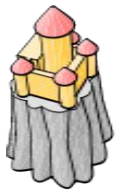
How to Perform a Brain Surgery on Yourself:
Safe Bootstrapping for Self-Supporting Systems

Christoph Thiede

Joint project with Marius Dörbandt, Eliot Miranda, Marcel Taeumel, and Robert Hirschfeld

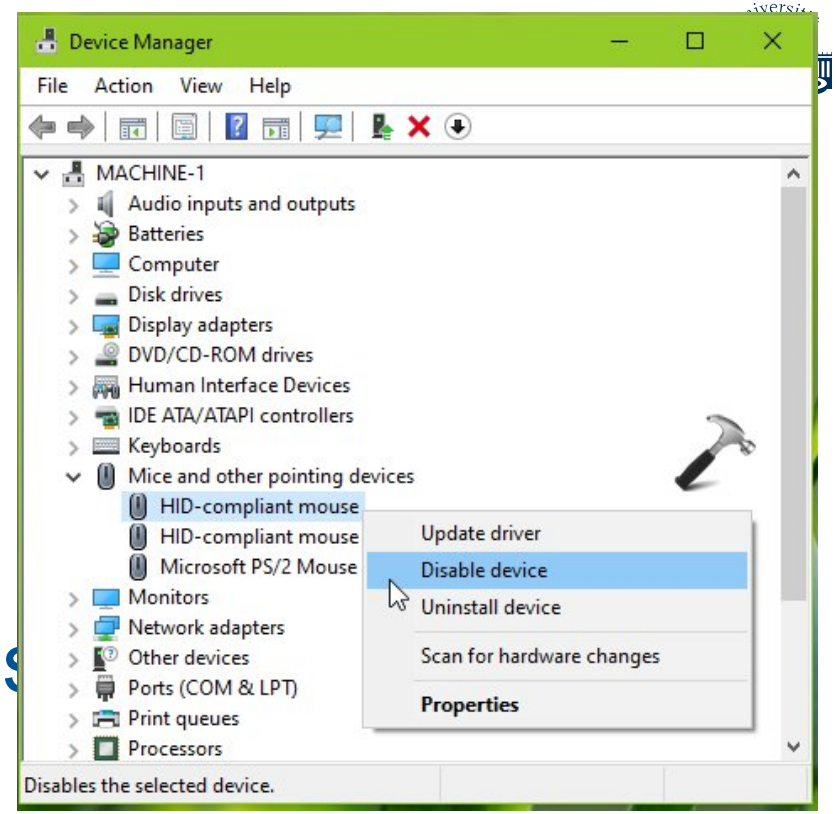
Software Architecture Group

Systems Design Research School Meeting 2026-06-03



Who here already has managed to destroy your own development environment?

```
$ pip install --upgrade pip
Traceback (most recent call last):
  File "/usr/local/bin/pip", line 7, in <module>
    from pip._internal import main
ModuleNotFoundError: No module named 'pip._internal'
```



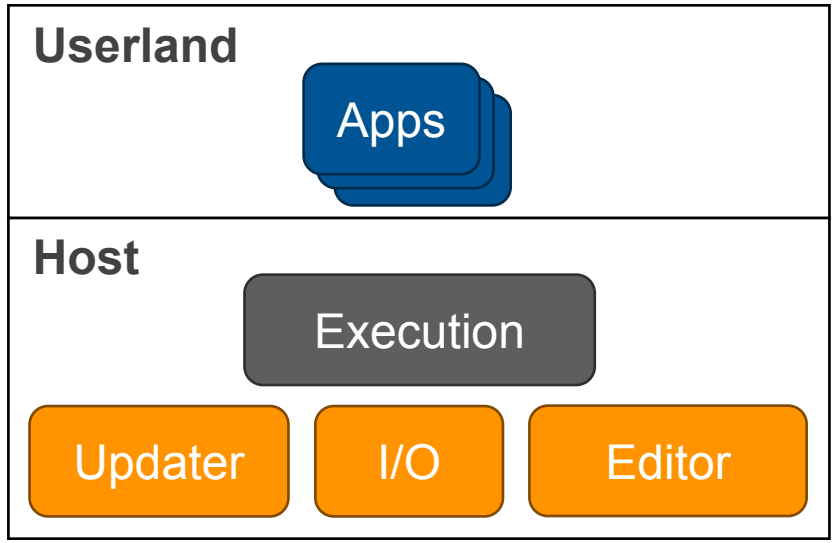
Who here already has managed to des your own development environment?

```
[FAILED] Failed to mount /boot.
See 'systemctl status boot.mount' for details.
[    1.984618] systemd[1]: Dependency failed for Local File Systems.
[DEPEND] Dependency failed for Local File Systems.
...
[    2.830035] usb 3-2.3: device descriptor read/64, error -32
...
You are in emergency mode. After logging in, type "journalctl -xb" to view
system logs, "systemctl reboot" to reboot, or "exit"
to continue bootup.
Give roor password for maintenance
(or press Control-D to continue):
```

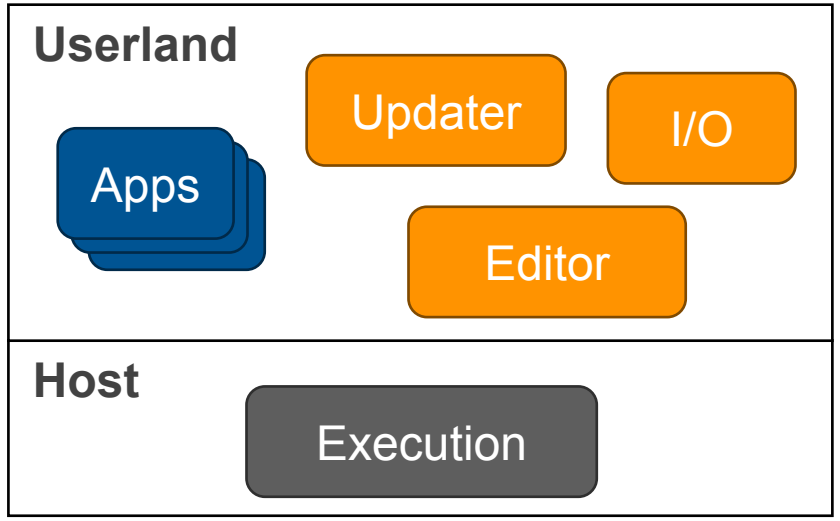
Self-Supporting Systems

- Self-supporting systems implement core components in themselves
  - Software updater, code editor, compiler, debugger, ...
  - Can be developed using userland language and tools
  - Can be changed at runtime
- But when they break, the entire system can become inoperable

[Jakubovic 2023, Ingalls 2020a, Ungar 2005]



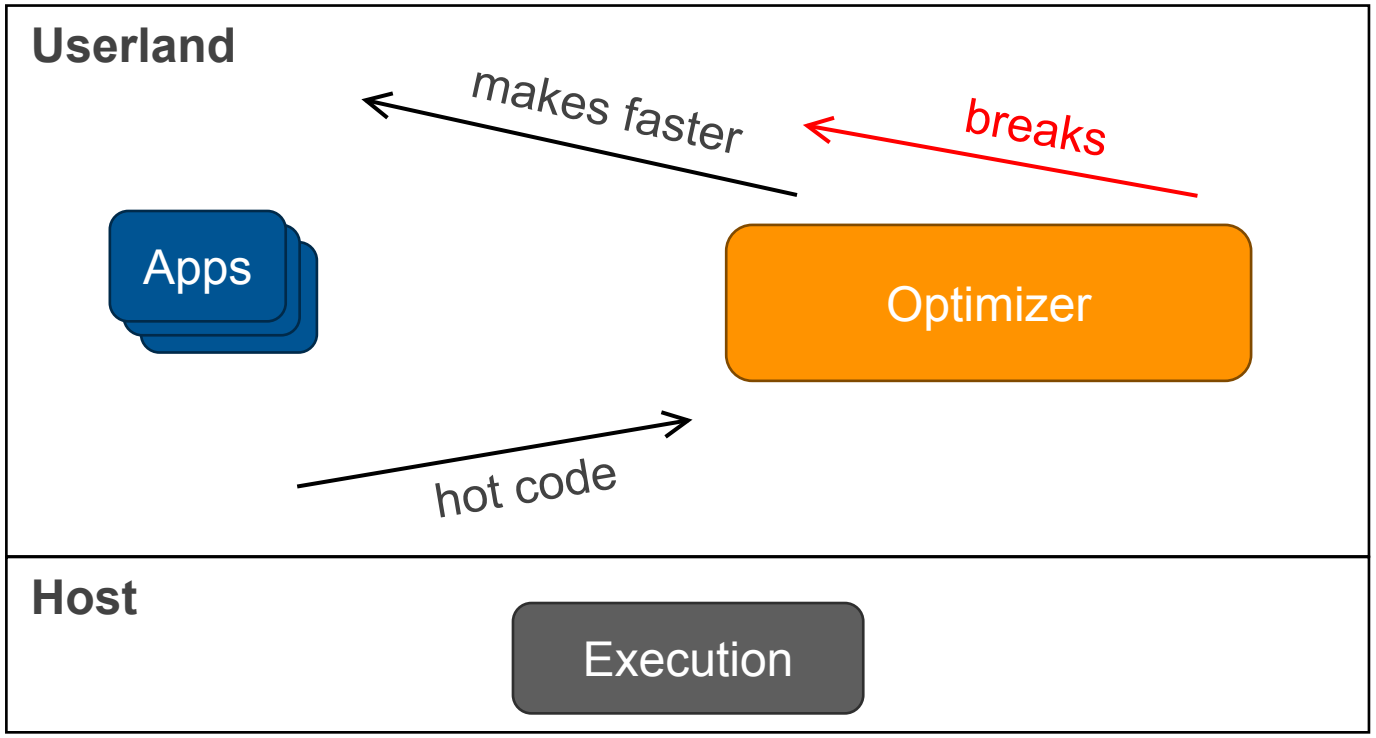
Conventional system



Self-supporting system

Sista/Scorch

- Goal: Develop an adaptive optimizer for Squeak/Smalltalk that dynamically optimizes hot methods
- Implement it in userland
- Originally proposed and prototyped by Béra & Miranda (2014)



[Béra 2014—2017]

How can we bootstrap self-supporting components before they are stable enough to sustain themselves?

Outline

1 **Why Self-Supporting Systems?**	2 **Live Programming for VMs**
3 **Surgery Strategies for Self-Supporting Systems**	4 **Contribution: Bootstrapping via Out-of-Place Execution**

Outline

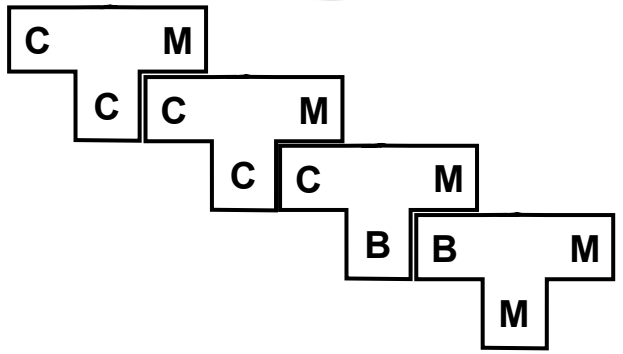
1 **Why Self-Supporting Systems?**	2 **Live Programming for VMs**
3 **Surgery Strategies for Self-Supporting Systems**	4 **Contribution: Bootstrapping via Out-of-Place Execution**

Dogs, Chickens, and Eggs

- Dogfooding: System programmers want to implement the system in itself

  - Reuse language, abstractions, and tools
  - Test them during development („eat your own dogfood“)
  - ➢ Self-hosted system: a system developed using a former version of itself
    - a C compiler written in C (e.g., clang, gcc)
    - a Linux kernel built on Linux
    - Microsoft Visual Studio is developed in Visual Studio
- The chicken-and-egg problem: how to develop the first version of the system?

  - ➢ Bootstrapping: Initialize a self-hosted development workflow through a seed step
    - The first C compiler was written in B + assembler
    - The first Linux was built on MINIX



[Terry 1997]

Live Programming

- Conventional programming systems separate compile time and runtime
  - To change the program, frequent restarts are required
  - Slow build chains, need to (manually) reproduce situations
  - ➢ Long feedback cycles
- Live programming systems aim to combine both stages
  - Change program at runtime
  - e.g., hot reloading or incremental compilation
  - ➢ Shorter feedback cycles, programming feels almost immediate

[Tanimoto 2013]

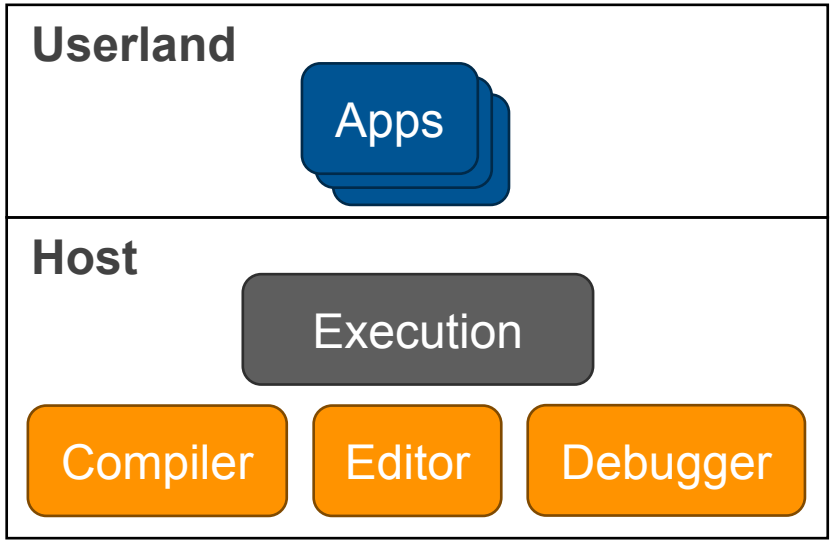
Exploratory Programming

- Agile development workflow
  - Gradually explore problem space and solution space
  - Run experiments and build prototypes in the live system
  - Browse code, run code snippets, inspect runtime objects, debug methods, ...
- Programming in the debugger: explore unhandled runtime error in a debugger, change methods on the stack, resume latest stack frame
- Debugging into existence: implement a program top-down (interface-first), implement missing methods on-the-fly in a debugger

[Sandberg 1988, Taeumel 2022, Rosson 1993]

Self-Supporting Systems

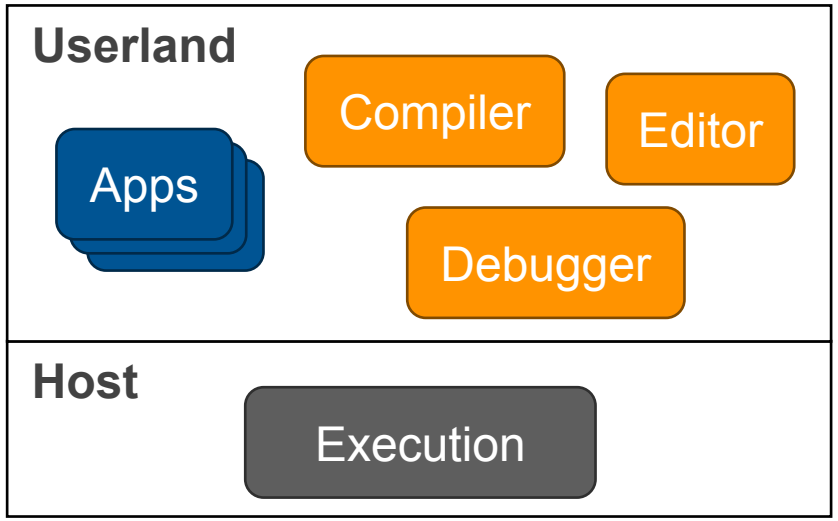
- Desire: Live and exploratory programming for self-hosted components
  - Recompile compiler at runtime
  - Change the code editor while using it
  - Debug a debugger invocation



[Jakubovic 2023, Ingalls 2020a, Ungar 2005]

Self-Supporting Systems

- Desire: Live and exploratory programming for self-hosted components
  - Recompile compiler at runtime
  - Change the code editor while using it
  - Debug a debugger invocation
- Solution: Co-locate system components together with apps in the same environment („userland“)
  - Develop and run them together

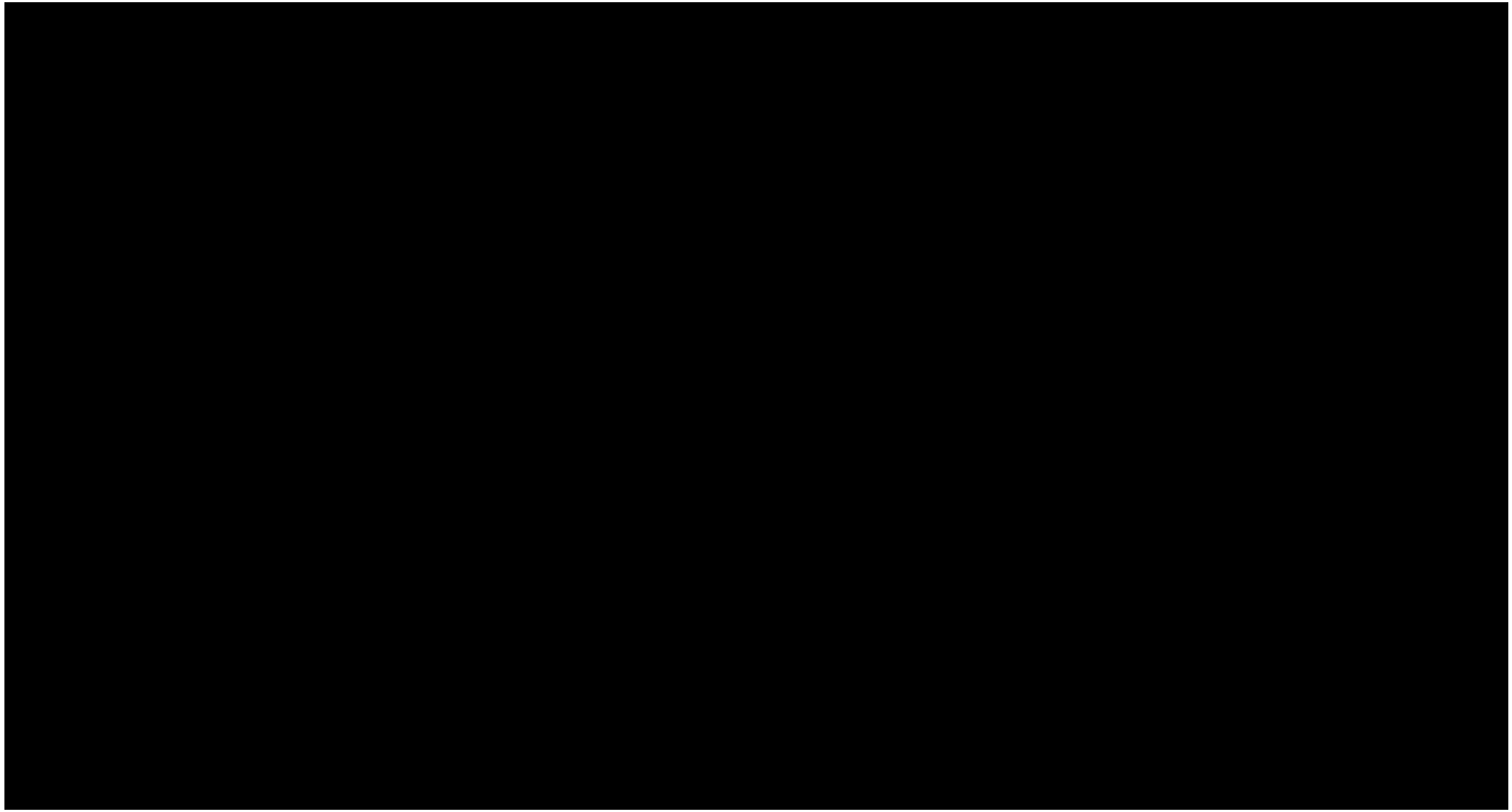


[Jakubovic 2023, Ingalls 2020a, Ungar 2005]

Self-Supporting Systems — Example: Compiler

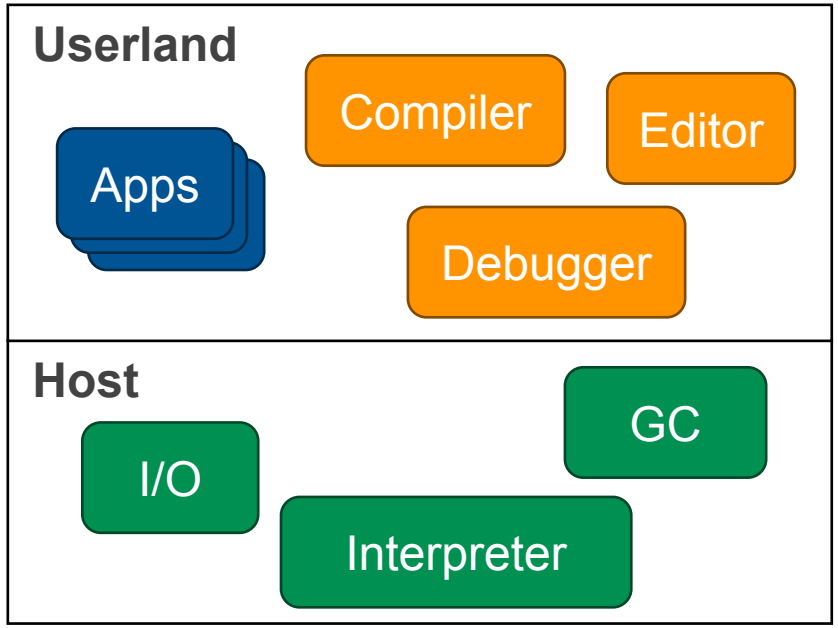
- Goal: Change syntax of number literals
- Traditional approach:
  - Check out compiler sources
  - Modify compiler
  - Rebuild compiler
  - Modify application
  - Rerun application
  - ...possibly repeat...
- In a self-supporting system:
  - Inspect or debug compiler at runtime
  - Modify compiler at runtime
  - Modify application at runtime
  - ➢ Shorter feedback cycles, more flexible development
  - You can break the development environment

Self-Supporting Systems — Example: Compiler



But every self-supporting system needs a substrate...

- Virtual machine / interpreter / runtime



- **...how can we live-program that substrate?**

Outline

1 **Why Self-Supporting Systems?**	2 **Live Programming for VMs**
3 **Surgery Strategies for Self-Supporting Systems**	4 **Contribution: Bootstrapping via Out-of-Place Execution**

Self-Hosted VMs

- Approach: Implement and compile VM using former version of itself
- Result:
  - Reuse high-level language and tools (e.g., editor, debugger)
  - Often metacircular: Implement execution semantics but offload some implementation details (e.g., calling conventions, object model) to host VM
- Still, cannot change VM components at runtime

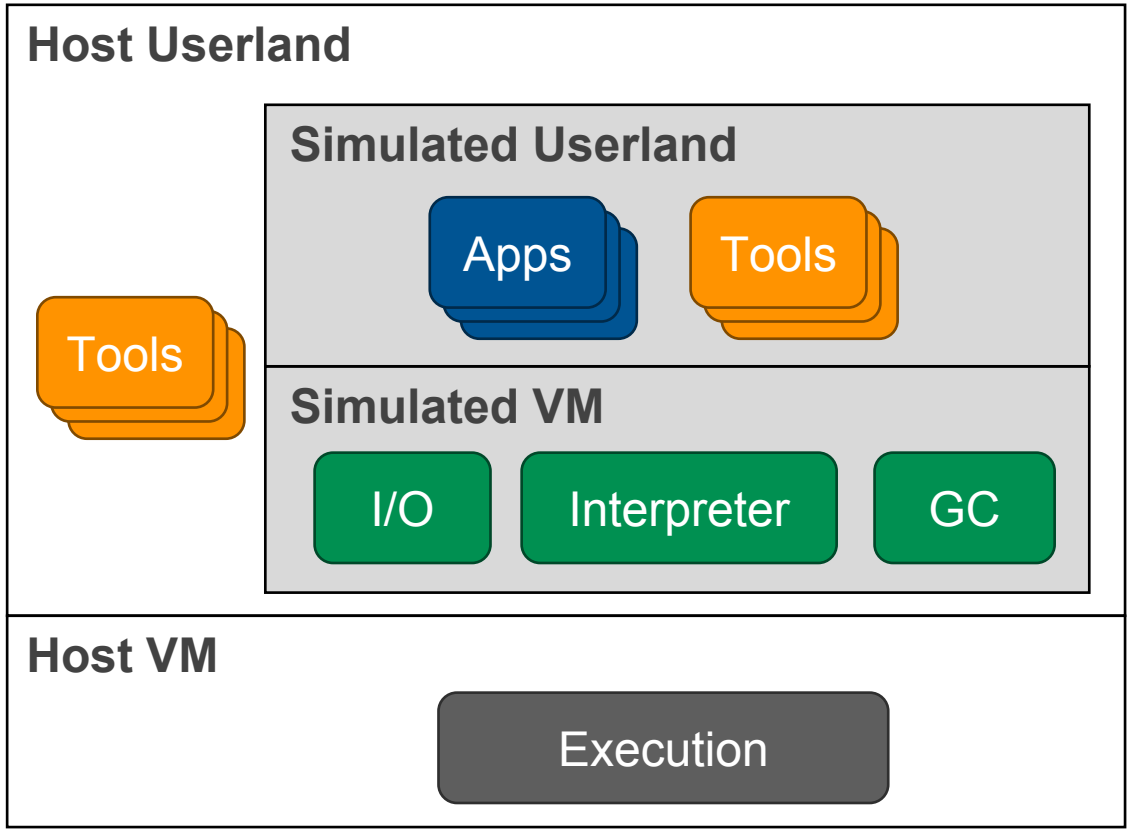
Klein/Self

Maxine VM

[Ungar 2005, Wimmer 2013, Ingalls 2020b]

OpenSmalltalk VM: Simulation

- Approach:
  - Self-hosted VM (implemented in subset of Smalltalk dubbed Slang)
  - Production: Transpile Slang to C and compile
  - Simulation: Run and debug Slang in host VM
- Result:
  - Reuse high-level language and tools (e.g., code browser, debugger, inspectors)
  - Change simulated VM at runtime

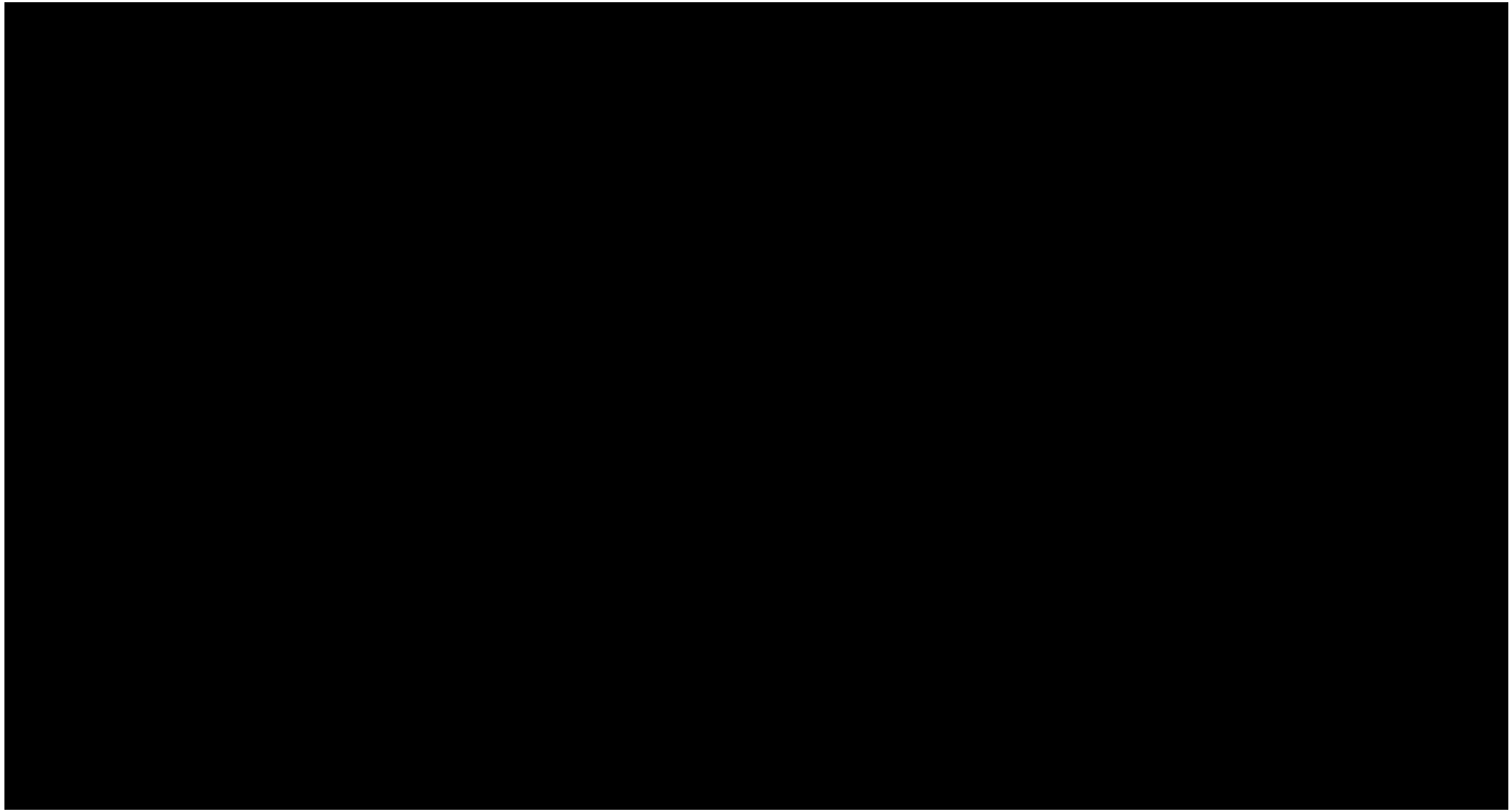


[Ingalls 2020b]

OpenSmalltalk VM: Simulation — Example

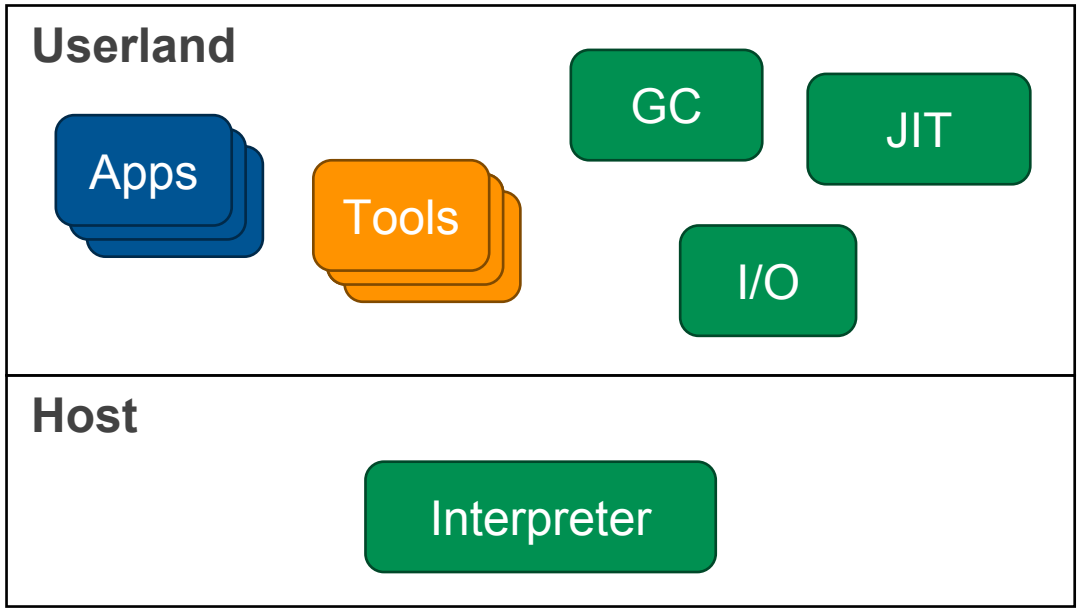
- Goal: Fix/change behavior of an arithmetic instruction
- Traditional approaches:
  - Reason about static source code of the VM
  - Debug VM with gdb/llvm/...
  - Rebuild VM, restart VM and program
  - ...possibly repeat...
- Using the OpenSmalltalk simulator:
  - Debug VM in high-level Smalltalk
  - Change interpreter at runtime in the debugger
  - Restart interpretation of single instruction
  - ➢ Faster feedback cycles
  - ➢ Exploratory programming for VM components
- The holy grail of live VM development?

OpenSmalltalk VM: Simulation — Example



Live Metacircular Runtimes

- Approach: Radically minimize substrate
  - Lift up VM components such as GC and JIT to userland



Bee Smalltalk

- **...but how can we change critical system components at runtime?**

[Pimas 2023]

Outline

1 **Why Self-Supporting Systems?**	2 **Live Programming for VMs**
3 **Surgery Strategies for Self-Supporting Systems**	4 **Contribution: Bootstrapping via Out-of-Place Execution**

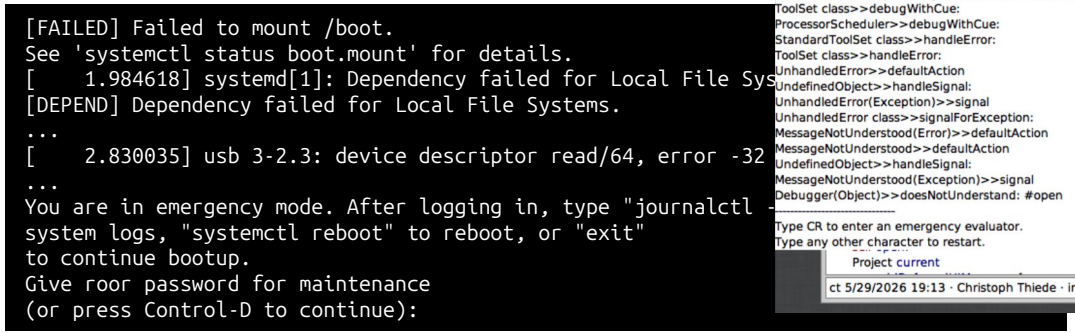
Self-Brain Surgeries

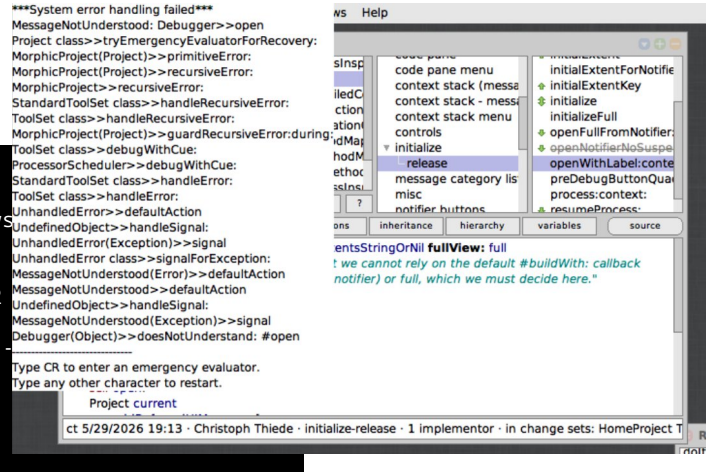
- Transactional updates: Prepare new code, then swap atomically
  - Multiple source code layers
  - Multiple object spaces managed via hypervisor in VM
  - ➢ Limits liveness
  - ➢ If the change was bad, the system will crash nonetheless

[Mattis 2017, Polito 2015]

Self-Brain Surgeries

- Transactional updates: Prepare new code, then swap atomically
  - Multiple source code layers
  - Multiple object spaces managed via hypervisor in VM
  - ➢ Limits liveness
  - ➢ If the change was bad, the system will crash nonetheless
- Fallback mechanisms
  - Emergency mode, native debugger, post-mortem analyzer, ...
  - ➢ Low-level tools
  - ➢ Disrupt live-programming flow

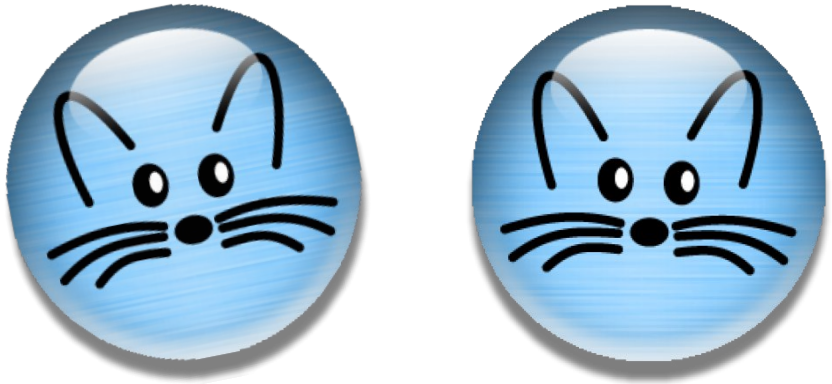




[Mattis 2017, Polito 20 15, Taeumel 2016]

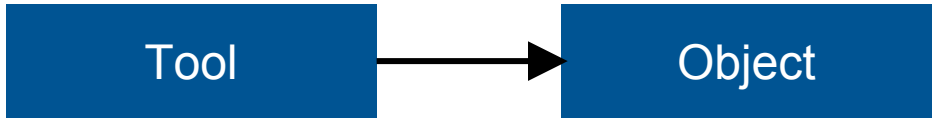
Self-Hosted, Out-of-Place Development

- Approach: Do not operate on yourself but on somebody else
  - Change unstable system from separate stable system
- Remote debugging
  - Tools in development system access target system in remote VM via bridge
  - Inspect objects and methods or change them
  - Reify remote objects to reuse local tools



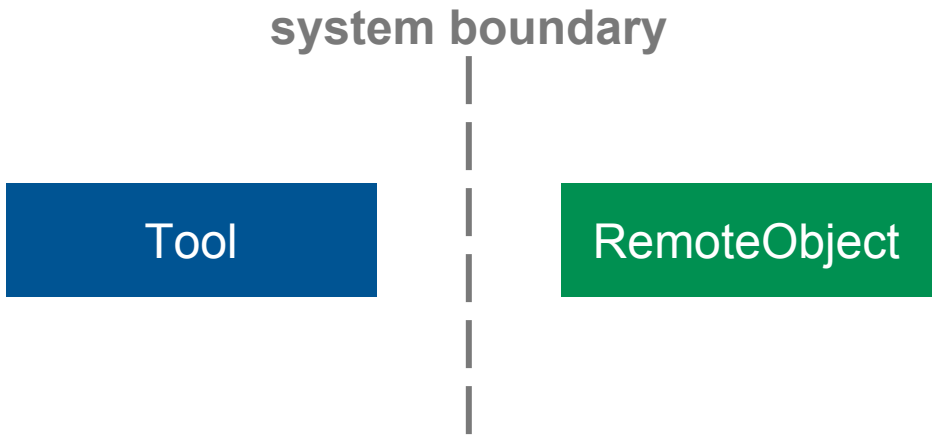
[Marra 2018, Ungar 2005, Wimmer 2013]

Self-Hosted, Out-of-Place Development: Reification Strategies




```
list = object.instanceVariables()
title = object.toString()
```

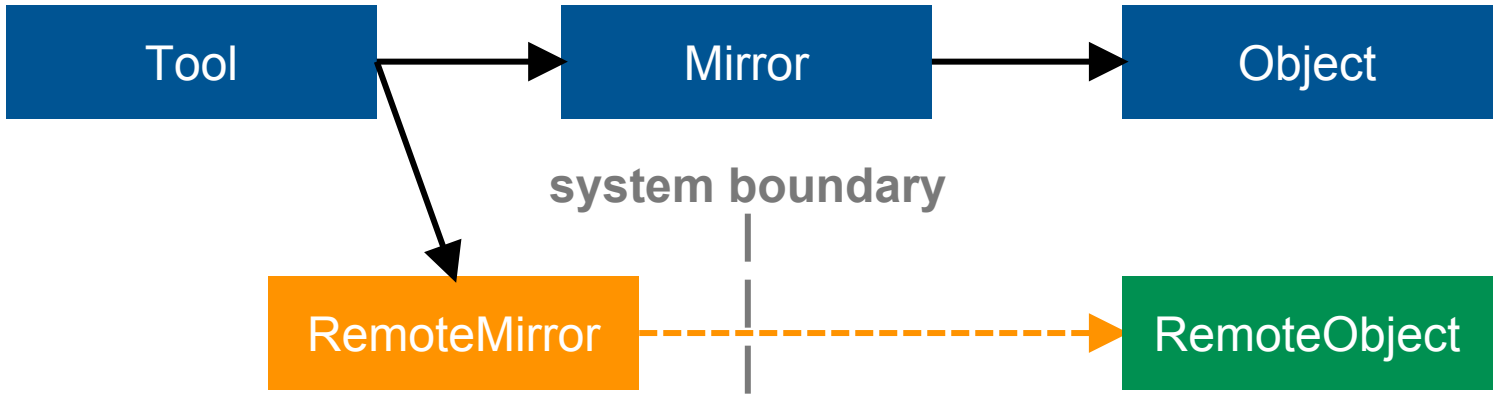
Self-Hosted, Out-of-Place Development: Reification Strategies



list = object.instanceVariables()
title = object.toString()

Self-Hosted, Out-of-Place Development: Reification Strategies

- Mirrors: Separate domain logic (base level) from reflection (meta-level)
  - Plug a remote mirror into existing local tools
  - Tools only have access to meta-level

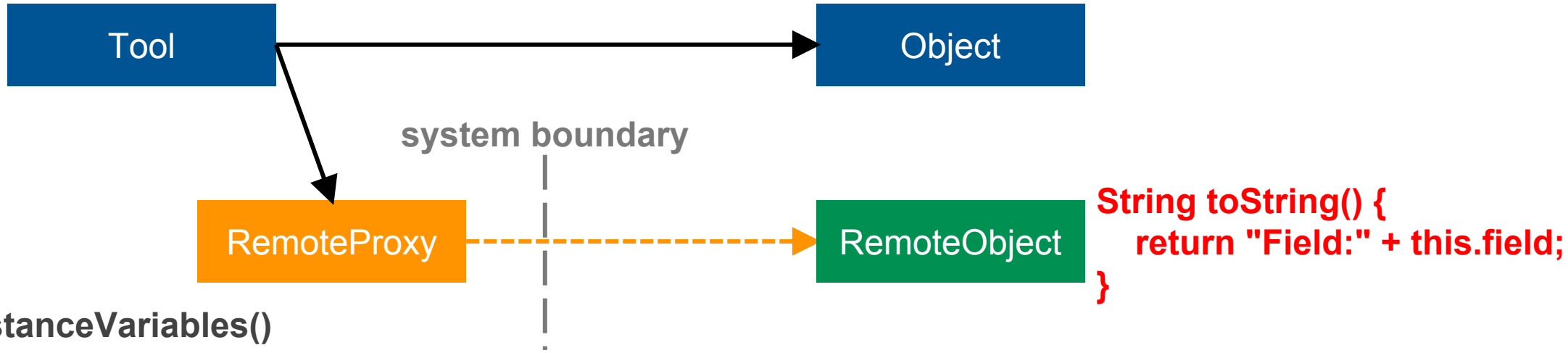


list = mirror.instanceVariables()
~~**title = object.toString()**~~

[Bracha 2004]

Self-Hosted, Out-of-Place Development: Reification Strategies

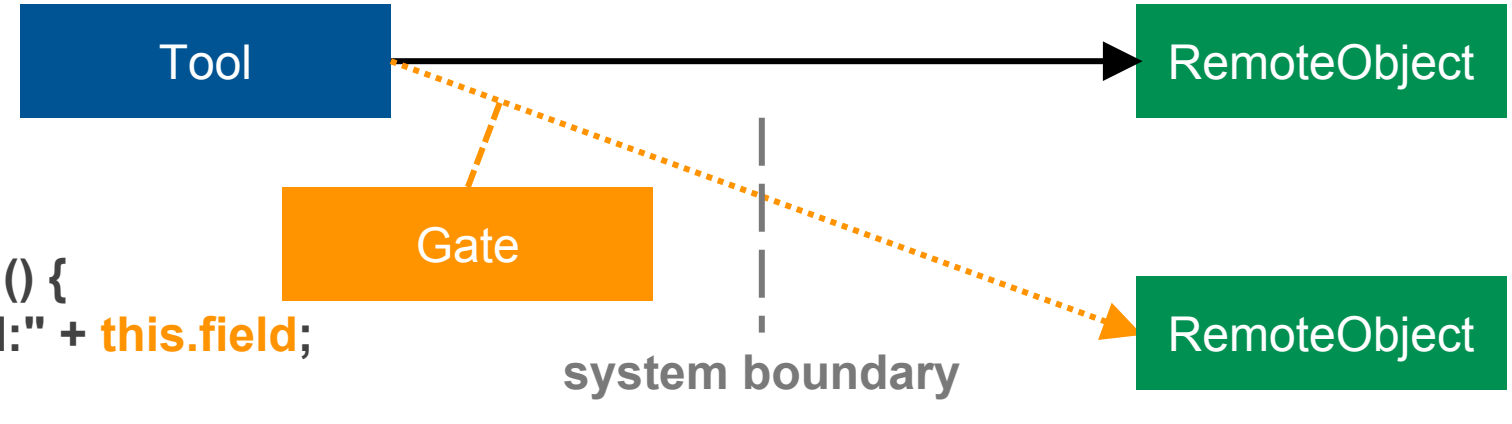
- Transparent proxies: Forward all accesses from tool to remote object
  - Execute methods in remote system
  - Remote system must be runnable (i.e., not suspended)



[Van Cutsem 2010]

Self-Hosted, Out-of-Place Development: Reification Strategies

- Metaphysics: Execute class methods locally against remote objects
  - Direct gates: Fetch method from target system
  - Mirage gates: Use equivalent method from development system
  - Use behavioral intercession to redirect object accesses (e.g., reading/writing fields) during interpretation
    - e.g., bytecode rewriting, virtualization, custom interpreter




```
String toString() {
    return "Field:" + this.field;
}
```

```
String toString() {
    return "Field:" + this.field;
}
```

```
list = mirage { object.instanceVariables() }
title = mirage { object.toString() }
```

[Pimas 2017]

Self-Hosted, Post-Mortem Debugging

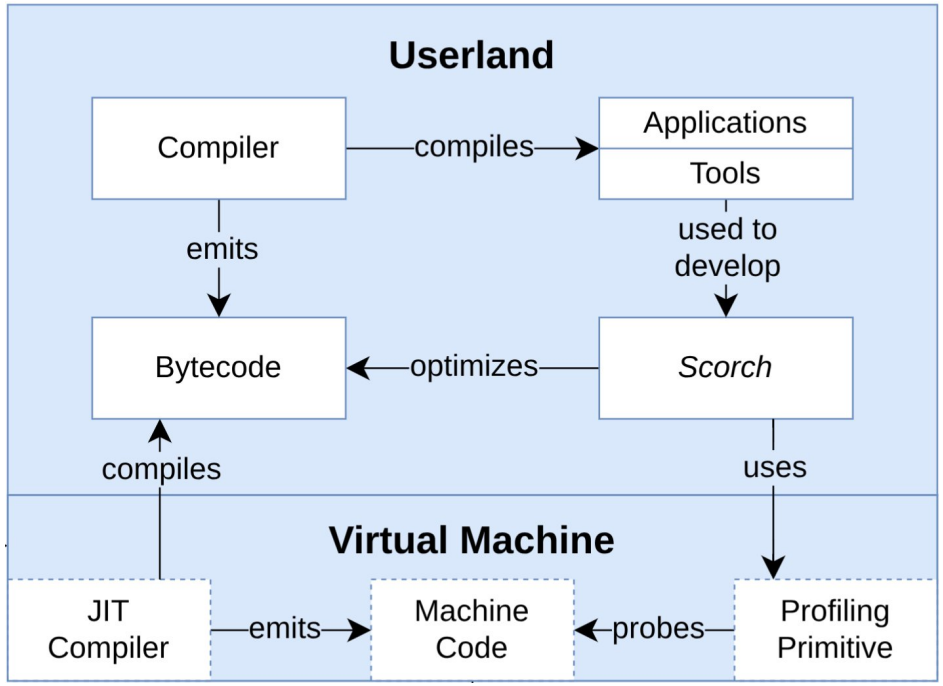
- Holographic objects: Reify objects from crash dump as read-only proxies
  - Reuse local tools
  - But no live programming

[Salkeld 2013]

The Case of Sista/Scorch

The Case of Sista/Scorch: Design

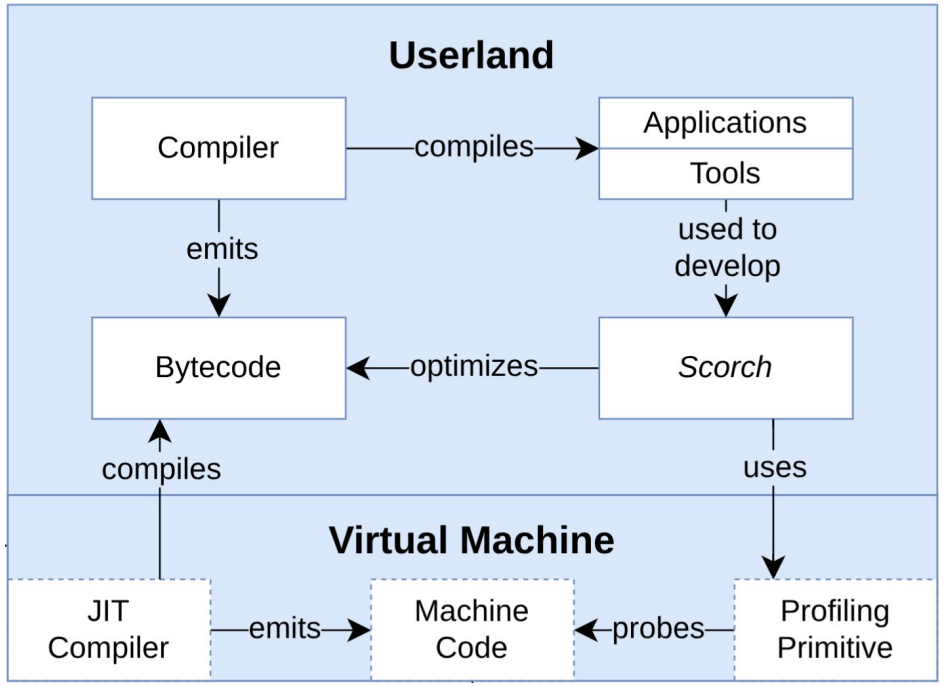
- Originally proposed and prototyped by Béra & Miranda (2014)
- Sista (Speculative Inlining SmallTalk Architecture):
  New bytecode set for method inlining
  - Introduces faster, unsafe instruction types
  - Processed by interpreter and JIT (just-in-time compiler)
- Scorch: Adaptive optimizer, running in userland
  - Notified by VM about hot code paths
  - Analyzes branch counters and inlines methods
  - Bytecode can trigger deoptimization by Scorch when an unexpected case occurs



[Béra 2014—2017]

The Case of Sista/Scorch: Design

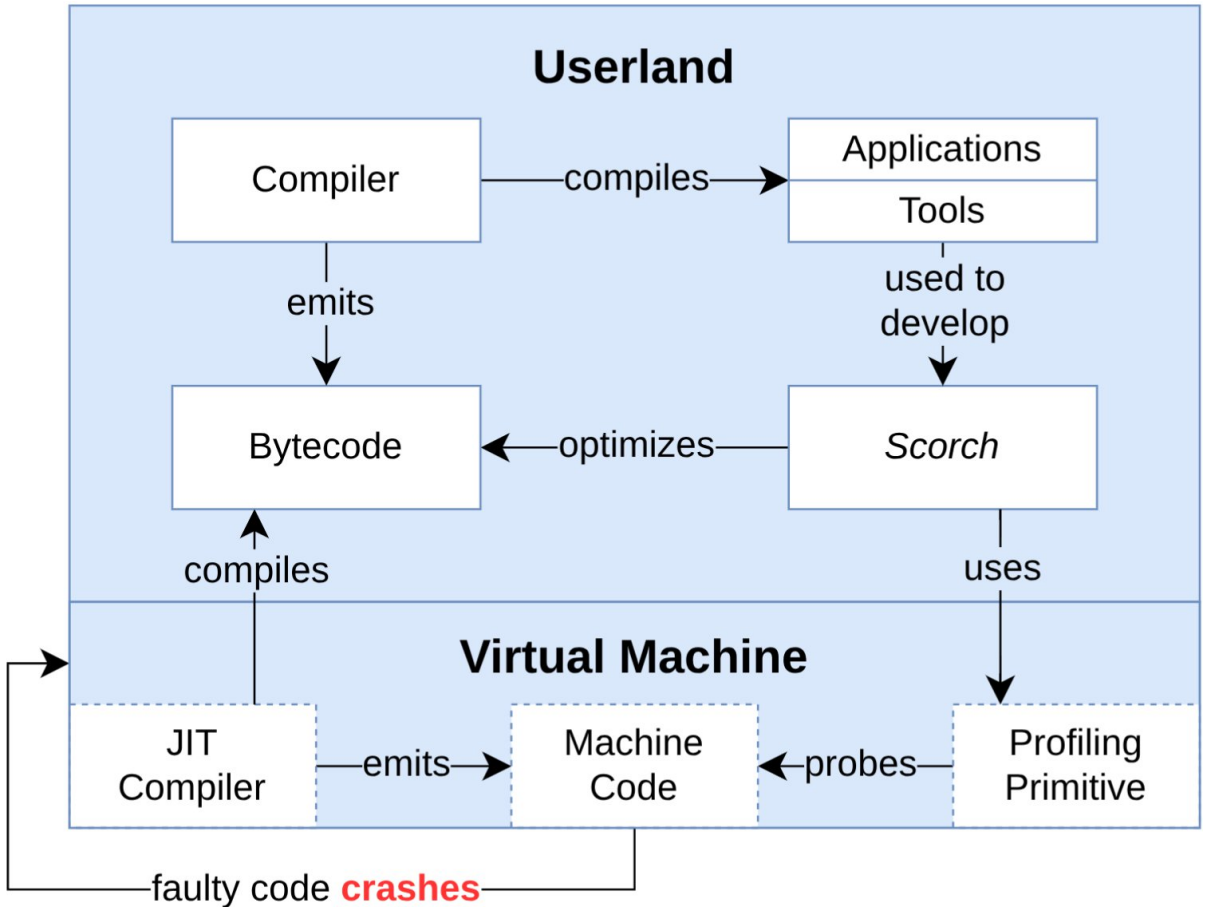
- Forces during bootstrapping:
  - Stability: Invalid bytecodes/interpretation can crash the VM (including tools)
  - Debugging interferences: Execution of tools or even optimizer itself can trigger new optimizations —> optimization context is hard to reproduce
  - Strong coupling between userland optimizer and VM-level executors: Need to develop both simultaneously



Bootstrapping Sista/Scorch

Approach #1: Develop in production

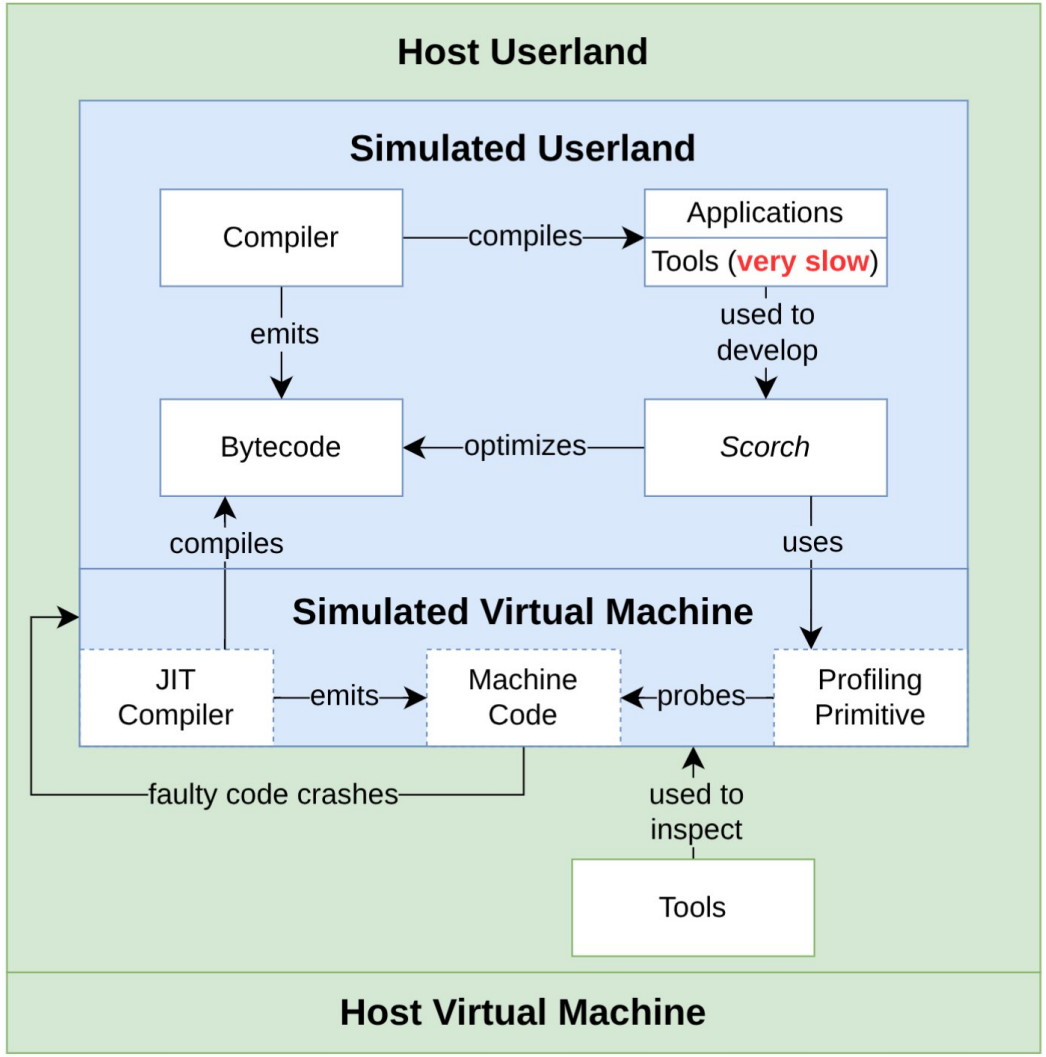
- Development:
  - Scorch: Live programming
  - Sista: No runtime changes; only low-level, post-mortem debugging
- Stability: Frequent crashes of system + tools
- Debugging interferences



Bootstrapping Sista/Scorch

Approach #2: Develop in simulation

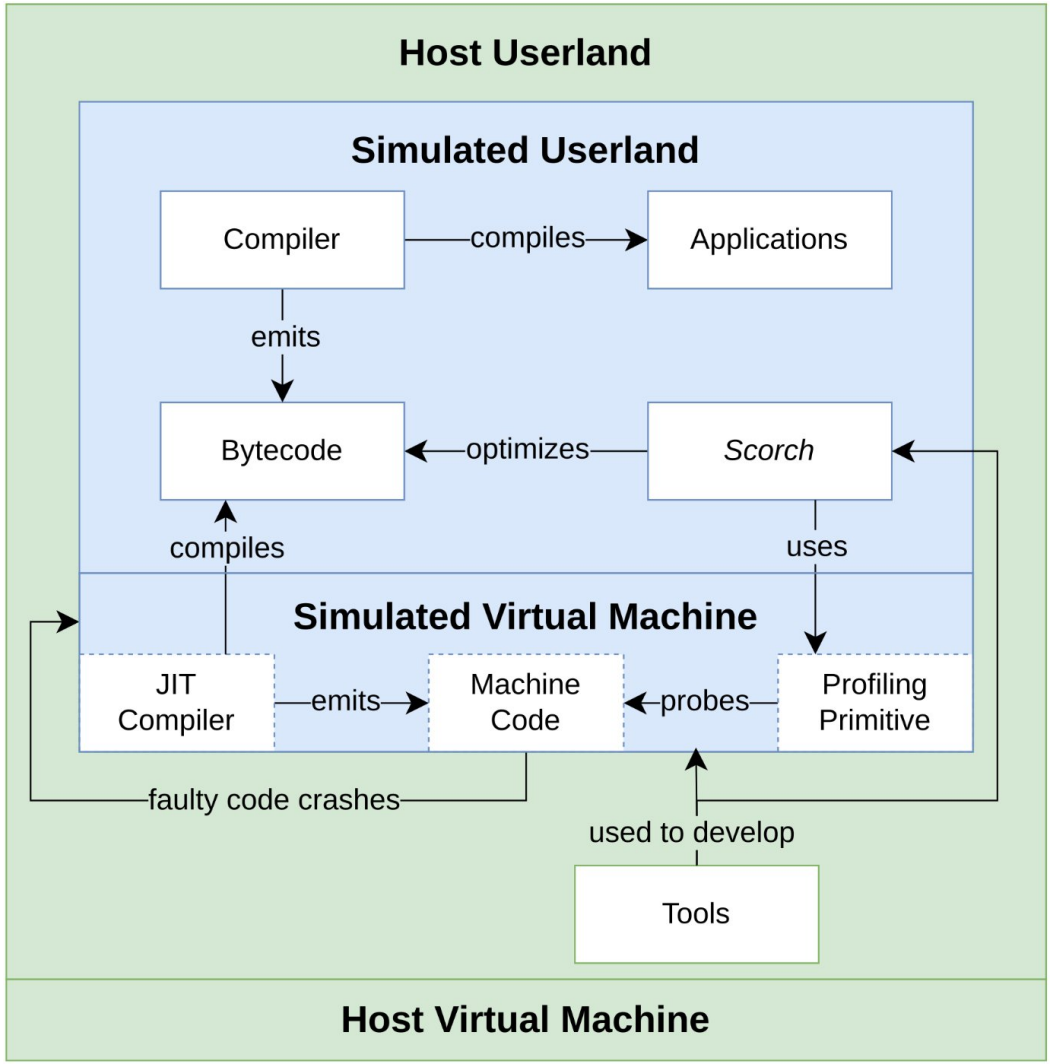
- Development:
  - Scorch: Live programming, but very slow in simulation
  - Sista: Live programming
- Stability:
  - Sista: Host system remain stable
  - Scorch: Simulated tools + code are lost on crash
- Debugging interferences



Bootstrapping Sista/Scorch

Approach #3: Out-of-place development + simulation

- Development:
  - Scorch: Live programming
  - Sista: Live programming
- Stability:
  - Sista: Host system remains stable
  - Scorch: Simulated code is lost on crash
- Debugging interferences



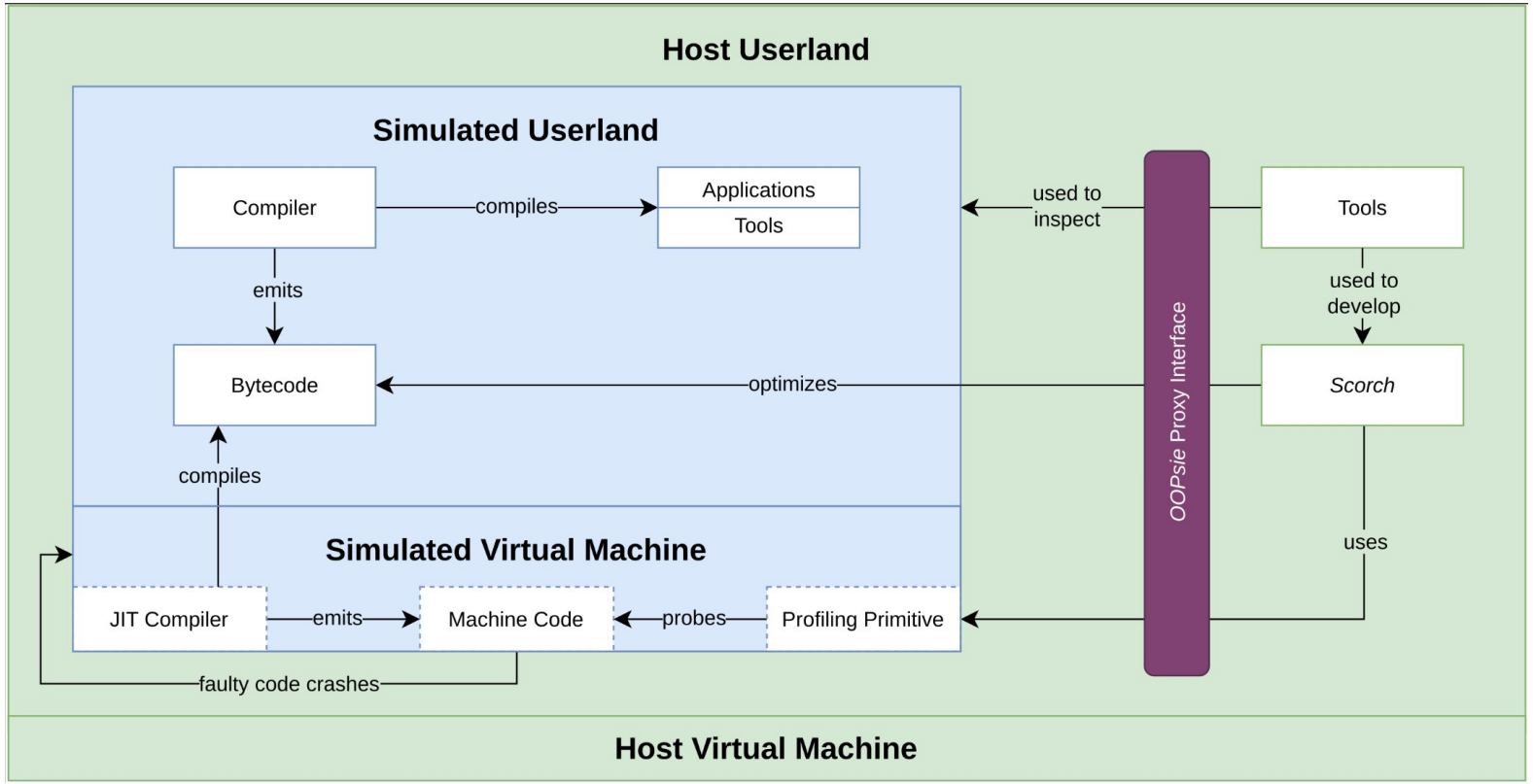
Outline

1 **Why Self-Supporting Systems?**	2 **Live Programming for VMs**
3 **Surgery Strategies for Self-Supporting Systems**	4 **Contribution: Bootstrapping via Out-of-Place Execution**

Bootstrapping Sista/Scorch

Our solution: Out-of-place execution + simulation

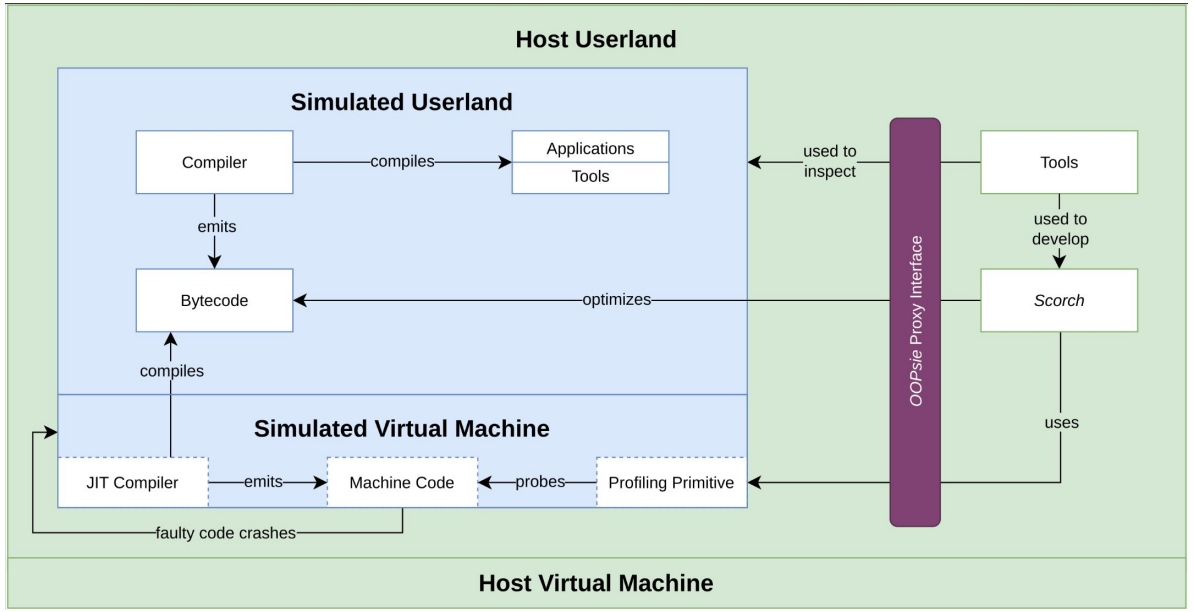
- Run and develop Scorch in the host system, outside of the simulation
- Interface it to the simulated system through a transparent proxy membrane
- Reify simulated objects using mirage gates



Bootstrapping Sista/Scorch

Our solution: Out-of-place execution + simulation

- Development:
  - Scorch: Live programming
  - Sista: Live programming
- Stability:
  - Sista: Host system remains stable
  - Scorch: Simulation still crashes, but all source code and tools survive in host
- No debugging interferences: All tools and Scorch run in host image, no implicit side-effects on simulation



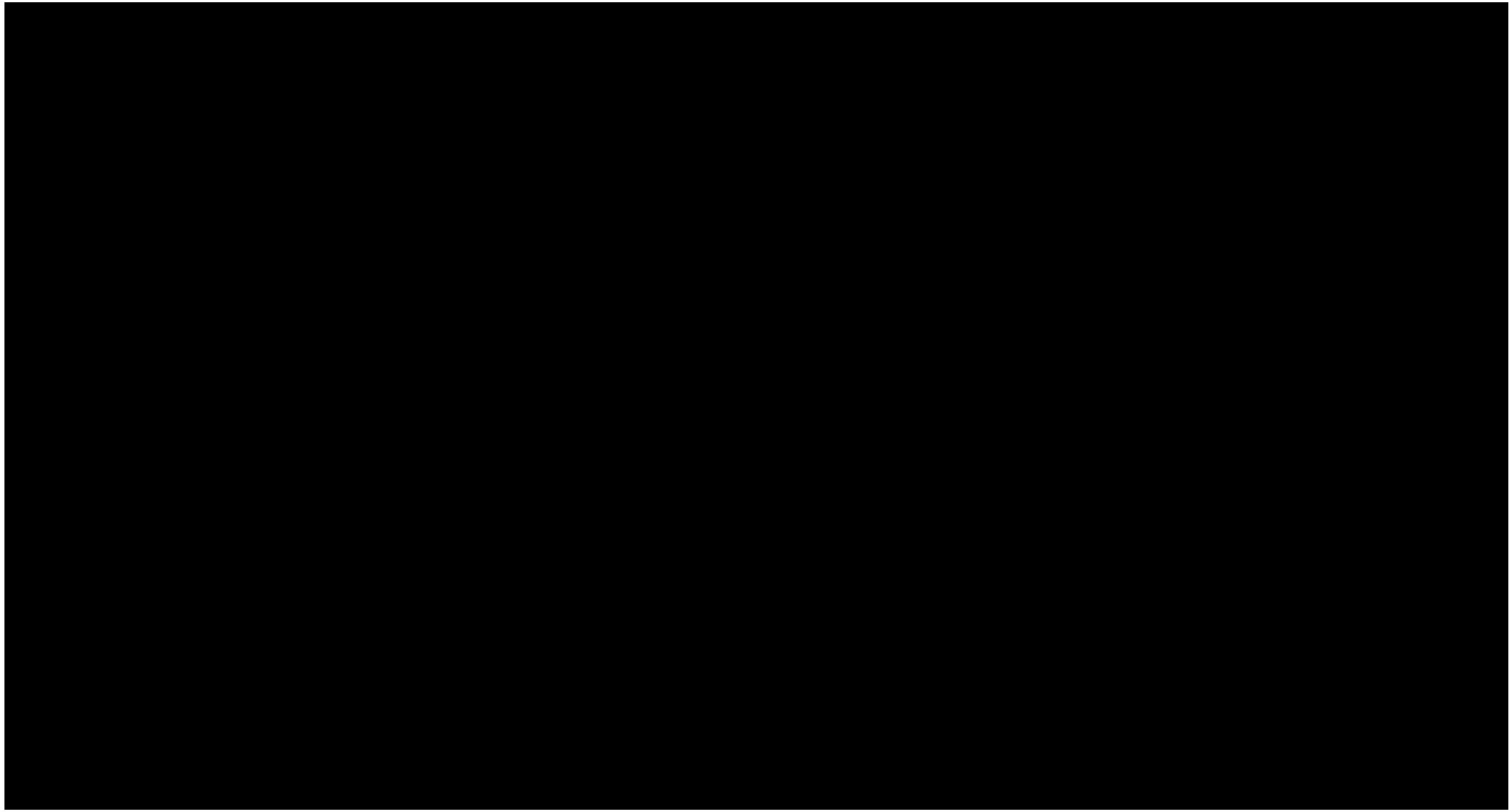
Out-of-Place Execution: Implementation

- OOPsie: Ordinary Object PointerS Interoperation Engine
  - Override Sista events (branch counter trip, deoptimization trap) in simulator, invoke Scorch in host image instead
  - Transparent proxy membrane via mirage gates
    - Redirect state accesses for behavioral intercession by overriding userland interpreter

Out-of-Place Execution: Demo

- Optimizing a Fibonacci benchmark
  - Run benchmark in simulation
  - Sista VM emits counter trip for hot method
  - Oopsie forwards counter trip and creates mirages for method/context objects
  - Scorch optimizes methods in host image
  - Optimized execution resumes in simulation
- Out-of-place development via Oopsie
  - Inspect mirage for method from simulation by reusing method inspector in host via mirage
  - Debug current call stack of simulated VM via proxy membrane in host debugger
  - Browse and change source code of simulated VM in host editors

Out-of-Place Execution: Demo

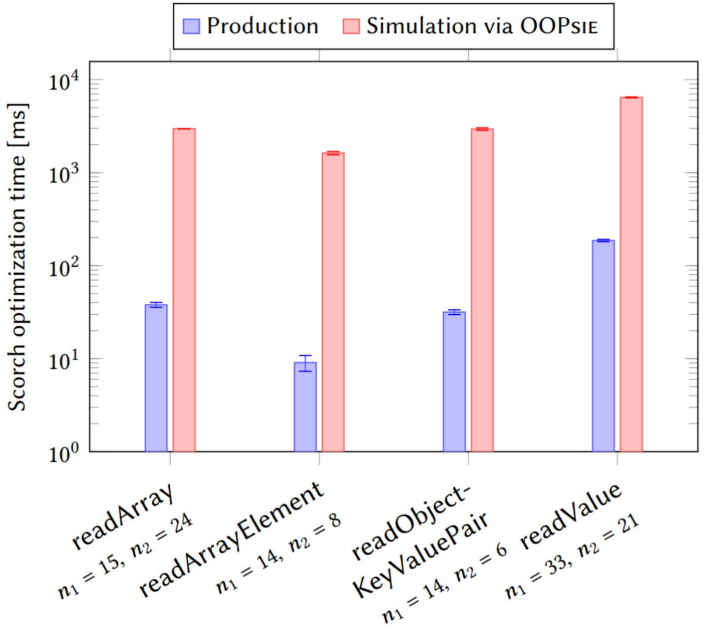


Out-of-Place Execution: Discussion

- Programming experience
  - Simultaneous live programming of userland and VM-level components
  - Stable development environment saves programmer flow during bootstrapping or later substantial changes
  - One implementation, two operation modes: Scorch can run either as self-supporting component in production or outside of simulation

Out-of-Place Execution: Discussion (ctd.)

- Performance
  - Code execution in simulation is slower by factor of 2000x-3000x
    - Still fast enough to run benchmarks and trigger optimization (~6 seconds for one AWFY benchmark)
  - Mirage gates slow down optimization by 35x-180x
    - Still fast enough for live programming (1-7 seconds per optimization cycle)
    - For comparison: Rebuilding production VM takes >60 seconds
  - Future work: Speed up mirage gates via bytecode rewriting or VM virtualization instead of userland interpreter



Future Work

- How will the programming experience evolve as we develop Sista/Scorch further?

- How can we enable safe brain surgeries yet uphold the experience of a singular self-supporting system?
  - Refactor VM components to use mirrors and reify them on demand in userland
  - Avoid upfront commitment to out-of-place execution and automatically fall back to simulation on errors

Conclusion

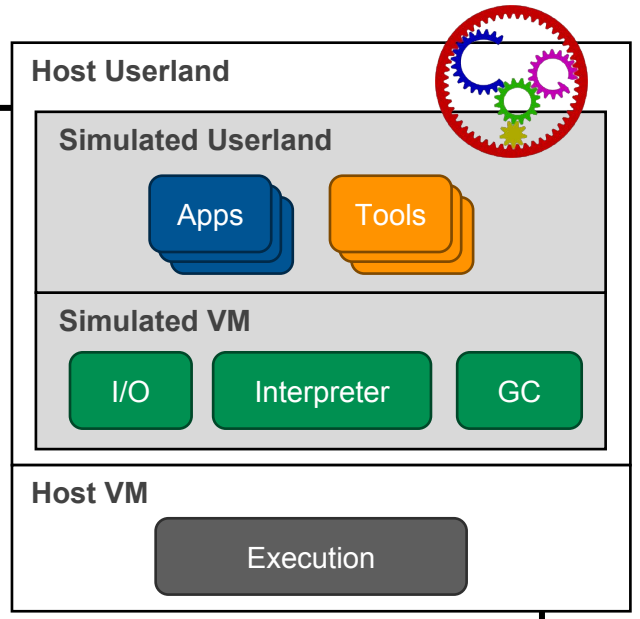
1   

Why Self-Supporting Systems?

2 

Bee Smalltalk

Live Programming for VMs

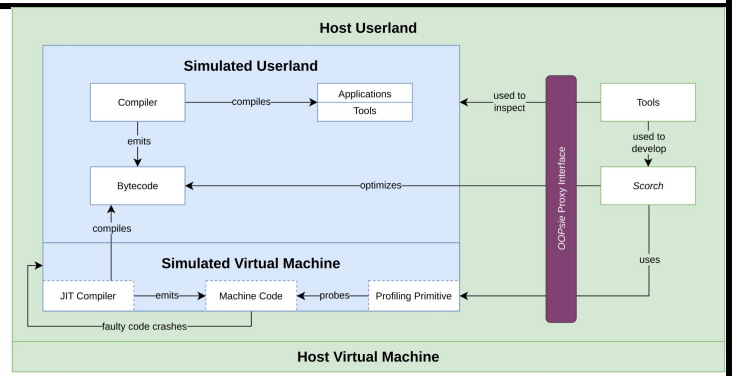


3   

Surgery Strategies for Self-Supporting Systems

4

Contribution: Bootstrapping via Out-of-Place Execution



Further Reading

- Christoph Thiede, Marius Dörbandt, Eliot Miranda, Marcel Taeumel, and Robert Hirschfeld. 2026. Proxies All the Way Down: Bootstrapping a Userland Speculative Optimizer for the OpenSmalltalk VM. Preprint, 15 pages.
- https://github.com/hpi-swa-lab/osvm-oopsie

Literature

Clément Béra. 2017. Sista: A Metacircular Architecture for Runtime Optimisation Persistence. Doctoral dissertation. Université de Lille 1.

Clément Béra and Eliot Miranda. 2014. A Bytecode Set for Adaptive Optimizations. *Proceedings of the International Workshop on Smalltalk Technologies (IWST 2014).*

Clément Béra, Eliot Miranda, Tim Felgentreff, Marcus Denker, and Stéphane Ducasse. 2017. Sista: Saving Optimized Code in Snapshots for Fast Start-Up. *Proceedings of the 14th International Conference on Managed Languages and Runtimes (ManLang 2017).*

Gilad Bracha and David Ungar. 2004. Mirrors: Design Principles for Meta-Level Facilities of Object-Oriented Programming Languages. *ACM SIGPLAN Notices* 39, 10 (Oct. 2004).

Daniel Ingalls. 2020. The Evolution of Smalltalk: From Smalltalk-72 through Squeak. *Proceedings of the ACM on Programming Languages* 4, HOPL, Article 85 (June 2020).

Daniel Ingalls, Eliot Miranda, Clément Béra, and Elisa Gonzalez Boix. 2020. Two Decades of Live Coding and Debugging of Virtual Machines through Simulation. *Software: Practice and Experience* 50, 9.

Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. 2023. Technical Dimensions of Programming Systems. *The Art, Science, and Engineering of Programming* 7, 3 (Feb. 2023).

Matteo Marra, Guillermo Polito, and Elisa Gonzalez Boix. 2018. Out-Of-Place Debugging: A Debugging Architecture to Reduce Debugging Interference. *The Art, Science, and Engineering of Programming* 3, 2 (Nov 2018).

Toni Mattis, Patrick Rein, and Robert Hirschfeld. 2017. Edit Transactions: Dynamically Scoped Change Sets for Controlled Updates in Live Programming. *The Art, Science, and Engineering of Programming* 1, 2 (1 April 2017).

Javier Pimás and Stefan Marr. 2017. Metaphysics: Towards a Robust Framework for Remotely Working with Potentially Broken Objects and Runtimes. *Proceedings of the 2nd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection (Meta '17).*

Javier E. Pimás, Stefan Marr, and Diego Garbervetsky. 2023. Live Objects All The Way Down: Removing the Barriers between Applications and Virtual Machines. *The Art, Science, and Engineering of Programming* 8, 2 (Oct. 2023).

Guillermo Polito, Stéphane Ducasse, Noury Bouraqadi, Luc Fabresse, and Max Mattone. 2015. Virtualization Support for Dynamic Core Library Update. *ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2015).*

Mary Beth Rosson and John M. Carroll. 1993. Active Programming Strategies in Reuse. *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP '93).*

Robin Salkeld and Gregor Kiczales. 2013. Interacting with Dead Objects. *ACM SIGPLAN Notices* 48 (Oct. 2013).

David W. Sandberg. 1988. Smalltalk and Exploratory Programming. *SIGPLAN Notices* 23, 10.

Marcel Taeumel and Robert Hirschfeld. 2016. Evolving User Interfaces From Within Self-supporting Programming Environments: Exploring the Project Concept of Squeak/Smalltalk to Bootstrap UIs. *Proceedings of the Programming Experience 2016 Workshop (PX/16).*

Marcel Taeumel, Jens Lincke, Patrick Rein, and Robert Hirschfeld. 2022. A Pattern Language of an Exploratory Programming Workspace. *Design Thinking Research: Achieving Real Innovation*, Christoph Meinel and Larry Leifer, 111–145

Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. *1st International Workshop on Live Programming (LIVE 2013).*

Patrick D. Terry. 1997. Compilers and compiler generators: an introduction with C++.

David Ungar, Adam Spitz, and Alex Ausch. 2005. Constructing a Metacircular Virtual Machine in an Exploratory Programming Environment. *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05).*

Tom Van Cutsem and Mark S. Miller. 2010. Proxies: Design Principles For Robust Object-Oriented Intercession APIs. *ACM SIGPLAN Notices* 45, 12 (Oct. 2010).

Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4, Article 30 (Jan. 2013).