Homework 5: Parallelism and Control Flow

15-814: Types and Programming Languages
Fall 2015
TA: Evan Cavallo (ecavallo@cs.cmu.edu)

Out: 11/5/15
Due: 11/17/15, 10:30am

1 Cost Dynamics

In this section, we will relate cost dynamics to sequential and parallel dynamics. Consider a language containing products.

$$\begin{array}{rcl} \tau & ::= & \cdots \mid \tau \times \tau \\ e & ::= & \cdots \mid \langle e, e \rangle \mid e \cdot \mathtt{L} \mid e \cdot \mathtt{R} \end{array}$$

In class, we defined a judgment $e \Downarrow^c v$, which states that e evaluates to v with cost graph c. We defined a grammar of cost graphs like so:

$$c ::= \mathbf{0} \mid \mathbf{1} \mid c \oplus c \mid c \otimes c$$

Here, $\mathbf{0}$ denotes a computation with zero cost and $\mathbf{1}$ represents a single unit of computation. The graph $c_1 \oplus c_2$ is the *sequential composition* of the graphs c_1 and c_2, while $c_1 \otimes c_2$ is the *parallel composition*. To each cost graph c we associate two numbers, the *work* $\mathrm{wk}(c)$ and *depth* $\mathrm{dp}(c)$, which measure sequential and parallel complexity respectively.

$$\begin{array}{lcl@{\qquad\qquad}lcl} \mathrm{wk}(\mathbf{0}) & = & 0 & \mathrm{dp}(\mathbf{0}) & = & 0 \\ \mathrm{wk}(\mathbf{1}) & = & 1 & \mathrm{dp}(\mathbf{1}) & = & 1 \\ \mathrm{wk}(c_1 \oplus c_2) & = & \mathrm{wk}(c_1) + \mathrm{wk}(c_2) & \mathrm{dp}(c_1 \oplus c_2) & = & \mathrm{dp}(c_1) + \mathrm{dp}(c_2) \\ \mathrm{wk}(c_1 \otimes c_2) & = & \mathrm{wk}(c_1) + \mathrm{wk}(c_2) & \mathrm{dp}(c_1 \otimes c_2) & = & \max\{\mathrm{dp}(c_1), \mathrm{dp}(c_2)\} \end{array}$$

The two differ only in the case of $c_1 \otimes c_2$ – while the work of a parallel composition is the sum of the work of the components, the depth is the longer of the component depths.

We also defined sequential and parallel versions of the structural operational dynamics, given by judgments $e \mapsto_{\mathsf{seq}} e'$ and $e \mapsto_{\mathsf{par}} e'$. These differ on the rules for evaluating pairs: while $\mapsto_{\mathsf{seq}}$ evaluates the first component of a pair before the second, $\mapsto_{\mathsf{par}}$ evaluates the two simultaneously.

$$\frac{e_1 \mapsto_{\mathsf{seq}} e_1'}{\langle e_1, e_2 \rangle \mapsto_{\mathsf{seq}} \langle e_1', e_2 \rangle}\ (\textsc{Pair-S1}) \qquad \frac{e_1\ \mathsf{val} \quad e_2 \mapsto_{\mathsf{seq}} e_2'}{\langle e_1, e_2 \rangle \mapsto_{\mathsf{seq}} \langle e_1, e_2' \rangle}\ (\textsc{Pair-S2})$$

$$\frac{e_1 \mapsto_{\mathsf{par}} e_1' \quad e_2 \mapsto_{\mathsf{par}} e_2'}{\langle e_1, e_2 \rangle \mapsto_{\mathsf{par}} \langle e_1', e_2' \rangle}\ (\textsc{Pair-P1}) \qquad \frac{e_1\ \mathsf{val} \quad e_2 \mapsto_{\mathsf{par}} e_2'}{\langle e_1, e_2 \rangle \mapsto_{\mathsf{par}} \langle e_1, e_2' \rangle}\ (\textsc{Pair-P2})$$

$$\frac{e_1 \mapsto_{\mathsf{par}} e_1' \quad e_2\ \mathsf{val}}{\langle e_1, e_2 \rangle \mapsto_{\mathsf{par}} \langle e_1', e_2 \rangle}\ (\textsc{Pair-P3})$$

The rules for evaluating projections are the same for both evaluation strategies (let $m \in \{\mathsf{seq}, \mathsf{par}\}$):

$$\frac{e \mapsto_m e'}{e \cdot \mathtt{L} \mapsto_m e' \cdot \mathtt{L}}\ (\textsc{ProjL-1}) \qquad \frac{e_1\ \mathsf{val} \quad e_2\ \mathsf{val}}{\langle e_1, e_2 \rangle \cdot \mathtt{L} \mapsto_m e_1}\ (\textsc{ProjL-2})$$

(The rules for $e \cdot \mathtt{R}$ are completely symmetric.) We want our definition $e \Downarrow^c v$ to cohere with the two operational dynamics definitions, in the sense of the following proposition.

Proposition 1 *Correction 11/16/15: Fixed statement.* *For an expression e and value v, $e \mapsto^w_{\mathsf{seq}} v$ if and only if there exists c with $e \Downarrow^c v$ and $\mathrm{wk}(c) = w$. Likewise, $e \mapsto^d_{\mathsf{par}} v$ if and only if there exists c with $e \Downarrow^c v$ and $\mathrm{dp}(c) = d$.*

Here, the judgment $e \mapsto^n_m e'$ expresses that e steps to e' in n steps, and is defined by the following rules:

$$\frac{}{e \mapsto^0_m e} \qquad \frac{e \mapsto_m e' \quad e' \mapsto^n_m e''}{e \mapsto^{n+1}_m e''}$$

Task 1 *Define rules for $e \Downarrow^c v$ covering the cost graph dynamics of products (pairing and projections). You may want to do this task in tandem with Task 2, which could help you determine what the cost graph annotations should be. Your definition should also satisfy Lemma 1 below.*

For a refresher on evaluation dynamics, see PFPL 7.

To show the forward direction(s) of Proposition 1, we use a head expansion lemma. Combined with Lemma 1 below, which you may use without proof, the theorem quickly follows.

Lemma 1 *If $e \Downarrow^c v$ and e val, then $\mathrm{wk}(c) = \mathrm{dp}(c) = 0$. Conversely, if e val then there exists c with $\mathrm{wk}(c) = \mathrm{dp}(c) = 0$ and $e \Downarrow^c e$.*

Task 2 (Head Expansion) *Using the definitions you gave in Task 1, show the following:*

(a) If $e' \Downarrow^{c'} v$ and $e \mapsto_{\mathsf{seq}} e'$, then there exists c with $e \Downarrow^c v$ and $\mathrm{wk}(c) = \mathrm{wk}(c') + 1$.

(b) If $e' \Downarrow^{c'} v$ and $e \mapsto_{\mathsf{par}} e'$, then there exists c with $e \Downarrow^c v$ and $\mathrm{dp}(c) = \mathrm{dp}(c') + 1$.

Go by induction on the derivation of $m \mapsto_e e'$. For (a), give the case for (Pair-S1)*. For (b), give the cases for* (Pair-P1), (Pair-P2), (ProjL-1), *and* (ProjL-2).

2 Continuations

In this section, we will write some programs using continuations and show how to encode continuations in a language without them. We'll start with a language with continuations which is evaluated using a control stack.

$$\begin{array}{rcl}
\tau & ::= & \mathtt{nat} \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau \mid \tau\ \mathtt{cont} \\
e & ::= & \cdots \mid \mathtt{cont}(k) \mid \mathtt{letcc}\{\tau\}(x.e) \mid \mathtt{throw}\{\tau\}(e; e) \\
k & ::= & \epsilon \mid k; f \\
f & ::= & \cdots \mid \mathtt{throw}\{\tau\}(-; e) \mid \mathtt{throw}\{\tau\}(e; -)
\end{array}$$

We use an eager evaluation strategy for all of the constructs in the language. Evaluation of continuations is given by the following rules (see PFPL 30.2 for more details and statics rules).

$$\frac{}{k \rhd \mathtt{cont}(k') \longmapsto k \lhd \mathtt{cont}(k')} \qquad \frac{}{k \rhd \mathtt{letcc}\{\tau\}(x.e) \longmapsto k \rhd [\mathtt{cont}(k)/x]e}$$

$$\frac{}{k \rhd \mathtt{throw}\{\tau\}(e_1; e_2) \longmapsto k; \mathtt{throw}\{\tau\}(-; e_2) \rhd e_1}$$

$$\frac{e_1\ \mathsf{val}}{k; \mathtt{throw}\{\tau\}(-; e_2) \lhd e_1 \longmapsto k; \mathtt{throw}\{\tau\}(e_1; -) \rhd e_2} \qquad \frac{e_1\ \mathsf{val}}{k; \mathtt{throw}\{\tau\}(e_1; -) \lhd \mathtt{cont}(k') \longmapsto k' \lhd e_1}$$

2.1 Programming with Continuations

To start off, we'll write a few basic programs with continuations. An example, which we covered in class and which may be useful in Task 3, is the function $\texttt{ccwf} : (\tau_1 \to \tau_2) \to (\tau_2\ \texttt{cont} \to \tau_1\ \texttt{cont})$, "compose continuation with function."

$$\begin{aligned}\texttt{ccwf} \triangleq\ & \lambda f{:}\tau_1 \to \tau_2.\ \lambda k{:}\tau_2\ \texttt{cont}.\\ & \quad \texttt{letcc}\{\tau_1\ \texttt{cont}\}(\mathit{ret}.\texttt{throw}\{\tau_1\ \texttt{cont}\}(f(\texttt{letcc}\{\tau_1\}(r.\texttt{throw}\{\tau_1\}(r;\mathit{ret})));k))\end{aligned}$$

How does this function work? Given $f : \tau_1 \to \tau_2$ and $k : \tau_2\ \texttt{cont}$, let's consider what would happen if we threw some value $e : \tau_1$ to the continuation $\texttt{ccwf}\ f\ k : \tau_1\ \texttt{cont}$. First, we push the throw onto the stack. (For our purposes, it doesn't matter what type we give the throw.)

$$\begin{aligned} & \epsilon & \rhd\ & \texttt{throw}\{\ldots\}(e;\texttt{ccwf}\ f\ k)\\ \mapsto^*\ & \epsilon;\texttt{throw}\{\ldots\}(e;-) & \rhd\ & \texttt{ccwf}\ f\ k\end{aligned}$$

Now we evaluate $\texttt{ccwf}\ f\ k$. After stepping through the application to f and k, we save the current continuation $(\epsilon;\texttt{throw}\{\ldots\}(e;-))$ in the variable ret (which we'll continue to write as ret for sake of space). We then begin computing a value of type τ_2 which we will eventually throw to k.

$$\mapsto^* \quad \epsilon;\texttt{throw}\{\ldots\}(e;-);\texttt{throw}\{\tau_2\}(-;k) \quad \rhd \quad f(\texttt{letcc}\{\tau_1\}(r.\texttt{throw}\{\tau_1\}(r;\mathit{ret})))$$

Assuming f is a value, we push an application frame onto the stack. We again save the current continuation, which at this point expects a value of type τ_1, in the variable r.

$$\begin{aligned}\mapsto^*\ & \epsilon;\texttt{throw}\{\ldots\}(e;-);\texttt{throw}\{\tau_2\}(-;k);f(-) & \rhd\ & \texttt{letcc}\{\tau_1\}(r.\texttt{throw}\{\tau_1\}(r;\mathit{ret}))\\ \mapsto^*\ & \epsilon;\texttt{throw}\{\ldots\}(e;-);\texttt{throw}\{\tau_2\}(-;k);f(-) & \rhd\ & \texttt{throw}\{\tau_1\}(r;\mathit{ret})\end{aligned}$$

Finally, we throw r to the continuation ret. The current stack is erased and replaced with the stack stored in ret, which was $\epsilon;\texttt{throw}\{\tau_1\}(e;-)$.

$$\mapsto^* \quad \epsilon;\texttt{throw}\{\ldots\}(e;-) \quad \rhd \quad r$$

Since r is a value, we execute the throw at the top of the stack. It replaces the stack with r, which is $\epsilon;\texttt{throw}\{\ldots\}(e;-);\texttt{throw}\{\tau_2\}(-;k);f(-)$, and continues by evaluating e.

$$\mapsto^* \quad \epsilon;\texttt{throw}\{\ldots\}(e;-);\texttt{throw}\{\tau_2\}(-;k);f(-) \quad \rhd \quad e$$

Let's say $f(e)$ evaluates to some value v. After said evaluation takes place, we execute the throw at the top of the stack, which throws v to k, leaving us in the state

$$\mapsto^* \quad k \quad \rhd \quad v$$

In the end, then, throwing a value e to $(\texttt{ccwf}\ f\ k)$ amounts to throwing $f(e)$ to k.

Task 3 *Define programs with the following types. You may use* ccwf *in your definitions.*

1. $\texttt{lem} : \tau + \tau\ \texttt{cont}$
2. $\texttt{dne} : \tau\ \texttt{cont}\ \texttt{cont} \to \tau$
3. $\texttt{cps} : (\tau_1 \to \tau_2) \to (\tau_1\ \texttt{cont} + \tau_2)$

(Notice: if we interpret these types as propositions by treating $(-)\ \texttt{cont}$ *as logical negation* $\neg(-)$*, these are tautologies of classical propositional logic!)*

Task 4 *Take $\tau = \texttt{int}$ in the previous task, and consider the following expression $e : \texttt{int}$ defined using your implementation of* `lem`*.*

$$\begin{array}{rcl} e & \triangleq & \texttt{case lem}\ \{ \\ & & \quad \texttt{L} \cdot x \hookrightarrow 2 * x; \\ & & \quad \texttt{R} \cdot c \hookrightarrow \texttt{throw}\{\texttt{int}\}(6; c) \\ & & \} \end{array}$$

What is the result of evaluating $\epsilon \triangleright e$? You don't have to write out the stack machine steps, but give an informal explanation of the evaluation process.

2.2 Translating Continuations

In this section, we will show how to translate programs in a language **K** with continuations into a language **I** without them. The theorem we prove will have the following form:

Theorem 1 *Fix a type ρ in* **I***. There are translations $||-||$, $|-|$ from types in* **K** *to types in* **I** *and a translation $\hat{-}$ from well-typed terms in* **K** *to terms in* **I***, all defined in terms of ρ. If $\Gamma \vdash_{\mathbf{K}} e : \tau$, then $||\Gamma|| \vdash_{\mathbf{I}} \hat{e} : |\tau|$. (Here, $||-||$ is extended to a translation of contexts by translating the type of each variable.)*

Of course, the value of this theorem depends on how we define $||-||$, $|-|$ and $\hat{-}$. (Our definition will satisfy a correctness theorem comparing the results of evaluating e and $\hat{e}$, but we will not discuss this here.) Let's by defining the translations of types $||\tau||$ and $|\tau|$. These are defined by mutual recursion. First, we have

$$|\tau| \quad \triangleq \quad (||\tau|| \to \rho) \to \rho$$

The type ρ represents the type of the "final result." A value of type $|\tau|$ accepts a "$||\tau||$ continuation," a term which computes a ρ from a $||\tau||$, and uses it to compute such a ρ. $||\tau||$ is defined inductively on the structure of τ. Let's say that **K** and **I** support natural numbers, products, and functions. We define

$$\begin{array}{rcl} ||\texttt{nat}|| & \triangleq & \texttt{nat} \\ ||\tau_1 \times \tau_2|| & \triangleq & ||\tau_1|| \times ||\tau_2|| \\ ||\tau_2 \to \tau_1|| & \triangleq & ||\tau_2|| \to |\tau_1| \\ ||\tau\ \texttt{cont}|| & \triangleq & ||\tau|| \to \rho \end{array}$$

The translation of context $|\Gamma|$ is simply given by translating the type of each variable. Now, we will given the translation of expressions. This is a *type-directed translation* – the translation of some expression e with $\Gamma \vdash_{\mathbf{K}} e : \tau$ is defined by induction on its typing derivation. We will specify it by defining a judgment $\Gamma \vdash e : \tau \rightsquigarrow \hat{e}$, which expresses that $\Gamma \vdash_{\mathbf{K}} e : \tau$ translates to $\hat{e}$ and has a case for each typing rule in **K**. After giving the definition, we will show that $||\Gamma|| \vdash_{\mathbf{I}} \hat{e} : |\tau|$. Let's start with the rules for variables and natural numbers.

$$\frac{}{\Gamma, x : \tau \vdash x : \tau \rightsquigarrow \lambda k.\ k(x)}\ (\textsc{Tr-Var})$$

$$\frac{}{\Gamma \vdash \texttt{z} : \texttt{nat} \rightsquigarrow \lambda k.\ k(\texttt{z})}\ (\textsc{Tr-Z}) \qquad \frac{\Gamma \vdash e : \texttt{nat} \rightsquigarrow \hat{e}}{\Gamma \vdash \texttt{s}(e) : \texttt{nat} \rightsquigarrow \lambda k.\ \hat{e}(\lambda x.\ k(\texttt{s}(x)))}\ (\textsc{Tr-S})$$

$$\frac{\Gamma \vdash e : \texttt{nat} \rightsquigarrow \hat{e} \quad \Gamma \vdash e_0 : \tau \rightsquigarrow \hat{e}_0 \quad \Gamma, x : \texttt{nat} \vdash e_1 : \tau \rightsquigarrow \hat{e}_1}{\Gamma \vdash \texttt{ifz}(e; e_0; x.e_1) : \tau \rightsquigarrow \lambda k.\ \hat{e}(\lambda y.\ \texttt{ifz}(y; \hat{e}_0 k; x.\hat{e}_1 k))}\ (\textsc{Tr-Ifz})$$

To understand these rules, remember that each translation $\hat{e}$ should be an expression of type $|\tau| = ||\tau|| \to \rho) \to \rho$, a function which takes a "continuation" and returns a result (a term of type ρ). It is important to distinguish $||\tau||$ from $(||\tau|| \to \rho) \to \rho$: although there is an obvious embedding from the former into the latter, taking e to $\lambda k.\ k(e)$, such a transformation is not reversible in general. In other words, while we might think of $\hat{e}$ as a function that, given a "continuation" $k : ||\tau|| \to \rho$, supplies "e" to k, this is not always accurate (although it is true in the case of the rules (Tr-Var) and (Tr-Z)). For one thing, this would require translating e into a $\mathbf{I}$ term of type $||\tau||$, which is not always possible. Moreover, $\hat{e}$ can use k in other ways. We can more accurately say that $\hat{e} : |\tau|$ will compute a ρ *if we tell it how we would compute a ρ given a* $||\tau||$, but it may accomplish this in ways other than "feeding in e."

With this in mind, let's go through the rules for `nat`. The translation for `z` is simple: given a "continuation" $k : \texttt{nat} \to \rho$, it simply invokes k on `z`. For $\texttt{s}(e)$, we start by translating e into $\hat{e}$. After taking an argument "continuation" $k : \texttt{nat} \to \rho$, we apply $\hat{e}$ to $\lambda x.\ k(\texttt{s}(x))$, the composition $k \circ \texttt{s}$. Put informally, we tell $\hat{e}$ that we want to "use e" by first taking its successor and then continuing with k. Finally, the rule for $\texttt{ifz}(e; e_0; x.e_1)$ supplies the translation $\hat{e}$ with the continuation $\lambda y.\ \texttt{ifz}(y; \hat{e}_0 k; x.\hat{e}_1 k)$, which uses $y : \texttt{nat}$ by checking if it is zero and continuing with either $\hat{e}_0 k$ or $\hat{e}_1 k$ as appropriate.

Task 5 *Define translation rules for the product type $\tau_1 \times \tau_2$ (i.e., for pairing and the two projections). Make sure your definitions have the correct types!*

The translation becomes more interesting at the function type.

$$\frac{\Gamma, x : \tau_2 \vdash e : \tau_1 \rightsquigarrow \hat{e}}{\Gamma \vdash (\lambda x{:}\tau_2.\ e) : \tau_2 \to \tau_1 \rightsquigarrow \lambda k.\ k(\lambda x{:}||\tau_2||.\ \hat{e})}\ (\textsc{Tr-Lam})$$

Recall that $||\tau_2 \to \tau_1|| \triangleq ||\tau_2|| \to |\tau_1|$. If our translation has the right type behavior, then $||\Gamma||, x : ||\tau_2|| \vdash \hat{e} : |\tau_1|$. Thus, $\lambda x.\ \hat{e}$ has type $||\tau_2 \to \tau_1||$, so we can supply it to the continuation $k : ||\tau_2 \to \tau_1|| \to \rho$. The key here is that hypotheses are translated with $||-||$ and conclusions with $|-|$, so it is natural to translate functions, which internalize a hypothetical, with $||\tau_2 \to \tau_1|| \triangleq ||\tau_2|| \to |\tau_1|$.

Task 6 *Give a translation rule for function application. For sake of intuition, you may find it useful to note the following: if we expand the definition of $||\tau_2 \to \tau_1||$ another step as $||\tau_2|| \to (||\tau_1|| \to \rho) \to \rho$, we see that a translated function takes an* argument *of type $||\tau_2||$ and a* "return address" continuation *which specifies how to continue with the result of type $||\tau_1||$.*

Finally, we come to continuations.

Task 7 *Give translation rules for $\texttt{letcc}\{\tau\}(x.e)$ and $\texttt{throw}\{\tau'\}(e_1; e_2)$. These are actually quite simple!*

This is enough to cover any program we would write in $\mathbf{K}$, but there is a form of expression arising in execution which we haven't handled – the term $\texttt{cont}(k)$ where k is a stack. In order to handle this case (and in order to state a correctness of translation result with respect to evaluation), we'd have to define a translation of stack frames. We won't pursue this here.