Lecture Notes on Bisimulation

15-814: Types and Programming Languages
Frank Pfenning

Lecture 13
Thursday, October 21, 2021

1 Introduction

In the last lecture we introduced the K Machine as an alternative way to define the dynamics of programs in our language. In this lecture we would like to prove (part of) the correctness of the machine. After this, we show how to add a simple form of exceptions to our language, as a second form of control construct after fixed points.

2 Eager Pairs in the K Machine

The fragment of the machine we pick to illustrate the proof technique is eager pairs. Since no new ideas are required, we just recall the small-step dynamics and then present the corresponding rules for the K machine.

$$\frac{e_1 \ \textit{value} \quad e_2 \ \textit{value}}{\langle e_1, e_2\rangle \ \textit{value}} \ \mathsf{val/pair}$$

$$\frac{v_1 \ \textit{value} \quad v_2 \ \textit{value}}{\mathsf{case}\ \langle v_1, v_2\rangle\ (\langle x_1, x_2\rangle \Rightarrow e_3) \mapsto [v_1/x_1][v_2/x_2]e_3} \ \mathsf{step/casep/pair}$$

$$\frac{e_1 \mapsto e_1'}{\langle e_1, e_2\rangle \mapsto \langle e_1', e_2\rangle} \ \mathsf{step/pair}_1 \qquad \frac{v_1 \ \textit{value} \quad e_2 \mapsto e_2'}{\langle v_1, e_2\rangle \mapsto \langle v_1, e_2'\rangle} \ \mathsf{step/pair}_2$$

$$\frac{e_0 \mapsto e_0'}{\mathsf{case}\ e_0\ (\langle x_1, x_2\rangle \Rightarrow e_3) \mapsto \mathsf{case}\ e_0'\ (\langle x_1, x_2\rangle \Rightarrow e_3)} \ \mathsf{step/casep}_0$$

$$
\begin{array}{rcl}
k \triangleright \langle e_1, e_2\rangle & \mapsto & k \circ \langle \_, e_2\rangle \triangleright e_1 \\
k \circ \langle \_, e_2\rangle \triangleleft v_1 & \mapsto & k \circ \langle v_1, \_\rangle \triangleright e_2 \\
k \circ \langle v_1, \_\rangle \triangleleft v_2 & \mapsto & k \triangleleft \langle v_1, v_2\rangle \\
k \triangleright \mathsf{case}\ e_0\ (\langle x_1, x_2\rangle \Rightarrow e) & \mapsto & k \circ \mathsf{case}\ \_\ (\langle x_1, x_2\rangle \Rightarrow e) \triangleright e_0 \\
k \circ \mathsf{case}\ \_\ (\langle x_1, x_2\rangle \Rightarrow e) \triangleleft \langle v_1, v_2\rangle & \mapsto & k \triangleright [v_1/x_1, v_2/x_2]e
\end{array}
$$

States	s	$::=$	$k \triangleright e$	evaluate e with continuation k
		$\mid$	$k \triangleleft v$	return value v to continuation k
Continuations	k	$::=$	$\epsilon \mid \cdots$	
		$\mid$	$k \circ \langle \_, e_2\rangle \mid k \circ \langle v_1, \_\rangle \mid k \circ (\mathsf{case}\ \_\ (\langle x_1, x_2\rangle \Rightarrow e))$	

3 Correctness of the K Machine

Given the relatively simple construction of the machine it is surprisingly tricky to prove its correctness. We refer to the textbook [Har16, Chapter 28] for a complete formal development. We already stated a central property:

> *For any continuation k, expression e and value v,*
> $k \triangleright e \mapsto^* k \triangleleft v$ *iff* $e \mapsto^* v$

This implies that $k \triangleright v \mapsto^* k \triangleleft v$ because $v \mapsto^0 v$.

A key step in the proof is to find a relation between expressions and machine states $k \triangleright e$ and $k \triangleleft v$. In this case we actually define this relation as a function that folds the state back into an expression. As stated in the property above, the state $k \triangleright e$ expects the value of e being passed to k. When we unravel the state we don't wait for evaluation finish, but we just substitute expression e back into k. Consider, for example,

$$k \triangleright e_1\ e_2 \mapsto k \circ (\_\ e_2) \triangleright e_1$$

If we plug e_1 into the hole of the continuation $(\_\ e_2)$ we recover $e_1\ e_2$, which we can then pass to k.

We write $k(e) = e'$ for the operation of *reconstituting* an expression from the state $k \triangleright e$ or $k \triangleleft e$ (ignoring the additional information that e is a value in the second case). We define this inductively over the structure of k. First, when the stack is empty we just take the expression.

$$\epsilon(e) = e$$

Otherwise, we plug the expression into the frame on top of the stack (which is the rightmost part of the continuation), and then recursively plug the result into the remaining continuation.

$$\begin{array}{lcl} \epsilon(e) & = & e \\ (k \circ \langle \_, e_2 \rangle)(e_1) & = & k\,\langle e_1, e_2 \rangle \\ (k \circ \langle v_1, \_ \rangle)(e_2) & = & k\,\langle v_1\, e_2 \rangle \\ (k \circ \mathsf{case}\ \_\ b)(e_0) & = & k(\mathsf{case}\ e_0\ b) \end{array}$$

Here we wrote b for the single branch of a case expression over pairs. We now observe that the rules of the K Machine that decompose an expression all reconstitute the same expression!

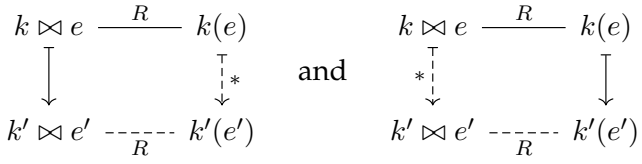
As a unifying notation for the two forms of machine state, we write $k \bowtie e$ to stand for either $k \triangleright e$ or $k \triangleleft e$. We relate machine states $k \bowtie e$ to expressions $k(e)$ written in infix notation as $k \bowtie e\ R\ k(e)$. This relation R is *weak bisimulation* if it satisfies

(i) If $k \bowtie e \mapsto k' \bowtie e'$ then $k(e) \mapsto^* k'(e')$

(ii) If $k(e) \mapsto k'(e')$ then $k \bowtie e \mapsto^* k' \bowtie e'$

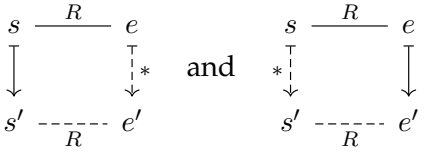
While the first statement is transparent, the second statement here has to be read carefully. In more detail, we are given an e_0, e_0', and transition $e_0 \mapsto e_0'$. Then for any $k \bowtie e$ such that $k \bowtie e\ R\ e_0$ there exist k' and e' such that $k \bowtie e \mapsto^* k' \bowtie e'$ and $k' \bowtie e'\ R\ e_0'$. Expanding the definition of R yields the second assertion above.

This form of relationship is often displayed in pictorial form, where solid lines denote given relationship and dashed lines denote relationship whose existence is to be proved. In this case we might display the two properties as



This is an example of a *weak bisimulation*, where "weak" indicates that the two sides do not have to proceed in lockstep. In the diagram this is represented by allowing zero or more steps $\mapsto^*$ in the transition we have to construct. Sometimes (actually: today) we can be more precise than just saying that there an unknown arbitrary number of steps. It is rare that a transformation we might consider will preserve individual steps exactly, so weak bisimulation is a more important notion that strong bisimulation.

A more generic depiction of a weak bisimulation that does not bake in the definition of R from this particular situation might look like



We now turn to our specific example, proving the first direction of the weak bisimulation.

Theorem 1 (Weak Bisimulation for the K Machine, Part 1, v1)
If $k \bowtie e \mapsto k' \bowtie e'$ *then* $k(e) \mapsto^* k'(e')$.

Proof: By cases on the definition of $k \bowtie e \mapsto k' \bowtie e'$. We write here "by cases" instead of "by induction" because none of the transition rules have any premises. A proof by cases is certainly a degenerate case of a proof by induction, but we would like to express the proof principle as precisely as possible.

Case: $k \triangleright \langle e_1, e_2 \rangle \mapsto k \circ \langle \_, e_2 \rangle \triangleright e_1$ where $e = \langle e_1, e_2 \rangle$, $k' = k \circ \langle \_, e_2 \rangle$ and $e' = e_1$. Then

$$k\langle e_1, e_2 \rangle = (k \circ \langle \_, e_2 \rangle)(e_1)$$

so

$$k(e) = k\langle e_1, e_2 \rangle \mapsto^0 (k \circ \langle \_, e_2 \rangle)(e_1) = k'(e')$$

Cases: The other cases where the rules just decompose the expression are analogous. In particular, on the side of the expressions no reductions are needed since the prestate and poststate of the transition reconstitute the same expression.

Case: $k_0 \circ (\mathsf{case}\ \_\ (\langle x_1, x_2 \rangle \Rightarrow e_3)) \triangleleft \langle v_1, v_2 \rangle \mapsto k_0 \triangleright [v_1/x_1, v_2/x_2]e_3$ where $k = k_0 \circ (\mathsf{case}\ \_\ (\langle x_1, x_2 \rangle \Rightarrow e_3))$, $e = \langle v_1, v_2 \rangle$, $k' = k_0$ and $e' = [v_1/x_1, v_2/x_2]e_3$. Then

$$k_0 \circ (\mathsf{case}\ \_\ (\langle x_1, x_2 \rangle \Rightarrow e_3))(\langle v_1, v_2 \rangle) = k_0(\mathsf{case}\ \langle v_1, v_2 \rangle\ (\langle x_1, x_2 \rangle \Rightarrow e_3))$$

and

$$\mathsf{case}\ \langle v_1, v_2 \rangle\ (\langle x_1, x_2 \rangle \Rightarrow e_3) \quad \mapsto \quad [v_1/x_1, v_2/x_2]e_3$$

So what we need to know is that

$$k_0(\mathsf{case}\ \langle v_1, v_2 \rangle\ (\langle x_1, x_2 \rangle \Rightarrow e_3) \quad \mapsto \quad k_0([v_1/x_1, v_2/x_2]e_3)$$

This does not follow directly from what we know in this case, but we can prove a suitable lemma (2), namely that $k(e) \mapsto k(e')$ whenever $e \mapsto e'$. Intuitively, this holds because k is just a representation of the congruence rules.

This lemma completes the proof

□

Lemma 2 (Continuation Congruence)
If $e \mapsto e'$ then $k(e) \mapsto k(e')$ for any k.

Proof: We are given a reduction from e to e', so the first instinct might be to prove this by rule induction on the derivation of $e \mapsto e'$. However, this reduction will simply be embedded in the reduction of $k(e) \mapsto k(e')$ rather than analyzed and decomposed, so this cannot be right.

Instead, we prove it by induction on the structure of k which is "wrapped around" e and e'.

Case: $k = \epsilon$. Then

$$k(e) = \epsilon(e) = e \mapsto e' = \epsilon(e') = k(e')$$

Case: $k = k_1 \circ \langle \_, e_2 \rangle$. Then

$k(e) = (k_1 \circ \langle \_, e_2 \rangle)(e)$	This case
$(k_1 \circ \langle \_\ e_2 \rangle)(e) = k_1\langle e, e_2 \rangle$	By definition of $k(-)$
$e \mapsto e'$	Assumption
$\langle e, e_2 \rangle \mapsto \langle e', e_2 \rangle$	By rule $\mathsf{step}/\mathsf{app}_1$
$k_1\langle e, e_2 \rangle \mapsto k_1\langle e', e_2 \rangle$	By ind. hyp. on k_1
$k_1\langle e', e_2 \rangle = (k_1 \circ \langle \_, e_2 \rangle)(e') = k(e')$	By definition of $k(-)$

Case: The remaining two cases for k are analogous.

□

We now refine the statement of the first direction of bisimulation to express that each step of the K machine is simulated by zero or one steps of reduction of our original dynamics. We write this in general as $e \mapsto^{0,1} e'$.

Theorem 3 (Weak Bisimulation for the K Machine, Part 1, v2)
If $k \bowtie e \mapsto k' \bowtie e'$ then $k(e) \mapsto^{0,1} k'(e')$.

Proof: See proof of Theorem 1. □

One ultimate end-to-end property we are interested in is for complete computations. We call this the *soundness* of the K machine because it expresses that if the K Machine returns a final answer, then this final answer is correct.

Corollary 4 (Soundness of the K Machine)
If $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft v$ *then* $e \mapsto^* v$.

Proof: We extend Theorem 1 to multistep reduction in the K Machine (by induction on the length of the reduction sequence) and then obtain the statement with $k = k' = \epsilon$. □

Unfortunately, the other direction of the bisimulation is more difficult to prove, so it remains a conjecture.

Conjecture 5 (Weak Bisimulation for the K Machine, Part 2)
If $e_0 \mapsto e_0'$ *where* $e_0 = k(e)$ *for some* k *and* e. *Then* $k \bowtie e \mapsto^* k' \bowtie e'$ *for some* k' *and* e' *with* $e_0' = k'(e')$.

Fortunately, the end-to-end result in the other direction is not in question. It states that if evaluation returns a value, then the K Machine will do so as well.

Theorem 6 (Completeness of the K Machine)
If $e \mapsto^* v$ *then* $\epsilon \triangleright e \mapsto^* \epsilon \triangleleft v$.

Proof: See the textbook [Har16, Chapter 28]. The proof uses an intermediate *big-step dynamics*. □

However, this theorem still does not capture everything we want, because it only talks about terminating computation. The bisimulation itself (if we could complete it) also speaks about arbitrary nonterminating computations.

4 Exceptions

Errors or exceptions are a fact of life. In programming, we have already seen this in the last lecture: when we analyzed the cases of the return value of v_1 of e_1 for an application $e_1\ e_2$ we only accounted for $e_1 = \lambda x.\, e_1'$. In fact, we

cannot continue with evaluation if v_1 is not a λ-expression. Furthermore, we expect our (object language) expression to be well-typed, so this should be the only possibility. But since we didn't check this in our implementation, we could certainly encounter a situation where the value (in the example of type E that represents expression) does not match any of the patterns.

One solution within our language so far would be to add those cases and simply not terminate when they are encountered. That would be difficult to analyze. Another solution would be to change our return type (in the metalanguage) to $E + 1$: we either return an actual value or none, indicating a runtime error. However, this change would require us to change code pervasively. What we would like instead is to signal an error in the form of an exception that aborts computation.

For this purpose we introduce a new form of expression.

$$\begin{array}{llll} \text{Expressions} & e & ::= & \cdots \mid \mathsf{raise}\ E \\ \text{Exceptions} & E & ::= & \mathbf{Match} \mid \mathbf{Error} \mid \mathbf{DivByZero} \mid \cdots \end{array}$$

Exceptions themselves are *not* first class expressions or values, but a new syntactic category. From the typing perspective, exception could be raised *anywhere*, that is, it has arbitrary type τ.

$$\frac{}{\Gamma \vdash \mathsf{raise}\ E : \tau}\ \mathsf{tp/raise}$$

This emphasizes that, as a control construct, it is in some sense orthogonal to all constructs already in the language, just as, for example, fixed points are.

In the K machine, it is almost trivial to specify the behavior of exceptions. We simply throw away the whole continuation (since we want to abort the computation) and immediately transition to a final state $\mathsf{raised}\ E$.

$$k \triangleright \mathsf{raise}\ E \quad \mapsto \quad \mathsf{raised}\ E$$

This means we now have three forms of machine states

$$\text{States} \quad s \quad ::= \quad k \triangleright e \mid k \triangleleft v \mid \mathsf{raised}\ E$$

The *final states* are now $\epsilon \triangleleft v$ (final answer is v) and $\mathsf{raised}\ E$ (the computation raised exception E).

If we want to go back to our original small-step semantics, it is not so easy. First, $\mathsf{raise}\ E$ is not value, but it also does not reduce. Instead, it is propagated up through the expression under evaluation using new version

of the congruence rules. For example, just for pairs we would have to *add* the following rules:

$$\frac{}{\langle \mathsf{raise}\ E, e_2\rangle \mapsto \mathsf{raise}\ E}\ \mathsf{step/pair/exn}_1 \qquad \frac{v_1\ \mathit{value}}{\langle v_1, \mathsf{raise}\ E\rangle \mapsto \mathsf{raise}\ E}\ \mathsf{step/pair/exn}_2$$

$$\frac{}{\mathsf{case}\ (\mathsf{raise}\ E)\ (\langle x_1, x_2\rangle \Rightarrow e_3) \mapsto \mathsf{raise}\ E}\ \mathsf{step/casep/exn}_0$$

We observe that we need a new version of every congruence rule to propagate exceptions. The same will be true for all the other constructs in the language, so we the presentation is far from modular.

This is an excellent example of a more general observation: the specification of language extensions, especially those that affect control, depends heavily on choosing the most suitable base semantics. Here, it is clear that exceptions are most naturally specified in the K machine.

But returning to our small-step semantics, let's examine our collection of theorems to see if they continue to hold.

Preservation continues to hold because the expression $\mathsf{raise}\ E$ can be of arbitrary type.

Progress no longer holds as stated before. Instead we have

> *If* $\cdot \vdash e : \tau$ *then either* $e \mapsto e'$ *for some* e*, or* e *value, or* $e = \mathsf{raise}\ E$*.*

Finality of Values does not change since we do not change the notion of value.

Determinacy also still holds, because in the new rule $\mathsf{step/pair/exn}_2$ we took care to require v_1 to be a value just as for $\mathsf{step/pair}_2$.

Canonical Forms also still holds exactly as stated since $\mathsf{raise}\ E$ is not a value.

Exercises

Exercise 1 One unnecessary expense in the K Machine is that values v may be evaluated many times. With recursive types values v can be arbitrarily large, so we would like avoid re-evaluation. For this purpose we introduce

a separate syntactic class of values w and a new expression constructor $\downarrow w$ that includes a value w as an expression. It is typed with

$$\frac{w\ \mathit{val} \quad \Gamma \vdash w : \tau}{\Gamma \vdash \downarrow w : \tau}\ \mathsf{tp/down}$$

and included in expressions with

$$\begin{array}{lrcl} \text{Expressions} & e & ::= & \ldots \mid \downarrow w \\ \text{Closed values} & w & ::= & \lambda x.\, e \mid \langle w_1, w_2 \rangle \mid \langle\,\rangle \mid i \cdot w \mid \mathsf{fold}\ w \end{array}$$

1. Update the K Machine so that the two machine states are $k \triangleright e$ and $k \triangleleft \downarrow w$. In order to avoid re-evaluation, only expressions $\downarrow w$ should be substituted for variables. Your rules should **not** appeal to the $e\ \mathit{val}$ judgment but simply construct closed values w as a natural part of the machine's operation. Only show the rules for functions and pairs.

2. Establish a weak bisimulation between the machine with marked values and those without, limiting yourself to eager pairs. This means you should

   (a) Define relation R between the states in the two machines.

   (b) Prove that R is a weak bisimulation, which requires two separate properties to be shown.

   (c) Sketch the proofs of any lemmas you might need regarding the operation of each of the two machines.

3. Analyze your proof in Part 3(b) to see if you can make a statement about how the number of steps in the two machines are related.

References

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, second edition, April 2016.