Stage-Splitting a Modal Language

Nicolas Feltman Carlo Anguili Umut Acar Kayvon Fatahalian
Carnegie Mellon University
emails

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. ([6]) 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...[4]), and their background theory and type-systems are well understood ([3]). 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 [5]), $[\![f]\!]$ 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

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

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

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

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

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

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

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

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. The $\lambda^{\mathbb{12}}$ Language

In this section, we describe $\lambda^{\mathbb{12}}$, a simple two-stage language used as the basis for defining and analyzing stage splitting. Every expression in $\lambda^{\mathbb{12}}$ has a statically known *stage*, either $\mathbb{1}$ or $\mathbb{2}$, which defines when it is fully reduced. Stage-$\mathbb{1}$ and stage-$\mathbb{2}$ types include products, sums, functions, and base types. In addition to these types, stage-$\mathbb{1}$ expressions may also be of type $\bigcirc A$, representing an enclosed stage-$\mathbb{2}$ expression of type A. (The type system of $\lambda^{\mathbb{12}}$ is adopted directly from that of the modal language $\lambda^{\bigcirc}$ [2], but is restricted to two stages for simplicity.) $\bigcirc A$ terms must be treated opaquely by stage-$\mathbb{1}$ code as they cannot be reduced (to values with type A) until stage $\mathbb{2}$. Like in $\lambda^{\bigcirc}$ there is no reciprocal modality.

Although there is a strict one-way dependence between the two stages at the type level, stage-$\mathbb{1}$ and stage-$\mathbb{2}$ terms are defined mutually recursively: the `next` construct embeds stage-$\mathbb{2}$ expressions into stage $\mathbb{1}$, whereas `prev` embeds stage-$\mathbb{1}$ expressions into stage $\mathbb{2}$. The `next` and `prev` constructs are the only expressions that interact with the stage of a term, and we surround their arguments with braces in $\lambda^{\mathbb{12}}$ syntax to clearly indicate stage boundaries within a program.

The grammar and type system for $\lambda^{\mathbb{12}}$ is given in Figures 1 and 3. (Typing judgments and context variables are annotated with stages.) Only $\bigcirc$ and its introductory and eliminatory forms `next` and `prev` affect the stage of a term or type. Specifically, given an argument with type A at stage $\mathbb{2}$, `next` forms a $\bigcirc A$ at stage $\mathbb{1}$. Stage $\mathbb{2}$ expressions can obtain the original stage $\mathbb{2}$ argument via the `prev` construct. Since `prev` operates at stage $\mathbb{2}$, this ensures no violation of causality [1]. The `hold` construct serves to wrap stage $\mathbb{1}$ integers for use in stage $\mathbb{2}$. Although it is possible to implement `hold` from other $\lambda^{\mathbb{12}}$ features, we provide it as a core primitive to simplify our examples.

3.1 Staged Evaluation

As a simple example of expressing staged programs in $\lambda^{\mathbb{12}}$, consider the following fast exponentiation algorithm, which calculates b^p in $\log p$ time:

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

The stage-$\mathbb{1}$ term below defines the function `fexp` that implements the fast exponentiation algorithm:

```
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,p/2)} in x*x}
    else
        next{prev{b} * prev{fexp(b,p-1)}}
```

Although the code for `fexp` looks very much like the unstaged mathematical definition above, it is in fact a staged program: the `if` predicates and exponent decomposition are all stage-$\mathbb{1}$ terms, since they occur within `prev` blocks. We note that our work does not consider the problem of *binding time analysis*—transforming an unstaged program into a staged $\lambda^{\mathbb{12}}$ program by inserting appropriate `next`, `prev`, and `hold` constructs. We assume that binding time analysis is performed manually by a programmer (as done above) or via automatic analysis of unstaged code to arrive at a valid $\lambda^{\mathbb{12}}$ program used as input for subsequent splitting.

A key attribute of the `fexp` code example is the nesting of stage-$\mathbb{1}$ and stage-$\mathbb{2}$ expressions. Ordinary term evaluation eliminates outermost redexes first, however in the case that stage $\mathbb{1}$ expressions are contained inside stage $\mathbb{2}$ ones (such as in the example above), this strategy conflicts with the precept of staged execution: that all stage $\mathbb{1}$ code be evaluated before the evaluation of stage $\mathbb{2}$ code. Accordingly, a suitably staged dynamic semantics for $\lambda^{\mathbb{12}}$ must evaluate *all* the stage $\mathbb{1}$ subexpressions of a term before its stage $\mathbb{2}$ subexpressions.

3.2 Non-Duplicating First-Stage Evaluation

To understand how this behavior can be achieved, consider evaluation of the stage-$\mathbb{2}$ term below, which contains nested `next` and `prev` blocks:

```
prev{
  let x = (next {1+2}, 3+4) in
  next{ prev{#1 x} * prev{#1 x} * hold{#2 x} }
}
```

Our goal is to evaluate away all of the stage-$\mathbb{1}$ portions of this code, but there is more than one reasonable way to do this. One option, taken by [2], is to treat `next{1+2}` as a value for the purposes of stage-$\mathbb{1}$ evaluation, consequently duplicating it during substitution for `x`. This produces:

```
(1+2) * (1+2) * 7
```

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

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

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

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

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

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

$$\frac{\Gamma \vdash e : \text{int} \mathbin{@} \mathbb{1}}{\Gamma \vdash \texttt{hold}\ e : \text{int} \mathbin{@} \mathbb{2}}\ \text{hold}$$

Figure 3. $\lambda^{\mathbb{12}}$ Static Semantics

We take a different approach. Instead of duplicating the contents of the first `next` expression, we bind them to some variable (here, `y`) and duplicate only a reference to that variable. This produces:

```
let y = 1+2 in y * y * 7
```

Achieving this behavior mechanically requires us to resolve a contradiction: we must substitute for `x` at stage $\mathbb{1}$, but we cannot evaluate inside the `next` block bound to it. Our solution is to replace the `next` with a new variable and create an explicit substitution (shown with a $\mapsto$) binding that variable to the `next`'s contents. This substitution then floats up to the top of the containing `prev` block:

```
prev {
[y ↦ 1+2]
  let x = (ŷ, 7) in
  next{prev{#1 x} * prev{#1 x} * hold{#2 x}}
}
```

Note that the variable left behind is rendered with a hat. By treating this *hatted variable* as a value, we're now free to perform the stage-$\mathbb{1}$ substitution for `x`.

```
prev {
[y ↦ 1+2]
  next{
    prev{#1 (ŷ, 7)} * prev{#1 (ŷ, 7)} *
    hold{#2 (ŷ, 7)}
  }
}
```

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. As a rule, these will reduce to hatted variables, and `prev` removes the hat, leaving the variable in place:

```
prev {
[y ↦ 1+2]
    next{ y * y * 7 }
}
```

Once again, we lift the contents of the `next` into a substitution:

```
prev {
[y ↦ 1+2]
[z ↦ y*y*7]
    ẑ
}
```

Finally, when evaluating the outer `prev`, we must *reify* the contained substitutions into let statements, yielding

```
let y = 1+2 in
let z = y * y * 7 in z
```

Thus we have evaluated all of the stage $\mathbb{1}$ expressions of this program without duplicating the contents of `next` blocks.

3.3 Dynamics

The algorithm described above creates three different kinds of expressions which cannot be evaluated further at a particular stage:

- Partial values (pvals) are stage $\mathbb{1}$ terms that have been fully evaluated, but which may contain hatted variables. In the example above, `(ŷ,7)`.
- Residuals (reses) are stage $\mathbb{2}$ terms whose stage $\mathbb{1}$ subexpressions have all been fully evaluated. In the example above, `(1+2)` and `(let z = y*y*7 in z)`.
- Values (vals) are stage $\mathbb{2}$ terms which are fully evaluated; these are the results of a computation after both stages have been completed. In the example above, the term evaluates to `63`.

The $\Downarrow_{\mathbb{1}}$ judgment takes an open stage $\mathbb{1}$ term to a *future environment* ξ and a partial value v. The future environment is a mapping from newly-created variables (which then may appear—hatted—in v) to residuals. It is the formal manifestation of the floating substitutions in our example. For all of the normal features of $\lambda^{\mathbb{12}}$(Figure 4), first stage evaluation has the same behavior and effect on values as standard evaluation, and the final future environment is gotten by merging the future environments of the subterms.

When this judgment encounters a `next` block (Figure 5), it searches into the block's stage $\mathbb{2}$ content to find any contained stage $\mathbb{1}$ subexpressions and evaluate them in place. We call this search process *speculation*. It is implemented by the $\Downarrow_{\mathbb{2}}$ judgment,

$$\frac{\cdot}{\Xi;\Gamma \vdash () \Downarrow_{\mathbb{1}} [\cdot, ()]}\ \text{unit}\ \Downarrow_{\mathbb{1}} \qquad \frac{\cdot}{\Xi;\Gamma \vdash i \Downarrow_{\mathbb{1}} [\cdot, i]}\ \text{int}\ \Downarrow_{\mathbb{1}} \qquad \frac{\cdot}{\Xi;\Gamma \vdash b \Downarrow_{\mathbb{1}} [\cdot, b]}\ \text{bool}\ \Downarrow_{\mathbb{1}} \qquad \frac{\cdot}{\Xi;\Gamma \vdash \lambda x{:}A.e \Downarrow_{\mathbb{1}} [\cdot, \lambda x.e]}\ \to \text{I}\ \Downarrow_{\mathbb{1}}$$

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

$$\frac{\Xi;\Gamma \vdash e_1 \Downarrow_{\mathbb{1}} [\xi_1, v_1] \quad \Xi;\Gamma \vdash e_2 \Downarrow_{\mathbb{1}} [\xi_2, v_2]}{\Xi;\Gamma \vdash (e_1, e_2) \Downarrow_{\mathbb{1}} [\xi_1 \circ \xi_2, (v_1, v_2)]}\ \times\text{I}\ \Downarrow_{\mathbb{1}} \qquad \frac{\Xi;\Gamma \vdash e \Downarrow_{\mathbb{1}} [\xi, (v_1, v_2)]}{\Xi;\Gamma \vdash \mathtt{pi1}\ e \Downarrow_{\mathbb{1}} [\xi, v_1]}\ \times\text{E}_1\ \Downarrow_{\mathbb{1}} \qquad \frac{\Xi;\Gamma \vdash e \Downarrow_{\mathbb{1}} [\xi, (v_1, v_2)]}{\Xi;\Gamma \vdash \mathtt{pi2}\ e \Downarrow_{\mathbb{1}} [\xi, v_2]}\ \times\text{E}_2\ \Downarrow_{\mathbb{1}}$$

$$\frac{\Xi;\Gamma \vdash e \Downarrow_{\mathbb{1}} [\xi, v]}{\Xi;\Gamma \vdash \mathtt{inl}\ e \Downarrow_{\mathbb{1}} [\xi, \mathtt{inl}\ v]}\ +\text{I}_1\ \Downarrow_{\mathbb{1}} \qquad \frac{\Xi;\Gamma \vdash e \Downarrow_{\mathbb{1}} [\xi, v]}{\Xi;\Gamma \vdash \mathtt{inr}\ e \Downarrow_{\mathbb{1}} [\xi, \mathtt{inr}\ v]}\ +\text{I}_2\ \Downarrow_{\mathbb{1}}$$

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

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

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

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

Figure 4. λ^{12} Dynamic Semantics: Core

$$\frac{\Xi;\Gamma \vdash e \Downarrow_2 q}{\Xi;\Gamma \vdash \mathtt{next}\ e \Downarrow_{\mathbb{1}} [y \mapsto q, \hat{y}]}\ \bigcirc\text{I}\ \Downarrow_{\mathbb{1}} \qquad \frac{\Xi;\Gamma \vdash e \Downarrow_{\mathbb{1}} [\xi, \hat{y}] \quad [\xi, y] \overset{R}{\Rightarrow} q}{\Xi;\Gamma \vdash \mathtt{prev}\ e \Downarrow_2 q}\ \bigcirc\text{E}\ \Downarrow_2 \qquad \frac{\Xi;\Gamma \vdash e \Downarrow_{\mathbb{1}} [\xi, i] \quad [\xi, i] \overset{R}{\Rightarrow} q}{\Xi;\Gamma \vdash \mathtt{hold}\ e \Downarrow_2 q}\ \text{hold}\ \Downarrow_2$$

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

Figure 5. λ^{12} Dynamic Semantics: `next` and `prev`

$$\frac{\cdot}{\Xi;\Gamma \vdash () \Downarrow_2 ()}\ \text{unit}\ \Downarrow_2 \qquad \frac{\cdot}{\Xi;\Gamma \vdash i \Downarrow_2 i}\ \text{int}\ \Downarrow_2 \qquad \frac{\cdot}{\Xi;\Gamma \vdash b \Downarrow_2 b}\ \text{bool}\ \Downarrow_2 \qquad \frac{x \in \Gamma}{\Xi;\Gamma \vdash x \Downarrow_2 x}\ \text{hyp}\ \Downarrow_2$$

$$\frac{\Xi;\Gamma, x \vdash e \Downarrow_2 q}{\Xi;\Gamma \vdash \lambda x{:}A.e \Downarrow_2 \lambda x{:}A.q}\ \to\text{I}\ \Downarrow_2 \qquad \frac{\Xi;\Gamma \vdash e_1 \Downarrow_2 q_1 \quad \Xi;\Gamma \vdash e_2 \Downarrow_2 q_2}{\Xi;\Gamma \vdash e_1\ e_2 \Downarrow_2 q_1\ q_2}\ \to\text{E}\ \Downarrow_2 \qquad \frac{\Xi;\Gamma \vdash e_1 \Downarrow_2 q_1 \quad \Xi;\Gamma \vdash e_2 \Downarrow_2 q_2}{\Xi;\Gamma \vdash (e_1, e_2) \Downarrow_2 (q_1, q_2)}\ \times\text{I}\ \Downarrow_2$$

$$\frac{\Xi;\Gamma \vdash e \Downarrow_2 q}{\Xi;\Gamma \vdash \mathtt{pi1}\ e \Downarrow_2 \mathtt{pi1}\ q}\ \times\text{E}_1\ \Downarrow_2 \qquad \frac{\Xi;\Gamma \vdash e \Downarrow_2 q}{\Xi;\Gamma \vdash \mathtt{pi2}\ e \Downarrow_2 \mathtt{pi2}\ q}\ \times\text{E}_2\ \Downarrow_2 \qquad \frac{\Xi;\Gamma \vdash e \Downarrow_2 q}{\Xi;\Gamma \vdash \mathtt{inl}\ e \Downarrow_2 \mathtt{inl}\ q}\ +\text{I}_1\ \Downarrow_2$$

$$\frac{\Xi;\Gamma \vdash e \Downarrow_2 q}{\Xi;\Gamma \vdash \mathtt{inr}\ e \Downarrow_2 \mathtt{inr}\ q}\ +\text{I}_2\ \Downarrow_2 \qquad \frac{\Xi;\Gamma \vdash e_1 \Downarrow_2 q_1 \quad \Xi;\Gamma, x_2 \vdash e_2 \Downarrow_2 q_2 \quad \Xi;\Gamma, x_3 \vdash e_3 \Downarrow_2 q_3}{\Xi;\Gamma \vdash \mathtt{case}\ e_1\ \mathtt{of}\ x_2.e_2 \mid x_3.e_3 \Downarrow_2 \mathtt{case}\ q_1\ \mathtt{of}\ x_2.q_2 \mid x_3.q_3}\ +\text{E}_1\ \Downarrow_2$$

$$\frac{\Xi;\Gamma \vdash e_1 \Downarrow_2 q_1 \quad \Xi;\Gamma, x \vdash e_2 \Downarrow_2 q_2}{\Xi;\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \Downarrow_2 \mathtt{let}\ x = q_1\ \mathtt{in}\ q_2}\ \text{let}\ \Downarrow_2 \qquad \frac{\Xi;\Gamma \vdash e_1 \Downarrow_2 q_1 \quad \Xi;\Gamma \vdash e_2 \Downarrow_2 q_2 \quad \Xi;\Gamma \vdash e_3 \Downarrow_2 q_3}{\Xi;\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \Downarrow_2 \mathtt{if}\ q_1\ \mathtt{then}\ q_2\ \mathtt{else}\ q_3}\ \text{if}_\text{T}\ \Downarrow_2$$

Figure 6. λ^{12} Dynamic Semantics: Speculation

which takes a stage $\mathbb{2}$ term to a residual. Once the contents of the `next` block are speculated into a residual (q), we then return a fresh hatted variable ($\hat{y}$), along with a future environment which maps that variable to our residual ($y \mapsto q$).

At all of the normal features (Figure 6), speculation does nothing but recursively speculate into every subexpression. Once speculation finds a `prev` block, we resume stage $\mathbb{1}$ evaluation of the contents, which produces a future context and (by canonical forms) a hatted variable ($\hat{y}$). The context is then reified (using the $\overset{R}{\Rightarrow}$ judgment) into a series of let bindings, with y—stripped of its hat—at the bottom.

Because the inputs to evaluation and speculation are sometimes open on variables, we adorn our judgments with two contexts. The first of these (Ξ) keeps track of those hatted variables that may appear in the input. Since hatted variables can only be introduced to terms via substitution (*i.e.* they do not appear in the original program), this context is augmented only at stage $\mathbb{1}$ substitution sites. The second context (Γ) keeps track of stage $\mathbb{2}$ variables, which are declared in original program.

If we change the `next` and `prev` rules to

$$\frac{\Xi; \Gamma \vdash e \Downarrow_{\mathbb{2}} q}{\Xi; \Gamma \vdash \texttt{next}\ e \Downarrow_{\mathbb{1}} [\cdot, \texttt{next}\ q]} \qquad \frac{\Xi; \Gamma \vdash e \Downarrow_{\mathbb{1}} [\cdot, \texttt{next}\ v]}{\Xi; \Gamma \vdash \texttt{prev}\ e \Downarrow_{\mathbb{2}} v}$$

and treat `next` q as a partial value, where q is a residual, then we get precisely the semantics from [2]. This essentially bypasses the environment bookkeeping in $\Downarrow_{\mathbb{1}}$, by inlining residuals instead of hoisting them in `let`-bindings. From this it's clear that the semantics of [2] and ours always produce the same value when they both terminate. But moreover, because a reified residual will always be evaluated whether or not its result is consumed, our semantics terminates strictly less often than that of [2].

3.4 A Partial Evaluation System

A function f which accepts inputs at both stages $\mathbb{1}$ and $\mathbb{2}$ can be given a type of the form $A \to \bigcirc(B \to C)$[1]. Given its stage $\mathbb{1}$ argument $a : A$, we can evaluate the partially-applied function: $\cdot; \cdot \vdash f\ a \Downarrow_{\mathbb{1}} [\xi, v]$. The result is an environment ξ and a partial value v of type $\bigcirc(B \to C)$. Next, we reify this environment into a sequence of `let`-bindings enclosing v, via $[\xi, v] \overset{R}{\Rightarrow} f_a$. A canonical forms theorem on v ensures that the resulting f_a has type $B \to C$ in the residual language. Finally, given a stage $\mathbb{2}$ argument $b : B$, we can stage-$\mathbb{2}$ evaluate the ultimate result of the function, $f_a\ b \downarrow_{\mathbb{2}} c$.

TODO: make sure this is true (haven't written the canonical forms theorems yet)

That this sequence of evaluations is in fact staged follows from our characterizations of partial values, residuals, and values, that $\Downarrow_{\mathbb{1}}$ outputs a partial value, and that $\overset{R}{\Rightarrow}$ outputs an expression in λ^2.

Remark 3.1. *For any $e : A \mathbin{@} \mathbb{1}$ containing no* `next` *subexpressions, $\Downarrow_{\mathbb{1}}$ will always compute an empty environment, and a partial value identical to the result of call-by-value evaluation of e.*

3.5 Metatheory

TODO: terminology for the contexts?

Definition 3.2. *Contexts Ξ and Γ are well-formed (Ξ wf, Γ wf) if they contain only stage-2 variables.*

Definition 3.3. *An environment 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* $\Xi, \mathsf{dom}(\xi'); \Gamma \vdash e\ \mathbb{1}\text{-val}@\mathbb{2}$.

Theorem 3.4. *If $\Xi; \Gamma \vdash e : A \mathbin{@} \mathbb{1}$ then Ξ wf, Γ wf, and A type $@\ \mathbb{1}$.*

Theorem 3.5. *If $\Xi; \Gamma \vdash e \Downarrow_{\mathbb{1}} [\xi, v]$ then:*

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

Theorem 3.6. *If $\Xi; \Gamma \vdash e \Downarrow_{\mathbb{2}} q$ then:*

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

[1] We can rewrite `fexp` in this form, or simply apply the following higher-order function which makes the adjustment:
TODO: code written, but listings don't work in footnotes! commented out in source until we get working

4. Splitting Algorithm

The basic idea is that we want to take a term which contains interleaved stage $\mathbb{1}$ and stage $\mathbb{2}$ code and untangle it into two separate terms: one with all of the stage $\mathbb{1}$ code and one with all the stage $\mathbb{2}$ code. To facilitate communication between the stages, there is a data structure passed between the stages, which is an output of the stage $\mathbb{1}$ term and and input to the stage $\mathbb{2}$ term.

As with the dynamics, the form of the splitting statement depends on the external stage of the term. We consider splitting for stage $\mathbb{2}$ terms first, as it has a simpler form. For example, take `hold{1+2} < 5`. We can split this into two separate terms, `1+2` and `fn l => l < 5`, which are called the *precomputation* and *residual*, respectively. For this splitting to be correct, the residual applied to the precomputation should yield the same result as the original term.

The form of splitting for stage $\mathbb{1}$ terms is more complicated. To see why, consider the term `(next{hold{1+2} < 5},3*4)`. As before, the comparison operation is part of stage $\mathbb{2}$, and the addition operation is a stage $\mathbb{1}$ precomputation, the result of which will eventually become input to the residual. The multiplication operation, however, is neither part of the residual nor a precomputation to support it. Instead it is the *immediate result* of stage $\mathbb{1}$, because it is available to the stage $\mathbb{1}$ context around our original term. For example:

```
let x = (next{hold{1+2} < 5},3*4) in
next{ if prev{#1 x} then 7 else hold {#2 x} }
```

To support this, stage $\mathbb{1}$ splitting produces two outputs: the *combined result* and the residual, where the combined result reduces to a tuple containing the immediate result and the precomputation.

4.1 Formal Setup

We now discuss the formal definition of splitting. The splitting algorithm comprises two judgments, $\overset{1}{\rightsquigarrow}$ and $\overset{2}{\rightsquigarrow}$. The $\overset{1}{\rightsquigarrow}$ judgment sends a stage $\mathbb{1}$ term (e) to a combined term (c) and residual ($l.r$), while $\overset{2}{\rightsquigarrow}$ sends a stage $\mathbb{2}$ term (e) to a precomputation (p) and a residual ($l.r$). For concision, we actually represent residuals as a term open on a single variable, rather than as functions. For both judgments, we use a context (Γ) to keep track of the open variables of e.

The rules[2] for splitting `next`, `prev`, and `hold` are given in Figure 7. The rule for `next` simply tuples up the precomputation of its subexpression with a trivial immediate result, while the rule for `prev` projects the combined result of its subexpression to just the precomputation. The `hold` rule treats the entire combined result of its subexpression as a precomputation, and projects out the integer result in the residual[3]

The rules of stage $\mathbb{2}$ splitting are given in Figure 8. Every rule works by bundling the precomputations of the constituent parts, and then unbundling them with a pattern. The rules of stage $\mathbb{1}$ splitting are given in Figure 7.

4.2 Speculation

Notice in the stage $\mathbb{2}$ rules for `if`s, `case`s, and functions, the precomputation is lifted out from within branches. This is the manifestation of the speculation behavior from the semantics.

4.3 Stage $\mathbb{1}$ Divergence

Consider the case of stage $\mathbb{1}$ `if` and `case` expressions. In both cases we have enough information at stage $\mathbb{1}$ to know what branch to take, so there's no need to speculate. Instead, we evaluate the stage $\mathbb{1}$ portion of only the active branch, and then inject precomputation into a sum type. Then in the residual, we case on that sum and resume the stage $\mathbb{2}$ portion of the correct branch.

4.4 Stage $\mathbb{1}$ Functions

The trick when splitting stage $\mathbb{1}$ functions is that the contents themselves may be multi-stage. We handle this by splitting them into two functions: one in the immediate result which handles all the stage $\mathbb{1}$ content of the original, and one in the residual which handles all of the stage $\mathbb{2}$ content. Note that stage $\mathbb{1}$ λ-expressions themselves have only a trivial precomputation.

Thus the boundary type appears as output of the first stage and input of the second stage. This makes it hard to type the output, because the boundary types are not represented in the original function type.

4.5 Boundary Type Worst Case

TODO: add the example which is the worst case for figuring out boundary types

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.

[2] The rules are written using patterns, including the open variable of the residual.

[3] The residual of an integer expression is usually trivial, but we have to include it here for termination purposes.

$$\frac{\cdot}{\Gamma \vdash () : \mathsf{unit} \overset{\mathbb{1}}{\rightsquigarrow} [((), ()), \_.()]}\ \mathsf{unit}\ \overset{\mathbb{1}}{\rightsquigarrow} \qquad \frac{\cdot}{\Gamma \vdash i : \mathsf{int} \overset{\mathbb{1}}{\rightsquigarrow} [(i, ()), \_.()]}\ \mathsf{int}\ \overset{\mathbb{1}}{\rightsquigarrow} \qquad \frac{x : A \mathbin{@} \mathbb{1} \in \Gamma}{\Gamma \vdash v : A \overset{\mathbb{1}}{\rightsquigarrow} [(x, ()), \_.()]}\ \mathsf{hyp}\ \overset{\mathbb{1}}{\rightsquigarrow}$$

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

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

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

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

$$\frac{\Gamma \vdash e_1 : A \to B \overset{\mathbb{1}}{\rightsquigarrow} [c_1, l_1.r_1] \quad \Gamma \vdash e_2 : A \overset{\mathbb{1}}{\rightsquigarrow} [c_2, l_2.r_2]}{\Gamma \vdash e_1\ e_2 : B \overset{\mathbb{1}}{\rightsquigarrow} \left[\begin{array}{l} \mathtt{let}\ ((y_1, z_1), (y_2, z_2)) = (c_1, c_2)\ \mathtt{in} \\ \mathtt{let}\ (y_3, z_3) = y_1\ y_2\ \mathtt{in} \\ (y_3, (z_1, z_2, z_3)) \end{array} \right., (l_1, l_2, l_3).(r_1\ (r_2, l_3)) \Bigg]}\ \to\mathrm{E}\ \overset{\mathbb{1}}{\rightsquigarrow}$$

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

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

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

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

Figure 7. Stage $\mathbb{1}$ Splitting

$$\frac{\cdot}{\Gamma \vdash i : \mathrm{int} \overset{2}{\rightsquigarrow} [(), \_.i]}\ \mathrm{int}\overset{2}{\rightsquigarrow} \qquad \frac{\cdot}{\Gamma \vdash () : \mathrm{unit} \overset{2}{\rightsquigarrow} [(), \_.()]}\ \mathrm{unit}\overset{2}{\rightsquigarrow} \qquad \frac{x : A \mathbin{@} 2 \in \Gamma}{\Gamma \vdash v : A \overset{2}{\rightsquigarrow} [(), \_.x]}\ \mathrm{hyp}\overset{2}{\rightsquigarrow}$$

$$\frac{\Gamma, x : A \mathbin{@} 2 \vdash e : B \overset{2}{\rightsquigarrow} [p, l.r]}{\Gamma \vdash \lambda x{:}A.e : A \to B \overset{2}{\rightsquigarrow} [p, l.(\lambda x.r)]}\ \to \mathrm{I}\overset{2}{\rightsquigarrow} \qquad \frac{\Gamma \vdash e_1 : A \to B \overset{2}{\rightsquigarrow} [p_1, l_1.r_1] \quad \Gamma \vdash e_2 : A \overset{2}{\rightsquigarrow} [p_2, l_2.r_2]}{\Gamma \vdash e_1\ e_2 : B \overset{2}{\rightsquigarrow} [(p_1, p_2), (l_1, l_2).r_1\ r_2]}\ \to \mathrm{E}\overset{2}{\rightsquigarrow}$$

$$\frac{\Gamma \vdash e_1 : A \overset{2}{\rightsquigarrow} [p_1, l_1.r_1] \quad \Gamma \vdash e_2 : B \overset{2}{\rightsquigarrow} [p_2, l_2.r_2]}{\Gamma \vdash (e_1, e_2) : A \times B \overset{2}{\rightsquigarrow} [(p_1, p_2), (l_1, l_2).(r_1, r_2)]}\ \times \mathrm{I}\overset{2}{\rightsquigarrow} \qquad \frac{\Gamma \vdash e : A \times B \overset{2}{\rightsquigarrow} [p, l.r]}{\Gamma \vdash \mathtt{pi1}\ e : A \overset{2}{\rightsquigarrow} [p, l.\mathtt{pi1}\ r]}\ \times \mathrm{E}_1\overset{2}{\rightsquigarrow}$$

$$\frac{\Gamma \vdash e : A \times B \overset{2}{\rightsquigarrow} [p, l.r]}{\Gamma \vdash \mathtt{pi2}\ e : B \overset{2}{\rightsquigarrow} [p, l.\mathtt{pi2}\ r]}\ \times \mathrm{E}_2\overset{2}{\rightsquigarrow} \qquad \frac{\Gamma \vdash e : A \overset{2}{\rightsquigarrow} [p, l.r]}{\Gamma \vdash \mathtt{inl}\ e : A + B \overset{2}{\rightsquigarrow} [l.\mathtt{inl}\ r]}\ + \mathrm{I}_1\overset{2}{\rightsquigarrow}$$

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

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

$$\frac{\Gamma \vdash e_1 : A \overset{2}{\rightsquigarrow} [p_1, l_1.r_1] \quad \Gamma, x : A \mathbin{@} 2 \vdash e_2 : B \overset{2}{\rightsquigarrow} [p_2, l_2.r_2]}{\Gamma \vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : B \overset{2}{\rightsquigarrow} [(p_1, p_2), (l_1, l_2).\mathtt{let}\ x = r_1\ \mathtt{in}\ r_2]}\ \mathrm{let}\overset{2}{\rightsquigarrow}$$

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

Figure 8. Stage 2 Splitting

$$\frac{\Gamma \vdash e : \bigcirc A \overset{1}{\rightsquigarrow} [c, l.r]}{\Gamma \vdash \mathtt{prev}\ e : A \overset{2}{\rightsquigarrow} [\mathtt{pi2}\ c, l.r]}\ \bigcirc \mathrm{E}\overset{2}{\rightsquigarrow} \qquad \frac{\Gamma \vdash e : A \overset{2}{\rightsquigarrow} [p, l.r]}{\Gamma \vdash \mathtt{next}\ e : \bigcirc A \overset{1}{\rightsquigarrow} [((), p), l.r]}\ \bigcirc \mathrm{I}\overset{1}{\rightsquigarrow}$$

$$\frac{\Gamma \vdash e : \mathrm{int} \overset{1}{\rightsquigarrow} [c, l.r]}{\Gamma \vdash \mathtt{hold}\ e : \mathrm{int} \overset{2}{\rightsquigarrow} [c, (a, l).(r; a)]}\ \mathrm{hold}\overset{2}{\rightsquigarrow}$$

Figure 9. `next` and `prev` Splitting

```
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 = hold{n} in
            case compare prev{i} n of
              () (*LT*) => prev {qs left i}
            | () (*EQ*) => 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 [6] 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 [6] 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 ([7]) 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 ([2]) 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, [3] 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 it. They note that where [circle] corresponds to the non-branching temporal logic, [square] corresponds to the branching version.

References

[1] A. Cave, F. Ferreira, P. Panangaden, and B. Pientka. Fair reactive programming. *SIGPLAN Not.*, 49(1):361–372, Jan. 2014. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/2578855.2535881`.

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

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

[4] 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`.

[5] 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`.

[6] 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`.

[7] 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`.