Tachyon: a Meta-circular Optimizing JavaScript Virtual Machine

Maxime Chevalier-Boisvert
Erick Lavoie
Marc Feeley
Bruno Dufour

{chevalma, lavoeric, feeley, dufour}@iro.umontreal.ca

DIRO - Université de Montréal

About the Tachyon Project

- Began in summer 2010
- Compiler lab at UdeM
- Two students:
  - Erick Lavoie (M.Sc.)
  - Maxime Chevalier-Boisvert (Ph.D.)
- Professor Marc Feeley
  - Gambit Scheme
- Professor Bruno Dufour
  - Dynamic program analysis
- Big project, because we like challenges

What's JavaScript?

- JavaScript ≠ Java in the browser
- Dynamic (scripting) language
- Dynamic typing
  - No type annotations
- Dynamic source loading, eval
- Basic types include:
  - Doubles (no int!), strings, booleans, objects, arrays, first-class functions
- Objects as hash maps
  - Can add/remove properties at any time
  - Prototype-based, no classes

Why JavaScript?

- JavaScript is very popular, it's everywhere
  - JavaScript is the only language for web applications.
  - Volume of JS code increasing fast, becoming more complex
- Many competing implementations
  - Push to move desktop apps to browsers
  - Performance is insufficient
- Compiling dynamic languages efficiently is challenging
  - Dynamic typing, eval, etc
  - Researchers definitely care!

State of the Art

- Firefox / JaegerMonkey
  - Tracing JIT
  - Compiles/specializes loop code traces
- Chrome / V8
  - Hidden classes
  - Inline caches, code patching
  - Very fast JIT compiler
  - Very efficient GC
- Is this the best we can do?
  - We believe there is potential for more optimization

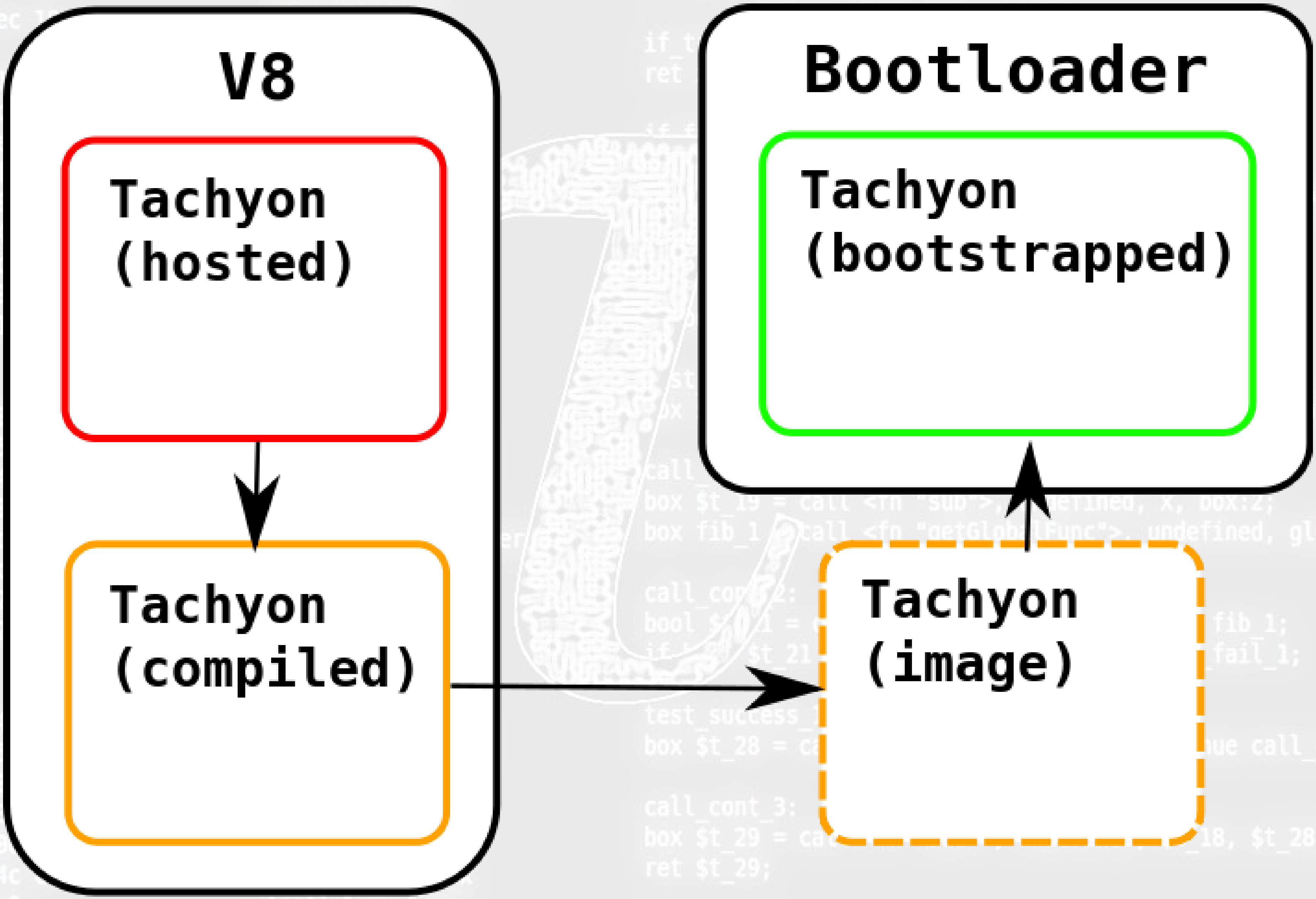
Our Objectives

- Full JavaScript (ECMAScript 5) support
- Retargetable JIT compiler (x86, x86-64)
- Meta-circularity of the VM
- Framework for dynamic language optimizations
  - Better object representations
  - Optimistic optimization w/ recompilation
  - Fast & efficient x86 back-end
- Integration into a web-browser
  - Demonstrate viability on “real” applications
- Free software / OSS

Meta-circularity

- Have your compiler compile itself
  - Less external dependencies
  - Forces you to test/debug
  - Self-optimization
  - Less optimization boundaries
- Fairly straightforward for a traditional static compiler (e.g.: gcc)
- Tricky for our virtual machine
  - Runtime support (on which VM does the VM run?)
  - Performance issues, self-optimization
  - Dynamic loading, dynamic constructs

Bootstrap


V8
Tachyon (hosted)
Tachyon (compiled)
Bootloader
Tachyon (bootstrapped)
Tachyon (image)

JavaScript Extensions

- JavaScript has no access to raw memory
  - Essential to implement a VM/JIT
- Tachyon is written in JS w/ “unsafe” extensions
  - Minimizes the need to write C code (FFI)
  - Maximizes performance
    - FFIs are optimization boundaries
- JS code translated to low-level typed IR
  - JS extensions: insert typed instructions in code as it is translated (Inline IR / IIR)

```
/**
Test if a boxed value is integer
*/
function boxIsInt(boxVal)
{
    "tachyon:inline";
    "tachyon:nothrow";
    "tachyon:ret bool";

    // Test if the value has the int tag
    return (boxVal & TAG_INT_MASK) == TAG_INT;
}
```

```
/**
Implementation of HIR less-than instruction
*/
function lt(v1, v2)
{
    "tachyon:inline";
    "tachyon:nothrow";

    // If both values are immediate integers
    if (boxIsInt(v1) && boxIsInt(v2))
    {
        // Compare immediate integers without unboxing
        var tv = iir.lt(v1, v2);
    }
    else
    {
        // Call a function for the general case
        var tv = ltGeneral(v1, v2);
    }

    return tv? true:false;
}
```

Intermediate Representation

- Inspired from LLVM
- SSA-based
- Type-annotated
  - Integers, floats, booleans, raw pointers
  - Boxed values
- Low-level
  - Mirrors instructions commonly found on most CPUs
    - add/sub/mul/div, and/or/shift, jump/if/call, load/store, etc.
  - Allows expressing more optimizations (specialization)

Optimistic Optimizations

- Traditional optimizations are conservative
  - Can't prove it, can't do it
  - Dynamic languages offer little static type information
  - Dynamic constructs problematic for analysis
    - *eval*, *load*
  - Often can't prove validity conservatively
- Optimistic optimizations
  - Valid now, assume valid until proven otherwise
  - Most dynamic programs not that dynamic
  - Many optimizations do apply

Example: Optimization Issues

```
function sum(list) {
    var sum = 0;
    for (var i = 0; i < list.length; ++i)
        sum += f(list[i]);
    return sum;
}

function f(v) { return v*v; }
print(sum([1,2,3,4,5]));
```

- Don't know type of list and its elements
  - Dynamic type checks needed
- Name f is global, can be redefined
  - Fetch from global object, is-function check needed
  - Can't trivially perform inlining
- What if we add an eval?

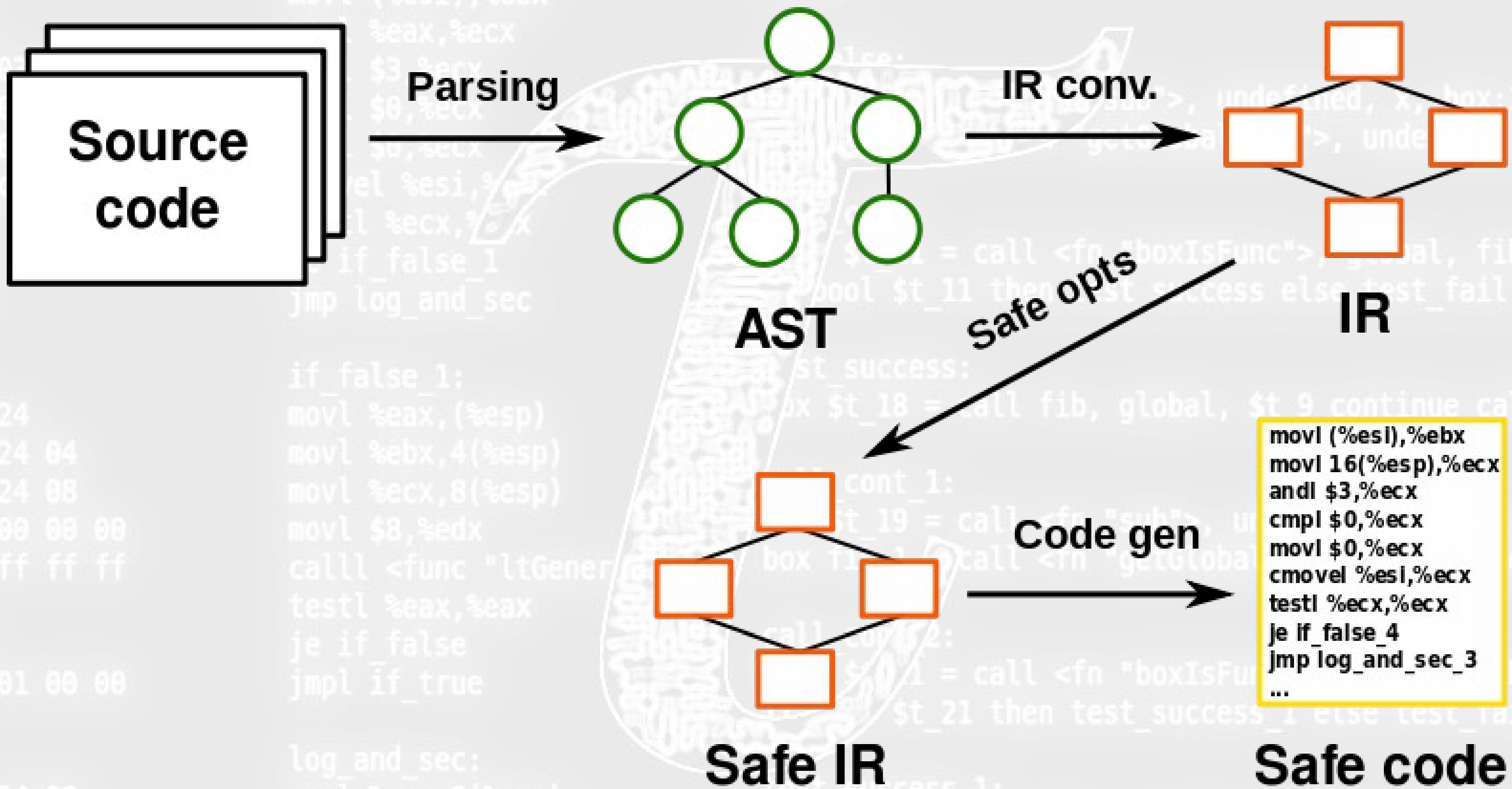
Realistic Assumptions

- As programmers, it's fairly obvious to us that:
  - function f is extremely unlikely to be redefined
  - list will likely always be array of integers
- Not obvious to a compiler, but, in general:
  - How often are global functions redefined?
  - How many call sites are truly polymorphic?
  - How many function arguments can have more than one type?
  - How often do people use eval to change local variable types?

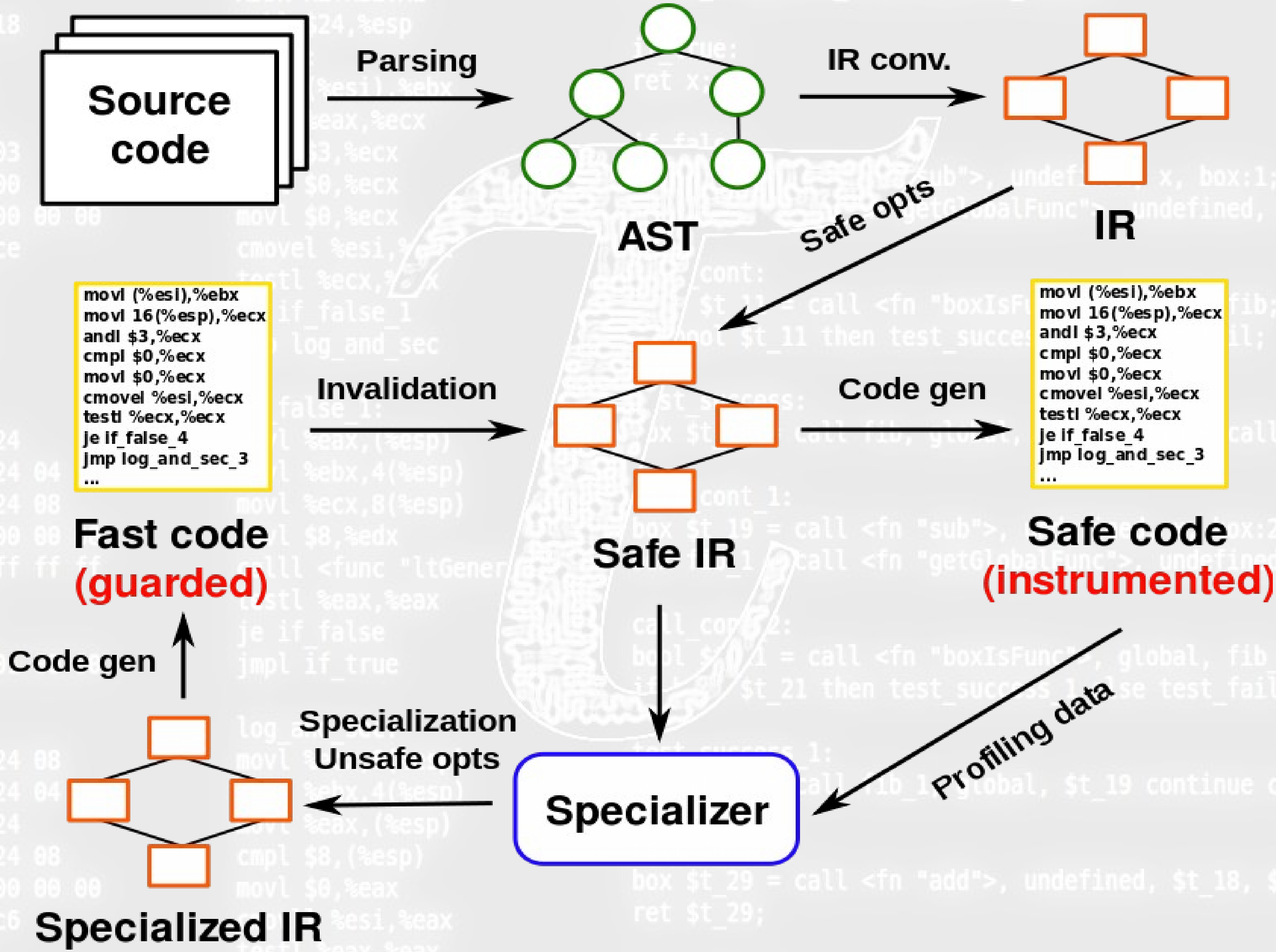
What Would Tachyon Do (WWTD)?

- A VM can observe the types of global variables as a program is executing
  - Can assume that these types will not change
    - e.g.: assume that f() will not be redefined
  - Compile functions with these assumptions
- A VM can observe what types input arguments to a function have
  - Can specialize functions based on these
    - e.g.: sum(list) is always called with arrays of integers
- Types inside of function bodies can be inferred from types of globals and arguments
  - Type propagation, simple dataflow analysis

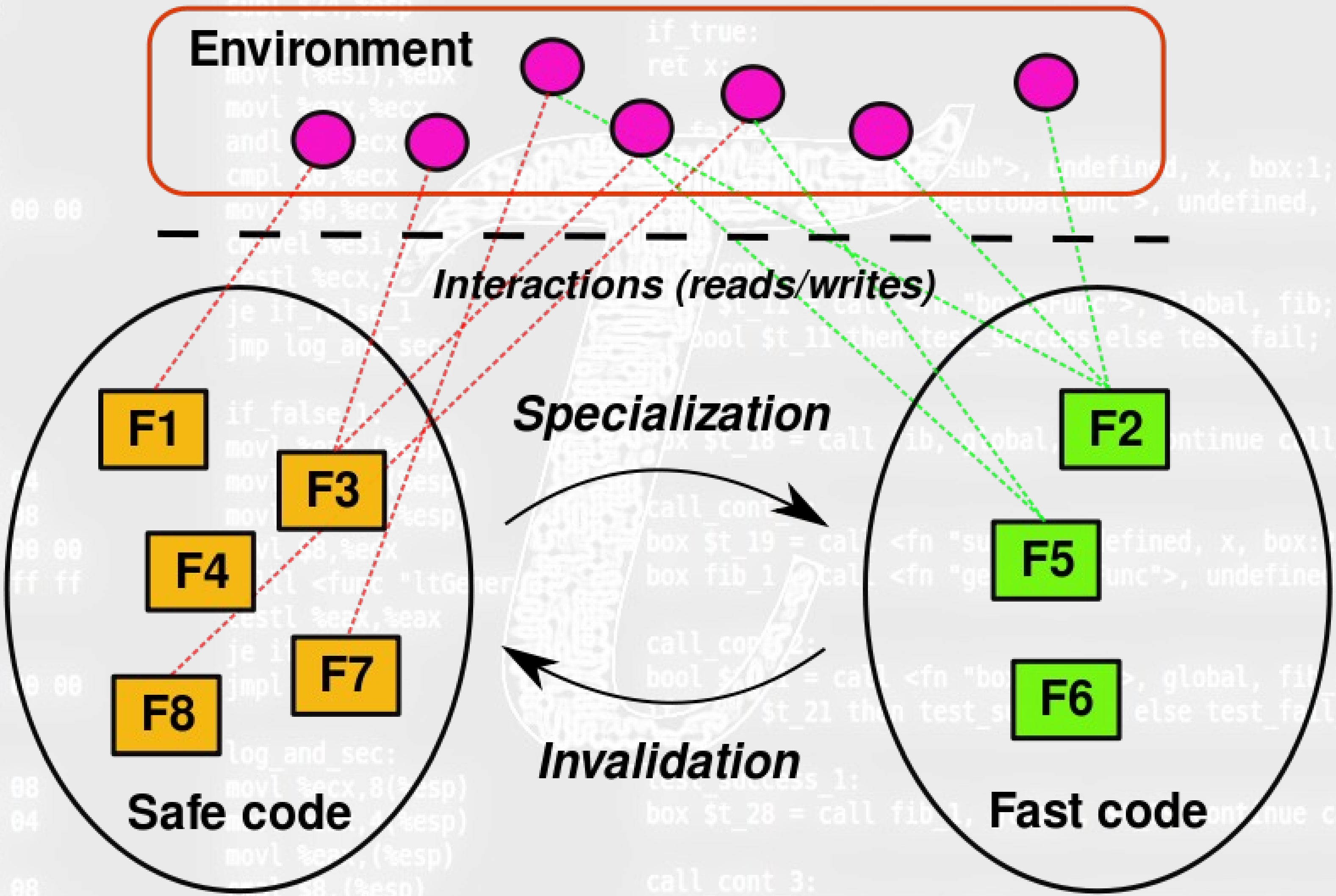
Naïve JavaScript Compilation



What Would Tachyon Do (WWTD)?



What Would Tachyon Do (WWTD)?



Key Ideas

- Crucial to capture info about run time behavior
- Program needs to be correct at all times
  - Don't need to run the same code at all times
  - Multiple optimized versions correct at different times
- Can make optimistic assumptions that *may be invalidated* later
  - So long as we *can repair* our mistakes in time
  - Code with broken assumptions must never be executed
  - Ideally, want invalidation to be unlikely

Type Profiling

- Type profiling can observe:
  - Frequency of calls
  - Types of arguments to calls
  - Types of values stored into globals
  - Types of values stored in object fields
- Goal: build fairly accurate profile of program behavior w.r.t. types

Type Propagation

- Form of type inference
- Dataflow analysis
- Local or whole program
- Rules depend on language semantics, e.g.:
  - add int, int → int
  - add float, float → float
  - mul m4x2, m2x1 → m4x1
  - getprop o, “a” → prop_type(o, “a”)
- In the local case, inputs are function argument types, globals types, closure variable types
- Output: local variable types, return type

Potential Difficulties

- Cost of profiling
  - Need accurate information
- Cost of recompilation
  - Usage of external threads
- Frequency of recompilation
  - Progressive pessimization
- Inherent complexity
  - Find more students!

Related work: Type Analysis

- M Chevalier-Boisvert, L Hendren, C Verbrugge. *Optimizing MATLAB through just-in-time specialization*. CC 2010.
- F Logozzo, H Venter. *RATA: Rapid Atomic Type Analysis by Abstract Interpretation–Application to JavaScript Optimization*. CC 2010.
- S Jensen, A Møller et al. *Type analysis for JavaScript*. SAS 2009.
- And much more...

Related work: Deoptimization

- I Pechtchanski, V Sarkar. *Dynamic optimistic interprocedural analysis: a framework and an application*. OOPSLA 2001.
  - Systematic optimistic interprocedural type analysis to optimize polymorphic call sites
- Speculative inlining
  - In Java, dynamic class loading can invalidate inlining decisions
  - Implemented in Java HotSpot VM
- Polymorphic inline cache

Related work: Tracing JITs

- HotpathVM, TraceMonkey, LuaJIT, etc.
- Tracing JITs are another dynamic compilation model
- Same basic underlying principle
  - Observe program as it runs, gather data about behavior
  - Assume current behavior will likely persist, use data to specialize program, minimize dynamic checks
- Main limitations
  - Local approach, detects & examines loops
  - Knows little about what goes on outside loops
  - No real way of dealing with global data, optimizing object layout, etc.

Related work: Meta-circularity

- JikesRVM: meta-circular Java VM
- Maxine VM: experimental project at Sun
- PyPy: Python in Python
  - JIT compiler generator based on language spec.
- Klein VM: Implementation of Self in Self

Distinguishing Features

- Meta-circular w/ dynamic language
- Self-optimizing
- Systematic optimistic optimizations
- Implementation flexibility
  - Function call protocol
  - Object layout
  - Intermediate representation
- Inline IR
- Multithreaded compiler

Project Status

- Working:
  - ECMAScript 5 parser
  - Translation of ASTs to SSA-based IR
  - Simple optimizations on IR
    - SCCP, value numbering, peepholes
  - x86 32/64 back-end w/ linear-scan reg. alloc.
  - Compilation of simple programs
    - Fibonacci, loops
  - Precise statistical profiler
- In progress:
  - Library of JS primitives (objects, strings)
  - Compilation of more complex programs
  - Back-end optimizations
  - Integration into Chrome

Thanks for Listening!

We welcome your questions/comments

Feel free to contact the Tachyon team:

{chevalma, lavoeric, feeley, dufour}@iro.umontreal.ca