Small-Step Operational Semantics

1 Background and Motivation

In order to implement a language compiler or interpreter, it is first necessary to know exactly how a language is supposed to work. While syntax tells us the particular symbols which a language is made up of, it is up to a language's *semantics* to apply meaning to these symbols. The equation $2 + 3$ is just a bunch of characters put together, but the knowledge that this evaluates to 5 speaks to the semantics of this expression.

1.1 Semantics can be Unintuitive

While the semantic meaning behind language symbols may occasionally seem obvious, frequently these can be problematic. For example, consider the following snippet of C code, with an embedded question:

```
int main(int argc, char** argv);
void foo(int x, int y);

void foo(int x, int y) { ... } // definition elided

int main(int argc, char** argv) {
  int x = 7;
  // QUESTION: what values will foo be called with?
  foo((x = 8), x);
  return 0;
}
```

Actually try to answer the question: what values will `foo` be called with? Recall that the assignment operation (`=`) will assign to the given variable, and then return the new value of the variable (which requires us to understand the semantics of C in the first place!). You may be tempted to run this code to see what happens, allowing you to actually see the parameters passed to `foo`. Surely this should answer the question, right?

As it turns out, according to the C standard, this would not really answer anything. According to the standard, **either** `8, 8` **or** `8, 7` are both possible values that can be passed along to `foo`. The underlying reason why is because the standard does not enforce the order in which the arguments to function calls are evaluated. The compiler is free to choose which one to pick, and the picks do not necessarily even have to be consistent (it may sometimes choose right to left, others left to right, or even a completely random order with more than two parameters). As such, even running the code is not enough to completely answer the question, because the compiler ultimately must make a choice. In this way, the semantics of something as seemingly simple as function calls in C are not at all straightforward.

These seemingly bizarre C semantics come from an intentional design decision made by the C standards committee. While these semantics are not straightforward for a C programmer to understand, the goal with this choice was not to simplify the job of the C programmer, but instead to simplify the job of the C compiler writer. For example, on some architectures, there are peformance benefits to evaluating arguments right to left as opposed to left to right. By leaving the standard open to interpretation here, the compiler writer can now make whatever choice is best for the underlying architecture. Additionally, this can open up optimization opportunities.

1.2 English and Natural Language Conveys Meaning Poorly

English and other natural languages are often employed for explaining programming language semantics, but this leads to its own set of problems. For example, C is often touted as being a simple language, and is backed by an English-based standard. However, this standard is currently over 700 pages long, and it shows no sign of getting any shorter. Moreover, discussions between compiler writers and the standards committee reveal that there are a multitude of points where the standard is unclear. For these reasons, writing a C compiler is quite a difficult task, as merely understanding what the language us *supposed* to do is non-trivial.

I have firsthand experience with the problem of using English to describe semantics, along with the repercussions (for the curious, see `https://github.com/Z3Prover/z3/issues/68`, particularly starting at `https://github.com/Z3Prover/z3/issues/68#issuecomment-98777443`). The short version of this story is that the word "underspecified" was ambiguous in the context it was applied, leading to *two* possible, but nonetheless incompatible, implementations. This revealed itself as a nasty bug in a widely-used tool. In the end, it was necessary to contact the standards committee for clarification before any action could be taken, because it was not clear what the correct behavior was.

Ultimately English suffers from being too verbose. Ironically, English also tends to lack descriptive power, leading to imprecision.

1.3 Definitional Interpreters are Problematic

Beyond English, another popular approach for defining language semantics is to write a *definitional interpreter* for a given language, as is done for Python, Ruby, and Perl. A definitional interpreter is an executable interpreter for a given language. The reasion it is called "definitional" is because the interpreter's behavior is defined to be the same as the behavior of **the underlying language**. For example, the definitive way to know what a Python program should do is to run it on the `cpython` interpreter (the default Python distribution), as `cpython` ultimately defines what Python (the language) is supposed to do.

Definitional interpreters offer a simplistic approach to defining language semantics, and they even give us a workable language interpreter in the process. However, this suffers from a fundamental problem. Consider the following example interaction with a Python shell, where the prompt is indicated by `>>`, and everything after the prompt was written by the programmer:

```
>> 2 + 2
5
>> 5 + 27
Segmentation Fault
```

Intuitively, we know that `2 + 2` should be `4`, and that `5 + 27` should not lead to a segmentation fault. However, by the very definition of what it means to be a definitional interpreter, these are not bugs, but **features**! By construction, a definitional interpreter defines what the language itself should do under a given input, which means that, also by construction, the interpreter does not contain any bugs. Anything that looks like a bug is, by definition, just something unintuitive but nonetheless correct.

Of course, we can still report these sort of issues to developers, but ultimately its up to the developers to decide if this needs attention or not. This, effectively, ends up being a communication with a standards committee, phrased as a bug report. This is hardly ideal.

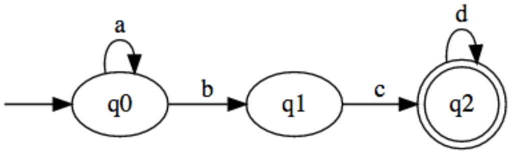
2 Math to the Rescue

As we did for type systems, here again we will appeal to mathematics to define language semantics. While the definitions may look scary and need lots of time to understand, they will be concise and unambiguous. With that in mind, consider what you would need to read if either English or a definitional interpreter approach had been used, and how long it would take you to read it.

2.1 Automata-Based Approach

The particular area of mathematics we will appeal to here is based on automata. You already have seen deterministic finite automata for handling regular expressions. As a quick refresher, consider the following regular expression, along with the automata that accepts it:

`a*bcd*`



While the above automata is small, it ends up capturing a lot of behavior, and ends up accepting an infinite set of sentences.

We will take a similar automata-based approach for defining language semantics. However, there is a problem: program variables can take on infinite numbers of values in general, and programs do not necessarily terminate. For these reasons, finite automata are not the appropriate basis for formulating program semantics.

A seemingly appealing automata-based replacement is that of Turing machines, particularly given the fact that they are used as a theoretical model of computation. Turing machines overcome the infinite data problem by introducing an auxilliary tape of infinite length, along with a finite automata describing program behavior. While this works, the end result tends to be burdensome and verbose, which is unappealing in the context of clearly defining language semantic behavior. As such, these are a non-starter for our purposes.

Instead, we will take a different approach to solving this problem. Rather than introducing an auxilliary component of infinite size, we will put the infinite component into the automata itself, allowing for an *infinite* number of states. While we thus loose the capability to write the automata out in general (as we would need an infinite amount of paper), this representation still has utility. In particular, with this infinite state representation, we tend to deemphasize exact what the automata *is*, but instead focus on *how it is being constructed* and what it is *doing*.

2.2 Automata Components

When using infinite state automata for modeling small-step semantics, there are two major components:

1. The definition of exactly what a **state** is
2. The definition of how we **transition between states**

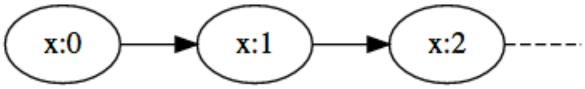
The state encapsulates all stateful parts of a program. For example, with C, this includes explicit things like variables in scope, the values of these variables, the contents of memory. Notably, this also includes more implicit things, such as where we are in the program (e.g., the line number, the call stack, and the actual code we are executing).

As for state transitions, these are defined via a *transition function*. The transition function takes an input state and produces an output state, and encodes the language components which do not change between programs. For example, with C, the transition function ultimately encodes the bulk of the semantics of C, as the transition function describes how program execution proceeds.

For a high-level example of how the state and the transition function interact, consider the following C snippet:

```
int x = 0;
while (1) {
  x++;
}
```

A graphical representation of a corresponding infinite state automaton for this program is as follows, showing only the variables:



As shown, the states encode the values of the variables in play (in reality a state holds additional information like where we are in the program, but this has been elided for clarity). The transitions between states effectively encode the semantics of both increment (`++`) and the `while` loop: the `++` operation ends up increasing the value of the variable `x`, yielding a new state. We keep yielding successor states in this fashion thanks to the behavior of `while`.

There is one component missing from the above automaton: an *initial state*. Because execution proceeds via repeated applications of the transition function, we must have some state to start with. A possible representation of the initial state in the above diagram would be an initial state wherein the variable `x` has an indeterminate value of some sort (e.g., some equivalent of `null`).

Altogether, this style of representing program semantics is known as *small-step operational semantics*. We say "small-step", because execution proceeds one complete step at a time, via multiple applications of the transition function. This is in contrast to another style called "big-step", which effectively goes from the initial state to a final result in one big jump. We say "operational" because we will model the transitions via an *abstract machine*, which ultimately defines execution rules. (Incidentally, a Turing machine is just another kind of abstract machine.)

3 Example: A Subset of Idealized MIPS Assembly

These concepts are best illustrated via an example. For these purposes, we will look at a subset of an idealized version of MIPS assembly. MIPS assembly serves as a good case study here, because individual instructions are intentionally as simple as possible. As such, we can model their semantic behaviors from a language standpoint without too much trouble.

To be clear, this is merely an *idealized* subset of MIPS assembly, and does not accurrately reflect how MIPS assembly works. In particular, this subset has the following limitations:

1. The vast majority of instructions are not specified
2. Instructions themselves are represented with syntactic forms, as opposed to being represented by various bit encodings
3. Memory is represented as an array of instructions, and instructions which manipulate memory are notably absent

These simplifications exist to make the semantics as simple as possible without deviating from the general behavior of MIPS assembly, with the ultimate goal of making the example easier to understand. We **could** define a much more realistic semantics of MIPS assembly, but this would be so complex as to be a research problem.

3.1 Instructions

We will model the following MIPS instructions. Brief English descriptions of what these instructions are supposed to do have been provided as a refresher.

- `addu` (Add-unsigned): Adds the values of two specified registers together, placing the result in a third specified register
- `beq` (Branch-if-equal): Jumps to a particular instruction if the values of two specified registers are equal
- `bne` (Branch-if-not-equal): Jumps to a particular instruction if the values of two specified registers are **not** equal
- `sltu` (Set-less-than-unsigned): Compare the results of two specified registers. If the first is less than the second, put a `1` in the specified destination register, else put a `0` in the specified destination register
- `li` (Load-immediate): Load a specified value into a specified register
- `halt` (Halt): Not actually a real MIPS instruction; stop program execution

A formalization of the syntax behind these instructions follows:

$$n \in \mathbb{N} \qquad v \in \mathit{BinaryValue}$$

$$\begin{aligned}
t \in \mathit{TempRegisterId} &::= \mathsf{\textbf{t}}_n \\
r \in \mathit{RegisterId} &::= \mathsf{\textbf{zero}} \mid t \\
a \in \mathit{Address} &::= n \\
i \in \mathit{Instruction} &::= \mathsf{\textbf{addu}}\ r_1\ r_2\ r_3 \mid \mathsf{\textbf{beq}}\ r_1\ r_2\ a \mid \mathsf{\textbf{bne}}\ r_1\ r_2\ a \mid \mathsf{\textbf{sltu}}\ r_1\ r_2\ r_3 \mid \mathsf{\textbf{li}}\ r\ v \mid \mathsf{\textbf{halt}}
\end{aligned}$$

3.2 State

The state for our idealized MIPS assembly will consist of three components:

1. An instruction pointer, indicating which instruction we are currently executing
2. A register file, indicating the values of various registers
3. Memory, holding an array of instructions to execute

A formalization of this state follows. Note that this reuses multiple components from the syntactic definition, reflecting the fact that we need the program text while we execute the program.

$$\begin{aligned}
\varsigma \in \mathit{State} &= \mathit{RegisterFile} \times \mathit{InstructionPointer} \times \mathit{Memory} \\
f \in \mathit{RegisterFile} &= \mathit{TempRegisterId} \rightarrow \mathit{BinaryValue} \\
ip \in \mathit{InstructionPointer} &= \mathit{Address} \\
mem \in \mathit{Memory} &= \mathit{Address} \rightarrow \mathit{Instruction}
\end{aligned}$$

3.3 Helper Functions

In addition to the state and the transition function (which will be defined shortly), we will also employ a number of helper functions. These helper functions will be used within the transition function to help abstract commonly-used routines. This will have the nice side benefit of cleaning up the transition function, making it look as streamlined as possible.

3.3.1 Register File Update

This helper will be used to update a register in the given register file with a particular value. This ends up being a little tricky thanks to the fact that updates to register **zero** do nothing. We use the notation $f[\mathbf{t}_n \mapsto v]$ to indicate updating map f to include a binding for the key $\mathbf{t}_n$ with the value v, yielding a new map in the process (as with updating a key in an immutable `Map` in Scala).

$\texttt{update} \in \mathit{RegisterFile} \times \mathit{RegisterId} \times \mathit{BinaryValue} \rightarrow \mathit{RegisterFile}$

$\texttt{update}(f, \textbf{zero}, v) = f$

$\texttt{update}(f, \mathbf{t}_n, v) = f[\mathbf{t}_n \mapsto v]$

3.3.2 Register File Lookup

This helper is used to get a particular register's value in the given register file. This is slightly non-trivial, as register **zero** must always return the binary value `0`. We use the notation $f(\mathbf{t}_n)$ to indicate retrieving the corresponding value for the key $\mathbf{t}_n$ in the map f.

$\texttt{lookup} \in \mathit{RegisterFile} \times \mathit{RegisterId} \rightarrow \mathit{BinaryValue}$

$\texttt{lookup}(f, \textbf{zero}) = 0$

$\texttt{lookup}(f, \mathbf{t}_n) = f(\mathbf{t}_n)$

3.3.3 Initial State

This helper is used to produce the initial program state, given some list of instructions to execute. It will call the `initialStateHelper` function in the process (which is subsequently defined).

$\texttt{initialState} \in \overrightarrow{\mathit{Instruction}} \rightarrow \mathit{State}$

$\texttt{initialState}(\vec{i}) =$

$$([\mathbf{t}_0 \mapsto 0, \mathbf{t}_1 \mapsto 0, ..., \mathbf{t}_n \mapsto 0], 0, \texttt{initialStateHelper}(\vec{i}, 0, []))$$

3.3.4 Initial State Helper

This helper is used by `initialState`, during the process of creating an initial program state.

$\texttt{initialStateHelper} \in \overrightarrow{\mathit{Instruction}} \times \mathbb{N} \times \mathit{Memory} \rightarrow \mathit{Memory}$

$\texttt{initialStateHelper}(\vec{i}, n, \mathit{mem}) =$

$$\begin{cases} \texttt{initialStateHelper}(\vec{i'}, n+1, \mathit{mem}[n \mapsto i]) & \texttt{if } \vec{i} = i :: \vec{i'} \\ \mathit{mem} & \text{otherwise} \end{cases}$$

3.3.5 Bitwise Addition

The $\hat{+}$ operation performs modular binary addition on two binary values, yielding a new binary value. This will generally be written with infix notation (e.g., $v_1 \hat{+} v_2$, as opposed to $\hat{+}(v_1, v_2)$). Given the straightforward definition, we will not bother to formally specify what this means.

$\hat{+} \in \mathit{BinaryValue} \times \mathit{BinaryValue} \rightarrow \mathit{BinaryValue}$

3.3.6 Bitwise Less-Than

The $\hat{<}$ operation returns **true** if the first value is less than the second value, else **false**. Given the straightforward definition, we will not bother to formally specify what this means.

$\hat{<} \in \mathit{BinaryValue} \times \mathit{BinaryValue} \rightarrow \mathit{Boolean}$

3.4 Transition Function

The transition function moves from an input state to an output state. It bears the following type signature:

$\mathcal{F} \in \mathit{State} \rightarrow \mathit{State}$

To clean things up as much as possible, we will represent this function in a table-based format. Each row of the table implements one particular kind of behavior, which tends to be based on the particular instruction involved. Each column shows one of the following:

- A component of the input state
- Arbitrary premises which must hold
- A value to produce for the output state, assuming the input state values match up with what was expected in the row, along with the arbitrary premises

We will also add a column named "#", indicating the number of the rule. This column is so we can easily refer to a particular rule (e.g., "rule 6 shows..."), and is purely for expository reasons.

Because the memory never changes, we will not bother to show the contents of memory in the table (phrased another way, *mem* behaves as a global immutable variable once execution begins). Additionally, because we will always execute the instruction that the instruction pointer points to, we show one of the columns in this table as $\mathit{mem}(\mathit{ip})$, that is, lookup the instruction in memory that the instruction pointer points to, and match on it. Without further ado, the table is below:

#	$\mathit{mem}(\mathit{ip})$	Premises	f_{new}	$\mathit{ip}_{\mathit{new}}$
1	**addu** r_1 r_2 r_3	$v = \texttt{lookup}(f, r_2) \hat{+} \texttt{lookup}(f, r_3)$	$\texttt{update}(f, r_1, v)$	$\mathit{ip} + 1$
2	**beq** r_1 r_2 a	$\texttt{lookup}(f, r_1) = \texttt{lookup}(f, r_2)$	f	a
3	**beq** r_1 r_2 a	$\texttt{lookup}(f, r_1) \neq \texttt{lookup}(f, r_2)$	f	$\mathit{ip} + 1$
4	**bne** r_1 r_2 a	$\texttt{lookup}(f, r_1) = \texttt{lookup}(f, r_2)$	f	$\mathit{ip} + 1$
5	**bne** r_1 r_2 a	$\texttt{lookup}(f, r_1) \neq \texttt{lookup}(f, r_2)$	f	a
6	**sltu** r_1 r_2 r_3	$\texttt{lookup}(f, r_2) \hat{<} \texttt{lookup}(f, r_3)$	$\texttt{update}(f, r_1, 1)$	$\mathit{ip} + 1$
7	**sltu** r_1 r_2 r_3	$\neg(\texttt{lookup}(f, r_2) \hat{<} \texttt{lookup}(f, r_3))$	$\texttt{update}(f, r_1, 0)$	$\mathit{ip} + 1$
8	**li** r v		$\texttt{update}(f, r, v)$	$\mathit{ip} + 1$
9	**halt**		f	ip

An English explanation of rule #1 follows. If the instruction at $\mathit{mem}(\mathit{ip})$ is an `addu`, derive the value v by looking up (via the `lookup` helper) the values of registers r_2 and r_3 in the register file f, and then add the two resulting values together via the $\hat{+}$ helper. Then update the register file (via the `update` helper) so that register r_1 maps to value v, yielding a new register file, f_{new} (which will be the input register file for the next state). Finally, update the program counter to point to the next instruction in memory.

As for rule #9 (which handles `halt`), note that the instruction does not update the instruction pointer (ip). As a result, the next instruction we will read upon ever encountering `halt` is the same `halt` instruction. This mechanism of representing normal program termination by trivially looping forever is standard in small-step operational semantics.