Starting with Semantics

Sylvan Clebsch
Microsoft Research, Cambridge
PLISS 2019
How does a computer work?


• How does memory work? Cache coherency? Prefetching? Virtual to physical address translation? Content-addressable memory?

• How does I/O work? PCIe? Programmed I/O? Direct memory access?
It’s a trick question

• Those details are specific to a particular type of computer

• CPU details like this are called microarchitecture, as opposed to the interface visible to the programmer, which is the architecture

• Both systems programmers and compilers for systems languages need to understand the architectural and microarchitectural details

• But what does the systems language itself understand? What machine does the language (as opposed to the compiled code) interact with?
Abstract machines

• An abstract machine is a theoretical model of a computer, expressed as step-by-step execution with architectural details omitted

• Abstract machines may be simplified or detailed, and may be implicit or explicit

• A Turing machine is a simple, explicit abstract machine

• The CLR and JVM are detailed, explicit abstract machines – one way to think about process virtual machines is as executable abstract machines

• C operates over a detailed, implicit abstract machine
Languages and abstract machines

• A programming language describes the steps to take on an abstract machine
• This is true even for non-imperative languages
• Example: the Warren Abstract Machine (WAM) for Prolog
• Example: SECD (stack-environment-control-dump) for strict functional languages
Semantics describe the interface

• There are many ways to describe how a programming language specifies the steps to take on an abstract machine
• The most popular is to write an interpreter or compiler with no specification
• Other approaches include axiomatic semantics, denotational semantics, and operational semantics
• We’ll use operational semantics, but for no reason other than personal preference
Operational semantics

• Big-step operational semantics is a divide-and-conquer approach to calculating the final result of a program – doesn’t handle concurrency

• Small-step operational semantics describes a set of transitions, each expressed as an inference rule, that can be applied non-deterministically

• We’ll use small-step operational semantics, but each has their purpose
Starting with syntax

• Many programming language designs start with syntax
• In some cases, this is because there is no attempt to intentionally design a semantics – perhaps the semantics was assumed
  • C, Perl, Python, Ruby, R, et al
• In some cases, this is because there is an existing semantics that is intentionally targeted
  • C++, F#, Scala, TypeScript, et al
• These are important and useful approaches!
Starting with the syntax makes sense

- Languages are user interfaces
- Allows the designer to focus on expressiveness, change the paradigm (e.g. functional-first on an existing abstract machine), address software engineering issues (modularity, reuse, etc.), and so much more
Why start with semantics?

• New hardware, e.g. shader languages on GPUs, targeting FPGAs
• New communications mechanisms, e.g. distributed programming languages, distributed transaction processing
• New runtime feature, e.g. garbage collection, hot code loading
• New deployment requirements, e.g. cross-platform, embedded devices
• New application domains, e.g. machine learning, reproducible science
• Evolving concrete machine, e.g. non-uniform memory access, non-volatile memory
Why start with semantics? Part 2!

• Safe languages are designed to mitigate specific security flaws
• Mitigations for languages not designed for them are sometimes possible, and sometimes not crippling expensive
• Another route is to design a safe semantics, where safe is always relative to some threat model
• Any language executing with that semantics is then safe, until we figure out what we left out of the threat model
Starting with semantics is harder

• You need to define the abstract machine
• Then you need to define the semantics (interface of language to abstract machine)
• You will still need a syntax
• That syntax will need to target a new and untested semantics
• We can talk about strategies to cope with this later
Let’s build a semantics

• If this is stuff you already know and you want to go faster, please say so
• If this is stuff you don’t feel well grounded in and you want to go slower, please say so
• If you have ideas for how to express things differently, or about something else you’d like to express, please say so
A-normal form

• The operational semantics doesn’t have to be a syntax-driven interpreter
• It can be an intermediate representation between the language and the abstract machine
• Some helpful IR elements are things like single static assignment (SSA), expression holes, continuation passing style (CPS), and A-normal form (ANF)
• We’ll use ANF: we let-bind non-trivial expressions – that is, we use a lot of local variables
\( x, y, z \in \text{LocalID} \)

\( \varphi \in \text{Frame} = \text{LocalID} \rightarrow \text{Value} \)

\( v \in \text{Value} = \text{Integer} \)
If this holds...

...we are allowed to do this
Frame, Expression $\leadsto$ Frame, Expression$|$Value

$\varphi, e \leadsto \varphi', e|v$
\[
\frac{x \in \text{dom}(\varphi)}{\varphi, x \rightsquigarrow \varphi, \varphi(x)}
\]
\[ \varphi, x = v \rightsquigarrow \varphi[x \rightarrow v], v \]
\[
\begin{align*}
  y & \in \text{dom}(\varphi) \\
  v & = \varphi(y) \\
  \varphi, x = y \leadsto \varphi[x \mapsto v], v
\end{align*}
\]
\[ x \in \text{dom}(\varphi) \]
\[ y \in \text{dom}(\varphi) \]
\[ v = \varphi(x) + \varphi(y) \]
\[ \varphi, z = x + y \leadsto \varphi[z \mapsto v], v \]
\[
\varphi, e_1 \leadsto \varphi', e_3 \quad \frac{\varphi, e_1; e_2 \leadsto \varphi', e_3; e_2}{\varphi, e_1; e_2 \leadsto \varphi', e_3; e_2}
\]

\[
\varphi, e_1 \leadsto \varphi', \nu \quad \frac{\varphi, e_1 \leadsto \varphi', \nu}{\varphi, e_1; e_2 \leadsto \varphi', e_2}
\]
\[ x \in \text{dom}(\varphi) \]
\[ \varphi(x) \neq 0 \]

\[ \therefore \varphi, \text{if}(x)\{e_1\}\text{else}\{e_2\} \rightsquigarrow \varphi, e_1 \]
\[ x \in \text{dom}(\varphi) \]
\[ \varphi(x) = 0 \]
\[ \varphi, \text{if}(x)\{e_1\}\text{else}\{e_2\} \leadsto \varphi, e_2 \]
$\varphi, \text{if}(\ast)\{e_1\}\text{else}\{e_2\} \leadsto \varphi, e_1$

$\varphi, \text{if}(\ast)\{e_1\}\text{else}\{e_2\} \leadsto \varphi, e_2$
\[ x \in \text{dom}(\varphi) \]

\[ \varphi(x) \neq 0 \]

\[ \varphi, \text{while}(x)\{e\} \leadsto \varphi, e; \text{while}(x)\{e\} \]
\[ x \in \text{dom}(\varphi) \]
\[ \varphi(x) = 0 \]
\[ \varphi, \text{while}(x)\{e\} \leadsto \varphi, 0 \]
A simple sequential semantics

- We can extend this with function calls, turning the frame into a stack of frames
- And add a heap, and allocation on the heap
- We can add manual free, or reference counting, or a non-deterministic garbage collector
- We can extend values to include addresses on the heap
- And add structured data with fields
- And extend that structured data with methods, dynamic dispatch, and dynamic type information (objects)
More complex sequential semantics?

• Tail call optimisation is just that – an implementation optimisation over the stack semantics
• Pattern matching, higher order functions, and call-by-need (lazy evaluation) can be expressed in terms of the simple sequential semantics
• So can multiple-dispatch, logic programming, exception handling, and more
System level sequential semantics

• Vectorisation (SIMD) and non-strict floating-point maths are relatively straight-forward extensions

• Interrupt handling and dynamic linkage are tricky, and system specific, such that finding the right abstract machine representation is hard

• Interrupt handling is not parallel, but it is asynchronous
A parallel semantics

- The standard abstract machine is a parallel random-access machine (PRAM)
- Add multiple stacks (i.e. threads) and non-deterministically select which stack to execute next
- Keep a single shared heap
- Add a collection of atomic operations, expressed as inference rules that complete the operation in a single small-step
- Small-step operational semantics expresses concurrency by using non-determinism to model parallelism
Concurrent = Two Queues One Coffee Machine

Parallel = Two Queues Two Coffee Machines
\[ \begin{align*}
\nu & \in \text{Value} = \text{Integer} \mid \text{Address} \\
\chi & \in \text{Heap} = \text{Address} \rightarrow \text{Object} \\
\sigma & \in \text{Stack} = \text{Frame} \\
\theta & \in \text{Thread} = \text{Stack} \times \text{Expression}
\end{align*} \]
\[ \chi, \sigma \cdot \varphi, e \sim \chi', \sigma \cdot \varphi', e|v \]
\[
\begin{align*}
\chi, \sigma, e \sim & \sim \chi', \sigma', e' \\
\chi, \bar{\theta} \cdot (\sigma, e) \cdot \bar{\theta}' \sim & \sim \chi', \bar{\theta} \cdot (\sigma', e') \cdot \bar{\theta}' \\
\chi, \sigma, e \sim & \sim \chi', \sigma', v \\
\chi, \bar{\theta} \cdot (\sigma, e) \cdot \bar{\theta}' \sim & \sim \chi', \bar{\theta} \cdot \bar{\theta}'
\end{align*}
\]
Memory ordering

• This approach results in a semantics for parallelism that is sequentially consistent – all reads and writes from all threads are locally in-order and globally interleaved in some total order

• There are many forms of more relaxed memory ordering on real CPUs, which are critical for performance – weak memory semantics is an active research area

• Even without relaxed memory orderings, concurrency in the PRAM model allows concurrent mutation (data races)
A use case for starting with semantics

• We can express an operational semantics that has mutation, is parallel, is efficiently implementable, but has no concurrent mutation

• This is an example of the abstract machine allowing less than the concrete machine

• Restricting the abstract machine can improve reasoning, safety, and performance

• The implicit C abstract machine, for example, has a flat, undifferentiated address space – restricting this is an active research area
Why remove concurrent mutation?

• Very roughly speaking, the top safety issues in systems programming languages, in order:
  • Spatial memory safety
  • Temporal memory safety
  • Data races
• If concurrent mutation isn’t necessary for your problem domain, removing it can improve safety and performance
• There are many approaches, from transactional memory to linear types – this is just a point in the design space
Building a PRAM without data-races

• In the PRAM model, there is a single shared heap
• We can instead associate a heap with each thread
• This eliminates data-races completely – each thread can only read or write its own heap and its own stack
• It also removes the only cross-thread communication mechanism, which was observing concurrent mutation in the heap
Adding data-race free communication

• We can associate a mailbox with each thread
• The mailbox can allow concurrent push but only allow the associated thread to pop – a multi-producer single-consumer (MPSC) queue
• If mailbox messages must be primitive values, we have data-race free communication
• But we want safe, efficient communication of structured data, including entire object graphs
Object messaging without data-races

- We could prevent the receiving thread from making progress and copy the message graph into the receiving heap – slow but effective (Erlang)
- We can ensure that the message graph does not reach any object reachable by the sending thread
- This is a form of separation logic – safe and fast messaging, but only if the reachability test is cheap
Simple separation logic encoded in semantics

- Use a single heap, segmented into regions, where every object belongs to one region and is reachable only from that region
- Allow objects in some region \( r \) to reference some other region \( r' \) but not the objects within \( r' \)
- Reference count regions
- Messages may then contain only regions with a zero reference count, and never object references
- This allows moving a region from one thread to another but not sharing a region, preventing concurrent mutation
Weakening the semantics

• This simple encoding of separation logic can be drastically improved, including by moving some elements from the dynamic domain (operational semantics) to the static (type checking)
• This can be allowed in the operational semantics by not specifying the way separation is enforced
• Instead, inference rules can have complex preconditions that may be enforced statically rather than dynamically
Another use case: memory constrained ML

• The other ML: machine learning
• Specifying vector operations needed for efficient ML is simple
• Less simple is scheduling parallel pipeline stage executions in a constrained memory environment
• Particularly for ML pipelines that are dynamically data dependent: recursive, looping, etc.
Key takeaways

• Starting with semantics means building a language to do something different rather than building a language to do something better

• Detailed, explicit small-step operational semantics are surprisingly easy to get right if they are there from the beginning

• A new semantics can free you from problems arising from your abstract machine that are due to features you don’t need for your problem domain