Lecture Notes on Bisimulation

15-814: Types and Programming Languages
Frank Pfenning

Lecture 13
Tuesday, October 15, 2019

1 Introduction

In the last lecture we introduced the K Machine as an alternative way to define the dynamics of programs in our language. We briefly summarize it on functions only, first recalling their small-step dynamics.

$$\frac{}{\lambda x.\, e\ \mathit{val}}\ \mathsf{val/lam}$$

$$\frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ \mathsf{step/app}_1 \qquad \frac{v_1\ \mathit{val} \quad e_2 \mapsto e_2'}{v_1\ e_2 \mapsto v_1\ e_2'}\ \mathsf{step/app}_2$$

$$\frac{}{(\lambda x.\, e_1')\ v_2 \mapsto [v_2/x]e_1'}\ \mathsf{step/beta}$$

The K Machine has two different forms of states

States	s	$::=$	$k \triangleright e$	evaluate e with continuation k
		$\mid$	$k \triangleleft v$	return value v to continuation k
Continuations	k	$::=$	ϵ	
		$\mid$	$k \circ (\_\ e_2) \mid k \circ (v_1\ \_)$	

with the following transitions

$$\begin{array}{rcllrcl}
k & \triangleright & \lambda x.\, e & \mapsto & k & \triangleleft & \lambda x.\, e \\
k & \triangleright & e_1\ e_2 & \mapsto & k \circ (_\ e_2) & \triangleright & e_1 \\
k \circ (_\ e_2) & \triangleleft & v_1 & \mapsto & k \circ (v_1\ _) & \triangleright & e_2 \\
k \circ ((\lambda x.\, e_1')\ _) & \triangleleft & v_2 & \mapsto & k & \triangleright & [v_2/x]e_1'
\end{array}$$

One of our guiding principles was

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

2 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 key 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 *unravels* 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.

$$\begin{array}{lcl} \epsilon(e) & = & e \\ (k \circ (\_\ e_2))(e_1) & = & k(e_1\, e_2) \\ (k \circ (v_1\ \_))(e_2) & = & k(v_1\, e_2) \end{array}$$

We now observe that the rules of the K Machine that decompose an expression leave the unravelling of a state unchanged.

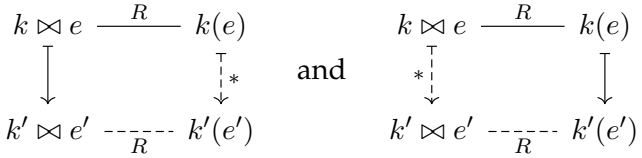
As a unifying notation for the two forms of machine state, we write $k \bowtie e$ to stand for either $k \triangleright e$ or $k \triangleleft e$. We relate machine states $k \bowtie e$ to expressions $k(e)$ written in infix notation as $k \bowtie e \; R \; k(e)$. This relation R is *weak bisimulation*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

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

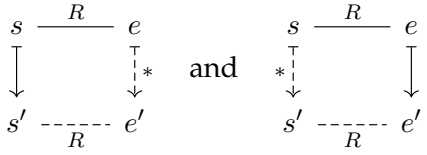
(ii) If $k(e) \mapsto k'(e')$ then $k \bowtie e \mapsto^* k' \bowtie e'$

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 side 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 this stronger property of the proof.

Case: $k \triangleright \lambda x_1.\, e_2 \mapsto k \triangleleft \lambda x_1.\, e_2$ where $e = \lambda x_1.\, e_2 = e'$ and $k' = k$. Then

$$k(e) = k(\lambda x_1.\, e_2) = k'(e')$$

and

$$k(e) \mapsto^0 k'(e')$$

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

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

so

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

Case: $k_0 \circ (\_\ e_2) \triangleleft v_1 \mapsto k_0 \circ (v_1\ \_) \triangleright e_2$ where $k = k_0 \circ (\_\ e_2)$, $e = v_1$, $k' = k_0 \circ (v_1\ \_)$ and $e' = e_2$. Then

$$(k_0 \circ \_\ e_2)(v_1) = k_0(v_1\, e_2) = (k_0 \circ v_1\ \_)(e_2)$$

so

$$k(e) = (k_0 \circ \_\ e_2)(v_1) \mapsto^0 (k_0 \circ v_1\ \_)(e_2) = k'(e')$$

What we have seen so far is the rule for values, or the rules that decompose expressions are actually identities as far as the global, small-step transitions are concerned. That because these transition rules just find the place in the expression where a "real reduction" (in this case, just β-reduction) can take place.

The final case will require a lemma, but let's see what that might be.

Case: $k_0 \circ ((\lambda x.\, e_1)\ \_) \triangleleft v_2 \mapsto k_0 \triangleright [v_2/x]e_2$. Then we need to show

$$k_0 \circ ((\lambda x.\, e_1)\ \_)(v_2) = k_0((\lambda x.\, e_1)\ v_2) \stackrel{?}{\mapsto}{}^* k_0([v_2/x]e_1)$$

The lemma that we need here, stated and proved below, expresses that small-step reduction behaves as a congruence but only for continuations k.

$k_0((\lambda x.\, e_1)\ v_2) \mapsto^1 k_0([v_2/x]e_1)$	By Lemma 2
$k(e) \mapsto^1 k'(e')$	By equality reasoning

□

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: $k = k_1 \circ \_\ e_2$. Then

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

Case: $k = k_1 \circ v_1\ \_$. Then

$k(e) = (k_1 \circ v_1\ \_)(e)$	This case
$(k_1 \circ v_1\ \_)(e) = k_1(v_1\ e)$	By definition of $k(-)$
$e \mapsto e'$	Assumption
$v_1\ e \mapsto v_1\ e'$	By rule $\mathsf{step/app}_2$
$k_1(v_1\ e) \mapsto k_1(v_1\ e')$	By ind. hyp. on k_1
$k_1(v_1\ e') = (k_1 \circ v_1\ \_)(e') = k(e')$	By definition of $k(-)$

□

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*. □

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 *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.

Exercise 2 Extend the K Machine to handle exceptions in the style of Section L11.6. There are two common technique to add exceptions.

1. We add a new form of state, $k \blacktriangleleft E$ expressing that an exception E has been raised and must be propagated or handled by k.

2. We have a pair of continuations: one is for handling normal return values, the other for handling exceptions directly. The goal is to avoid explicit unwinding of the stack because the most recent handler is directly accessible.

Write two extended versions of the K Machine following these two approaches, limiting yourself to functions, raising exceptions with $\mathsf{raise}\ E$, and the $\mathsf{try}\ e_1\ e_2$ construct. You should make sure that your machines remain faithful to the original semantics in L11.6, but you do not need to prove it.

References

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