General Stage Untangling with Applications to Algorithm Derivation

Nicolas Feltman et al.

February 14, 2014

1 Introduction

Many computational tasks are premised on the notion of multiple domains, places and times at which computation occurs. Yet programming languages traditionally view their execution environments as monolithic, specifying the What and the How, but not the When nor the Where. As always, where the language falls short, a programmer instead must bear the burden. Inevitably, this comes at the cost of expressivity, safety, or both. In this work, we present a language for specifying certain multi-domain computations, sacrificing neither the expresivity nor the safety that programmers have come to expect in single-domain languages.

Consider the programming of a pipeline system. Generally in such a system, data comes in to one processor, is transformed to an intermediate result, and then fed to later processors, possibly with some buffering, duplication, and reording in between. We can view an algorithm for such as system as one computation, split accross two spatial domains (processors). For a concrete example, we draw inspiration from graphics systems. Lets say that we want to map a function f : PixelCoord $\rightarrow$ Color over every pixel on a 1024x768 screen. Due to geometric coherence inherent in the problem, it's often advantageous to compute some information at the level of coarse 32x32 tiles (e.g. a list of the scene primitives which interact with that part of the screen) whose results can be used to accelerate the final per-pixel calculation. In adition to yielding algorithmic benefits, this decomposition strategy maps well to parallel hardware.

Often when programming high-performance systems, a programmer will wish to specialize a bivariate function $f : \alpha \times \beta \rightarrow \gamma$ to one of its inputs, $x : \alpha$, while leaving the other, $y : \beta$, for later. This need may arise because x varies at a higher frequency than y, and we would wish to avoid repeated calculation. It may also be the case that x is available at a time when computational resources are less expensive than when y becomes available. Here we say that the calculation is split accross two temporal domains. Pulling another example from graphics, we consider a raytracer wherein x : `Geom` is a specification of the scene geometry (perhaps a list of triangles), y : `Ray` is a ray cast from some point in the scene, and f computes the first intersection of the ray and the geometry. In modern raytracers, we may cast millions of rays without varying the scene geometry, so it could be advantageous to precompute as much of the algorithm as possible based on only scene geometry.

In addition to the partitioning of a calculation across multiple domains, the examples above exhibit one further similarity: communication between domains is one-directional. In the spatial case, information always flows from one processor to the next, but never backwards. In the temporal case too, information made available at the second time cannot be observed earlier (indeed this seems to be a defining feature of time, generally). For this reason, we refer to the domains as stages. This property, that information flow is acyclic, informs the rest of this paper.

[Talk about how languages without inherent support for domains must instead tackle the problem by writing various passes/shaders. This is bad because it's not compositional. For instance, I cannot write a function that talks about multiple stages at once.]

Figure 1: $L^{\mathbb{12}}$ Syntax

$$
\begin{array}{l}
\langle \mathbb{1}\text{-}type \rangle ::= \text{unit} \mid \text{int} \mid \text{bool} \\
\qquad \mid \langle \mathbb{1}\text{-}type \rangle \times \langle \mathbb{1}\text{-}type \rangle \\
\qquad \mid \bigcirc \langle \mathbb{2}\text{-}type \rangle \\
\langle \mathbb{1}\text{-}exp \rangle ::= () \mid \langle int \rangle \mid \langle bool \rangle \\
\qquad \mid \texttt{let}\ \langle var \rangle = \langle \mathbb{1}\text{-}exp \rangle\ \texttt{in}\ \langle \mathbb{1}\text{-}exp \rangle \\
\qquad \mid \langle var \rangle \\
\qquad \mid (\langle \mathbb{1}\text{-}exp \rangle, \langle \mathbb{1}\text{-}exp \rangle) \\
\qquad \mid \pi_1\ \langle \mathbb{1}\text{-}exp \rangle \mid \pi_2\ \langle \mathbb{1}\text{-}exp \rangle \\
\qquad \mid \texttt{if}\ \langle \mathbb{1}\text{-}exp \rangle\ \texttt{then}\ \langle \mathbb{1}\text{-}exp \rangle\ \texttt{else}\ \langle \mathbb{1}\text{-}exp \rangle \\
\qquad \mid \texttt{next}\ \langle \mathbb{2}\text{-}exp \rangle \\
\langle \mathbb{1}\text{-}val \rangle ::= () \mid \langle int \rangle \mid \langle bool \rangle \\
\qquad \mid (\!|\langle \mathbb{1}\text{-}val \rangle, \langle \mathbb{1}\text{-}val \rangle|\!) \\
\qquad \mid \textbf{next}\ \langle \mathbb{2}\text{-}val \rangle \\
\langle \mathbb{12}\text{-}cont \rangle ::= \bullet \\
\qquad \mid \langle \mathbb{12}\text{-}cont \rangle, \langle var \rangle : \langle \mathbb{1}\text{-}type \rangle^{\mathbb{1}} \\
\qquad \mid \langle \mathbb{12}\text{-}cont \rangle, \langle var \rangle : \langle \mathbb{2}\text{-}type \rangle^{\mathbb{2}}
\end{array}
$$

$$
\begin{array}{l}
\langle \mathbb{2}\text{-}type \rangle ::= \text{unit} \mid \text{int} \mid \text{bool} \\
\qquad \mid \langle \mathbb{2}\text{-}type \rangle \times \langle \mathbb{2}\text{-}type \rangle \\
\langle \mathbb{2}\text{-}exp \rangle ::= () \mid \langle int \rangle \mid \langle bool \rangle \\
\qquad \mid \texttt{let}\ \langle var \rangle = \langle \mathbb{2}\text{-}exp \rangle\ \texttt{in}\ \langle \mathbb{2}\text{-}exp \rangle \\
\qquad \mid \langle var \rangle \\
\qquad \mid (\langle \mathbb{2}\text{-}exp \rangle, \langle \mathbb{2}\text{-}exp \rangle) \\
\qquad \mid \pi_1\ \langle \mathbb{2}\text{-}exp \rangle \mid \pi_2\ \langle \mathbb{2}\text{-}exp \rangle \\
\qquad \mid \texttt{if}\ \langle \mathbb{2}\text{-}exp \rangle\ \texttt{then}\ \langle \mathbb{2}\text{-}exp \rangle\ \texttt{else}\ \langle \mathbb{2}\text{-}exp \rangle \\
\qquad \mid \texttt{prev}\ \langle \mathbb{1}\text{-}exp \rangle \\
\langle \mathbb{2}\text{-}val \rangle ::= () \mid \langle int \rangle \mid \langle bool \rangle \\
\qquad \mid (\!|\langle \mathbb{2}\text{-}val \rangle, \langle \mathbb{2}\text{-}val \rangle|\!)
\end{array}
$$

[Because of the acyclic nature of the problem, we can take algorithms specified in our multi-domain language and tease them apart, essentially recovering the old way of doing things. This is called stage untangling. It gets us all the safety and expression without messing up the rest of the compiler.]

[Talk about standard partial evaluation and pass separation, then place our work in the second, modulo binding time analysis.]

2 A Simple Two-Stage Language

We'll start by analyzing a simple two-stage language called $L^{\mathbb{12}}$. We say it is simple in that it supports boolean, integer, and product types, as well as let bindings, if expressions, and various primitive operations. Notably, we leave out functions, fixed point operators, and full sum types, which will be added later.

The grammar for types, terms, values, and contexts in $L^{\mathbb{12}}$are given in Figure 1. The first thing to notice is that stage $\mathbb{1}$ and stage $\mathbb{2}$ code are syntactically seperated at the type, term, and value levels, despite heavy overlap. It would be tempting to merge the two stages into one syntactic class and maintain the stage invariants with the typing judgements. Such a trick would reduce the total number of rules presented later, but at the cost of making those rules type derivation-directed, rather than syntax directed. Alternatively, we chose to have one context merged accross both stages, rather than one context for each stage. The intended effect of this decision is for bindings in one stage to shadow binding to the same variables in the other stage. The reasons for this will become clear later.

At the type and value levels, stage $\mathbb{1}$ can refer to stage $\mathbb{2}$, but stage $\mathbb{2}$ cannot refer back to stage $\mathbb{1}$. Alternatively, the two stages are mutually recursive at the term level. These two patterns together fulfill the promise of *mixed code, staged evalutation.*

In the rest of this document, I will tend to use $\{A, B, C\}$ for types, $\{v, u, w, t\}$ for values, $\{x, y\}$ for variables, e for terms, i for integers, and b for booleans.

Figure 2: $\mathrm{L}^{\mathbb{1}\mathbb{2}}$ Static Semantics

$$\frac{\cdot}{\Gamma \Vdash () : \mathrm{unit}}\ \text{unit-}\mathbb{1} \qquad \frac{\cdot}{\Gamma \Vdash i : \mathrm{int}}\ \text{int-}\mathbb{1} \qquad \frac{\cdot}{\Gamma \Vdash b : \mathrm{bool}}\ \text{bool-}\mathbb{1} \qquad \frac{x : A^{\mathbb{1}} \in \Gamma}{\Gamma \Vdash x : A}\ \text{hyp-}\mathbb{1}$$

$$\frac{\cdot}{\Gamma \Vdash () : \mathrm{unit}}\ \text{unit-}\mathbb{2} \qquad \frac{\cdot}{\Gamma \Vdash i : \mathrm{int}}\ \text{int-}\mathbb{2} \qquad \frac{\cdot}{\Gamma \Vdash b : \mathrm{bool}}\ \text{bool-}\mathbb{2} \qquad \frac{x : A^{\mathbb{2}} \in \Gamma}{\Gamma \Vdash x : A}\ \text{hyp-}\mathbb{2}$$

$$\frac{\Gamma \Vdash e_1 : A \quad \Gamma, x : A^{\mathbb{1}} \Vdash e_2 : B}{\Gamma \Vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : B}\ \text{let-}\mathbb{1} \qquad \frac{\Gamma \Vdash e_1 : A \quad \Gamma, x : A^{\mathbb{2}} \Vdash e_2 : B}{\Gamma \Vdash \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 : B}\ \text{let-}\mathbb{2}$$

$$\frac{\Gamma \Vdash e_1 : A \quad \Gamma \Vdash e_2 : B}{\Gamma \Vdash (e_1, e_2) : A \times B}\ \times\text{I-}\mathbb{1} \qquad \frac{\Gamma \Vdash e : A \times B}{\Gamma \Vdash \pi_1\ e : A}\ \times\text{E}_1\text{-}\mathbb{1} \qquad \frac{\Gamma \Vdash e : A \times B}{\Gamma \Vdash \pi_2\ e : B}\ \times\text{E}_2\text{-}\mathbb{1}$$

$$\frac{\Gamma \Vdash e_1 : A \quad \Gamma \Vdash e_2 : B}{\Gamma \Vdash (e_1, e_2) : A \times B}\ \times\text{I-}\mathbb{2} \qquad \frac{\Gamma \Vdash e : A \times B}{\Gamma \Vdash \pi_1\ e : A}\ \times\text{E}_1\text{-}\mathbb{2} \qquad \frac{\Gamma \Vdash e : A \times B}{\Gamma \Vdash \pi_2\ e : B}\ \times\text{E}_2\text{-}\mathbb{2}$$

$$\frac{\Gamma \Vdash e_1 : \mathrm{bool} \quad \Gamma \Vdash e_2 : A \quad \Gamma \Vdash e_3 : A}{\Gamma \Vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : A}\ \text{if-}\mathbb{1} \qquad \frac{\Gamma \Vdash e_1 : \mathrm{bool} \quad \Gamma \Vdash e_2 : A \quad \Gamma \Vdash e_3 : A}{\Gamma \Vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : A}\ \text{if-}\mathbb{2}$$

$$\frac{\Gamma \Vdash e : A}{\Gamma \Vdash \mathtt{next}\ e : \bigcirc A}\ \bigcirc\text{I-}\mathbb{1} \qquad \frac{\Gamma \Vdash e : \bigcirc A}{\Gamma \Vdash \mathtt{prev}\ e : A}\ \bigcirc\text{E-}\mathbb{2}$$

2.1 Static Semantics

The static semantics for $\mathrm{L}^{\mathbb{1}\mathbb{2}}$ is given in Figure 2. Modulo the two-stage restriction and current absence of functions, it's based off of the system in [1], which was inspired by temporal logic (the source of our $\bigcirc$ modality). The static semantics comprises two judgements. Those are:

Name	Pattern
Typing at $\mathbb{1}$	$\langle \mathbb{1}\mathbb{2}\text{-}\mathit{cont} \rangle \Vdash \langle \mathbb{1}\text{-}\mathit{exp} \rangle : \langle \mathbb{1}\text{-}\mathit{type} \rangle$
Typing at $\mathbb{2}$	$\langle \mathbb{1}\mathbb{2}\text{-}\mathit{cont} \rangle \Vdash \langle \mathbb{2}\text{-}\mathit{exp} \rangle : \langle \mathbb{2}\text{-}\mathit{type} \rangle$

We see that the typing judgements for both stages are each what one would expect for an unstaged language. That is, there are introduction and elimination forms for base types and products at both stage $\mathbb{1}$ and stage $\mathbb{2}$. Furthermore, these rules preserve stage. For these reasons, we can think of the base type and product features as essentially being orthogonal to the staging features. The terms `next` and `prev`, which are the only terms that allow us to change stage, are respectively the introduction and elimination forms for $\bigcirc$.

2.2 Erasure Dynamic Semantics

The dynamics semantics for $\mathrm{L}^{\mathbb{1}\mathbb{2}}$ is given in Figure 3 in big-step style. Predictably, it comprises two judgements:

Name	Pattern
Reduction at $\mathbb{1}$	$\langle \mathbb{1}\text{-}\mathit{exp} \rangle \Downarrow_{\mathbb{1}} \langle \mathbb{1}\text{-}\mathit{val} \rangle$
Reduction at $\mathbb{2}$	$\langle \mathbb{2}\text{-}\mathit{exp} \rangle \Downarrow_{\mathbb{2}} \langle \mathbb{2}\text{-}\mathit{val} \rangle$

The primary goal for this semantics was simplicity. This was achieved in that, like the static semantics, the judgements for each level are extremely similar to those for an unstaged

Figure 3: $\mathrm{L}^{\mathbb{1}\mathbb{2}}$ Erasure Dynamic Semantics

$$\frac{\cdot}{() \Downarrow_{\mathbb{1}} ()}\ \text{unit}\ \Downarrow_{\mathbb{1}} \qquad \frac{\cdot}{i \Downarrow_{\mathbb{1}} i}\ \text{int}\ \Downarrow_{\mathbb{1}} \qquad \frac{\cdot}{b \Downarrow_{\mathbb{1}} b}\ \text{bool}\ \Downarrow_{\mathbb{1}} \qquad \frac{e_1 \Downarrow_{\mathbb{1}} v_1 \quad [v_1/x]e_2 \Downarrow_{\mathbb{1}} v_2}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow_{\mathbb{1}} v_2}\ \text{let}\ \Downarrow_{\mathbb{1}}$$

$$\frac{\cdot}{() \Downarrow_{\mathbb{2}} ()}\ \text{unit}\ \Downarrow_{\mathbb{2}} \qquad \frac{\cdot}{i \Downarrow_{\mathbb{2}} i}\ \text{int}\ \Downarrow_{\mathbb{2}} \qquad \frac{\cdot}{b \Downarrow_{\mathbb{2}} b}\ \text{bool}\ \Downarrow_{\mathbb{2}} \qquad \frac{e_1 \Downarrow_{\mathbb{2}} v_1 \quad [v_1/x]e_2 \Downarrow_{\mathbb{2}} v_2}{\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Downarrow_{\mathbb{2}} v_2}\ \text{let}\ \Downarrow_{\mathbb{2}}$$

$$\frac{e_1 \Downarrow_{\mathbb{1}} v_1 \quad e_2 \Downarrow_{\mathbb{1}} v_2}{(e_1, e_2) \Downarrow_{\mathbb{1}} (\!|v_1, v_2|\!)}\ \times\text{I}\ \Downarrow_{\mathbb{1}} \qquad \frac{e \Downarrow_{\mathbb{1}} (\!|v_1, v_2|\!)}{\pi_1\ e \Downarrow_{\mathbb{1}} v_1}\ \times\text{E}_1\ \Downarrow_{\mathbb{1}} \qquad \frac{e \Downarrow_{\mathbb{1}} (\!|v_1, v_2|\!)}{\pi_2\ e \Downarrow_{\mathbb{1}} v_2}\ \times\text{E}_2\ \Downarrow_{\mathbb{1}}$$

$$\frac{e_1 \Downarrow_{\mathbb{2}} v_1 \quad e_2 \Downarrow_{\mathbb{2}} v_2}{(e_1, e_2) \Downarrow_{\mathbb{2}} (\!|v_1, v_2|\!)}\ \times\text{I}\ \Downarrow_{\mathbb{2}} \qquad \frac{e \Downarrow_{\mathbb{2}} (\!|v_1, v_2|\!)}{\pi_1\ e \Downarrow_{\mathbb{2}} v_1}\ \times\text{E}_1\ \Downarrow_{\mathbb{2}} \qquad \frac{e \Downarrow_{\mathbb{2}} (\!|v_1, v_2|\!)}{\pi_2\ e \Downarrow_{\mathbb{2}} v_2}\ \times\text{E}_2\ \Downarrow_{\mathbb{2}}$$

$$\frac{e_1 \Downarrow_{\mathbb{1}} \textbf{true} \quad e_2 \Downarrow_{\mathbb{1}} v}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow_{\mathbb{1}} v}\ \text{if}_1\ \Downarrow_{\mathbb{1}} \qquad \frac{e_1 \Downarrow_{\mathbb{1}} \textbf{false} \quad e_3 \Downarrow_{\mathbb{1}} v}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow_{\mathbb{1}} v}\ \text{if}_2\ \Downarrow_{\mathbb{1}}$$

$$\frac{e_1 \Downarrow_{\mathbb{2}} \textbf{true} \quad e_2 \Downarrow_{\mathbb{2}} v}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow_{\mathbb{2}} v}\ \text{if}_1\ \Downarrow_{\mathbb{2}} \qquad \frac{e_1 \Downarrow_{\mathbb{2}} \textbf{false} \quad e_3 \Downarrow_{\mathbb{2}} v}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow_{\mathbb{2}} v}\ \text{if}_2\ \Downarrow_{\mathbb{2}}$$

$$\frac{e \Downarrow_{\mathbb{2}} v}{\texttt{next}\ e \Downarrow_{\mathbb{1}} \textbf{next}\ v}\ \bigcirc\text{I}\ \Downarrow_{\mathbb{1}} \qquad \frac{e \Downarrow_{\mathbb{1}} \textbf{next}\ v}{\texttt{prev}\ e \Downarrow_{\mathbb{2}} v}\ \bigcirc\text{E}\ \Downarrow_{\mathbb{2}}$$

language. This simplicity came at the cost of some descriptiveness, however. The inadequacy of this semantics will become apparent shortly.

First, we discuss a few desirable theorems. Chiefly among them is type preservation (I haven't defined the judgements for the types of values, but it should be rather clear). As will be the pattern for many theorems to come, our inductive hypothesis comprises two mutually dependent lemmas, one for stage $\mathbb{1}$, and the other for stage $\mathbb{2}$:

If $\bullet \Vdash e : A$	If $\bullet \Vdash\!\models e : A$
and $e \Downarrow_{\mathbb{1}} v$	and $e \Downarrow_{\mathbb{2}} v$
then $v : A$	then $v : A$

[Proof not done, but it looks trivial in every case. [Wait no, substitutions kill me again.]]

We also want some formalization of the idea that stage two computation cannot affect stage one computation. At a minimum, the following lemma should be true [this needs lots of work],

If $\exists v \in \langle \mathbb{2}\text{-}val \rangle$ s.t. $[v/x]e \Downarrow_{\mathbb{1}} u$
then $\forall v \in \langle \mathbb{2}\text{-}val \rangle$ s.t. $[v/x]e \Downarrow_{\mathbb{1}} u'$, $u \simeq u'$

As it turns out, this is not true! [Example forthcoming.]

2.3 Splitting

We now cover splitting, the process by which mixed code is teased apart into two individual chunks of code, one for each stage. To begin, we first introduce the target lagnuage (for both stage), L. The grammar for L is given in figure 4. We see that L is (as one would hope) a traditional single-stage language, with all the same basetypes as $\mathrm{L}^{\mathbb{1}\mathbb{2}}$, in addition to products, if expressions, and full sums (injections and case expressions). The presence of this last feature is a bit curious, since $\mathrm{L}^{\mathbb{1}\mathbb{2}}$ did not support full sums. The necessity for sums should become

Figure 4: L Syntax

$$
\begin{array}{lll}
\langle exp\rangle ::= () \mid \langle int\rangle \mid \langle bool\rangle & & \langle type\rangle ::= \text{unit} \mid \text{int} \mid \text{bool} \\
\quad \mid \texttt{let}\ \langle var\rangle = \langle exp\rangle\ \texttt{in}\ \langle exp\rangle & & \quad \mid \langle type\rangle \times \langle type\rangle \\
\quad \mid \langle var\rangle & & \quad \mid \langle type\rangle + \langle type\rangle \\
\quad \mid (\langle exp\rangle, \langle exp\rangle) & & \langle val\rangle ::= () \mid \langle int\rangle \mid \langle bool\rangle \\
\quad \mid \pi_1\ \langle exp\rangle \mid \pi_2\ \langle exp\rangle & & \quad \mid (\!|\langle val\rangle, \langle val\rangle|\!) \\
\quad \mid \texttt{inl}\ \langle exp\rangle \mid \texttt{inr}\ \langle exp\rangle & & \quad \mid \mathbf{inl}\ \langle exp\rangle \mid \mathbf{inr}\ \langle exp\rangle \\
\quad \mid \texttt{if}\ \langle exp\rangle\ \texttt{then}\ \langle exp\rangle\ \texttt{else}\ \langle exp\rangle & & \langle cont\rangle ::= \bullet \\
\quad \mid \texttt{case}\ \langle exp\rangle\ \texttt{of}\ x_1.\langle exp\rangle \mid x_2.\langle exp\rangle & & \quad \mid \langle cont\rangle, \langle var\rangle : \langle type\rangle
\end{array}
$$

Figure 5: Type Splitting

$$\frac{\cdot}{\text{unit} \overset{\mathbb{1}}{\rightsquigarrow} [\text{unit}, \text{unit}]}\ \text{unit} \overset{\mathbb{1}}{\rightsquigarrow} \qquad \frac{\cdot}{\text{int} \overset{\mathbb{1}}{\rightsquigarrow} [\text{int}, \text{unit}]}\ \text{int} \overset{\mathbb{1}}{\rightsquigarrow} \qquad \frac{\cdot}{\text{bool} \overset{\mathbb{1}}{\rightsquigarrow} [\text{bool}, \text{unit}]}\ \text{bool} \overset{\mathbb{1}}{\rightsquigarrow}$$

$$\frac{A \overset{\mathbb{1}}{\rightsquigarrow} [A_1, A_2] \quad B \overset{\mathbb{1}}{\rightsquigarrow} [B_1, B_2]}{A \times B \overset{\mathbb{1}}{\rightsquigarrow} [A_1 \times B_1, A_2 \times B_2]}\ \times \overset{\mathbb{1}}{\rightsquigarrow} \qquad \frac{A \overset{\mathbb{2}}{\rightsquigarrow} A'}{\bigcirc\ A \overset{\mathbb{1}}{\rightsquigarrow} [\text{unit}, A']}\ \bigcirc \overset{\mathbb{1}}{\rightsquigarrow}$$

apparent in time. The static and dynamic semantics of L are not given, but should be obvious. They are given by the judgements:

Name	Pattern
Typing	$\Gamma \vdash \langle exp\rangle : \langle val\rangle$
Reduction	$\langle exp\rangle \Downarrow \langle val\rangle$

The splitting judgements come in seven parts, two each for types, terms, and values, and one for contexts. In tabular form, they are:

Name	Pattern
Value Splitting at $\mathbb{1}$	$\langle \mathbb{1}\text{-}val\rangle \overset{\mathbb{1}}{\hookrightarrow} [\langle val\rangle, \langle val\rangle]$
Value Splitting at $\mathbb{2}$	$\langle \mathbb{2}\text{-}val\rangle \overset{\mathbb{2}}{\hookrightarrow} \langle val\rangle$
Type Splitting at $\mathbb{1}$	$\langle \mathbb{1}\text{-}type\rangle \overset{\mathbb{1}}{\rightsquigarrow} [\langle type\rangle, \langle type\rangle]$
Type Splitting at $\mathbb{2}$	$\langle \mathbb{2}\text{-}type\rangle \overset{\mathbb{2}}{\rightsquigarrow} \langle type\rangle$
Context Splitting	$\langle \mathbb{1}\mathbb{2}\text{-}cont\rangle \rightsquigarrow [\langle cont\rangle, \langle cont\rangle]$
Term Splitting at $\mathbb{1}$	$\Gamma \Vdash \langle \mathbb{1}\text{-}exp\rangle : \langle \mathbb{1}\text{-}type\rangle \rightsquigarrow [\langle exp\rangle, \langle var\rangle.\langle exp\rangle]$
Term Splitting at $\mathbb{2}$	$\Gamma \Vdash \langle \mathbb{2}\text{-}exp\rangle : \langle \mathbb{2}\text{-}type\rangle \rightsquigarrow [\langle exp\rangle, \langle var\rangle.\langle exp\rangle]$

[...more...]

2.4 Splitting Theorems

2.4.1 It preserves types

The theorem we want to show has two parts, one for the $\mathbb{1}$ code, and one for the $\mathbb{2}$ code.

Figure 6: Value Splitting

$$\frac{\cdot}{() \overset{\mathbb{1}}{\rightsquigarrow} [(),()]}\ \text{unit}\overset{\mathbb{1}}{\rightsquigarrow} \qquad \frac{\cdot}{i \overset{\mathbb{1}}{\rightsquigarrow} [i,()]}\ \text{int}\overset{\mathbb{1}}{\rightsquigarrow} \qquad \frac{\cdot}{b \overset{\mathbb{1}}{\rightsquigarrow} [b,()]}\ \text{bool}\overset{\mathbb{1}}{\rightsquigarrow}$$

$$\frac{v_1 \overset{\mathbb{1}}{\rightsquigarrow} [u_1,w_1] \quad v_2 \overset{\mathbb{1}}{\rightsquigarrow} [u_2,w_2]}{(\!|v_1,v_2|\!) \overset{\mathbb{1}}{\rightsquigarrow} [(\!|u_1,u_2|\!),(\!|w_1,w_2|\!)]} \times\overset{\mathbb{1}}{\rightsquigarrow} \qquad \frac{v \overset{\mathbb{2}}{\rightsquigarrow} v'}{\textbf{next}\ v \overset{\mathbb{1}}{\rightsquigarrow} [(),v']} \bigcirc\overset{\mathbb{1}}{\rightsquigarrow}$$

Figure 7: Context Splitting

$$\frac{\cdot}{\bullet \rightharpoonup [\bullet\ ;\ \bullet]} \qquad \frac{\Gamma \rightharpoonup [\Gamma_1;\Gamma_2] \quad A \overset{\mathbb{1}}{\rightharpoonup} [A_1,A_2]}{\Gamma, v : A^{\mathbb{1}} \rightharpoonup [\Gamma_1, v : A_1; \Gamma_2, v : A_2]} \qquad \frac{\Gamma \rightharpoonup [\Gamma_1;\Gamma_2] \quad A \overset{\mathbb{2}}{\rightharpoonup} A'}{\Gamma, v : A^{\mathbb{2}} \rightharpoonup [\Gamma_1;\Gamma_2, v : A']}$$

If $\Gamma \Vdash e : A \rightsquigarrow [c, l.r]$	If $\Gamma \Vdash e : A \rightsquigarrow [p, l.r]$
then $\Gamma \rightharpoonup [\Gamma_1;\Gamma_2]$	then $\Gamma \rightharpoonup [\Gamma_1;\Gamma_2]$
and $A \overset{\mathbb{1}}{\rightharpoonup} [A_1, A_2]$	and $A \overset{\mathbb{2}}{\rightharpoonup} A'$
and $\Gamma_1 \vdash c : A_1 \times \tau$	and $\Gamma_1 \vdash p : \tau$
and $\Gamma_2, p : \tau \vdash r : A_2$	and $\Gamma_2, p : \tau \vdash r : A'$

2.4.2 It preserves meaning

The theorem we want to show has two parts, one for the $\mathbb{1}$ code, and one for the $\mathbb{2}$ code.

If $\bullet \Vdash e : A \rightsquigarrow [c, l.r]$	If $\bullet \Vdash e : B \rightsquigarrow [p, l.r]$
and $e \Downarrow_{\mathbb{1}} v$	and $e \Downarrow_{\mathbb{2}} v$
and $c \Downarrow (\!|t,u|\!)$	and $p \Downarrow u$
and $[u/l]r \Downarrow w$	and $[u/l]r \Downarrow w$
then $v \overset{\mathbb{1}}{\rightsquigarrow} [t,w]$	then $v \overset{\mathbb{2}}{\rightsquigarrow} w$

We proceed by cases of the term splitting judgement.

- **Int.** Suppose that

  1. $\Gamma \Vdash i : \text{int} \rightsquigarrow [(i,()), \_.()]$

  Since reduction is a function, we also have

  1. $i \Downarrow_{\mathbb{1}} i$
  2. $(i,()) \Downarrow_{\mathbb{1}} (\!|i,()|\!)$
  3. $[()/\_]() \Downarrow_{\mathbb{1}} ()$

  And since value splitting is a function,

  1. $i \overset{\mathbb{1}}{\rightsquigarrow} [i,()]$

  Which is what we need to show.

- **Tuple.** Suppose that

Figure 8: Term Splitting

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

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

$$\frac{\Gamma \Vdash e_1 : A \rightsquigarrow [c_1, l_1.r_1] \quad \Gamma \Vdash e_2 : B \rightsquigarrow [c_2, l_2.r_2]}{\Gamma \Vdash (e_1, e_2) : A \times B \rightsquigarrow \left[\begin{array}{l} (\texttt{let}\ y_1 = c_1\ \texttt{in let}\ y_2 = c_2\ \texttt{in}\ ((\pi_1 y_1, \pi_1 y_2), (\pi_2 y_1, \pi_2 y_2))), \\ l.(\texttt{let}\ l_1 = \pi_1 l\ \texttt{in}\ r_1, \texttt{let}\ l_2 = \pi_2 l\ \texttt{in}\ r_2) \end{array}\right]}\ \times\mathrm{I}\ \overset{\mathbb{1}}{\rightsquigarrow}$$

$$\frac{\Gamma \Vdash e_1 : A \rightsquigarrow [p_1, l_1.r_1] \quad \Gamma \Vdash e_2 : B \rightsquigarrow [p_2, l_2.r_2]}{\Gamma \Vdash (e_1, e_2) : A \times B \rightsquigarrow [(p_1, p_2), l.(\texttt{let}\ l_1 = \pi_1\ l\ \texttt{in}\ r_1, \texttt{let}\ l_2 = \pi_2\ l\ \texttt{in}\ r_2)]}\ \times\mathrm{I}\ \overset{\mathbb{2}}{\rightsquigarrow}$$

$$\frac{\Gamma \Vdash e : A \times B \rightsquigarrow [c, l.r]}{\Gamma \Vdash \pi_1\ e : A \rightsquigarrow [(\pi_1(\pi_1 c), \pi_2 c), l.\pi_1\ r]}\ \times\mathrm{E}_1\ \overset{\mathbb{1}}{\rightsquigarrow} \qquad \frac{\Gamma \Vdash e : A \times B \rightsquigarrow [p, l.r]}{\Gamma \Vdash \pi_1\ e : A \rightsquigarrow [p, l.\pi_1\ r]}\ \times\mathrm{E}_1\ \overset{\mathbb{2}}{\rightsquigarrow}$$

$$\frac{\Gamma \Vdash e : A \times B \rightsquigarrow [c, l.r]}{\Gamma \Vdash \pi_2\ e : B \rightsquigarrow [(\pi_2(\pi_1 c), \pi_2 c), l.\pi_2\ r]}\ \times\mathrm{E}_2\ \overset{\mathbb{1}}{\rightsquigarrow} \qquad \frac{\Gamma \Vdash e : A \times B \rightsquigarrow [p, l.r]}{\Gamma \Vdash \pi_2\ e : B \rightsquigarrow [p, l.\pi_2\ r]}\ \times\mathrm{E}_2\ \overset{\mathbb{2}}{\rightsquigarrow}$$

$$\frac{x : A^{\mathbb{1}} \in \Gamma}{\Gamma \Vdash v : A \rightsquigarrow [(x, ()), \_.()]}\ \mathrm{unit}\ \overset{\mathbb{1}}{\rightsquigarrow} \qquad \frac{x : A^{\mathbb{2}} \in \Gamma}{\Gamma \Vdash v : A \rightsquigarrow [(), \_.x]}\ \mathrm{unit}\ \overset{\mathbb{2}}{\rightsquigarrow}$$

$$\frac{\Gamma \Vdash e_1 : A \rightsquigarrow [c_1, l_1.r_1] \quad \Gamma, x : A^{\mathbb{1}} \Vdash e_2 : B \rightsquigarrow [c_2, l_2.r_2]}{\Gamma \Vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : B \rightsquigarrow [\cdots]}\ \mathrm{let}\ \overset{\mathbb{1}}{\rightsquigarrow}$$

$$\frac{\Gamma \Vdash e_1 : A \rightsquigarrow [p_1, l_1.r_1] \quad \Gamma, x : A^{\mathbb{2}} \Vdash e_2 : B \rightsquigarrow [p_2, l_2.r_2]}{\Gamma \Vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : B \rightsquigarrow [(p_1, p_2), l.\texttt{let}\ x = (\texttt{let}\ l_1 = \pi_1 l\ \texttt{in}\ r_1)\ \texttt{in let}\ l_2 = \pi_2 l\ \texttt{in}\ r_2]}\ \mathrm{let}\ \overset{\mathbb{2}}{\rightsquigarrow}$$

$$\frac{\Gamma \Vdash e : A \rightsquigarrow [p, l.r]}{\Gamma \Vdash \texttt{next}\ e : \bigcirc A \rightsquigarrow [((), p), l.r]}\ \bigcirc\mathrm{I}\ \overset{\mathbb{1}}{\rightsquigarrow} \qquad \frac{\Gamma \Vdash e : \bigcirc A \rightsquigarrow [c, l.r]}{\Gamma \Vdash \texttt{prev}\ e : A \rightsquigarrow [\pi_2\ c, l.r]}\ \bigcirc\mathrm{E}\ \overset{\mathbb{2}}{\rightsquigarrow}$$

$$\frac{\Gamma \Vdash e_1 : \mathrm{bool} \rightsquigarrow [c_1, l_1.r_1] \quad \Gamma \Vdash e_2 : A \rightsquigarrow [c_2, l_2.r_2] \quad \Gamma \Vdash e_3 : A \rightsquigarrow [c_3, l_3.r_3]}{\Gamma \Vdash \left(\begin{array}{l} \texttt{if}\ e_1 \\ \texttt{then}\ e_2 \\ \texttt{else}\ e_3 \end{array}\right) : A \rightsquigarrow \left[\begin{array}{l} \left(\begin{array}{l} \texttt{if}\ \pi_1 c_1 \\ \texttt{then let}\ y = c_2\ \texttt{in}\ (\pi_1 y, \texttt{inl}(\pi_2 y)) \\ \texttt{else let}\ y = c_3\ \texttt{in}\ (\pi_1 y, \texttt{inr}(\pi_2 y)) \end{array}\right), \\ l.(\texttt{case}\ \pi_1 l\ \texttt{of}\ l_2.r_2 \mid l_3.r_3) \end{array}\right]}\ +\mathrm{E}\ \overset{\mathbb{1}}{\rightsquigarrow}$$

$$\frac{\Gamma \Vdash e_1 : \mathrm{bool} \rightsquigarrow [p_1, l_1.r_1] \quad \Gamma \Vdash e_2 : A \rightsquigarrow [p_2, l_2.r_2] \quad \Gamma \Vdash e_3 : A \rightsquigarrow [p_3, l_3.r_3]}{\Gamma \Vdash \left(\begin{array}{l} \texttt{if}\ e_1 \\ \texttt{then}\ e_2 \\ \texttt{else}\ e_3 \end{array}\right) : A \rightsquigarrow \left[(p_1, p_2, p_3), l.\ \left(\begin{array}{l} \texttt{if}\ (\texttt{let}\ l_1 = \pi_1 l\ \texttt{in}\ r_1) \\ \texttt{then}\ (\texttt{let}\ l_2 = \pi_2 l\ \texttt{in}\ r_2) \\ \texttt{else}\ (\texttt{let}\ l_3 = \pi_3 l\ \texttt{in}\ r_3) \end{array}\right)\right]}\ +\mathrm{E}\ \overset{\mathbb{2}}{\rightsquigarrow}$$

1. $\Gamma \Vdash (e_1, e_2) : A \times B \rightsquigarrow \left[\begin{array}{l} (\texttt{let } y_1 = c_1 \texttt{ in let } y_2 = c_2 \texttt{ in } ((\pi_1 y_1, \pi_1 y_2), (\pi_2 y_1, \pi_2 y_2))), \\ l.(\texttt{let } l_1 = \pi_1 l \texttt{ in } r_1, \texttt{let } l_2 = \pi_2 l \texttt{ in } r_2) \end{array}\right]$
2. $(e_1, e_2) \Downarrow_{\mathbb{1}} v$
3. $\texttt{let } y_1 = c_1 \texttt{ in let } y_2 = c_2 \texttt{ in } ((\pi_1 y_1, \pi_1 y_2), (\pi_2 y_1, \pi_2 y_2)) \Downarrow (\!|t, u|\!)$
4. $[u/l](\texttt{let } l_1 = \pi_1 l \texttt{ in } r_1, \texttt{let } l_2 = \pi_2 l \texttt{ in } r_2) \Downarrow w$

Since the splitting and evaluation rules are syntax-directed, it follows that

1. $\Gamma \Vdash e_1 : A \rightsquigarrow [c_1, l_1.r_1]$
2. $\Gamma \Vdash e_2 : B \rightsquigarrow [c_2, l_2.r_2]$
3. v has the form $(\!|v_1, v_2|\!)$, by inversion of tuple evaluation
4. $(e_1, e_2) \Downarrow_{\mathbb{1}} (\!|v_1, v_2|\!)$
5. $e_1 \Downarrow_{\mathbb{1}} v_1$
6. $e_2 \Downarrow_{\mathbb{1}} v_2$
7. By inversion of evaluation, t has the form $(\!|t_1, t_2|\!)$ and u has the form $(\!|u_1, u_2|\!)$
8. $[(\!|t_1, u_1|\!), (\!|t_2, u_2|\!)/y_1, y_2]((\pi_1 y_1, \pi_1 y_2), (\pi_2 y_1, \pi_2 y_2)) \Downarrow_{\mathbb{1}} (\!|t, u|\!)$
9. $c_1 \Downarrow_{\mathbb{1}} (\!|t_1, u_1|\!)$, by inversion of let eval
10. $c_2 \Downarrow_{\mathbb{1}} (\!|t_2, u_2|\!)$, by inversion of let eval
11. By inversion of evaluation, w has the form $(\!|w_1, w_2|\!)$
12. $[(\!|u_1, u_2|\!)/l](\texttt{let } l_1 = \pi_1 l \texttt{ in } r_1, \texttt{let } l_2 = \pi_2 l \texttt{ in } r_2) \Downarrow (\!|w_1, w_2|\!)$, substituting for u, w
13. $[u_1/l_1]r_1 \Downarrow w_1$, inversion of evaluation
14. $[u_2/l_2]r_2 \Downarrow w_2$, inversion of evaluation

We now have enough to invoke the inductive hypothesis,

1. $v_1 \overset{\mathbb{1}}{\hookrightarrow} [t_1, w_1]$
2. $v_2 \overset{\mathbb{1}}{\hookrightarrow} [t_2, w_2]$
3. $(\!|v_1, v_2|\!) \overset{\mathbb{1}}{\hookrightarrow} [(\!|t_1, t_2|\!), (\!|w_1, w_2|\!)]$
4. $v \overset{\mathbb{1}}{\hookrightarrow} [t, w]$

Which is what we need to show.

3 Implementation

Not much to discuss. It exsists. Talk about how crufty the naive transformation is.

4 Related Work in Stage Seperation

Our starting language, λ^{12}, is essentially a two-stage version of $\lambda^{\bigcirc}$ from [1], but beefed up with products, sums, and fixed-points. That work motivates $\lambda^{\bigcirc}$ as an image of linear temporal logic under the Curry-Howard correspondence, and it makes the argument that $\lambda^{\bigcirc}$ is a good model for binding time analysis. Although $\lambda^{\bigcirc}$ was designed with partial evaluation in mind, it turns out to be a good model for pass separation as well. In contrast, [2] presents $\lambda^{\Box}$, which is based off of branching temporal logic. As that work shows, $\lambda^{\Box}$ is a good system for program generation, although we find the closed-code requirement to be more restricting than we need.

Pass seperation was introduced in [4]. They focused on motivating pass separation as a technique for compiler generation, and hinted at wider applicability. Their approach was not mechanized or automated in any sense, and their examples were separated by hand. We believe that out work represents the fulfillment of their prediction that "the [pass separation] approach will elude full automation for some time."

[3] also uses a pass separation technique for generating compiler/evaluator pairs from interpretters specified as term-rewrite systems. This heavy restriction on the form of the input prevents generalization of their style of pass separation. [I don't know enough about term-rewrite systems yet to be able to comment further.] Their method, while amenable to mechanization, was not actually implemented.

[5] represents the first attempt at fully automatic pass separation for code approaching general-purpose. In particular, they implemented pass separation for a C-like shading language including basic arithmetic and if statements. The main goal of their work, like ours, was to minimize recomputation. That said, their starting language was sufficiently restricted for recursive boundary types to be inexpressible. Given that memory is at a premium in shading languages, so much of that work was also focused on minimizing the memory-footprint of the boundary type.

References

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

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

[3] John Hannan. Operational semantics-directed compilers and machine architectures. *ACM Trans. Program. Lang. Syst.*, 16(4):1215–1247, 1994.

[4] Ulrik Jørring and William 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.

[5] Todd B. Knoblock and Erik 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.