lec4: Soundness and completeness; type soundness

Jana Dunfield

January 21, 2022

1 Soundness and completeness

Soundness and completeness are tools for describing the relationship between logical theories (also called systems, theories, logical systems, logics, or semantics). Since they describe relationships between theories, at least two theories must be involved: it doesn't make sense to say that big-step evaluation is sound, unless we say what it is sound *with respect to*.

Informally, we might say that a theory is *sound*, in order to say that the theory is sound with respect to our (informal) understanding. For our theory of big-step evaluation, that means we expect that it gives results (values v in $e \Downarrow v$) that are consistent with our mental model of integer addition ("MMIA"), we could conjecture that big-step semantics is sound with respect to MMIA:

Conjecture 1. *For all expressions e and integers n, if $e \Downarrow n$ then $e = n$ in MMIA (interpreting `(+` e_1 e_2`)` as adding e_1 and e_2).*

Attempting to prove this conjecture may seem questionable, since the proof would have to refer to a mental model of something rather than a formal theory. However, believing *any* mathematical proof depends on a mental model.

In any case, we will focus on the soundness and completeness of different formal theories with respect to each other, rather than with respect to to an informal one. Given two theories, if we "believe" one more than the other—because it is simpler, or because we have more experience using it, or because "everyone" knows it—we designate that theory as *ground truth*. Then we can test the other theory, the one we believe less, by asking whether it is sound with respect to our chosen ground truth.

If we take big-step evaluation as ground truth, we can ask if our system of small-step rules (the small-step theory or small-step semantics) is sound with respect to big-step. For example, if `(+ 1 2)` $\mapsto^* 3$ then we would expect `(+ 1 2)` $\Downarrow 3$.

Conjecture 2 (Soundness of small-step with respect to big-step).
For all expressions e and values v, if $e \mapsto^* v$ then $e \Downarrow v$.

We need to use the zero-or-more-steps judgment $\mapsto^*$ and not $\mapsto$, because it is not the case that "If $e \mapsto e'$ then $e \Downarrow e'$." That is because e' is not always a value, for example, when e is `(+ (+ 1 2) 10)`.

Soundness tells us that a theory (system of rules) does not show wrong things. If we define a silly big-step semantics that has only one rule—which doesn't even have meta-variables—we will have a semantics that "works" for exactly one expression:

$\boxed{e \Downarrow_2 v}$ expression e big-step evaluates to v, but in a silly way

$$\frac{}{\texttt{(+ 2 2)} \Downarrow_2 4}\ \text{twoeval-twoplustwo}$$

This semantics only gives an answer (a value) for one expression, `(+ 2 2)`, but since it gives a correct answer—an answer consistent with our big-step semantics—it is sound:

Theorem 1 (Soundness of silly big-step with respect to big-step).
For all expressions e *and values* v*, if* $e \Downarrow_2 v$ *then* $e \Downarrow v$*.*

Proof. Since the $\Downarrow_2$ system has only one rule, we know that the concluding rule of the derivation of $e \Downarrow_2 v$ is twoeval-twoplustwo.

By inversion on twoeval-twoplustwo, $e = (+\ 2\ 2)$ and $v = 4$.

By rule eval-const, $2 \Downarrow 2$.

By rule eval-add (with $e_1 = 2$ and $e_2 = 2$), we have $(+\ 2\ 2) \Downarrow 4$, which is $e \Downarrow v$. □

Completeness is the converse of soundness:

Conjecture 3 (Completeness of silly big-step with respect to big-step).
For all expressions e *and values* v*, if* $e \Downarrow v$ *then* $e \Downarrow_2 v$*.*

This conjecture claims that if we know something using the big-step theory, we also know it using the silly big-step theory. However, the silly big-step theory only works when $e = (+\ 2\ 2)$, so we can disprove the conjecture by giving a counterexample: Let $e = (+\ 1\ 1)$ and $v = 2$.

If we (unwisely) took the silly big-step semantics as ground truth, we could state soundness and completeness of big-step with respect to silly big-step:

Conjecture 4 (Soundness of big-step with respect to silly big-step).
For all expressions e *and values* v*, if* $e \Downarrow v$ *then* $e \Downarrow_2 v$*.*

Theorem 2 (Completeness of big-step with respect to silly big-step).
For all expressions e *and values* v*, if* $e \Downarrow_2 v$ *then* $e \Downarrow v$*.*

These statements are *identical* to the earlier two statements of soundness and completeness, but swapped! Soundness of big-step with respect to silly big-step is the same as completeness of silly big-step with respect to big-step (so it is disproved by our counterexample): this is the claim that if big-step semantics says something, the silly big-step system says it too. Completeness of big-step with respect to silly big-step is the same as soundness of silly big-step with respect to big-step, which we proved. This says that if the silly big-step semantics gives an answer, the big-step semantics gives the same answer.

An even sillier semantics would have no rules at all—but it would be sound with respect to big-step semantics, and indeed every other theory. However, it would be complete with respect to no other theories. (It would be complete with respect to itself, but the point of completeness and soundness is to compare two theories; every theory is sound with respect to itself, and complete with respect to itself, because "if X then X" is always valid reasoning in a proof.)

Apart from their intrinsic interest, soundness (and completeness) results provide leverage: If we have proved a result like determinacy in big-step semantics, and we also prove that another system—say, small-step semantics—is sound with respect to big-step, then the determinacy result carries over:

Theorem 3 (Determinacy of big-step).
For all expressions e *and values* v_1 *and* v_2*,*
if $e \Downarrow v_1$ *and* $e \Downarrow v_2$
then $v_1 = v_2$*.*

Conjecture 5 (Determinacy of small-step).
For all expressions e *and values* v_1 *and* v_2,
if $e \mapsto^* v_1$ *and* $e \mapsto^* v_2$
then $v_1 = v_2$.

Proof. **(Assuming Conjecture 2—soundness of small-step with respect to big-step—which we haven't proved yet!)**

$e \mapsto^* v_1$	Given
$e \Downarrow v_1$	By Conjecture 2
$e \mapsto^* v_2$	Given
$e \Downarrow v_2$	By Conjecture 2
$v_1 = v_2$	By Theorem 3

□

If we also proved completeness of small-step with respect to big-step, we could leverage results that are easier to prove for the small-step semantics.

2 Excursion: CompCert

If the above seems rather theoretical, consider a more practical question. Outside of extremely low-level settings (some hardware drivers, embedded systems, parts of OS kernels), virtually no one writes programs in assembly language. Instead, we write in "high-level" languages C, which—while verbose, unsafe, rather low-level, and generally unpleasant to use—is much easier to use than assembly.

By not programming in assembly, we rely on (say) a C compiler turning something like

```
int m, n;
...
m = n - 1;
```

into assembly code that reads the value of `n`, subtracts one, and puts the result into `m`—handling all the situations in which `m` and `n` are stored in CPU registers, or in memory, or some combination of those. This is a very simple example; actual C compilers must handle function calls, arrays, optimizations, conversions between different sizes of integers, and so on. Real compilers also implement a variety of optimizations, to improve the performance of the generated assembly code.

How do we know that the compiler generates assembly code that corresponds to the program we wrote? For nearly all compilers, we don't know. Compiler developers do run their compilers with extensive test suites, but compilers can be extremely complex.

An exception, where we do know, is the CompCert C compiler. It has been proved to be sound: given a C program, the assembly it generates has the same meaning. To prove this, the designers of CompCert had to define a formal semantics of x86 assembly, a formal semantics of C, and formal semantics for the intermediate languages. If you are not familiar with compilers, compilers gradually transform source programs into assembly by way of increasingly lower-level "intermediate languages". Then they had to prove that, at each transition from a higher-level language to a lower-level one, the transition is sound: it preserves meaning.

3 Typing

A [static] *type system* keeps out sort-of-nonsense:

```
(+ "no" 1)
```

Like small-step and big-step operational semantics, type systems can be defined by rules.

$$\frac{}{n : \mathsf{int}}\ \text{type-const} \qquad \frac{e_1 : \mathsf{int} \qquad e_2 : \mathsf{int}}{(+\ e_1\ e_2) : \mathsf{int}}\ \text{type-add}$$

This rule says that if e_1 is an integer, and e_2 is an integer, then their sum is an integer.

This is not a terribly interesting type system: every possible expression has the same type, int.

Exercise 1*. State and prove (by induction) that every expression e has type int.

Type systems become more interesting with more than one type.

3.1 Type soundness

Conjecture 6 (Type soundness with respect to big-step).
For all expressions e*, values* v *and types* A*,*
if $e : A$ *and* $e \Downarrow v$*,*
then $v : A$*.*

3.2 Totality

Type soundness is conditional: *if* e evaluates to v, then v has the same type as e. In our current tiny language with only integer addition, the condition is always satisfied:

Conjecture 7 (Totality).
For all expressions e*,*
there exists v *such that* $e \Downarrow v$*.*

The name "totality" comes from functions in mathematics: a total function is defined on all inputs in the function's domain, while a partial function is not. For example, addition on the integers is total, but division is not:

1. Divisions that would produce non-integers are not defined: 1 and 2 are integers, but 1/2 is not.
2. Division by zero is not defined: $1/0$ is not defined; even $0/0$ is not.

The first "undefinedness" could be solved fairly easily by defining $(/\ e_1\ e_2)$ to produce $\lfloor \frac{n_1}{n_2} \rfloor$, where $\lfloor - \rfloor$ rounds down to the nearest integer. It could also be solved by changing our language to have rational numbers instead of integers (we already assume arbitrary-precision integers, sometimes called "bignums", which are not supported directly in hardware).

But the second "undefinedness" is harder to solve. The only reasonable interpretation of a rule

$$\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{(/\ e_1\ e_2) \Downarrow \lfloor n_1 / n_2 \rfloor}$$

would be to mentally add a third premise specifying that n_2 is not zero, because then n_1/n_2 is defined.

$$\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2 \qquad n_2 \neq 0}{(\texttt{/}\ e_1\ e_2) \Downarrow \lfloor n_1/n_2 \rfloor}$$

This leaves the situation when n_2 *is* zero. One approach would be to say that division by zero produces zero (or some other arbitrary number), but that would destroy the algebraic properties of arithmetic in our language. A better, but more difficult approach is to use a type system to check that the type of e_2 is NonzeroInt, or something like that. Such type systems exist, but at this point in the course we don't have the tools to understand how they work.

The most common approach, however, is to expand the semantics to allow exceptional states or errors. We can do this by introducing another judgment form, e error:

$\boxed{e\ \mathsf{error}}$ evaluating expression e is an error

$$\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow 0}{(\texttt{/}\ e_1\ e_2)\ \mathsf{error}}\ \text{error-divzero}$$

which can also be read "e crashes". However, it is possible for a semantics to express error recovery, such as exception handling; then an error does not necessarily lead to a crash.

Exercise 2. The above rule for e error is necessary but not sufficient. For example, the judgment `(+ (/ 1 0) 5)` error is not derivable. How would you fix that?

(Subtraction and multiplication on integers are total functions and could be included without any issues.)