Stage-Splitting a Modal Language

Name1
Affiliation1
Email1

Name2 Name3
Affiliation2/3
Email2/3

Abstract

This is the text of the abstract.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms term1, term2

Keywords keyword1, keyword2

1. Introduction

We often find ourselves in situations where the input to a computation arrives in two parts, at different points in time.

The natural response is to try and do some part of the computation early, and the rest of the computation once the second input arrives.

Since the computation runs in two parts, the total cost can be split as well, into $m + n$.

Additionally, we can run the second stage multiple times. Cost is now $m + bn$.

Jorring et al. ([5]) identify three classes of staging techniques: meta-programming, partial evaluation, and stage-splitting. The first two of these have received significantly more attention than the third. Countless meta-programming systems exist (...twelve thousand citations...[3]), and their background theory and type-systems are well understood ([2]). Partial evaluation, too, is well-understood. Partial evaluation systems exist... . This paper explores both theory and applications for stage-splitting.

2. Stage-Splitting Definition and Comparison to Partial Evaluation

First, we review the definition of partial evaluation. Informally, a partial evaluator takes the code for a function f, as well as the first-stage input x to that function, and produces the code for a version of that function *specialized* to the first input, often called f_x. This f_x function can then be evaluated with the second-stage input to produce the same final answer that f would have. The goal of the process is that f_x should be cheaper to evaluate than f, although this can't be guaranteed for all inputs. We now state this theorem more formally: a partial evaluator is some function p such that,

$$\forall f, x. \exists f_x. [p(f, x) = f_x \text{ and } \forall y. [\![f]\!](x, y) = [\![f_x]\!](y)]$$

where (borrowing notation from [4]), $[\![f]\!]$ means the mathematical function corresponding to the code given by f.

Informally, we define stage-splitting to be the process of taking some function f into two other functions, f_1 and f_2, where f_1 computes a partial result from the first-stage input, and f_2 uses that partial result and the second-stage input to compute a final result which is the same as if we had just run the original f on both inputs. Again, more formally, a stage-splitter is some s such that,

$$\forall f. \exists f_1, f_2. [s(f) = (f_1, f_2) \text{ and } \forall x, y. [\![f]\!](x, y) = [\![f_2]\!]([\![f_1]\!](x), y)]$$

We first discuss a few similarities between partial-evaluation and stage-splitting. First off, both techniques have the same form of input, namely a bivariate function where the first input comes at stage one, and the second input comes at stage two.

Again in both cases, the governing equations are too weak to fully determine the definitions of p and s. Indeed, both admit completely trivial definitions. Consider the stage-splitter which always returns the identity for f_1 and f for f_2, or analogously the partial evaluator which always returns an f_x that just closes over the input x and internally calls f once y is available. The ambiguity of these equations (modulo standard program equivalence of the outputs) can be resolved by adding annotations to f to clearly specify the parts of the computation that are first stage and the parts that are second stage. Later, we show that the same annotations suffice for both partial evaluation and stage-splitting.

The differences between stage-splitting and partial evaluation are likewise evident from these governing equations. For instance in partial evaluation, the existential f_x depends on x, which means that the partial evaluator cannot be run until x is known. Moreover, if one wishes to specialize f for multiple x's, then the partial evaluator must be run several times. Depending on the use case and cost of partial evaluation, this may be prohibitively expensive. Alternatively, a stage-splitter need only be run once, and this can be done entirely before any x is known.

2.1 Partial Evaluator from Stage-Splitter

We can recover a valid partial evaluator from a stage-splitter by stage-splitting the input function f into f_1 and f_2, computing $[\![f_1]\!](x)$ to obtain $\bar{x}$, and then returning an f_x such that $[\![f_x]\!](y) = [\![f_2]\!](\bar{x}, y)$. Note that this does not mean that stage-splitting is a strict generalization of partial evaluation. In practice, partial evaluators easily perform optimizations (such as branch elimination, discussed later) which are beyond the scope of stage-splitting, and would require further technology than has been developed here. It is best to think of stage-splitting as simply the first half of partial evaluation, where the back half is an optimizer. [Might be able to come up with a futamura projection-like statement here, which would be really really really cool.]

2.2 Stage-Splitter from Partial Evaluator

Likewise, we can easily recover a stage-splitter from a partial evaluator. If p is a valid partial evaluator, then we can define a stage

$$
\begin{array}{rl}
w & ::= \mathbb{1} \mid \mathbb{2} \\
\tau & ::= \text{unit} \mid \text{int} \mid \text{bool} \mid \tau \times \tau \mid \tau + \tau \\
& \mid \tau \to \tau \mid \bigcirc \tau \mid \alpha \mid \mu\alpha.\tau \\
e & ::= () \mid i \mid b \mid x \mid \lambda x{:}\tau.e \mid e\ e \\
& \mid (e, e) \mid \texttt{pi1}\ e \mid \texttt{pi2}\ e \mid \texttt{inl}\ e \mid \texttt{inr}\ e \\
& \mid \texttt{case}\ e\ \texttt{of}\ x.e \mid x.e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
& \mid \texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{next}\ e \mid \texttt{prev}\ e \mid \texttt{hold}\ e \\
\Gamma & ::= \bullet \mid \Gamma, x : \tau\ @\ w
\end{array}
$$

Figure 1. $\lambda^{\mathbb{12}}$ Syntax

$$
\frac{}{\texttt{unit type} @ w}\ \text{unit} \qquad \frac{}{\texttt{int type} @ w}\ \text{int}
$$

$$
\frac{}{\texttt{bool type} @ w}\ \text{bool} \qquad \frac{A \text{ type} @ w \quad B \text{ type} @ w}{A \times B \text{ type} @ w}\ \times
$$

$$
\frac{A \text{ type} @ w \quad B \text{ type} @ w}{A + B \text{ type} @ w}\ +
$$

$$
\frac{A \text{ type} @ w \quad B \text{ type} @ w}{A \to B \text{ type} @ w}\ \to \qquad \frac{A \text{ type} @ \mathbb{2}}{\bigcirc A \text{ type} @ \mathbb{1}}\ \bigcirc
$$

Figure 2. $\lambda^{\mathbb{12}}$ Valid Types

$$
\frac{}{() \text{ val} @ w}\ \text{unit} \qquad \frac{}{i \text{ val} @ w}\ \text{int} \qquad \frac{}{b \text{ val} @ w}\ \text{bool}
$$

$$
\frac{v_1 \text{ val} @ w \quad v_2 \text{ val} @ w}{(v_1, v_2) \text{ val} @ w}\ \times \qquad \frac{v \text{ val} @ w}{\texttt{inl}\ v \text{ val} @ w}\ +_1
$$

$$
\frac{v \text{ val} @ w}{\texttt{inr}\ v \text{ val} @ w}\ +_2 \qquad \frac{}{\lambda x{:}A.e \text{ val} @ w}\ \to
$$

$$
\frac{}{\Xi \vdash \hat{y} \text{ val} @ \mathbb{1}}\ \bigcirc
$$

Figure 3. $\lambda^{\mathbb{12}}$ Valid Values

splitter s such that $s(f) = (f_1, f_2)$, where

$$[\![f_1]\!](x) = p(f, x)$$
$$[\![f_2]\!](l, y) = [l](y)$$

This implicitly requires that the languages in which f_1 and f_2 are expressed are strong enough to write a partial evaluator, but that is the case in this paper. A stage-splitter defined this way leaves much to be desired. Firstly, partial evaluation of f may be too expensive for the context in which f_1 needs to run. Additionally, the intermediate data structure created this way may be much larger than necessary, as it would contain all of the residual code.

3. A Two-Stage Modal Language

Every expression in $\lambda^{\mathbb{12}}$ can be understood as having a *stage*, either $\mathbb{1}$ or $\mathbb{2}$, in addition to a type. Just as a type describes *which* values an expression reduces to, the stage tells us *when* it will be fully reduced. The reason for a two-stage design (as opposed to [1], which allows an infinite number of stages) will be discussed later.

The only types that exist at stage $\mathbb{2}$ are products, sums, functions, and base types. In addition to this usual set, stage $\mathbb{1}$ also has one more type: $\bigcirc A$, where A is a stage-$\mathbb{2}$ type. This can be understood as *promising* a value of type A at stage $\mathbb{2}$. We'll see shortly that the promise given by $\bigcirc A$ can only be treated opaquely at stage one; it cannot be redeemed for an actual A until the second stage.

Although there is a strict one-way dependence between the two stages at the type level, their terms are defined mutually recursively: `next` embeds stage-$\mathbb{2}$ expressions into stage $\mathbb{1}$, whereas `prev` embeds stage-$\mathbb{1}$ expressions into stage $\mathbb{2}$. These `next` and `prev` constructs are the only expressions that interact with the stage of a term, and we surround their arguments with {braces} in our concrete syntax to clearly indicate the stage boundaries with in a program. The problem of how to turn an unstaged program into a staged one by inserting these constructs is called *binding time analysis*. We do not consider that problem here, and instead assume that it has been solved either automatically or by a programmer.

The grammar and type system for our language, $\lambda^{\mathbb{12}}$, are given in 1 and 4. We formulated our typing judgments in the style of [1], where the whole judgment is annotated with a stage. Variables in the context are annotated with a stage and must be used at the same stage at which they were declared. All of the non-stage constructs have the expected type-level behavior. This is made manifest as rules which are entirely abstract over stage.

In addition to determining the stage, `next` and `prev` are the introduction and elimination forms for $\bigcirc$ types. Specifically, `next`, given an argument with type A at stage $\mathbb{2}$, forms a $\bigcirc A$ at stage $\mathbb{1}$. That is, it forms the promise of a future A out of a construction for an A at the next timestep. This promise can only be redeemed be the `prev` construct. Since `prev` operates at stage $\mathbb{2}$, this ensures no violation of causality.

The `hold` construct is a mechanism to wrap stage $\mathbb{1}$ integers so that they can be used at stage $\mathbb{2}$. Although `hold` can be implemented from our other features, we provide it as a primitive to simplify our examples.

3.1 Staged Fast Exponentiation

Fast exponentiation is a method for calculating b^p in $\log p$ time:

$$
\mathit{fexp}(b, p) = \begin{cases} 1 & p = 0 \\ \mathit{fexp}(b, p/2)^2 & p \text{ even} \\ b \cdot \mathit{fexp}(b, p-1) & p \text{ odd} \end{cases}
$$

It is often observed that the decomposition of the power depends only on the power at prior recursive levels. For this reason, we can stage an implementation of fast exponentiation so that the power is available at the first stage and the base is available at the second stage. This implies that the result is then available only at the second stage.

We implemented staged fast exponentiation in $\lambda^{\mathbb{12}}$ below:

```
let fexp (b : $int, e : int) : $int =
    if e == 0 then
        next{1}
    else if (e mod 2) == 0 then
        next{let x = prev{fexp(b,e/2)} in x*x}
    else
        next{prev{b} * prev{fexp(b,e-1)}}
```

The type of the recursive function is $\bigcirc\text{int} \times \text{int} \to \bigcirc\text{int}$. Note that the type system is sufficient to prove the observation that all of the decomposition of the exponent can be done at the first stage.

Given this definition of fast exponent, the question the remains how it can be run. For this, we need a semantics for $\lambda^{\mathbb{12}}$.

$$\frac{\cdot}{\Gamma \vdash () : \mathrm{unit} \mathbin{@} w}\ \mathrm{unit} \qquad \frac{\cdot}{\Gamma \vdash i : \mathrm{int} \mathbin{@} w}\ \mathrm{int} \qquad \frac{\cdot}{\Gamma \vdash b : \mathrm{bool} \mathbin{@} w}\ \mathrm{bool} \qquad \frac{x : A \mathbin{@} w \in \Gamma}{\Gamma \vdash x : A \mathbin{@} w}\ \mathrm{hyp}$$

$$\frac{A\ \mathrm{type} \mathbin{@} w \quad \Gamma, x : A \mathbin{@} w \vdash e : B \mathbin{@} w}{\Gamma \vdash \lambda x{:}A.e : A \to B \mathbin{@} w} \to \mathrm{I} \qquad \frac{\Gamma \vdash e_1 : A \to B \mathbin{@} w \quad \Gamma \vdash e_2 : A \mathbin{@} w}{\Gamma \vdash e_1\ e_2 : B \mathbin{@} w} \to \mathrm{E}$$

$$\frac{\Gamma \vdash e_1 : A \mathbin{@} w \quad \Gamma, x : A \mathbin{@} w \vdash e_2 : B \mathbin{@} w}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : B \mathbin{@} w}\ \mathrm{let} \qquad \frac{\Gamma \vdash e_1 : A \mathbin{@} w \quad \Gamma \vdash e_2 : B \mathbin{@} w}{\Gamma \vdash (e_1, e_2) : A \times B \mathbin{@} w} \times \mathrm{I} \qquad \frac{\Gamma \vdash e : A \times B \mathbin{@} w}{\Gamma \vdash \mathtt{pi1}\ e : A \mathbin{@} w} \times \mathrm{E}_1$$

$$\frac{\Gamma \vdash e : A \times B \mathbin{@} w}{\Gamma \vdash \mathtt{pi2}\ e : B \mathbin{@} w} \times \mathrm{E}_2 \qquad \frac{\Gamma \vdash e : A \mathbin{@} w}{\Gamma \vdash \mathtt{inl}\ e : A + B \mathbin{@} w} + \mathrm{I}_1 \qquad \frac{\Gamma \vdash e : B \mathbin{@} w}{\Gamma \vdash \mathtt{inr}\ e : A + B \mathbin{@} w} + \mathrm{I}_2$$

$$\frac{\Gamma \vdash e_1 : A + B \mathbin{@} w \quad \Gamma, x_2 : A \mathbin{@} w \vdash e_2 : C \mathbin{@} w \quad \Gamma, x_3 : B \mathbin{@} w \vdash e_3 : C \mathbin{@} w}{\Gamma \vdash \mathtt{case}\ e_1\ \mathtt{of}\ x_2.e_2 \mid x_3.e_3 : C \mathbin{@} w} + \mathrm{E}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{bool} \mathbin{@} w \quad \Gamma \vdash e_2 : A \mathbin{@} w \quad \Gamma \vdash e_3 : A \mathbin{@} w}{\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : A \mathbin{@} w}\ \mathrm{if} \qquad \frac{\Gamma \vdash e : A \mathbin{@} \mathbb{2}}{\Gamma \vdash \mathtt{next}\ e : \bigcirc A \mathbin{@} \mathbb{1}} \bigcirc \mathrm{I} \qquad \frac{\Gamma \vdash e : \bigcirc A \mathbin{@} \mathbb{1}}{\Gamma \vdash \mathtt{prev}\ e : A \mathbin{@} \mathbb{2}} \bigcirc \mathrm{E}$$

$$\frac{\Gamma \vdash e : \mathrm{int} \mathbin{@} \mathbb{1}}{\Gamma \vdash \mathtt{hold}\ e : \bigcirc \mathrm{int} \mathbin{@} \mathbb{2}}\ \mathrm{hold}$$

Figure 4. $\lambda^{\mathbb{12}}$ Static Semantics

$$\frac{\cdot}{\Xi; \Gamma \vdash \lambda x{:}A.e \Downarrow_{\mathbb{1}} [\cdot, \lambda x.e]} \to \mathrm{I} \Downarrow$$

$$\frac{\Xi; \Gamma \vdash e_1 \Downarrow_{\mathbb{1}} [\xi_1, \lambda x.e'] \quad \Xi; \Gamma \vdash e_2 \Downarrow_{\mathbb{1}} [\xi_2, v_2] \quad \Xi, \mathsf{dom}(\xi_1), \mathsf{dom}(\xi_2); \Gamma \vdash [v_2/x]e' \Downarrow_{\mathbb{1}} [\xi', v']}{\Xi; \Gamma \vdash e_1\ e_2 \Downarrow_{\mathbb{1}} [\xi_1 \circ \xi_2 \circ \xi', v']} \to \mathrm{E} \Downarrow$$

$$\frac{\cdot}{\Xi; \Gamma \vdash () \Downarrow_{\mathbb{1}} [\cdot, ()]}\ \mathrm{unit} \Downarrow \qquad \frac{\Xi; \Gamma \vdash e_1 \Downarrow_{\mathbb{1}} [\xi_1, v_1] \quad \Xi; \Gamma \vdash e_2 \Downarrow_{\mathbb{1}} [\xi_2, v_2]}{\Xi; \Gamma \vdash (e_1, e_2) \Downarrow_{\mathbb{1}} [\xi_1 \circ \xi_2, (v_1, v_2)]} \times \mathrm{I} \Downarrow \qquad \frac{\Xi; \Gamma \vdash e \Downarrow_{\mathbb{1}} [\xi, (v_1, v_2)]}{\Xi; \Gamma \vdash \mathtt{pi1}\ e \Downarrow_{\mathbb{1}} [\xi, v_1]} \times \mathrm{E}_1 \Downarrow$$

$$\frac{\Xi; \Gamma \vdash e \Downarrow_{\mathbb{1}} [\xi, (v_1, v_2)]}{\Xi; \Gamma \vdash \mathtt{pi2}\ e \Downarrow_{\mathbb{1}} [\xi, v_2]} \times \mathrm{E}_2 \Downarrow \qquad \frac{\Xi; \Gamma \vdash e \Downarrow_{\mathbb{1}} [\xi, v]}{\Xi; \Gamma \vdash \mathtt{inl}\ e \Downarrow_{\mathbb{1}} [\xi, \mathtt{inl}\ v]} + \mathrm{I}_1 \Downarrow \qquad \frac{\Xi; \Gamma \vdash e \Downarrow_{\mathbb{1}} [\xi, v]}{\Xi; \Gamma \vdash \mathtt{inr}\ e \Downarrow_{\mathbb{1}} [\xi, \mathtt{inr}\ v]} + \mathrm{I}_2 \Downarrow$$

$$\frac{\Xi; \Gamma \vdash e_1 \Downarrow_{\mathbb{1}} [\xi_1, \mathbf{inl}\ v] \quad \Xi, \mathsf{dom}(\xi_1); \Gamma \vdash [v/x_2]e_2 \Downarrow_{\mathbb{1}} [\xi_2, v_2]}{\Xi; \Gamma \vdash \mathtt{case}\ e_1\ \mathtt{of}\ x_2.e_2 \mid x_3.e_3 \Downarrow_{\mathbb{1}} [\xi_1 \circ \xi_2, v_2]} + \mathrm{E}_1 \Downarrow$$

$$\frac{\Xi; \Gamma \vdash e_1 \Downarrow_{\mathbb{1}} [\xi_1, \mathbf{inr}\ v] \quad \Xi, \mathsf{dom}(\xi_1); \Gamma \vdash [v/x_3]e_3 \Downarrow_{\mathbb{1}} [\xi_3, v_3]}{\Xi; \Gamma \vdash \mathtt{case}\ e_1\ \mathtt{of}\ x_2.e_2 \mid x_3.e_3 \Downarrow_{\mathbb{1}} [\xi_1 \circ \xi_3, v_3]} + \mathrm{E}_2 \Downarrow \qquad \frac{\cdot}{\Xi; \Gamma \vdash i \Downarrow_{\mathbb{1}} [\cdot, i]}\ \mathrm{int} \Downarrow \qquad \frac{\cdot}{\Xi; \Gamma \vdash b \Downarrow_{\mathbb{1}} [\cdot, b]}\ \mathrm{bool} \Downarrow$$

$$\frac{\Xi; \Gamma \vdash e_1 \Downarrow_{\mathbb{1}} [\xi_1, v_1] \quad \Xi, \mathsf{dom}(\xi_1); \Gamma \vdash [v_1/x]e_2 \Downarrow_{\mathbb{1}} [\xi_2, v_2]}{\Xi; \Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \Downarrow_{\mathbb{1}} [\xi_1 \circ \xi_2, v_2]}\ \mathrm{let} \Downarrow \qquad \frac{\Xi; \Gamma \vdash e_1 \Downarrow_{\mathbb{1}} [\xi_1, \mathbf{true}] \quad \Xi; \Gamma \vdash e_2 \Downarrow_{\mathbb{1}} [\xi_2, v]}{\Xi; \Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \Downarrow_{\mathbb{1}} [\xi_1 \circ \xi_2, v]}\ \mathrm{if_T} \Downarrow$$

$$\frac{\Xi; \Gamma \vdash e_1 \Downarrow_{\mathbb{1}} [\xi_1, \mathbf{false}] \quad \Xi; \Gamma \vdash e_3 \Downarrow_{\mathbb{1}} [\xi_3, v]}{\Xi; \Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \Downarrow_{\mathbb{1}} [\xi_1 \circ \xi_3, v]}\ \mathrm{if_F} \Downarrow$$

Figure 5. $\lambda^{\mathbb{12}}$ Dynamic Semantics: Core

$$\frac{\Xi;\Gamma \vdash e \downarrow q}{\Xi;\Gamma \vdash \texttt{next}\ e \Downarrow_{\mathbb{1}} [z \mapsto \texttt{let}\ y = q\ \texttt{in}\ z, \hat{y}]}\ \bigcirc\text{I} \Downarrow \qquad \frac{\Xi;\Gamma \vdash e \Downarrow_{\mathbb{1}} [\xi, \hat{y}] \quad [\xi, y] \overset{R}{\Rightarrow} q}{\Xi;\Gamma \vdash \texttt{prev}\ e \downarrow q}\ \bigcirc\text{E} \downarrow$$

$$\frac{\Xi;\Gamma \vdash e \Downarrow_{\mathbb{1}} [\xi, i]}{\Xi;\Gamma \vdash \texttt{hold}\ e \Downarrow_{\mathbb{1}} [z \mapsto \xi(\texttt{let}\ y = i\ \texttt{in}\ z), \hat{y}]}\ \text{hold} \Downarrow \qquad \frac{\cdot}{\Xi;\Gamma \vdash \hat{y} \Downarrow_{\mathbb{1}} [\cdot, \hat{y}]}\ \text{hattedvar} \Downarrow \qquad \frac{[\xi, q_2] \overset{R}{\Rightarrow} q'}{[(y \mapsto q_1) \circ \xi, q_2] \overset{R}{\Rightarrow} \texttt{let}\ y = q_1\ \texttt{in}\ q'}$$

$$\frac{\cdot}{[\cdot, q] \overset{R}{\Rightarrow} q}$$

Figure 6. $\lambda^{\mathbb{1}2}$ Dynamic Semantics: `next` and `prev`)

$$\frac{\Xi;\Gamma, x \vdash e \downarrow q}{\Xi;\Gamma \vdash \lambda x{:}A.e \downarrow \lambda x{:}A.q} \to \text{I} \downarrow$$

$$\frac{\Xi;\Gamma \vdash e_1 \downarrow q_1 \quad \Xi;\Gamma \vdash e_2 \downarrow q_2}{\Xi;\Gamma \vdash e_1\ e_2 \downarrow q_1\ q_2} \to \text{E} \downarrow$$

$$\frac{\cdot}{\Xi;\Gamma \vdash () \downarrow ()}\ \text{unit} \downarrow \qquad \frac{\Xi;\Gamma \vdash e_1 \downarrow q_1 \quad \Xi;\Gamma \vdash e_2 \downarrow q_2}{\Xi;\Gamma \vdash (e_1, e_2) \downarrow (q_1, q_2)} \times\text{I} \downarrow$$

$$\frac{\Xi;\Gamma \vdash e \downarrow q}{\Xi;\Gamma \vdash \texttt{pi1}\ e \downarrow \texttt{pi1}\ q} \times\text{E}_1 \downarrow \qquad \frac{\Xi;\Gamma \vdash e \downarrow q}{\Xi;\Gamma \vdash \texttt{pi2}\ e \downarrow \texttt{pi2}\ q} \times\text{E}_2 \downarrow$$

$$\frac{\Xi;\Gamma \vdash e \downarrow q}{\Xi;\Gamma \vdash \texttt{inl}\ e \downarrow \texttt{inl}\ q} +\text{I}_1 \downarrow \qquad \frac{\Xi;\Gamma \vdash e \downarrow q}{\Xi;\Gamma \vdash \texttt{inr}\ e \downarrow \texttt{inr}\ q} +\text{I}_2 \downarrow$$

$$\frac{\Xi;\Gamma \vdash e_1 \downarrow q_1 \quad \Xi;\Gamma, x_2 \vdash e_2 \downarrow q_2 \quad \Xi;\Gamma, x_3 \vdash e_3 \downarrow q_3}{\Xi;\Gamma \vdash \texttt{case}\ e_1\ \texttt{of}\ x_2.e_2 \mid x_3.e_3 \downarrow \texttt{case}\ q_1\ \texttt{of}\ x_2.q_2 \mid x_3.q_3} +\text{E}_1 \downarrow$$

$$\frac{\cdot}{\Xi;\Gamma \vdash i \downarrow i}\ \text{int} \downarrow \qquad \frac{\cdot}{\Xi;\Gamma \vdash b \downarrow b}\ \text{bool} \downarrow$$

$$\frac{\Xi;\Gamma \vdash e_1 \downarrow q_1 \quad \Xi;\Gamma, x \vdash e_2 \downarrow q_2}{\Xi;\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \downarrow \texttt{let}\ x = q_1\ \texttt{in}\ q_2}\ \text{let} \downarrow$$

$$\frac{\Xi;\Gamma \vdash e_1 \downarrow q_1 \quad \Xi;\Gamma \vdash e_2 \downarrow q_2 \quad \Xi;\Gamma \vdash e_3 \downarrow q_3}{\Xi;\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \downarrow \texttt{if}\ q_1\ \texttt{then}\ q_2\ \texttt{else}\ q_3}\ \text{if}_\text{T} \downarrow$$

Figure 7. $\lambda^{\mathbb{1}2}$ Dynamic Semantics: Speculation

3.2 Semantics

In this section, we discuss the dynamic semantics of our language. Before we can get into the details of the semantics, we have to first make some decisions about the behavior we want it to have. Similar to [1], we want our semantics to have the form of a judgment that partially evaluates an expression to create a residual that can be evaluated at the second stage in the standard way. Again like Davies, we choose a call-by-value paradigm as our baseline at both stages. The only significant freedom that this leaves is how to handle the staging constructs.

We present here two reasonable interpretations of these staging constructs, in the context of the following stage-$\mathbb{2}$ example:

```
prev{
let x = (next {4+5}, 7+8) in
next{
    prev{#1 x} * prev{#1 x} * prev{hold (#2 x)}
}
}
```

The first option is to duplicate the contents of the first `next` expression to produce,

```
(4+5) * (4+5) * 15
```

Essentially, we're treating `next` as a value a stage-$\mathbb{1}$, so long as all of its contained `prev`s are reduced. This is the technique of [1]. Instead of this, we take a second, nearly equivalent approach. That is to bind the contents of the first `next` expression to some variable (here, $x1$) and then duplicate that variable,

```
let x1 = 4+5 in
x1 * x1 * 15
```

We think that this interpretation makes more sense from the point of view of cost, since the contents of a `next` always appear once in the residual, regardless of how many times the result is used. This is *nearly* equivalent to the first expression, because they have different termination behavior when a non-terminating stage-$\mathbb{2}$ expression is declared but never used.

Now that we know *what* the behavior is that we want, we can explain *how* our semantics achieves that behavior. We start with the same example:

```
prev {
let x = (next {4+5}, 7+8) in
next{
    prev{#1 x} * prev{#1 x} * prev{hold (#2 x)}
}
}
```

The first thing we need to do is evaluate the binding for x. Of course, we can't evaluate inside the `next` statement, but we still need to produce a value that will get substituted. What we do instead is replace the body of the `next` by a variable and make an explicit substitution, which then floats up to the top of the containing `prev`:

```
prev{
[4+5/x1]
let x = (next{x1}, 15) in
next{
    prev{#1 x} * prev{#1 x} * prev{hold (#2 x)}
}
}
```

We're now free to perform the stage-$\mathbb{1}$ substitution for x.

```
prev{
[4+5/x1]
next{
    prev{#1 (next{x1}, 15)}
    * prev{#1 (next{x1}, 15)}
    * prev{hold (#2 (next{x1}, 15))}
```

```
}
}
```

To evaluate the remaining `next`, we must first partially evaluate the body by finding all of the contained stage-$\mathbb{1}$ terms and reducing them:

```
prev{
[4+5/x1]
    next{ x1 * x1 * 15 }
}
```

The final step in evaluating the outer `prev` is reifying the contained substitutions into let statements, yielding

```
let x1 = 4+5 in
 x1 * x1 * 15
```

Once again, the point here is that we did not duplicate the $4 + 5$ in our residual. Instead we lifted it out into a substitution, leaving a hatted variable [note: add hats above] in its place, and we reified the substitution into a let-binding at the enclosing `prev`.

3.3 more formally...

Abstractly, we can think of our evaluation as proceeding in the standard way for stage-1 code. When the evaluator encounters a `next`$\{e\}$ expression, it places e off to the side in a context and replaces the whole expression with a reference to the context entry. These references are then passed around as stage-1 values for $\bigcirc$ types. But what if e contains `prev` expressions? To ensure that all stage-1 code is evaluated before any stage-2 code, we must evaluate all of the 1-code contained in e before inserting it into the table. This entails searching e for all contained `prev`s and evaluating them in place.

More formally, evaluation comprises the following judgments:

Judgment	Reads	Conditions
$\Xi \vdash e$ val $@\ \mathbb{1}$	"e is a $\mathbb{1}$-value under context Ξ"	...
$\Xi; \Gamma \vdash e \Downarrow_{\mathbb{1}} [\xi, v]$	"e evaluates to future table ξ and value v"	$\Gamma \vdash e : A\ @\ \mathbb{1}$ $\Xi \vdash v$ val $@\ \mathbb{1}$ ...
$\Xi; \Gamma \vdash e \downarrow q$	"e speculates to q"	...
...	...	...

Evaluation and speculation are the main judgments here, the rest being largely administrative. The evaluation judgment operates on stage-1 code, whereas the speculation operates on stage-2 code. Since `next` and `prev` are the crossover points between 1-code and 2-code, they are correspondingly the only places where the evaluation and speculation judgments depend on the other.

The evaluation judgment is very similar to standard call-by-value evaluation, as goal 1 would suggest. The input to evaluation is a stage-1 expression (usually e), as well as two administrative contexts (Γ and Ξ), covered later. Evaluation has two outputs: the *future context* (usually ξ) and the *partial value* (usually v). We cover the latter first. The partial value is essentially the result of the first-stage portion of e, and must be a $\mathbb{1}$-value of the same type as e. As usual, this means that v must be composed only of base primitives, tuples, injections, and lambdas (corresponding to base types, products, sums, and functions). Analogously, the value corresponding to $\bigcirc$ is a construct called a *hatted variable* (written $\hat{y}$), which signifies a reference to some stage-2 computation. The mapping between these hatted variables and stage-two expressions is

...

3.4 Other nice properties

1. For some $e : A\ @\ \mathbb{1}$ containing no `next` expressions, the semantics is derivationally equivalent to standard call-by-value evaluation.

3.5 Metatheory

TODO: unfinished; where should this go? combine with table above
TODO: terminology for Ξ, Γ, ξ, v, q?
TODO: need to define 1-vals at 1 and 2, and 2-vals at 2. what is the terminology? (?, residuals, results?)
TODO: put Ξ and Γ in the judgments where appropriate
TODO: be uniform about which judgments have which contexts, etc

Definition 3.1. *Contexts Ξ and Γ are well-formed (Ξ wf, Γ wf) if they contain only stage-2 variables.*

TODO: don't call it a table
TODO: what is the notation for ξs? how do you build them?
TODO: define reification explicitly as a metafunction

Definition 3.2. *A table ξ is well-formed (ξ wf) if either:*

1. $\xi = \cdot$; *or*
2. $\xi = \xi', x \mapsto e$ *where* $\Xi, \mathsf{dom}(\xi'); \Gamma \vdash e : B@\mathbb{2}$ *and* $\Xi, \mathsf{dom}(\xi'); \Gamma \vdash e\ \mathbb{1}\text{-}\mathsf{val}@\mathbb{2}$.

Theorem 3.3. *If $\Xi; \Gamma \vdash e : A\ @\ \mathbb{1}$ then Ξ wf, Γ wf, and A type $@\ \mathbb{1}$.*

Theorem 3.4. *If $\Xi; \Gamma \vdash e \Downarrow_{\mathbb{1}} [\xi, v]$ then:*

1. $\Gamma \vdash e : A\ @\ \mathbb{1}$ *for some 1-type A;*
2. ξ wf;
3. $\Xi, \mathsf{dom}(\xi); \cdot \vdash v : A@\mathbb{1}$; *and*
4. $\Xi, \mathsf{dom}(\xi); \cdot \vdash v\ \mathbb{1}\text{-}\mathsf{val}@\mathbb{1}$.

Theorem 3.5. *If $\Xi; \Gamma \vdash e \downarrow q$ then:*

1. $\Gamma \vdash e : A\ @\ \mathbb{2}$ *for some 2-type A;*
2. $\Xi; \Gamma \vdash q : A@\mathbb{2}$; *and*
3. $\Xi; \Gamma \vdash q\ \mathbb{2}\text{-}\mathsf{val}@\mathbb{2}$.

4. Splitting Algorithm

[Present the splitting judgement. Give statements of type and value correctness for splitting. Give all of the splitting rules. Talk through a few of them.]

5. Examples for Staged Pipelines

Give the gist of one-to-one pipeline example (like client/server). Then talk about a one-to-many pipeline. Then talk about a many-to-one pipeline like spark. It clear how to target something like this for known base types on the boundary, and for product types. But sums on the boundary are hard! We leave many-to-one as future-work.

6. Examples of Algorithm Derivation

Fast exponent example.

```
let exp (b : $int, e : int) : $int =
    if e == 0 then
        next{1}
    else if (e mod 2) == 0 then
        next{let x = prev{exp(b,e/2)} in x*x}
    else
        next{prev{b} * prev{exp (b,e-1)}}
```

splits into

```
let exp (b, e) =
    ((), roll (
        if e == 0 then
            inL ()
        else
            inR (
                if (e mod 2) == 0 then
                    inL (#2 (exp (b,e/2)))
                else
                    inR (#2 (exp (b,e-1)))
            )
    ))
```

and

```
let exp ((b, e), p) =
    case unroll p of
      () => 1
    | d =>
        case d of
          r => let x = exp ((b,()),r) in x*x
        | r => b * exp ((b,()),r)
```

Quickselect example.

```
let qs (l : μα.() + int * α, i: $int) =
    case unroll l of
      () => next {0}
    | (h,t) =>
        let (left,right,n) = partition h t in
        next{
            let n = prev{hold n} in
            case compare prev{i} n of
              () (*LT*) => prev {qs left i}
            | () (*EQ*) => prev {hold h}
            | () (*GT*) =>
                prev {qs right next{prev{i}-n-1}}
        }
```

Things to try: an interpreter which, partially evaluated, does cps or something.

For each of these examples, talk about what partial evaluation would do and why that might be bad.

[Meta-ML eases off on this restriction but does not (I think?) eliminate it.]

[What's going on with names and necessity?]

[Our work bears a lot of similarity to ML5, which also uses a modal type system. The difference is that we target stages systems (each stage talks to the next), whereas they target distributed ones (all stages talk to all others). The type systems reflect this directly in the world accessibility relation. There might be some analogue of stage-splitting in the ML5 work, but I have not yet isolated it (might be buried in CPS conversion).]

7. Related Work

Our stage-splitting algorithm was first suggested in [5] under the name *pass separation*. They essentially proposed that a function f could be split into two others, f_1 and f_2, such that $f(x, y) = f_2(f_1(x), y)$. They did not distinguish binding time analysis from stage splitting, and so pass separation inherits the former's ambiguity. The main goal of [5] was to motivate pass separation and other staging transformations as a powerful way to think about compilation and optimization. Accordingly, their approach was entirely informal, with no implementation realized. Moreover, they predicted that "the [pass separation] approach will elude full automation for some time."

Implementations of the stage-splitting algorithm have appeared in the literature exclusively (and coincidentally) in the context of graphics pipelines. The first of these ([6]) uses a binding time analysis to separate those parts of graphics shaders that are input-invariant from those which are not, and then uses a stage splitting algorithm to factor that into two shaders, thereby minimizing recomputation. Their shaders are written in a C-like language with basic arithmetic and if statements. Although their analysis does not give an explicit account of the type-level behavior of the splitting algorithm, it effectively can synthesize product and sum boundary type.

Like the previous example, the Spark language ([**?**]) uses staging to minimize recomputation in real-time rendering applications. But instead of using a binding-time analysis, Spark allows the programmer to manually target stages of the graphics pipeline. Since the modern graphics pipeline is inherently a many-to-one system, this is difficult to reconcile with sum types on the boundary. Fortunately, Spark has a set of syntactic restrictions that prevent sum boundary types. Spark does not clearly identify this conflict, but the authors did note that first-stage if statements were difficult to provide meaning to [need quote].

[RTSL and SH]

[Discuss Yong's recent paper here. It does some pretty sophisticated binding time analysis, with a somewhat straightforward splitting after that. They have the same many-to-one use case as Spark, but syntactic restrictions prevent sum types on the boundary, sort of. If we wanted to faithfully represent their system in ours, we would need some mechanisms for abstraction over stage, which we do not have.]

Davies ([1]) explored the connection between linear temporal logic and its corresponding type system [circle](which we adapted into [circle sub 2]), and showed the equivalence between [circle] and existing systems for binding time analysis. That work provided β and η rules for the next and prev operators, but did not consider a full dynamic semantics for the whole language. Whereas [name of our type system] is appropriate for stage-splitting and partial evaluation, [2] provides a similar system, [square], that is appropriate for meta-programming. The main difference is that terms inside a [prev] operator do not see any stage-2 bindings declared outside of

$$\frac{\cdot}{\Gamma \vdash () : \mathrm{unit} \overset{\mathbb{1}}{\rightsquigarrow} [((),()), \_.()]}\ \mathrm{unit}\ \overset{\mathbb{1}}{\rightsquigarrow} \qquad \frac{\cdot}{\Gamma \vdash () : \mathrm{unit} \overset{\mathbb{2}}{\rightsquigarrow} [(), \_.()]}\ \mathrm{unit}\ \overset{\mathbb{2}}{\rightsquigarrow} \qquad \frac{\cdot}{\Gamma \vdash i : \mathrm{int} \overset{\mathbb{1}}{\rightsquigarrow} [(i,()), \_.()]}\ \mathrm{int}\ \overset{\mathbb{1}}{\rightsquigarrow}$$

$$\frac{\cdot}{\Gamma \vdash i : \mathrm{int} \overset{\mathbb{2}}{\rightsquigarrow} [(), \_.i]}\ \mathrm{int}\ \overset{\mathbb{2}}{\rightsquigarrow}$$

$$\frac{\Gamma \vdash e_1 : A \overset{\mathbb{1}}{\rightsquigarrow} [c_1, l_1.r_1] \quad \Gamma \vdash e_2 : B \overset{\mathbb{1}}{\rightsquigarrow} [c_2, l_2.r_2]}{\Gamma \vdash (e_1, e_2) : A \times B \overset{\mathbb{1}}{\rightsquigarrow} \left[\begin{array}{l} (\mathtt{let}\ (y_1, z_1) = c_1\ \mathtt{in}\ \mathtt{let}\ (y_2, z_2) = c_2\ \mathtt{in}\ ((y_1, y_2), (z_1, z_2)))\ , \\ (l_1, l_2).(r_1, r_2) \end{array}\right]}\ \times\mathrm{I}\ \overset{\mathbb{1}}{\rightsquigarrow}$$

$$\frac{\Gamma \vdash e_1 : A \overset{\mathbb{2}}{\rightsquigarrow} [p_1, l_1.r_1] \quad \Gamma \vdash e_2 : B \overset{\mathbb{2}}{\rightsquigarrow} [p_2, l_2.r_2]}{\Gamma \vdash (e_1, e_2) : A \times B \overset{\mathbb{2}}{\rightsquigarrow} [(p_1, p_2), (l_1, l_2).(r_1, r_2)]}\ \times\mathrm{I}\ \overset{\mathbb{2}}{\rightsquigarrow}$$

$$\frac{\Gamma \vdash e : A \times B \overset{\mathbb{1}}{\rightsquigarrow} [c, l.r]}{\Gamma \vdash \mathtt{pi1}\ e : A \overset{\mathbb{1}}{\rightsquigarrow} [\mathtt{let}\ (y, z) = c\ \mathtt{in}\ (\mathtt{pi1}\ y, z), l.\mathtt{pi1}\ r]}\ \times\mathrm{E}_1\ \overset{\mathbb{1}}{\rightsquigarrow}$$

$$\frac{\Gamma \vdash e : A \times B \overset{\mathbb{1}}{\rightsquigarrow} [c, l.r]}{\Gamma \vdash \mathtt{pi2}\ e : B \overset{\mathbb{1}}{\rightsquigarrow} [\mathtt{let}\ (y, z) = c\ \mathtt{in}\ (\mathtt{pi2}\ y, z), l.\mathtt{pi2}\ r]}\ \times\mathrm{E}_2\ \overset{\mathbb{1}}{\rightsquigarrow} \qquad \frac{\Gamma \vdash e : A \times B \overset{\mathbb{2}}{\rightsquigarrow} [p, l.r]}{\Gamma \vdash \mathtt{pi1}\ e : A \overset{\mathbb{2}}{\rightsquigarrow} [p, l.\mathtt{pi1}\ r]}\ \times\mathrm{E}_1\ \overset{\mathbb{2}}{\rightsquigarrow}$$

$$\frac{\Gamma \vdash e : A \times B \overset{\mathbb{2}}{\rightsquigarrow} [p, l.r]}{\Gamma \vdash \mathtt{pi2}\ e : B \overset{\mathbb{2}}{\rightsquigarrow} [p, l.\mathtt{pi2}\ r]}\ \times\mathrm{E}_2\ \overset{\mathbb{2}}{\rightsquigarrow} \qquad \frac{x : A\ @\ \mathbb{1} \in \Gamma}{\Gamma \vdash v : A \overset{\mathbb{1}}{\rightsquigarrow} [(x,()), \_.()]}\ \mathrm{hyp}\ \overset{\mathbb{1}}{\rightsquigarrow} \qquad \frac{x : A\ @\ \mathbb{2} \in \Gamma}{\Gamma \vdash v : A \overset{\mathbb{2}}{\rightsquigarrow} [(), \_.x]}\ \mathrm{hyp}\ \overset{\mathbb{2}}{\rightsquigarrow}$$

$$\frac{\Gamma \vdash e_1 : A \overset{\mathbb{1}}{\rightsquigarrow} [c_1, l_1.r_1] \quad \Gamma, x : A\ @\ \mathbb{1} \vdash e_2 : B \overset{\mathbb{1}}{\rightsquigarrow} [c_2, l_2.r_2]}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : B \overset{\mathbb{1}}{\rightsquigarrow} \left[\begin{array}{l} \left(\begin{array}{l} \mathtt{let}\ (x, z_1) = c_1\ \mathtt{in} \\ \mathtt{let}\ (y, z_2) = c_2\ \mathtt{in} \\ (y, (z_1, z_2)) \end{array}\right), \\ (l_1, l_2).\mathtt{let}\ x = r_1\ \mathtt{in}\ r_2 \end{array}\right]}\ \mathrm{let}\ \overset{\mathbb{1}}{\rightsquigarrow}$$

$$\frac{\Gamma \vdash e_1 : A \overset{\mathbb{2}}{\rightsquigarrow} [p_1, l_1.r_1] \quad \Gamma, x : A\ @\ \mathbb{2} \vdash e_2 : B \overset{\mathbb{2}}{\rightsquigarrow} [p_2, l_2.r_2]}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : B \overset{\mathbb{2}}{\rightsquigarrow} [(p_1, p_2), (l_1, l_2).\mathtt{let}\ x = r_1\ \mathtt{in}\ r_2]}\ \mathrm{let}\ \overset{\mathbb{2}}{\rightsquigarrow} \qquad \frac{\Gamma \vdash e : A \overset{\mathbb{2}}{\rightsquigarrow} [p, l.r]}{\Gamma \vdash \mathtt{next}\ e : \bigcirc A \overset{\mathbb{1}}{\rightsquigarrow} [((), p), l.r]}\ \bigcirc\mathrm{I}\ \overset{\mathbb{1}}{\rightsquigarrow}$$

$$\frac{\Gamma \vdash e : \bigcirc A \overset{\mathbb{1}}{\rightsquigarrow} [c, l.r]}{\Gamma \vdash \mathtt{prev}\ e : A \overset{\mathbb{2}}{\rightsquigarrow} [\mathtt{pi2}\ c, l.r]}\ \bigcirc\mathrm{E}\ \overset{\mathbb{2}}{\rightsquigarrow}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{bool} \overset{\mathbb{1}}{\rightsquigarrow} [c_1, l_1.r_1] \quad \Gamma \vdash e_2 : A \overset{\mathbb{1}}{\rightsquigarrow} [c_2, l_2.r_2] \quad \Gamma \vdash e_3 : A \overset{\mathbb{1}}{\rightsquigarrow} [c_3, l_3.r_3]}{\Gamma \vdash \left(\begin{array}{l} \mathtt{if}\ e_1 \\ \mathtt{then}\ e_2 \\ \mathtt{else}\ e_3 \end{array}\right) : A \overset{\mathbb{1}}{\rightsquigarrow} \left[\begin{array}{l} \left(\begin{array}{l} \mathtt{let}\ (y_1, z_1) = c_1\ \mathtt{in} \\ \quad \mathtt{if}\ y_1 \\ \quad \mathtt{then}\ \mathtt{let}\ (y_2, z_2) = c_2\ \mathtt{in}\ (y_2, (z_1, \mathtt{inl}\ z_2)) \\ \quad \mathtt{else}\ \mathtt{let}\ (y_3, z_3) = c_2\ \mathtt{in}\ (y_3, (z_1, \mathtt{inl}\ z_3)) \end{array}\right), \\ (l_1, l_b).(r_1; \mathtt{case}\ l_b\ \mathtt{of}\ l_2.r_2 \mid l_3.r_3) \end{array}\right]}\ \mathrm{if}\ \overset{\mathbb{1}}{\rightsquigarrow}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{bool} \overset{\mathbb{2}}{\rightsquigarrow} [p_1, l_1.r_1] \quad \Gamma \vdash e_2 : A \overset{\mathbb{2}}{\rightsquigarrow} [p_2, l_2.r_2] \quad \Gamma \vdash e_3 : A \overset{\mathbb{2}}{\rightsquigarrow} [p_3, l_3.r_3]}{\Gamma \vdash \left(\begin{array}{l} \mathtt{if}\ e_1 \\ \mathtt{then}\ e_2 \\ \mathtt{else}\ e_3 \end{array}\right) : A \overset{\mathbb{2}}{\rightsquigarrow} \left[(p_1, p_2, p_3), (l_1, l_2, l_3).\left(\begin{array}{l} \mathtt{if}\ r_1 \\ \mathtt{then}\ r_2 \\ \mathtt{else}\ r_2 \end{array}\right)\right]}\ \mathrm{if}\ \overset{\mathbb{2}}{\rightsquigarrow}$$

Figure 8. Term Splitting

$$\frac{\Gamma, x : A \mathbin{@} \mathbb{1} \vdash e : B \overset{\mathbb{1}}{\rightsquigarrow} [c, l.r]}{\Gamma \vdash \lambda x{:}A.e : A \to B \overset{\mathbb{1}}{\rightsquigarrow} [\lambda x.c, \_.(\lambda(x,l).r)]} \to \mathrm{I} \overset{\mathbb{1}}{\rightsquigarrow} \qquad \frac{\Gamma, x : A \mathbin{@} \mathbb{2} \vdash e : B \overset{\mathbb{2}}{\rightsquigarrow} [p, l.r]}{\Gamma \vdash \lambda x{:}A.e : A \to B \overset{\mathbb{2}}{\rightsquigarrow} [p, l.(\lambda x.M)]} \to \mathrm{I} \overset{\mathbb{2}}{\rightsquigarrow}$$

$$\frac{\Gamma \vdash e_1 : A \to B \overset{\mathbb{1}}{\rightsquigarrow} [c_1, l_1.r_1] \quad \Gamma \vdash e_2 : A \overset{\mathbb{1}}{\rightsquigarrow} [c_2, l_2.r_2]}{\Gamma \vdash e_1\ e_2 : B \overset{\mathbb{1}}{\rightsquigarrow} [\ldots]} \to \mathrm{E} \overset{\mathbb{1}}{\rightsquigarrow}$$

$$\frac{\Gamma \vdash e_1 : A \to B \overset{\mathbb{2}}{\rightsquigarrow} [p_1, l_1.r_1] \quad \Gamma \vdash e_2 : A \overset{\mathbb{2}}{\rightsquigarrow} [p_2, l_2.r_2]}{\Gamma \vdash e_1\ e_2 : B \overset{\mathbb{2}}{\rightsquigarrow} [(p_1, p_2), (l_1, l_2).r_1\ r_2]} \to \mathrm{E} \overset{\mathbb{2}}{\rightsquigarrow}$$

$$\frac{\Gamma \vdash e : A \overset{\mathbb{1}}{\rightsquigarrow} [c, l.r]}{\Gamma \vdash \mathtt{inl}\ e : A + B \overset{\mathbb{1}}{\rightsquigarrow} [\mathtt{let}\ (y,z) = c\ \mathtt{in}\ (\mathtt{inl}\ y, z), l.r]} +\mathrm{I}_1 \overset{\mathbb{1}}{\rightsquigarrow}$$

$$\frac{\Gamma \vdash e : A \overset{\mathbb{1}}{\rightsquigarrow} [c, l.r]}{\Gamma \vdash \mathtt{inr}\ e : A + B \overset{\mathbb{1}}{\rightsquigarrow} [\mathtt{let}\ (y,z) = c\ \mathtt{in}\ (\mathtt{inr}\ y, z), l.r]} +\mathrm{I}_2 \overset{\mathbb{1}}{\rightsquigarrow} \qquad \frac{\Gamma \vdash e : A \overset{\mathbb{2}}{\rightsquigarrow} [p, l.r]}{\Gamma \vdash \mathtt{inl}\ e : A + B \overset{\mathbb{2}}{\rightsquigarrow} [l.\mathtt{inl}\ r]} +\mathrm{I}_1 \overset{\mathbb{2}}{\rightsquigarrow}$$

$$\frac{\Gamma \vdash e : A \overset{\mathbb{2}}{\rightsquigarrow} [p, l.r]}{\Gamma \vdash \mathtt{inr}\ e : A + B \overset{\mathbb{2}}{\rightsquigarrow} [l.\mathtt{inr}\ r]} +\mathrm{I}_1 \overset{\mathbb{2}}{\rightsquigarrow}$$

$$\frac{\Gamma \vdash e_1 : A + B \overset{\mathbb{1}}{\rightsquigarrow} [c_1, l_1.r_1] \quad \Gamma, x_2 : A \mathbin{@} \mathbb{1} \vdash e_2 : C \overset{\mathbb{1}}{\rightsquigarrow} [c_2, l_2.r_2] \quad \Gamma, x_3 : B \mathbin{@} \mathbb{1} \vdash e_3 : C \overset{\mathbb{1}}{\rightsquigarrow} [c_3, l_3.r_3]}{\Gamma \vdash \begin{pmatrix} \mathtt{case}\ e_1\ \mathtt{of} \\ x_2.e_2 \\ \mid x_3.e_3 \end{pmatrix} : C \overset{\mathbb{1}}{\rightsquigarrow} \left[\begin{array}{l} \left(\begin{array}{l} \mathtt{let}\ (y_1, z_1) = c_1\ \mathtt{in} \\ \quad \mathtt{case}\ y_1\ \mathtt{of} \\ \quad\ x_2.\mathtt{let}\ (y_2, z_2) = c_2\ \mathtt{in}\ (y_2, (z_1, \mathtt{inl}(z_2))) \\ \quad \mid x_3.\mathtt{let}\ (y_3, z_3) = c_3\ \mathtt{in}\ (y_3, (z_1, \mathtt{inr}(z_3))) \end{array} \right), \\ (l_1, l_b).\ \left(\begin{array}{l} \mathtt{let}\ z = r_1\ \mathtt{in} \\ \quad \mathtt{case}\ l_b\ \mathtt{of} \\ \quad\ l_2.\mathtt{let}\ x_2 = z\ \mathtt{in}\ r_2 \\ \quad \mid l_3.\mathtt{let}\ x_3 = z\ \mathtt{in}\ r_3 \end{array} \right) \end{array} \right]} +\mathrm{E} \overset{\mathbb{1}}{\rightsquigarrow}$$

$$\frac{\Gamma \vdash e_1 : A + B \overset{\mathbb{2}}{\rightsquigarrow} [p_1, l_1.r_1] \quad \Gamma, x_2 : A \mathbin{@} \mathbb{2} \vdash e_2 : C \overset{\mathbb{2}}{\rightsquigarrow} [p_2, l_2.r_2] \quad \Gamma, x_3 : B \mathbin{@} \mathbb{2} \vdash e_3 : C \overset{\mathbb{2}}{\rightsquigarrow} [p_3, l_3.r_3]}{\Gamma \vdash \begin{pmatrix} \mathtt{case}\ e_1\ \mathtt{of} \\ x_2.e_2 \\ \mid x_3.e_3 \end{pmatrix} : C \overset{\mathbb{2}}{\rightsquigarrow} \left[(p_1, p_2, p_3), (l_1, l_2, l_3).\ \left(\begin{array}{l} \mathtt{case}\ r_1\ \mathtt{of} \\ x_2.r_2 \\ \mid x_3.r_3 \end{array} \right) \right]} +\mathrm{E} \overset{\mathbb{2}}{\rightsquigarrow}$$

Figure 9. Sum and Function Splitting

it. They note that where [circle] corresponds to the non-branching temporal logic, [square] corresponds to the branching version.

References

[1] R. Davies. A temporal-logic approach to binding-time analysis. In *LICS*, pages 184–195, 1996.

[2] R. Davies and F. Pfenning. A modal analysis of staged computation. *J. ACM*, 48(3):555–604, 2001.

[3] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: A multi-stage language for high-performance computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 105–116, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. . URL `http://doi.acm.org/10.1145/2491956.2462166`.

[4] N. D. Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3):480–503, Sept. 1996. ISSN 0360-0300. . URL `http://doi.acm.org/10.1145/243439.243447`.

[5] U. Jørring and W. L. Scherlis. Compilers and staging transformations. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '86, pages 86–96, New York, NY, USA, 1986. ACM. . URL `http://doi.acm.org/10.1145/512644.512652`.

[6] T. B. Knoblock and E. Ruf. Data specialization. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, pages 215–225, New York, NY, USA, 1996. ACM. ISBN 0-89791-795-2. . URL `http://doi.acm.org/10.1145/231379.231428`.