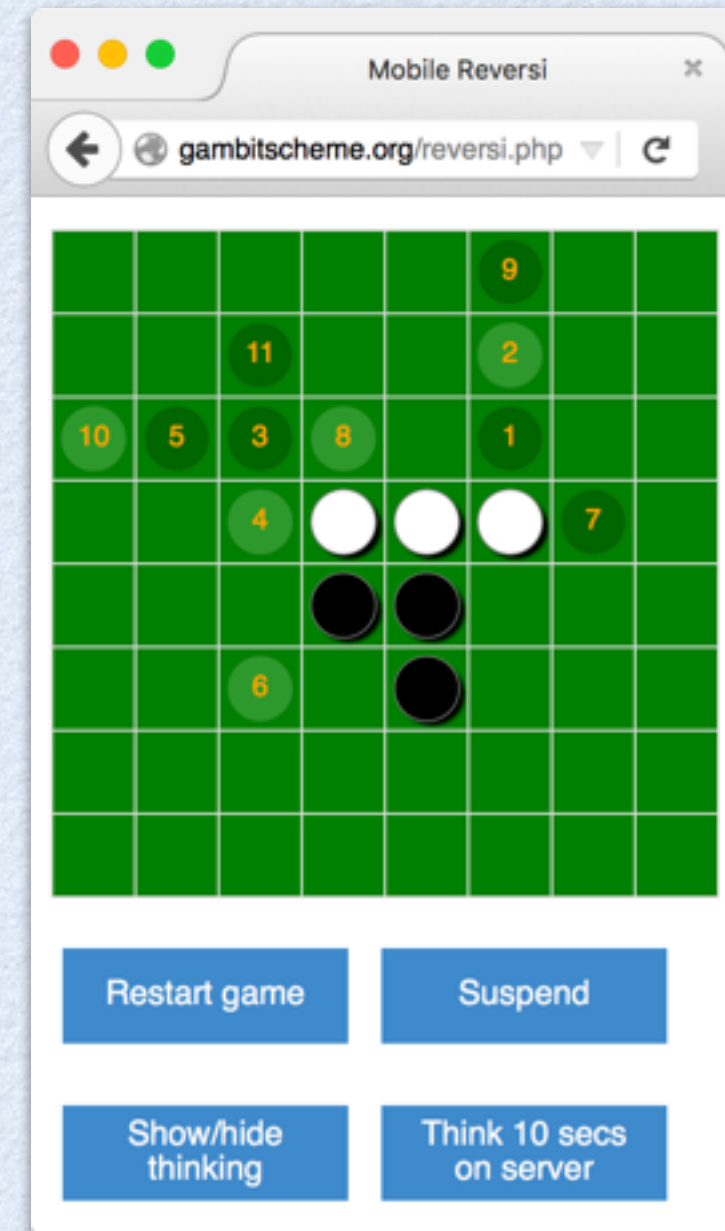
Task Migration

- Process of transferring a running computation to another computational node




Mobile Reversi Web App

- Reversi game against computer in browser
- **Continuous search for best replies** to human's legal moves while human is thinking (this allows instant response when human plays a move)
- **"Suspend"** button sends an email with a URL to **resume game on any other device**
- **"Think on server"** button migrates to web server for 10 secs and then back to browser (useful when using a powerful web server)



Implementation

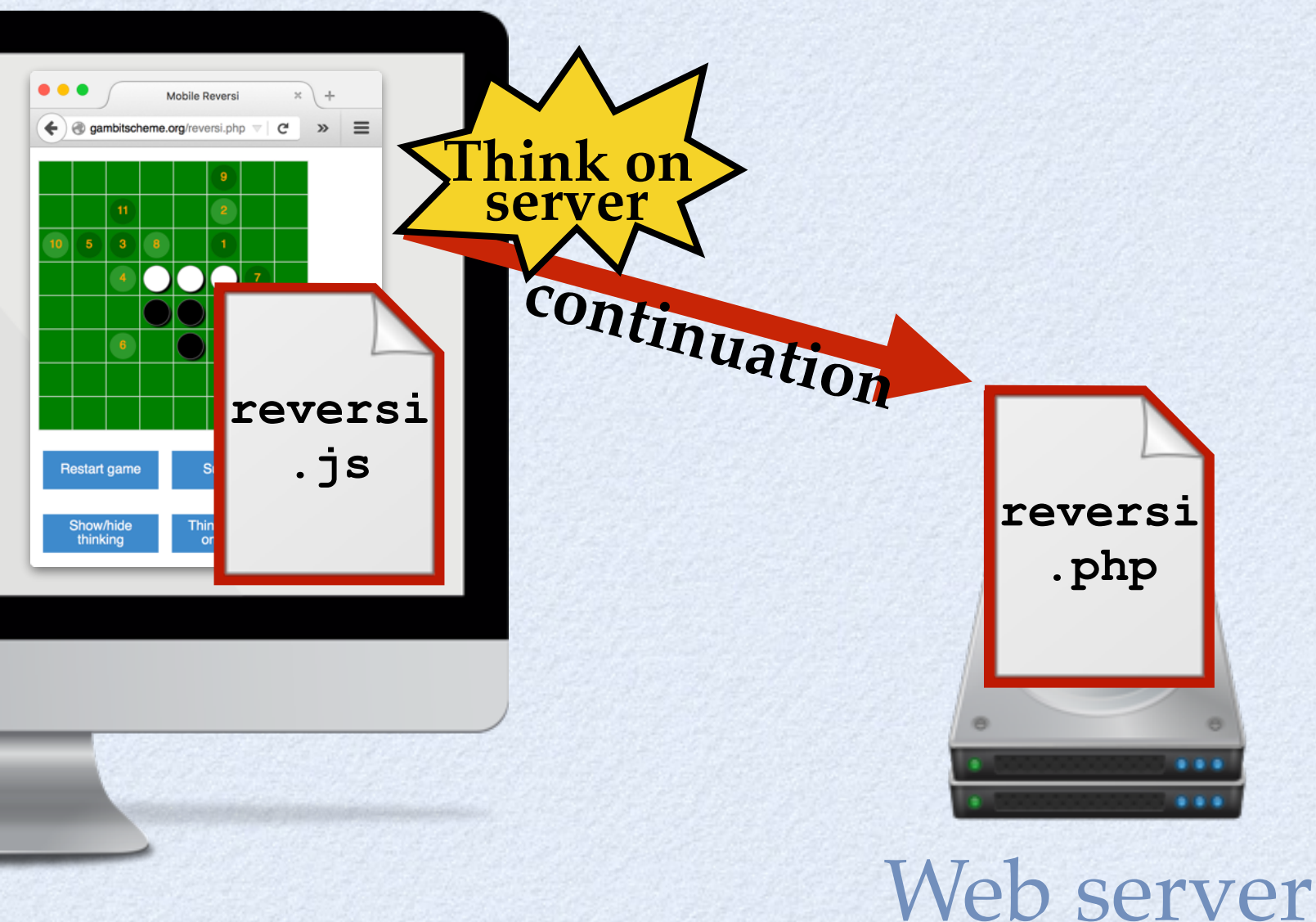
- Program logic is implemented by a single **Scheme** program compiled to **JS** and **PHP**

reversi.scm  **reversi.php**
reversi.js

- To facilitate migration, browser and server run **same program** but at different entry points
- Task migration consists in sending the program's **current continuation** to the destination node and resuming it

Trace for “Think on server” (1)

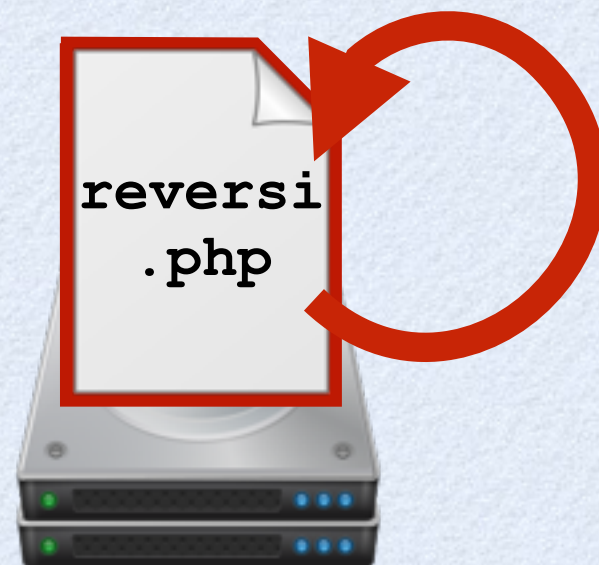
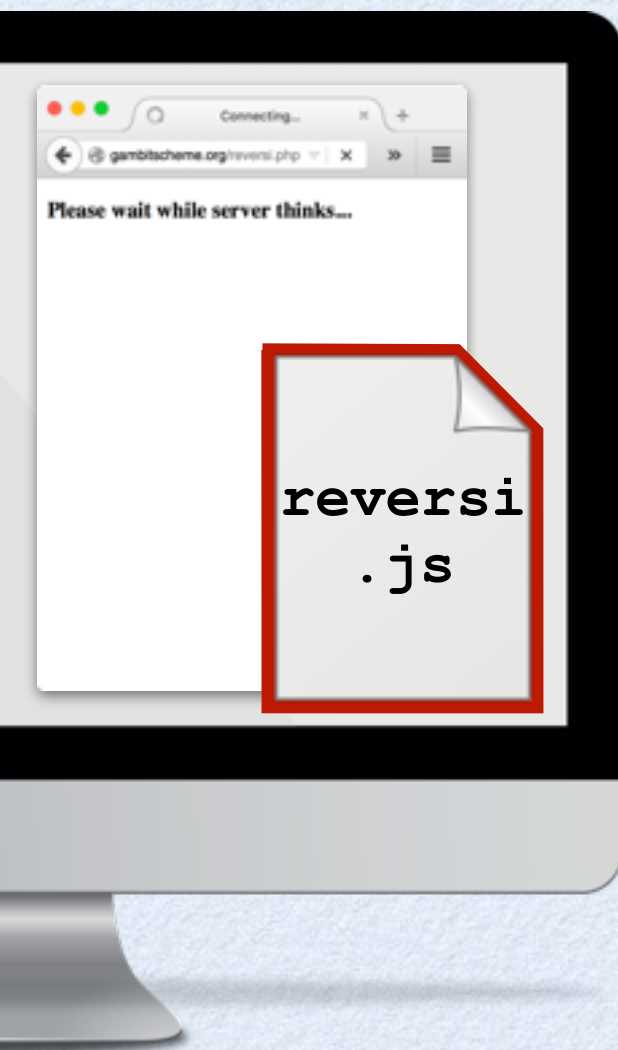
- Clicking “Think on server” sends the current continuation to the server



Trace for "Think on server" (2)

6

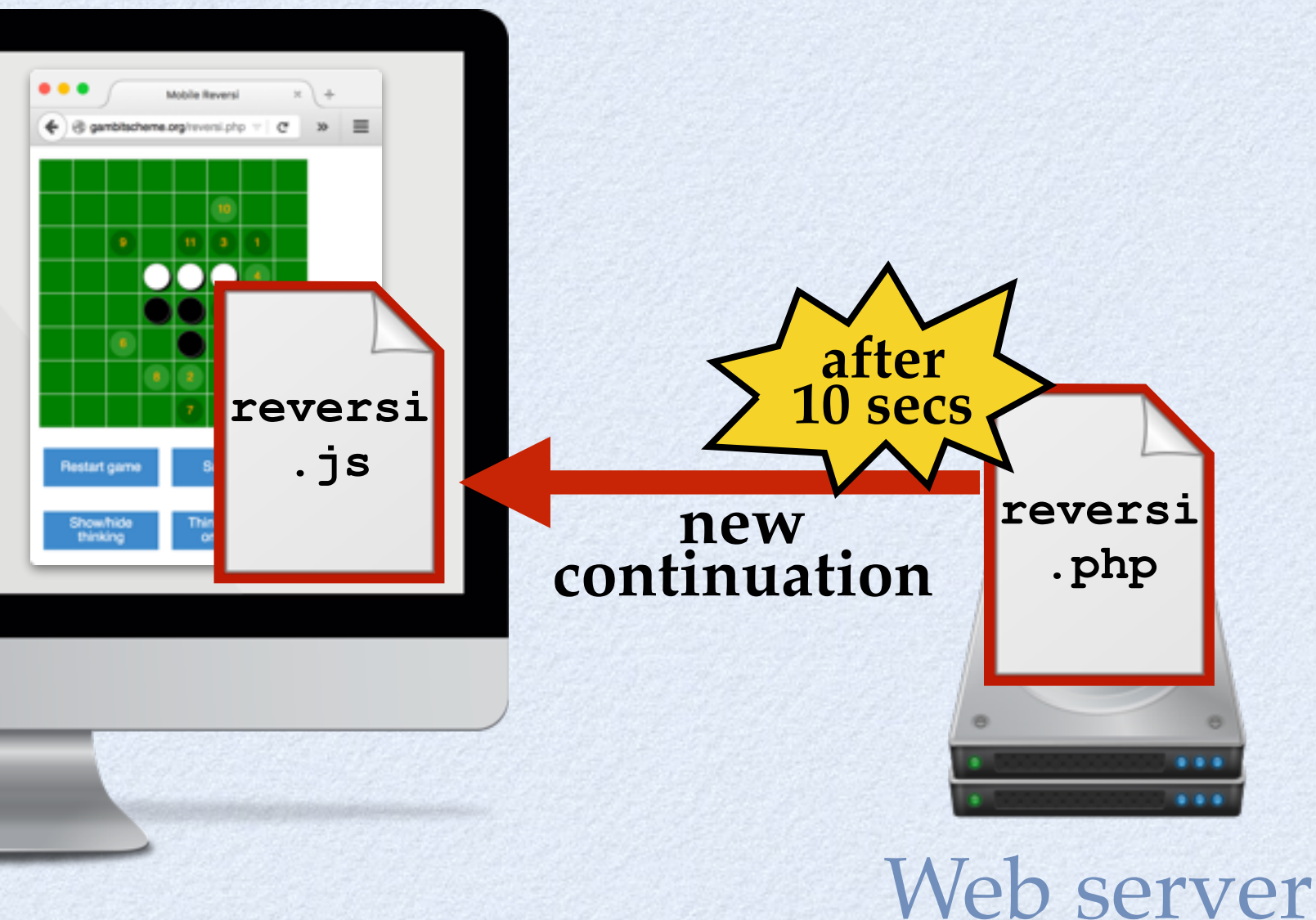
- The **server** resumes the continuation to continue searching for best replies



Web server

Trace for "Think on server" (3)

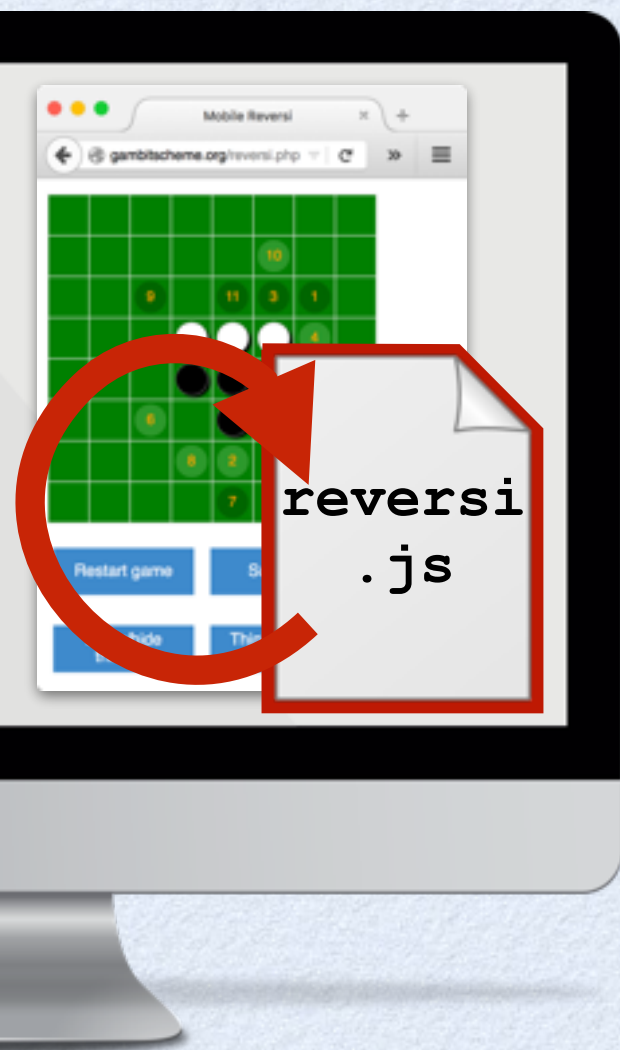
- After 10 seconds, the server sends its current continuation to the browser



Trace for "Think on server" (4)

8

- The **browser** resumes the continuation to continue searching for best replies



Web server

Migration

- Gist of “Think on server” implementation

runs on browser

runs on server

runs on browser

```
(define (search ...)  
  .  
  .  
  .  
  (migrate-task web-server) ;; click  
  .  
  .  
  .  
  (migrate-task web-browser) ;; 10 secs  
  .  
  .  
  .  
)
```


migrate-task

- Migration is performed with **migrate-task**:

```
(define (migrate-task dest)
  (call/cc
    (lambda (k)
      (send dest (make-request-resume-task k))
      (halt)))))
```

- **Object serialization/deserialization** is hidden in the **(send dest msg)** operation that sends the message **msg** to the destination **dest**
- **(halt)** terminates task

Requirements

- To support this task migration approach, **continuations** (as well as the other data types) must be:
 - **serializable**
 - **compatible across the target languages**
- Our contribution is to show how to do this efficiently for a Scheme compiler (Gambit) targeting JS, PHP, Python, Ruby (and Java)

Our Approach

Closures, Continuations and Tail Calls

- In general, target languages **don't have a natural mapping** for closures, continuations and tail calls
- continuations and tail calls are **unsupported**
- closures of the target language are not **serializable** because they are opaque (no way to access their free variables)

```
function smallerThan(n) {  
  return function (x) {  
    return x < n;  
  };  
}  
  
var neg = smallerThan(0);
```

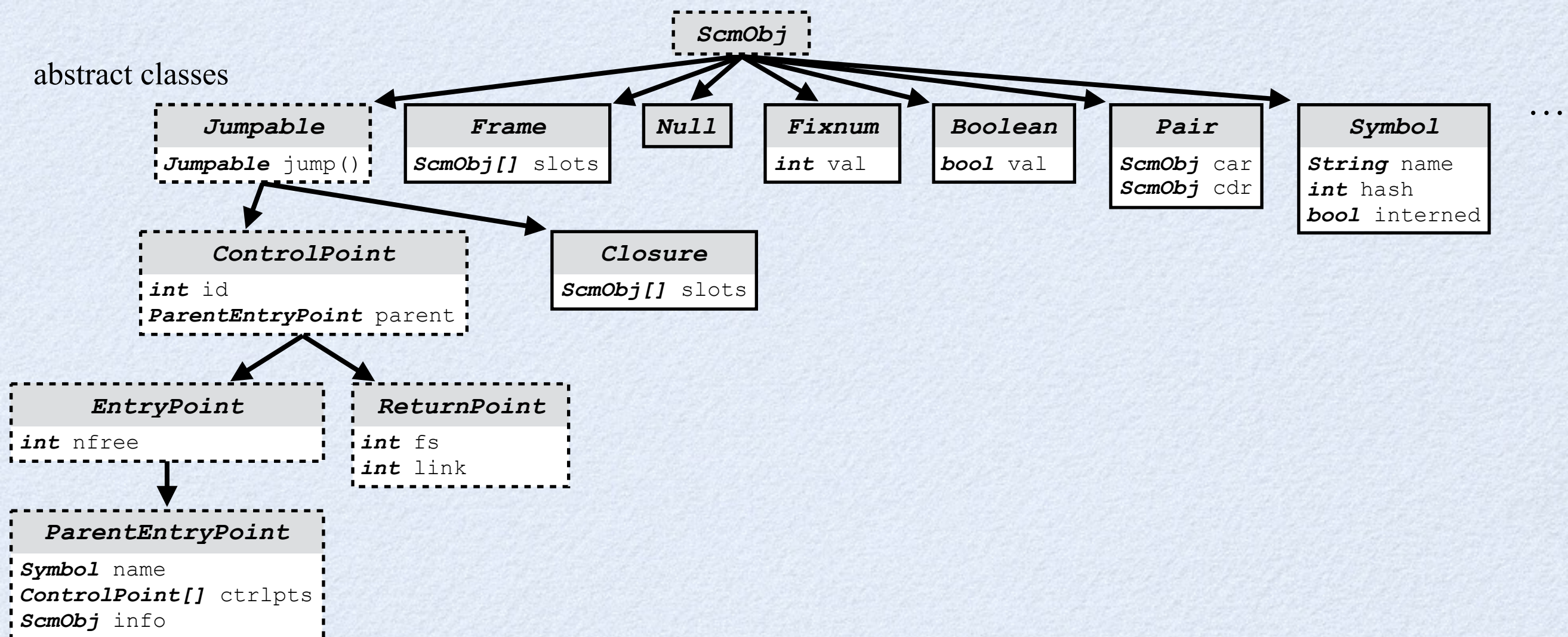
No way to get **n**
from **neg**

Virtual Machine Approach

- Our approach uses an **explicit representation** of Scheme closures and continuations
 - **Trampoline for tail calls**
 - **Explicit call stack management**
 - **Efficient incremental stack/heap implementation of continuations**
- This amounts to using the target language as a **high-level assembly language** to implement a virtual machine for Scheme

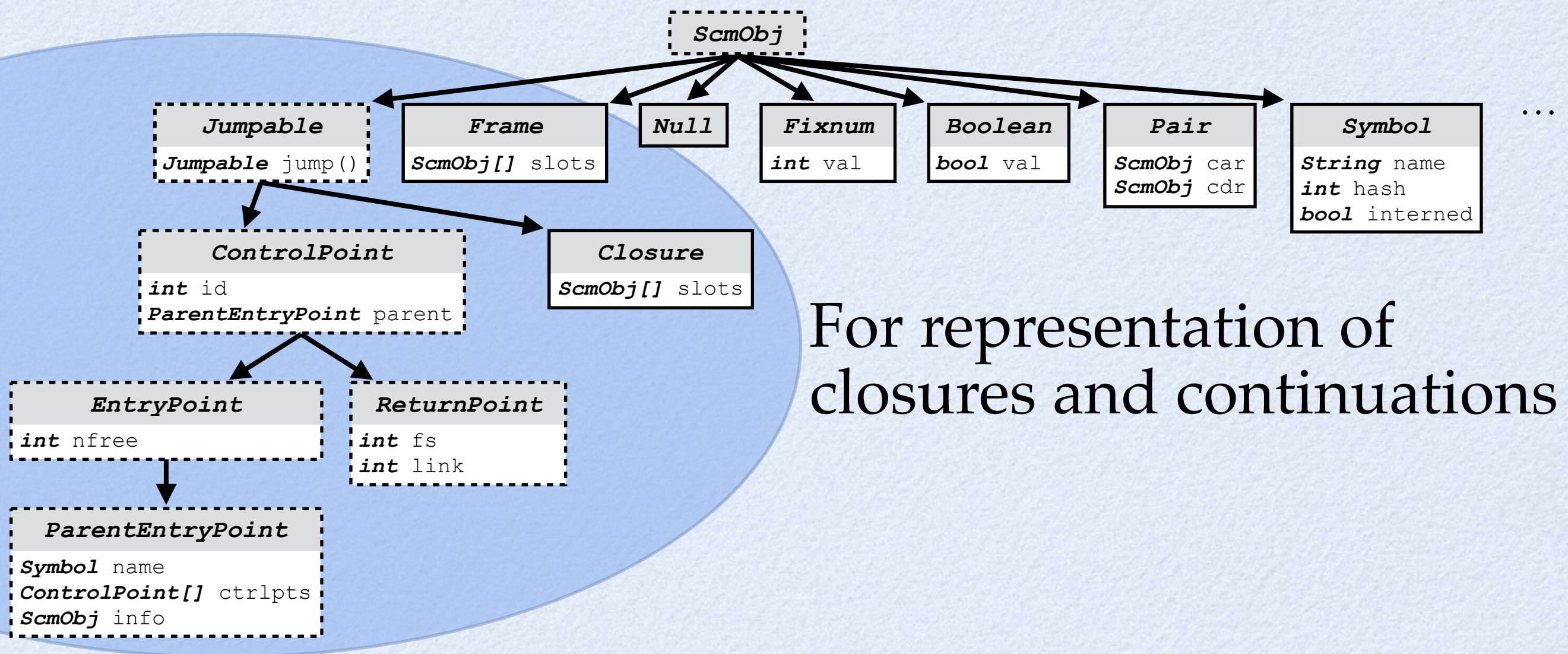
Abstract Object Representation

- Target language compatibility of data is achieved by a **common abstract representation** of Scheme objects that is serializable



Abstract Object Representation

- Target language compatibility of data is achieved by a **common abstract representation** of Scheme objects that is serializable



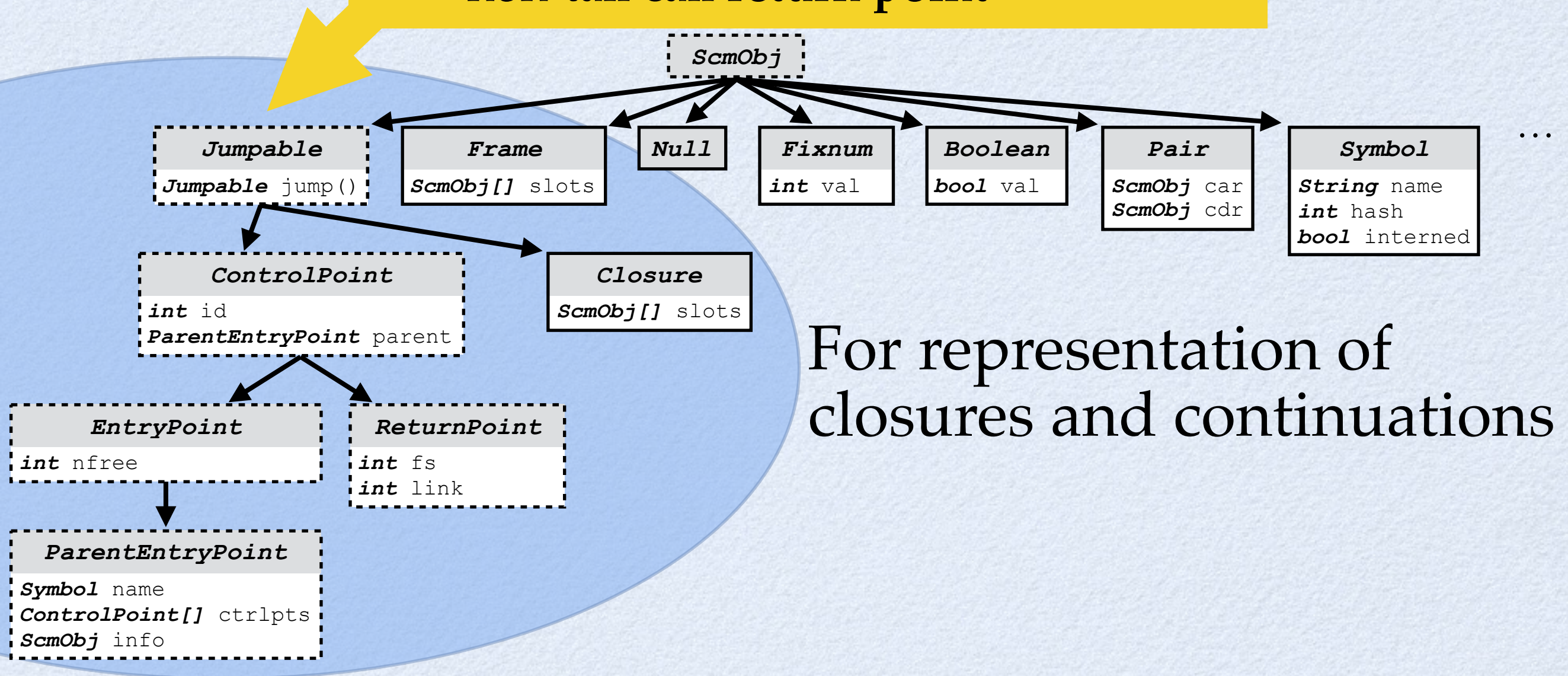
Abstract

Jumpable is the type of objects that can be jumped to by the trampoline:

- closures and continuations
- control points in the code:
 - procedure entry point
 - non-tail call return point

Representation

is
representation



Mapping Object Representation

- The abstract object representation is **mapped to the target languages independently**
- Java uses **actual classes** for all object types
- For efficiency, dynamic languages use a **natural mapping** when possible, typically:
 - **Boolean** -> target language **Booleans**
 - **Fixnum** -> target language **numbers**
 - **Vector** -> target language **arrays**
 - **Jumpable** -> target language **functions**

Gambit Virtual Machine

- GVM is the intermediate code representation of the compiler (Scheme \longrightarrow GVM \longrightarrow target)
- A procedure's code is a **set of basic-blocks** with **explicit jump instructions** for local and global control flow
 - Each BB is a **control point** of the program
- The GVM has a set of general purpose **registers** and a **stack**, both used by the procedure call protocol

GVM Example

```
(define (foreach f lst)
  (let loop ((lst lst))
    (if (pair? lst)
        (begin
          (f (car lst))
          (loop (cdr lst)))
        #f)))
```

GVM basic blocks of **foreach**

```
#1 fs=0 entry-point nparams=2 ()
  jump fs=0 #3
```

```
#2 fs=3 return-point
  r2 = (cdr frame[3])
  r1 = frame[2]
  r0 = frame[1]
  jump/poll fs=0 #3
```

```
#3 fs=0
  if (pair? r2) jump fs=0 #4 else #5
```

```
#4 fs=0
  frame[1] = r0
  frame[2] = r1
  frame[3] = r2
  r1 = (car r2)
  r0 = #2
  jump fs=3 frame[2] nargs=1
```

```
#5 fs=0
  r1 = '#f
  jump fs=0 r0
```

tail call
loop

non-tail
call
f

create continuation
frame

pass return addr.

Call protocol of **foreach** and **loop**

```
r0 = return address
r1 = f
r2 = lst
```

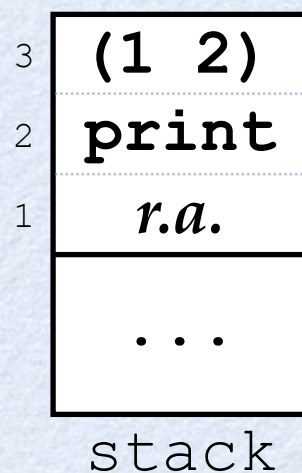
Call protocol of **f**

```
r0 = return address
r1 = arg1
```


Trace of (foreach print '(1 2))

```
r0 = r.a.  
r1 = print  
r2 = (1 2)
```

```
r0 = #2  
r1 = 1
```



GVM basic blocks of **foreach**

```
#1 fs=0 entry-point nparams=2 ()  
jump fs=0 #3
```

```
#2 fs=3 return-point  
r2 = (cdr frame[3])  
r1 = frame[2]  
r0 = frame[1]  
jump/poll fs=0 #3
```

```
#3 fs=0  
if (pair? r2) jump fs=0 #4 else #5
```

```
#4 fs=0  
frame[1] = r0  
frame[2] = r1  
frame[3] = r2  
r1 = (car r2)  
r0 = #2  
jump fs=3 frame[2] nargs=1
```

```
#5 fs=0  
r1 = '#f  
jump fs=0 r0
```

tail call
loop

non-tail
call
f

create continuation
frame

pass return addr.

print

Trace of (foreach print ' (1 2))

```
r0 = r.a.  
r1 = print  
r2 = (1 2)
```

GVM basic blocks of **foreach**

tail call
loop

jump/poll instruction performs a check for interrupts (UI events, etc) in addition to the actual jump

The compiler guarantees that a bounded number of instructions are executed between polling operations

```
#1 fs=0 entry-point nparams=2 ()  
jump fs=0 #3
```

```
#2 fs=3 return-point  
r2 = (cdr frame[3])  
r1 = frame[2]  
r0 = frame[1]  
jump/poll fs=0 #3
```

```
fs=0  
if (pair? r2) jump fs=0 #4 else #5
```

```
#4 fs=0  
frame[1] = r0  
frame[2] = r1  
frame[3] = r2  
r1 = (car r2)  
r0 = #2  
jump fs=3 frame[2] nargs=1
```

create continuation
frame

pass return addr.

```
#5 fs=0  
r1 = '#f  
jump fs=0 r0
```

print

Translating Jumps

- Except for jumps, GVM instructions are easily translated to the target language
- To implement jumps, each basic block is translated to a **parameter-less function** (in the case of Java a class derived from **Jumpable**)
- Their execution is chained by a **trampoline**

Translation to JS

Runtime system for JS

```
var r0, r1, r2, r3; // GVM registers
var nargs;          // argument count
var stack = [null]; // runtime stack
var sp = 0;          // stack pointer
```

```
function trampoline(pc) {
  while (pc !== null)
    pc = pc();
}
```

```
function poll(dest) {
  // ...check for interrupts here...
  return dest;
}
```

Generated JS code

```
function bb1_foreach() {
  if (nargs !== 2)
    return wrong_nargs(bb1_foreach);
  return bb3_foreach;
}
```

```
function bb2_foreach() {
  r2 = stack[sp].cdr;
  r1 = stack[sp-1];
  r0 = stack[sp-2];
  sp -= 3;
  return poll(bb3_foreach);
}
```

```
function bb3_foreach() {
  if (r2 instanceof Pair) {
    stack[sp+1] = r0;
    stack[sp+2] = r1;
    stack[sp+3] = r2;
    r1 = r2.car;
    r0 = bb2_foreach;
    sp += 3;
    nargs = 1;
    return stack[sp-1];
  } else {
    r1 = false;
    return r0;
  }
}
```


Translation to JS

Runtime system for JS

```
var r0, r1, r2, r3; // GVM registers
var nargs;          // argument count
var stack = [null]; // runtime stack
var sp = 0;          // stack pointer
```

```
function trampoline(pc) {
  while (pc !== null)
    pc = pc();
}
```

```
function poll(dest) {
  // ...check for interrupts here...
  return dest;
}
```

poll function returns its argument after checking for interrupts

Generated JS code

```
function bb1_foreach() {
  if (nargs !== 2)
    return wrong_nargs(bb1_foreach);
  return bb3_foreach;
}
```

```
function bb2_foreach() {
  r2 = stack[sp].cdr;
  r1 = stack[sp-1];
  r0 = stack[sp-2];
  sp -= 3;
  return poll(bb3_foreach);
}
```

```
function bb3_foreach() {
  if (r2 instanceof Pair) {
    stack[sp+1] = r0;
    stack[sp+2] = r1;
    stack[sp+3] = r2;
    r1 = r2.car;
    r0 = bb2_foreach;
    sp += 3;
    nargs = 1;
    return stack[sp-1];
  } else {
    r1 = false;
    return r0;
  }
}
```


Translation to JS

Runtime system for JS

```
var r0, r1, r2, r3; // GVM registers
var nargs;          // argument count
var stack = [null]; // runtime stack
var sp = 0;          // stack pointer
```

```
function trampoline(pc) {
  while (pc !== null)
    pc = pc();
}
```

```
function poll(dest) {
  // ...check for interrupts here...
  return dest;
}
```

Statically known jumps
to singly used or short
basic blocks are inlined

Generated JS code

```
function bb1_foreach() {
  if (nargs !== 2)
    return wrong_nargs(bb1_foreach);
  return bb3_foreach;
}
```

```
function bb2_foreach() {
  r2 = stack[sp].cdr;
  r1 = stack[sp-1];
  r0 = stack[sp-2];
  sp -= 3;
  return poll(bb3_foreach);
}
```

```
function bb3_foreach() {
  if (r2 instanceof Pair) {
    stack[sp+1] = r0;
    stack[sp+2] = r1;
    stack[sp+3] = r2;
    r1 = r2.car;
    r0 = bb2_foreach;
    sp += 3;
    nargs = 1;
    return stack[sp-1];
  } else {
    r1 = false;
    return r0;
  }
}
```

inlined basic
block #4

inlined basic
block #5

Translation to JS

Runtime system for JS

```
var r0, r1, r2, r3; // GVM registers
var nargs;          // argument count
var stack = [null]; // runtime stack
var sp = 0;          // stack pointer
```

```
function trampoline(pc) {
  while (pc !== null)
    pc = pc();
}
```

```
function poll(dest) {
  // ...check for interrupts here...
  return dest;
}
```

Note: only basic control flow statements are used, namely **return** and **if** (no loops or exception handling or closures)

This makes optimization by the target VM easier

Generated JS code

```
function bb1_foreach() {
  if (nargs !== 2)
    return wrong_nargs(bb1_foreach);
  return bb3_foreach;
}
```

```
function bb2_foreach() {
  r2 = stack[sp].cdr;
  r1 = stack[sp-1];
  r0 = stack[sp-2];
  sp -= 3;
  return poll(bb3_foreach);
}
```

```
function bb3_foreach() {
  if (r2 instanceof Pair) {
    stack[sp+1] = r0;
    stack[sp+2] = r1;
    stack[sp+3] = r2;
    r1 = r2.car;
    r0 = bb2_foreach;
    sp += 3;
    nargs = 1;
    return stack[sp-1];
  } else {
    r1 = false;
    return r0;
  }
}
```


Trace of call trampoline (bb1_foreach)

28

Runtime system for JS

```
var r0, r1, r2, r3; // GVM registers
var nargs;          // argument count
var stack = [null]; // runtime stack
var sp = 0;         // stack pointer
```

```
function trampoline(pc) {
  while (pc !== null)
    pc = pc();
}
```

```
function poll(dest) {
  // ...check for interrupts here...
  return dest;
}
```

Generated JS code

```
function bb1_foreach() {
  if (nargs !== 2)
    return wrong_nargs(bb1_foreach);
  return bb3_foreach;
}
```

```
function bb2_foreach() {
  r2 = stack[sp].cdr;
  r1 = stack[sp-1];
  r0 = stack[sp-2];
  sp -= 3;
  return poll(bb3_foreach);
}
```

```
function bb3_foreach() {
  if (r2 instanceof Pair) {
    stack[sp+1] = r0;
    stack[sp+2] = r1;
    stack[sp+3] = r2;
    r1 = r2.car;
    r0 = bb2_foreach;
    sp += 3;
    nargs = 1;
    return stack[sp-1];
  } else {
    r1 = false;
    return r0;
  }
}
```

bb2_foreach

bb3_foreach

procedure
f

f

1

4

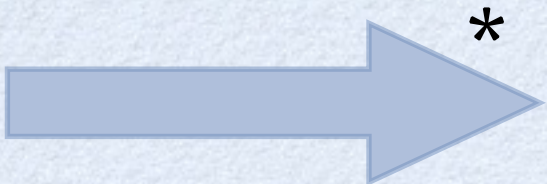
2

3

Improvements

- This basic trampoline implementation has a relatively **high overhead**
- Each jump to a control point = 1 call + 1 return
- The single centralized function call dispatch in the trampoline **hinders inline caching by the target VM**
- This can be improved...

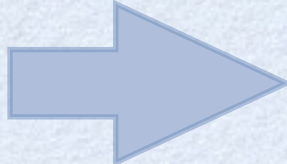
“Jump Destination Call” Optimization

`return X;`  `return X();`

* except when `X` is `poll(...)`

- Allows the target VM to optimize the call to `X`:
 - **inlining and inline caching of functions**
 - **TCO**
- Target VM stack overflows are avoided because the calls to `poll` unwind the stack back to the trampoline at regular intervals

“Intermittent Polling” Optimization

`return poll(X) ;`  `if (--pollcount == 0)
 return poll(X) ;
else
 return X() ;`

- Lowers overhead of calling **poll**
- Exposes more call optimizations, such as inlining of *X*

Stack Space Leak Issue

- Target languages typically implement the stack using an array that **grows on demand**
- When a procedure returns, the data it saved on the stack is still there above **sp** => **space leak!**
- The **poll** function does the stack shrinking:

```
function poll(dest) {  
    pollcount = 100;  
    stack.length = sp+1;  
    // ...check for interrupts...  
    return dest;  
}
```

garbage on stack
persists at most
until next call to
poll

Serialization

- For serializing closures and continuations, **meta information** is attached to control points:
- **Unique identifier** (parent + local id), used to recover the control point when deserializing it
- **ReturnPoint**: size of frame + location of RA, used by **call/cc** implementation
- **EntryPoint**: number of free variables

JS Code with Meta Information

```
function bb1_foreach() { ... } // ParentEntryPoint
function bb2_foreach() { ... } // ReturnPoint
function bb3_foreach() { ... } // Jumpable
```

```
bb1_foreach.id = 0;
bb1_foreach.parent = bb1_foreach;
bb1_foreach.nfree = -1; // not a closure
bb1_foreach.name = "foreach";
bb1_foreach.ctrlpts = [bb1_foreach, bb2_foreach];
```

```
bb2_foreach.id = 1;
bb2_foreach.parent = bb1_foreach;
bb2_foreach.fs = 3;
bb2_foreach.link = 1;
```

```
peps["foreach"] = bb1_foreach;
```

peps = table of parent entry points
(used for deserialization)

Serialization: `bb2_foreach` \longrightarrow `<"foreach", 1>`

Deserialization: `peps["foreach"].ctrlpts[1]`

Evaluation

Other Systems

- No other Scheme system targets multiple languages and has serializable continuations
- The closest point of comparison is **Scheme2JS** and **Spock** that target JS and whose continuations could be made serializable with a moderate amount of work

Scheme Systems Compared

- **Scheme2JS** (Florian Loitsch):
 - **call/cc** based on Replay-C algorithm that uses exceptions to iterate over stack frames
- **Spock** (Felix Winkelmann):
 - CPS conversion and exceptions to unwind stack
- **Gambit-JS** is our approach with JS backend

JS VMs Evaluated

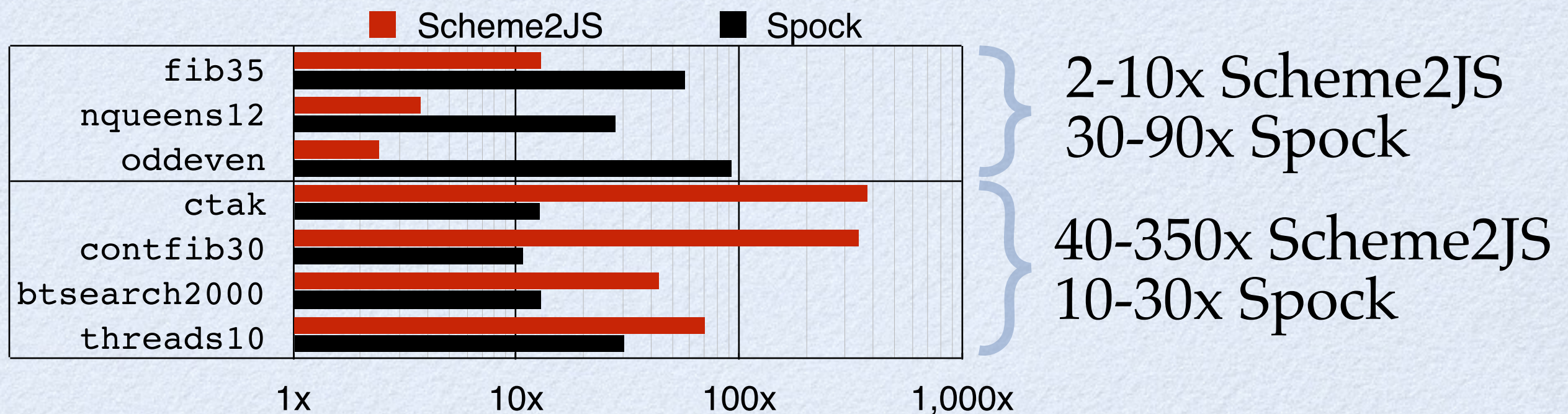
- **Microsoft Chakra**
- **Google V8**
- **Mozilla SpiderMonkey**
- **Apple Nitro**

Benchmarks

- Benchmarks without uses of **call/cc**
 - Recursive: **fib35**, **nqueens12**
 - Tail calls only: **oddeven**
- Benchmarks which use **call/cc**
 - Recursive: **ctak**, **contfib30**
 - Backtracking: **btsearch2000**
 - Threads: **threads10**

Execution Speed

- Gambit-JS consistently faster on all benchmarks and on all JS VMs
- Execution time relative to Gambit-JS averaged over the 4 JS VMs:



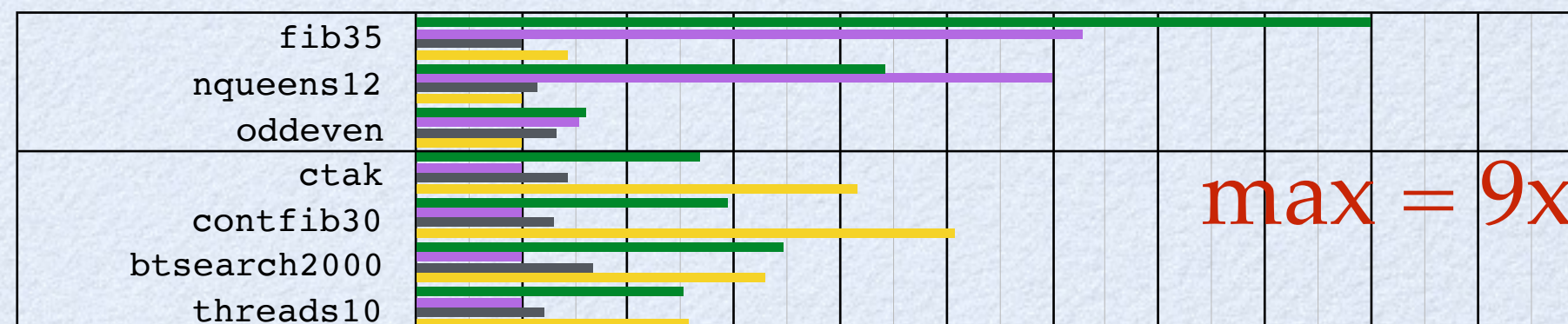
Performance Portability

- Execution speed difference between VMs:

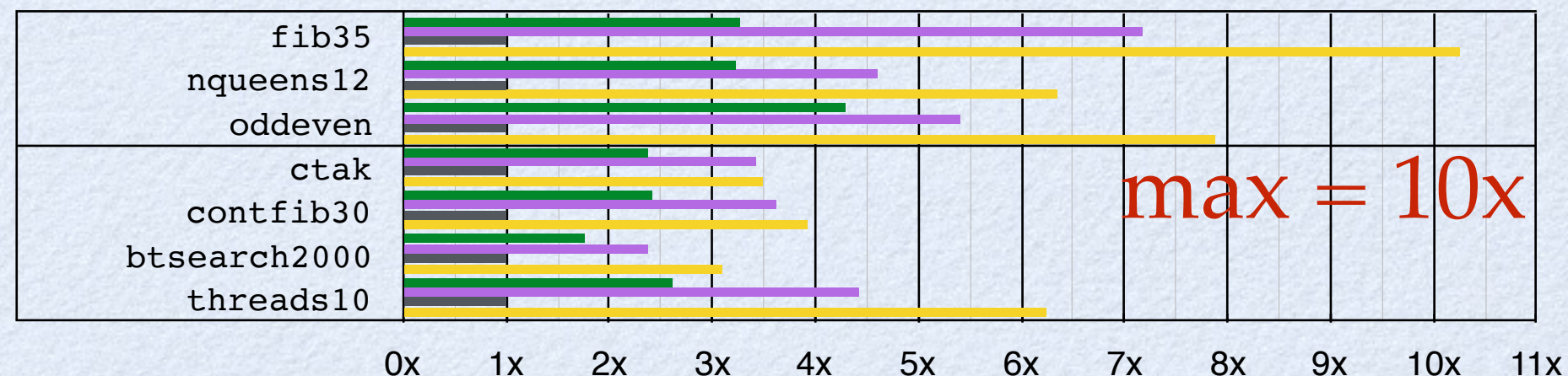
Gambit-JS



Scheme2JS



Spock



Conclusions

Conclusions

- Serializable continuations compatible with multiple targets can be implemented efficiently
- For JS, the VM approach is **1 to 2 orders of magnitude faster** than CPS-conversion and the Replay-C algorithm
- The VM approach also has a more **consistent performance** between target VMs, due to the easier to optimize generated code

Questions?