Assignment #6: Bounded Implementations of Nested Parallelism

15-312: Principles of Programming Languages

Out: Tuesday, April 14, 2015
Due: Thursday, April 30, 2015 1:29PM

Introduction

In this assignment, we will write a bounded implementation of a language with fork-join parallelism. The work will be divided into two stages. First, we will implement the semantics of the “local steps” that are performed by a single processor. Second, we will implement a scheduler that assigns these local steps to some finite number of processors.

Submission

We will collect exactly the following files from the /afs/andrew/course/15/312/ directory:

- handin/<yourandrewid>/assn6/global_dynamics.sml
- handin/<yourandrewid>/assn6/local_dynamics.sml

Make sure that your files have the right names (especially assn6.pdf!) and are in the correct directories.

1 A Data Parallel Language

The language of study in this assignment features a new parallel let construct. This evaluates two expressions in parallel and then substitutes them into the body of the let when they are both values. To model this parallel evaluation, our semantics will have tasks that represent independent pieces of computation that can be scheduled on a processor. When a task executes \( \text{par } x_0 = e_0 \text{ and } e_1 = x_1 \text{ in } e \) it will fork off two new tasks to compute \( e_0 \) and \( e_1 \) and wait for them to finish. To represent that a task is waiting for the results of other tasks, we have expressions of the form \( \text{join} [a_0,\ldots,a_n](x_0,\ldots,x_n.e) \), which indicates that this computation is waiting on the results of \( a_0,\ldots,a_n \). When these tasks finish, their results will be substituted in for \( x_0,\ldots,x_n \) (respectively) in \( e \). In this assignment, we will only need to wait on at most two tasks at a time. These join points cannot appear at the source level of programs; they only arise during execution. Types and expressions are given in Figure 1. This language is the familiar PCF language extended with the constructs just described.
<table>
<thead>
<tr>
<th>Sort</th>
<th>Abstract Form</th>
<th>Concrete Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type  ( \tau ) ::=</td>
<td>nat</td>
<td>nat</td>
</tr>
<tr>
<td>( \text{parr}(\tau_1;\tau_2) )</td>
<td>( \tau_1 \to \tau_2 )</td>
<td></td>
</tr>
<tr>
<td>( \text{unit} )</td>
<td>( \text{unit} )</td>
<td></td>
</tr>
<tr>
<td>( \text{prod}(\tau_1;\tau_2) )</td>
<td>( \tau_1 \times \tau_2 )</td>
<td></td>
</tr>
<tr>
<td>( \text{void} )</td>
<td>( \text{void} )</td>
<td></td>
</tr>
<tr>
<td>( \text{sum}(\tau_1;\tau_2) )</td>
<td>( \tau_1 + \tau_2 )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exp  ( e \ ::= )</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>( x )</td>
<td></td>
</tr>
<tr>
<td>( z )</td>
<td>( z )</td>
<td></td>
</tr>
<tr>
<td>( s(e) )</td>
<td>( s(e) )</td>
<td></td>
</tr>
<tr>
<td>( \text{ifz}(e; e_0; x.e_1) )</td>
<td>( \text{ifz} { z \Rightarrow e_0</td>
<td>s(x) \Rightarrow e_1 } )</td>
</tr>
<tr>
<td>( \text{let}(e_0; x.e_1) )</td>
<td>( \text{let} { x = e_0 \ \text{in} \ e_1 } )</td>
<td></td>
</tr>
<tr>
<td>( \text{par}(e_0; e_1; x_0.x_1.e) )</td>
<td>( \text{par} { x_0 = e_0 \ \text{and} \ e_1 = x_1 \ \text{in} \ e } )</td>
<td></td>
</tr>
<tr>
<td>( \text{join}(a_0,..,a_n)(x_0,..,x_n.e) )</td>
<td>( \text{join} { x_0 \leftarrow a_0,..,x_n \leftarrow a_n \ \text{in} \ e } )</td>
<td></td>
</tr>
<tr>
<td>( \text{lam}<a href="x.e">\tau</a> )</td>
<td>( \text{fn} { x:\tau \Rightarrow e } )</td>
<td></td>
</tr>
<tr>
<td>( \text{ap}(e_1; e_2) )</td>
<td>( e_1(e_2) )</td>
<td></td>
</tr>
<tr>
<td>( \text{fix}<a href="x.e">\tau</a> )</td>
<td>( \text{fix} { x:\tau \Rightarrow e } )</td>
<td></td>
</tr>
<tr>
<td>( \text{triv} )</td>
<td>( \langle \rangle )</td>
<td></td>
</tr>
<tr>
<td>( \text{pair}(e_1; e_2) )</td>
<td>( (e_1, e_2) )</td>
<td></td>
</tr>
<tr>
<td>( \text{pr}<a href="e">l</a> )</td>
<td>( e \cdot l )</td>
<td></td>
</tr>
<tr>
<td>( \text{pr}<a href="e">r</a> )</td>
<td>( e \cdot r )</td>
<td></td>
</tr>
<tr>
<td>( \text{abort}<a href="e">\tau</a> )</td>
<td>( \text{abort}<a href="e">\tau</a> )</td>
<td></td>
</tr>
<tr>
<td>( \text{in}[\tau_1;\tau_2]<a href="e">l</a> )</td>
<td>( \text{inl}<a href="e">\tau_1;\tau_2</a> )</td>
<td></td>
</tr>
<tr>
<td>( \text{in}[\tau_1;\tau_2]<a href="e">r</a> )</td>
<td>( \text{inr}<a href="e">\tau_1;\tau_2</a> )</td>
<td></td>
</tr>
<tr>
<td>( \text{case}(e; x_1.e_1; x_2.e_2) )</td>
<td>( \text{case} { e_1 \Rightarrow e</td>
<td>e_2 \Rightarrow e } )</td>
</tr>
</tbody>
</table>

Figure 1: Expressions and types in our parallel language
2 Bounded Implementation

The bounded implementation we are going to describe gives a picture of execution that is closer to a realistic implementation than the parallel semantics we discussed in class. We describe the global state of a parallel computation as a mapping from symbols $a_i$, which we call “tasks”, to expressions $e_i$. These tasks collectively capture the steps that need to happen to finish the global computation. The global states have the form $\nu \Sigma \{\mu\}$, where the signature $\Sigma$ associates types to each task:

$$\Sigma = a_1 \sim \tau_1, \ldots, a_n \sim \tau_n$$

and $\mu$ associates an expression with each task:

$$\mu = a_1 \hookrightarrow s_1 \otimes \ldots \otimes a_n \hookrightarrow s_n$$

We treat the symbols $a_i$ as being bound in $\Sigma$ and therefore subject to renaming, though this renaming will not be visible in our implementation – as with variables (or assignables in Algol) we will make sure each newly generated task is given a fresh name. The orderings in $\Sigma$ and $\mu$ are relevant – we do not consider the $\otimes$ operation to be commutative. The importance of this ordering will become clear later.

We will say that $\mu$ is a substate of $\mu'$ if there exists $\mu_l, \mu_r$ (each may possibly be empty) such that $\mu' = \mu_l \otimes \mu \otimes \mu_r$. If $\mu_l$ is non-empty and $\mu_r$ is empty, we further say that $\mu$ is a strict right-substate of $\mu'$. Similarly, if $\mu_l$ is empty and $\mu_r$ is non-empty, then $\mu$ is a strict left-substate of $\mu'$.

3 Statics

The static semantics for most of the expressions in the language remain the same as in PCF. The rule for parallel let is a natural generalization of the rule for let:

$$\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \Gamma, x_1 : \tau_1, x_2 : \tau_2 \vdash e : \tau$$

$$\Gamma \vdash \text{par}(e_1; e_2; x_1.x_2.e) : \tau$$

(par)

Because the statics of the source level language are so similar to the languages you’ve already implemented, we will not ask you to implement them in this assignment. The statics of the global states are more complicated.

We are giving you a typechecker in the handout which can typecheck both expressions and states. However, our typechecker does not enforce a number of invariants that should hold of well-formed states. For example, it does not check that at most one join point refers to any given task. Rather, it checks that the expression associated with each task is well typed, and that join points are well-typed given the types of tasks.

4 Local Steps

A local transition deals only with a restricted portion of the memory. Each local step has the form:

$$\nu a, \Sigma \{a \hookrightarrow e \otimes \mu\} \rightarrow_{\text{loc}} \nu a, \Sigma' \{a \hookrightarrow e' \otimes \mu'\}$$

\footnote{Do remember, though, that this is still a semantics, a specification of how implementations should behave and not a guide to implementation. There are elements to this presentation, like the presence of substitution, that you wouldn’t want in a realistic implementation.}
We do not write out the types in $\Sigma$ because they do not affect the dynamics. The left most task ($a$ above) is called the primary task for each transition. We say that a state $\mu$ is runnable if $\nu \Sigma \{ \mu \} \rightarrow_{loc} \nu \Sigma' \{ \mu' \}$ for some $\mu'$. The full list of local transition rules can be found in Appendix B. These rules can be roughly categorized into three groups: those that perform some piece of computation, those that fork off new tasks to do some subcomputation, and those that collect results from forked tasks. Let’s look at the rules for $\text{ifz}(e; e_0; x.e_1)$:

\[
\nu a, \Sigma \{ a \leftarrow \text{ifz}(e; e_0; x.e_1) \otimes \mu \} \rightarrow_{loc} \nu a, a', \Sigma \{ a \leftarrow \text{join}(a')(x'.\text{ifz}(x'; e_0; x.e_1)) \otimes a' \leftarrow e \otimes \mu \}
\]

\[
\nu a, \Sigma \{ a \leftarrow \text{ifz}(z; e_0; x.e_1) \otimes \mu \} \rightarrow_{loc} \nu a, \Sigma \{ a \leftarrow e_0 \otimes \mu \}
\]

Rules L3 and L4 should seem familiar. They perform a local piece of computation that corresponds exactly to the normal rules we have for PCF. Meanwhile, L2 is an example of a transition that forks off a new task to perform a subcomputation. The rule first checks that $e$ is not a value. If it’s not, then we spawn a new task $a'$ to finish the computation of $e$. The expression for task $a$ is updated to a join to indicate that it depends on $a'$. Note that we insert $a'$ immediately to the right of $a$, before any of the tasks that appear in $\mu$.

We have another rule LJ that substitutes the results of tasks into joins:

\[
\nu a, a_1, \ldots, a_n, \Sigma \{ a \leftarrow \text{join}(a_1, \ldots, a_n)(x_1, \ldots, x_n.e) \otimes a_1 \leftarrow e_1 \otimes \cdots \otimes a_n \leftarrow e_n \otimes \mu \} \rightarrow_{loc} \nu a, \Sigma \{ a \leftarrow [e_1, \ldots, e_n/x_1, \ldots, x_n]e \otimes \mu \}
\]

Notice that after we substitute the results into the join, tasks $a_1$ through $a_n$ go away. Most of the rules we have for forking new tasks only create a single new task. The obvious exception is the rule for the parallel let:

\[
\nu a, \Sigma \{ a \leftarrow \text{par}(e_1; x_1; e_2, x_2.e) \otimes \mu \} \rightarrow_{loc} \nu a, a_1, a_2, \Sigma \{ a \leftarrow \text{join}(a_1, a_2)(x_1, x_2.e) \otimes a_1 \leftarrow e_1 \otimes a_2 \leftarrow e_2 \otimes \mu \}
\]

Again, the task $a_1$ that computes $e_1$ appears to the left of the task $a_2$ that computes $e_2$.

We have set up the rules for local transitions so that we have a property called right weakening:

**Lemma 1** (Right Weakening). If $\nu \Sigma \{ \mu \} \rightarrow_{loc} \nu \Sigma \{ \mu' \}$, then for all $\mu_f$ and $\Sigma_f$ such that the domains of $\mu$ and $\mu_f$ are disjoint, $\nu \Sigma_f \{ \mu \otimes \mu_f \} \rightarrow_{loc} \nu \Sigma_f' \{ \mu' \otimes \mu_f \}$.

**Task 4.1** (25%). Implement the local step semantics in local_dynamics.sml. You need to implement two functions to finish this structure. The first, isVal, should take in an expression $e$ and return true if $e$ is a value, and return false otherwise. The second, trystep should take in some $\mu$ and return $\text{STEP} \mu'$ if $\mu$ takes a local step to $\mu'$, or $\text{VAL}$ if the primary task of $\mu$ is a value. You may raise the exception Malformed if neither case is applicable. Note that we do not pass around $\Sigma$, because it’s not needed. (However, it is included in the rules we gave because it’s needed if we want to prove a type safety theorem).
5 Scheduling

The local steps describe how to perform computation on a single processor. Now we need to give rules that describe how a $p$-processor machine computes multiple steps in parallel. We call this a *global transition*. The general rule for a global transition is:

$$n \leq p \quad \text{for each } i \leq n, \nu \Sigma_i \{\mu_i\} \mapsto \text{loc} \nu \Sigma'_i \{\mu'_i\}$$

$$\nu \Sigma_1, \ldots, \Sigma_n \{\mu_{\text{pause}} \otimes \mu_1 \otimes \cdots \otimes \mu_n\} \mapsto \text{glo} \nu \Sigma'_1, \ldots, \Sigma'_n \{\mu_{\text{pause}} \otimes \mu'_1 \otimes \cdots \otimes \mu'_n\}$$

(G – nondet)

This transition represents a situation where $n$ of the $p$ processors performed a local step. To apply this rule, we first partition the state into some $\mu_{\text{pause}}, \mu_1, \ldots, \mu_n$ where each of the $\mu_i$ can take a local transition.

Notice that this rule is highly non-deterministic. There may be many ways to partition the state into $n \leq p$ runnable sub-states. The decision about how we should perform this partitioning is known as *scheduling*. The overall progression of a parallel computation can be viewed as a *series-parallel* diagram like the one to the right, where the dots represent a local transition that must be made for the computation to complete. Any global state that could arise during evaluation corresponds to a cut through this graph. The cut labeled 1 to the right corresponds to a global state where there are four mappings from assignables to states. Three of these mappings (call them $a_1 \leftrightarrow s_1$, $a_2 \leftrightarrow s_2$, and $a_3 \leftrightarrow s_3$) would correspond to the states that are ready to make local transitions labeled o, g, and h, respectively. The transition corresponding to h involves forking off three subtasks (in our language the language can only fork off at most two new tasks, but the idea generalizes). The fourth mapping would be of the form $a_0 \leftrightarrow \text{join}[a_1, a_2, a_3](x_1, x_2, x_3, e)$, as it is a state awaiting the termination of these three other computations; the next local step that this state will take corresponds to the node labeled q.

A scheduler provides some way of mapping the work that needs to be done onto a fixed number of processors that are available to do work. *Greedy* schedulers are ones that always keep processors working if there’s available work to do. The most obvious greedy scheduling strategies that have been studied in practice are $p$-BFS and $p$-DFS (breadth-first and depth-first scheduling on $p$ processors). Breadth-first scheduling is a simple strategy that can be implemented with a queue: if there’s more available work than available parallelism, always prioritize one of the units of work that has been waiting longest to be scheduled. Depth-first scheduling always schedules the *leftmost* work from the perspective of the series-parallel diagram.

For the computation associated with the series-parallel diagram in Figure 2, Figure 3 gives an example of how scheduling would proceed under strict 2-DFS and 2-BFS scheduling.

**Task 5.1 (20%).** For each of the computations in Figure 4, give the order that work would be scheduled on 2 processors under both 2-BFS and 2-DFS.
<table>
<thead>
<tr>
<th>Round #</th>
<th>2-DFS Scheduling</th>
<th>2-BFS Scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Scheduled</td>
<td>Also ready</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>b, c</td>
<td>d</td>
</tr>
<tr>
<td>3</td>
<td>o, e</td>
<td>d</td>
</tr>
<tr>
<td>4</td>
<td>g, d</td>
<td>d</td>
</tr>
<tr>
<td>5</td>
<td>l, f</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>h</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>i, j</td>
<td>k</td>
</tr>
<tr>
<td>8</td>
<td>m, n</td>
<td>k</td>
</tr>
<tr>
<td>9</td>
<td>k</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>p</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>q</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3: Scheduling the diagram in Figure 2

Figure 4: Series-parallel diagrams for task 2.1
In general, $p$-BFS and $p$-DFS can have rather different performance depending on the shape of the computation; however, any greedy scheduler will take, at worst, twice the number of transitions that the best greedy scheduler would take. In practice, $p$-BFS and $p$-DFS scheduling (and more advanced variants, like work stealing schedulers, that we won’t consider here) are important because they make a huge difference in space usage; unfortunately, this is outside the scope of this assignment.

6 $p$-DFS Global Steps

Now that we’ve discussed some general approaches to scheduling, let’s consider how we formally encode $p$-DFS scheduling in our semantics. Because we have been careful about the ordering of tasks, we can specify $p$-DFS scheduling in terms of a few properties about the partitions that we use in global steps. Let $\mu_{\text{pause}} \otimes \mu_1 \otimes \cdots \otimes \mu_n$ be some partition of $\mu$ with $n \leq p$. Then we say this is a $p$-DFS partition if:

1. No substate of $\mu_{\text{pause}}$ is runnable.
2. Each $\mu_i$ is runnable.
3. For $i < n$, no strict right-substate of $\mu_i$ is runnable.
4. In the case that $n < p$, no strict right-substate of $\mu_n$ is runnable.

As long as every global transition we take uses a $p$-DFS partition, the result is a $p$-DFS scheduled execution. We will not prove that this matches $p$-DFS, but it is useful to think about why this would be the case. Intuitively, the scheduler is greedy because if $n < p$, then the above constraints require that there are no additional runnable substates we could add in. Moreover, such partitions select the first $n$ runnable tasks starting from the left, which is what we want in DFS.

**Task 6.1** (35%). Implement the function `partition` in `global_dynamics.sml`. This function should take in a memory $\mu$ and a number $p$ and return $(\mu_{\text{pause}}, [\mu_1, \ldots, \mu_n])$ where $\mu_{\text{pause}} \otimes \mu_1 \otimes \cdots \otimes \mu_n$ is a $p$-DFS partition of $\mu$.

**Task 6.2** (10%). Implement the function `ispDFS`. This should take in a state $\mu$, a number $p$, and a partition `part` and returns true if and only if `part` is a $p$-DFS partition of $\mu$. (according to the above specification).

**Task 6.3** (10%). Implement $p$-DFS global transitions in `trystep` in `global_dynamics.sml`. You should use the partition function you wrote in the previous task and the local transitions you implemented earlier.

A Putting The Code Together

As usual, you can compile your files using `CM.make "sources.cm"`.

Copyright © 2015, Carnegie Mellon University
A.1 Interpreter

This time, the interpreter is parameterized over the number of processors. To run the interpreter first do
dstructure TopLevel = MkToplevel (val p = 2);, replacing “2” with the number of processors you want to use. Then run TopLevel.repl (); as usual.

The syntax for each term construct is as close as possible to the concrete syntax mentioned for it. (The concrete syntax is the second column in the table which introduces the syntax for the language.) We provide below the grammar that the interpreter accepts.

 identifiers ::= (a letter followed by alphanumeric characters, _, or ‘ ’) numerals ::= (a series of digits *) string constants ::= (a standard C-escaped string like "foo" or "why \\
hello\" " *)

directives ::= eval <expression>;
| eval;
| step <expression>;
| step;
| load <expression>;
| use <string constant>;

types ::= nat | <type> -> <type> | unit | <type> * <type> | void | <type> + <type>

expressions ::= | z
| s <expression>
| <numeral>
| ifz <expression> { z => <expression> | s <ident> => <expression> }
| fn (<ident> : <type>) <expression>
| <>
| <<expression>, <expression>>
| <expression>.l
| <expression>.r
| abort[<type>] <expression>
| inl[<type>,<type>] <expression>
| inr[<type>,<type>] <expression>
| case e { inl <ident> => exp | inr <ident> => exp }
| match e { <rules> }
| let <declarations> in <expression> end
| fix <ident> : <type> is <expression>
| par <ident> = <expression> and <ident> = <expression> in <expression> end
| (<expression>)

declarations ::= <declaration> | <declaration> <declarations>

declaration ::= val <ident> = <expression>

A.2 Reference implementation

You can use the reference interpreter to test your code and help debug your interpreter.
The reference interpreter can be loaded like this:
$ rlwrap sml @SMLload=ref_impl
- structure TopLevel = MkTopLevel (val p = 3);
- TopLevel.repl ();
B  Local Dynamics

B.1  Values

There’s nothing particularly surprising about the values for this language.

\[
\begin{array}{cccc}
\text{lam}[\tau](x.e) & \text{val} & (\text{lam-val}) \\
\text{null} & \text{val} & (\text{triv-val}) \\
e_1 & \text{val} & (\text{pair-val}) \\
e_2 & \text{val} & (\text{z-val}) \\
\end{array}
\]

\[
\begin{array}{cccc}
\nu a, \{ \_ \}, \Sigma & \nu a, \{ \_ \}, \Sigma & \nu a, \{ \_ \}, \Sigma & \nu a, \{ \_ \}, \Sigma \\
\text{in} \{ \tau_1; \tau_2 \}[1](e) & \text{in} \{ \tau_1; \tau_2 \}[\mathtt{r}](e) & \text{in} \{ \tau_1; \tau_2 \}[\mathtt{l}](e) & \text{in} \{ \tau_1; \tau_2 \}[\mathtt{i}](e) \\
& & (\text{inl-val}) \\
& & (\text{inr-val}) \\
\end{array}
\]

B.2  Local Transitions

\[
\frac{\nu a, \Sigma \{ a \to s(e) \otimes \mu \} \rightarrow \text{loc} \nu a, a', \Sigma \{ a \to \text{join}[a'](x.s(x)) \otimes a' \rightarrow e \otimes \mu \}}{(L1)}
\]

\[
\frac{\nu a, \Sigma \{ a \to \text{fix}(e; e_0; x.e_1) \otimes \mu \} \rightarrow \text{loc} \nu a, a', \Sigma \{ a \to \text{join}[a'](x'. \text{fix}(x'; e_0; x.e_1)) \otimes a' \rightarrow e \otimes \mu \}}{(L2)}
\]

\[
\frac{\nu a, \Sigma \{ a \to (x.e) \otimes \mu \} \rightarrow \text{loc} \nu a, \Sigma \{ a \to e_0 \otimes \mu \}}{(L3)}
\]

\[
\frac{\nu a, \Sigma \{ a \to \text{let}(e_1; x.e_2) \otimes \mu \} \rightarrow \text{loc} \nu a, a', \Sigma \{ a \to \text{join}[a'](e_1.e_2) \otimes a' \rightarrow e_1 \otimes \mu \}}{(L4)}
\]

\[
\frac{\nu a, \Sigma \{ a \to \text{par}(e_1; e_2; x_1.x_2.e) \otimes \mu \} \rightarrow \text{loc} \nu a, a_1, a_2, \Sigma \{ a \to \text{join}[a_1](x_1, x_2.e) \otimes a_1 \rightarrow e_1 \otimes a_2 \rightarrow e_2 \otimes \mu \}}{(L5)}
\]

\[
\frac{\nu a, \Sigma \{ a \to e_1(e_2) \otimes \mu \} \rightarrow \text{loc} \nu a, a', \Sigma \{ a \to \text{join}[a'](x.e.e_2) \otimes a' \rightarrow e_1 \otimes \mu \}}{(L6)}
\]

\[
\frac{\nu a, \Sigma \{ a \to e_2(e_2) \otimes \mu \} \rightarrow \text{loc} \nu a, a', \Sigma \{ a \to \text{join}[a'](x.e_1(x)) \otimes a' \rightarrow e_2 \otimes \mu \}}{(L7)}
\]

\[
\frac{\nu a, \Sigma \{ a \to \text{fix}[\tau](x.e_1.e_2) \otimes \mu \} \rightarrow \text{loc} \nu a, \Sigma \{ a \to [e_2/x_1]e_1 \otimes \mu \}}{(L8)}
\]

\[
\frac{\nu a, \Sigma \{ a \to \text{lam}[\tau](x.e_1.e_2) \otimes \mu \} \rightarrow \text{loc} \nu a, \Sigma \{ a \to [e_2/x]e_1 \otimes \mu \}}{(L9)}
\]

\[
\frac{\nu a, \Sigma \{ a \to \text{fix}[\tau](x.e) \otimes \mu \} \rightarrow \text{loc} \nu a, \Sigma \{ a \to [\text{fix}[\tau](x.e)/x]e \otimes \mu \}}{(L10)}
\]

\[
\frac{\nu a, \Sigma \{ a \to \text{in} \{ \tau_1; \tau_2 \}[\mathtt{r}](e) \otimes \mu \} \rightarrow \text{loc} \nu a, a', \Sigma \{ a \to \text{join}[a'](x.\text{in} \{ \tau_1; \tau_2 \}[\mathtt{l}](x)) \otimes a' \rightarrow e \otimes \mu \}}{(L11)}
\]
\[ \neg (e \text{ val}) \]
\[
\nu a, \Sigma \{ a \leftarrow \text{in}[\tau_1; \tau_2][r](e) \otimes \mu \} \mapsto_{loc} \nu a, a', \Sigma \{ a \leftarrow \text{join}[a'][x.\text{in}[\tau_1; \tau_2][r](x)] \otimes a' \leftarrow e \otimes \mu \}
\]

\[ \neg (e \text{ val}) \]
\[
\nu a, \Sigma \{ a \leftarrow \text{case}(e; x_1.e_1; x_2.e_2) \otimes \mu \} \mapsto_{loc} \nu a, a', \Sigma \{ a \leftarrow \text{join}[a'][x.\text{case}(x; x_1.e_1; x_2.e_2)] \otimes a' \leftarrow e \otimes \mu \}
\]

\[ e \text{ val} \]
\[
\nu a, \Sigma \{ a \leftarrow \text{case(in}[\tau_1; \tau_2][l](e); x_1.e_1; x_2.e_2) \otimes \mu \} \mapsto_{loc} \nu a, \Sigma \{ a \leftarrow [e/x_1]e_1 \otimes \mu \}
\]

\[ e \text{ val} \]
\[
\nu a, \Sigma \{ a \leftarrow \text{case(in}[\tau_1; \tau_2][r](e); x_1.e_1; x_2.e_2) \otimes \mu \} \mapsto_{loc} \nu a, \Sigma \{ a \leftarrow [e/x_2]e_2 \otimes \mu \}
\]

\[ \neg (e_1 \text{ val}) \]
\[
\nu a, \Sigma \{ a \leftarrow \text{pair}(e_1; e_2) \otimes \mu \} \mapsto_{loc} \nu a, a', \Sigma \{ a \leftarrow \text{join}[a'][x.\text{pair}(x; e_2)] \otimes a' \leftarrow e_1 \otimes \mu \}
\]

\[ e_1 \text{ val} \]
\[
\nu a, \Sigma \{ a \leftarrow \text{pair}(e_1; e_2) \otimes \mu \} \mapsto_{loc} \nu a, a', \Sigma \{ a \leftarrow \text{join}[a'][x.\text{pair}(e_1; x)] \otimes a' \leftarrow e_2 \otimes \mu \}
\]

\[ \neg (e \text{ val}) \]
\[
\nu a, \Sigma \{ a \leftarrow \text{pr}[1](e) \otimes \mu \} \mapsto_{loc} \nu a, a', \Sigma \{ a \leftarrow \text{join}[a'][x.\text{pr}[1](x)] \otimes a' \leftarrow e \otimes \mu \}
\]

\[ e_1 \text{ val} \]
\[
\nu a, \Sigma \{ a \leftarrow \text{pr}[1](\text{pair}(e_1; e_2)) \otimes \mu \} \mapsto_{loc} \nu a, \Sigma \{ a \leftarrow e_1 \otimes \mu \}
\]

\[ \neg (e \text{ val}) \]
\[
\nu a, \Sigma \{ a \leftarrow \text{pr}[r](e) \otimes \mu \} \mapsto_{loc} \nu a, a', \Sigma \{ a \leftarrow \text{join}[a'][x.\text{pr}[r](x)] \otimes a' \leftarrow e \otimes \mu \}
\]

\[ e_1 \text{ val} \]
\[
\nu a, \Sigma \{ a \leftarrow \text{pr}[r](\text{pair}(e_1; e_2)) \otimes \mu \} \mapsto_{loc} \nu a, \Sigma \{ a \leftarrow e_2 \otimes \mu \}
\]

\[ \neg (e \text{ val}) \]
\[
\nu a, \Sigma \{ a \leftarrow \text{abort}[r](e) \otimes \mu \} \mapsto_{loc} \nu a, a', \Sigma \{ a \leftarrow \text{join}[a'][x.\text{abort}[r](x)] \otimes a' \leftarrow e \otimes \mu \}
\]

\[ e_1 \text{ val} \]
\[
\nu a, \Sigma \{ a \leftarrow \text{join}(a_1, \ldots, a_n)(x_1, \ldots, x_n,e) \otimes a_1 \leftarrow e_1 \otimes \ldots \otimes a_n \leftarrow e_n \otimes \mu \} \mapsto_{loc} \nu a, \Sigma \{ a \leftarrow [e_1, \ldots, e_n/x_1, \ldots, x_n]e \otimes \mu \}
\]