Automatically Splitting a Two-Stage Lambda Calculus

Nicolas Feltman, Carlo Angiuli, Umut A. Acar, and Kayvon Fatahalian

Carnegie Mellon University

Abstract. Staged programming languages assign a stage to each program expression and evaluate each expression in its assigned stage. A common use of staged languages is to describe programs where inputs arrive at different times or rates. In this paper we present an algorithm for statically *splitting* these mixed-staged programs into two unstaged, but dependent, programs where the outputs of the first program can be efficiently reused across multiple invocations of the second. While previous algorithms for performing this transformation (also called *pass separation* and *data specialization*) were limited to operate on simpler, imperative languages, we define a splitting algorithm for an explicitly-two-stage, typed lambda calculus $\lambda^{\mathbb{I}2}$ with a $\bigcirc$ modality denoting computation at a later stage, and a ∇ modality noting purely first-stage code. Most notably, the algorithm splits mixed-stage recursive and higher-order functions. We prove the dynamic correctness of our splitting algorithm with respect to a partial-evaluation semantics, and mechanize this proof in Twelf. We also implement the algorithm in a prototype compiler, and demonstrate that the ability to split programs in a language featuring recursion and higher-order features enables non-trivial algorithmic transformations that improve code efficiency and also facilitates modular expression of staged programs.

1 Introduction

Consider a function F which, given $x, y_1, \ldots, y_m$, computes $f(x, y_i)$ for each y_i:

$\texttt{fun}\ F(x, y_1, y_2 \ldots, y_m)\ =$
$\quad f(x, y_1); f(x, y_2); \ldots; f(x, y_m)$

Observe that this implementation would be wasteful if $f(x, y_i)$ does a significant amount of work that does not depend on y_i. In such a case, it would be advantageous to find the computations in f that depend only on x and then stage execution of f so they are performed only once at the beginning of F.

Jørring and Scherlis classify automatic program staging transformations, such as the one described above, as forms of *frequency reduction* or *precomputation* [15]. To perform frequency reduction, one identifies and hoists computations that are performed multiple times, in order to compute them only once. To perform precomputation, one identifies computations that can be performed in advance and does so—for example, at compile time if the relevant inputs are statically known.

```
datatype list = Empty
              | Cons of int * list

fun part (p, Empty) =(0,Empty,Empty)
  | part (p, Cons (h,t)) =
    let val (n,le,ri) = part (p,t)
    in if h < p
       then (n+1,Cons(h,le),ri)
       else (n,le,Cons(h,ri))


qsel: list * int -> int
fun qsel (Empty, k) = 0
  | qsel (Cons ht, k) =
    let val (i,le,ri) = part ht in
    case compare k i of
      LT => qsel (le, k)
    | EQ => #1 ht
    | GT => qsel (ri, k-i-1)
```

(a) Unstaged quickselect.

```
@gr{
datatype list = Empty
              | Cons of int * list
fun part (p, Empty) = ...
}

qss : ∇list * ◯int -> ◯int
fun qss (gr{Empty},_) = next {0}
  | qss (gr{Cons ht},next{k}) =
    let
      @gr{val (i0,le,ri) = part ht}
      val next{i} = hold gr{i0}
    in next{
    case compare k i of
      LT => prev {
            qss (gr{le}, next{k})}
    | EQ => prev {hold gr{#1 ht}}
    | GT => prev {
       qss (gr{ri}, next{k-i-1})}
    }
```

(b) Staged quickselect in $\lambda^{\mathbb{I}2}$.

Fig. 1: Quickselect: traditional and staged.

One common precomputation technique is partial evaluation [9,13], which relies on dynamic compilation to specialize functions to known argument values. Going back to our example, if f specializes to a particular v, written f_v, such that $f(v, y) = f_v(y)$, then F can be specialized to v as

$$\texttt{fun}\ F_v(y_1, y_2, \ldots, y_m)\ = \\ f_v(y_1); f_v(y_2); \ldots; f_v(y_m)\,.$$

This eliminates the need to compute m times those parts of $f(v, -)$ which do not depend on the second argument.

Closely related to partial evaluation is *metaprogramming*, where known values represent program code to be executed in a later stage [4,31,5,22]. Metaprogramming enables fine-grained control over specialization by requiring explicit *staging annotations* that mark the stage of each expression.

While simple forms of frequency reduction include standard compiler optimizations such as loop hoisting and common subexpression elimination, Jørring and Scherlis proposed the more general transformation of *splitting* a program into multiple subfunctions (called pass separation in [15]). In our example, splitting transforms the function f into two others f_1 and f_2 such that $f(x, y_i) = f_2(f_1(x), y_i)$. Then we evaluate F by evaluating f_1 on x, and using the result z to evaluate the second function on each y_i.

$$\texttt{fun}\ F_{\text{multipass}}(x, y_1, y_2, \ldots, y_m)\ = \\ \texttt{let}\ z = f_1(x)\ \texttt{in}\ f_2(z, y_1); f_2(z, y_2); \ldots; f_2(z, y_m)$$

The key difference between splitting and partial evaluation (or metaprogramming) is that the former can be performed without access to the first argument x; $F_{\text{multipass}}$ works for any x, while F_v is defined only for $x = v$. Therefore, unlike partial evaluation, splitting is a static program transformation ("metastatic" in partial evaluation terminology) and does not require dynamic code generation.

Prior work on partial evaluation and metaprogramming has demonstrated automatic application of these techniques on higher order functional languages. In contrast, automatic splitting transformations have been limited to simpler languages [17,24,8,11].

In this paper, we present a splitting algorithm for $\lambda^{\mathbb{1}2}$, a two-staged typed lambda calculus in the style of Davies [4], with support for recursion and first-class functions. Like Davies, $\lambda^{\mathbb{1}2}$ uses a $\bigcirc$ modality to denote computation in the second stage, but to aid splitting we also add a ∇ modality to denote purely first-stage computations. The dynamic semantics (Section 3) of $\lambda^{\mathbb{1}2}$ are that of Davies, modified to provide an eager behavior between stages, which we believe is more intuitive in the context of splitting. We then prove the correctness of our splitting algorithm (Section 4) for $\lambda^{\mathbb{1}2}$ with respect to the semantics. Finally, we discuss our implementation of this splitting algorithm (Section 5) and demonstrate its power and behavior for a number of staged programs ranging from straight-line arithmetic operations to recursive and higher-order functions (Section 6).

We also demonstrate that splitting a recursive mixed-stage f yields an f_1 which computes a recursive data structure and an f_2 which traverses that structure in light of information available at the second stage. In the case of the quickselect algorithm, which we discuss next, the split code executes asymptotically faster than an unstaged evaluation of f.

2 Overview

Suppose that we wish to perform a series of order statistics queries on a list `l`. To this end, we can use the quickselect algorithm [12], which given a list `l` and an integer `k`, returns the element of `l` with rank `k` (i.e. the kth-largest element). As implemented in an ML-like language in Figure 1(a), `qs` partitions `l` using the first element as a pivot and then recurs on one of the two resulting sides, depending on the relationship of `k` to the size `i` of the first half, in order to find the desired element. If the index is out of range, a default value of `0` is returned. Assuming that the input list, with size n, is uniformly randomly ordered (which can be achieved by pre-permuting it), `qs` runs in expected $\Theta(n)$ time. Using `qs`, we can perform m different order statistics queries with ranks `k1`, ..., `km` as follows:

```
(qs l k1, qs l k2, ..., qs l km)
```

Unfortunately, this approach requires $\Theta(n \cdot m)$ time.

We can attempt to improve on this algorithm by factoring out computations shared between these calls to `qs`. In particular, we can construct a binary search tree out of `l`, at cost $\Theta(n \log n)$, and then simply look for the `k`th leftmost element of that tree. Doing lookups efficiently, in $\Theta(\log n)$ time, requires one

more innovation—augmenting the tree by storing at each node the size of its left subtree. Thus in total this approach has expected runtime $\Theta(n \log n + m \log n)$.

Is this method, wherein we precompute a data structure, better than the direct $\Theta(mn)$ method? The answer depends on the relationship between the number of lookups m and the size of the list n. If m is constant, then the direct method is superior for sufficiently large n. This is because the precomputed method does unnecessary work sorting parts of the list where there are no query points. However, if the number of queries grows with the size of the list, specficially in $\omega(\log n)$, then the precomputed method will be asymptotically faster.

Rewriting algorithms in this way—to precompute some intermediate results that depend on constant (or infrequently-varying) inputs—is non-trivial, as it requires implementing more complex data structures and algorithms. In this paper, we present a splitting algorithm which does much of this task automatically.

2.1 Staging

The idea behind staged programming is to use staging annotations—in our case, guided by types—to indicate the stage of each subterm.

In Figure 1(b) we show a staged version of `qs`, called `qss`, where first-stage code is colored red, and second-stage blue. In `qss`, we regard the input list `l` as arriving in the first stage (with type ∇list, a list "now"), the input rank `k` as arriving in the second stage (with type $\bigcirc$int, an integer in the "future"), and the result as being produced in the second stage (with type $\bigcirc$int).

`qss` is obtained from `qs` by wrapping certain computations with `prev` and `next`, signaling transitions between first- and second-stage code. Additionally, `gr` (*ground*) annotations mark certain first-stage components as being purely first-stage, rather than mixed-stage. We also use a function `hold : ∇int -> ◯int`, to promote first-stage integers to second-stage integers. Our type system ensures that the staging annotations in `qss` are consistent, in the sense that computations marked as first-stage cannot depend on ones marked as second-stage.

The process of automatically adding staging annotations to unstaged code, called *binding time analysis*, has been the subject of extensive research (Section 7). In this paper, we do not consider this problem, instead assuming that the annotations already exist. In the case of `qss`, we have specifically chosen annotations which maximize the work performed in the first stage.

2.2 Splitting Staged Programs

In the rest of this section, we present a high-level overview of the main ideas behind our splitting algorithm, applied to `qss`. Splitting `qss` yields a two-part program that creates a probabilistically balanced, augmented binary search tree as an intermediate data structure. In particular, its first part (`qs1`) constructs such a binary search tree and its second part (`qs2`) traverses the tree, using the embedded size information to find the element of the desired rank. The code for `qs1` and `qs2` is in Figure 2.

```
datatype tree = Leaf
              | Branch of int * int * tree * tree
```

```
datatype list = Empty
              | Cons of int * list
fun part (p : int, l : list) = ...

fun qs1 (l : list) : tree =
  case l of
    Empty => Leaf
  | Cons ht =>
      let val (i,le,ri) = part ht in
      Branch (i,#1 ht,qs1 le,qs1 ri)
```

```
fun qs2 (p : tree, k : int) :int=
  case p of
    Leaf => 0
  | Branch (i,h,p1,p2) =>
      case compare k i of
        LT => qs2 (p1,k)
      | EQ => h
      | GT => qs2 (p2,k-i-1)
```

Fig. 2: Two-pass implementation of quickselect.

Our splitting algorithm scans `qss` for first-stage computations, gathering them into `qs1`. Given `l`, this function performs these computations and places the information needed by the subsequent function into a boundary data structure. In particular, `qs1` performs all recursive calls and evaluates all instances of `part` (since it depends only on `l`). It produces a boundary data structure that collects the results from these recursive calls, tagged by the branch (`LT`, `EQ`, or `GT`) in which that call occurred. Since the recursive calls occur in two different branches (`LT` and `GT`) the boundary structure is a binary tree. Lastly, it records `i` (the size of the left subtree) and `#1 ht` (the pivot/head of the list) in the boundary structure, because those computations are *held* for use in the second stage. The final result is a binary search tree augmented with size information, and whose keys are the pivots.

Our splitting algorithm simultaneously scans `qss` for second-stage computations, gathering them into `qs2`. This function is given the boundary data structure and the rank `k`, and it finishes the computation. Now that `k` is known, the conditional on `compare k i` can be evaluated, choosing which recursive call of `qss` is actually relevant for this `k`. Since the boundary data structure contains the first-stage data for *all* of the recursive calls, performing these comparisons essentially walks the tree, using the rank along with the size data `i` to look up the `k`th leftmost node in the tree.

3 $\lambda^{\mathbb{1}\mathbb{2}}$ Statics and Dynamics

We express two-stage programs as terms in $\lambda^{\mathbb{1}\mathbb{2}}$, a typed, modal lambda calculus. Although $\lambda^{\mathbb{1}\mathbb{2}}$ describes computations that occur in two stages, we find it helpful for the specification of splitting to codify terms using one of three *worlds*. A world is essentially a slightly finer classification than a stage. Whereas there is only one world, $\mathbb{2}$, for second-stage computations, there are two worlds corresponding to the first stage: $\mathbb{1}\mathbb{M}$ for *mixed* first-stage computations, which may contain second-stage subterms within `next` blocks, and $\mathbb{1}\mathbb{G}$ for *ground* first-stage computations,

$$
\begin{array}{rrl}
\text{Worlds} & w & ::= \mathbb{1}\mathbb{M} \mid \mathbb{1}\mathbb{G} \mid \mathbb{2} \\
\{\mathbb{1}\mathbb{M},\mathbb{2}\}\text{-Types} & \tau & ::= \text{unit} \mid \tau \times \tau \mid \tau \to \tau \mid \alpha \mid \mu\alpha.\tau \mid \tau + \tau \\
& \tau_{\mathbb{M}} & ::= \text{unit} \mid \tau_{\mathbb{M}} \times \tau_{\mathbb{M}} \mid \tau_{\mathbb{M}} \to \tau_{\mathbb{M}} \mid \alpha \mid \mu\alpha.\tau_{\mathbb{M}} \mid \bigcirc\, \tau \mid \nabla\tau \\
\text{Contexts} & \Gamma & ::= \bullet \mid \Gamma, x_w : \tau_w \mathbin{@} w \\
\text{Terms} & e_w & ::= \texttt{app}(e_w\texttt{;}e_w) \mid \texttt{<}e_w, e_w\texttt{>} \mid \texttt{pi1}(e_w) \mid \texttt{pi2}(e_w) \\
& & \quad \mid \texttt{inl}(e_w) \mid \texttt{inr}(e_w) \mid \texttt{case}(e_w\texttt{;}x.e_w\texttt{;}x.e_w) \\
& & \quad \mid \texttt{roll}(e_w) \mid \texttt{unroll}(e_w) \\
& e_{\mathbb{M}} & ::= \underline{v} \mid \texttt{next}(e_{\mathbb{2}}) \mid \texttt{gr}(e_{\mathbb{G}}) \mid \texttt{letg}(e_{\mathbb{M}}\texttt{;}x_{\mathbb{G}}.e_{\mathbb{M}}) \\
& & \quad \mid \texttt{caseg}(e_{\mathbb{M}}\texttt{;}x_{\mathbb{M}}.e_{\mathbb{M}}\texttt{;}x_{\mathbb{M}}.e_{\mathbb{M}}) \\
& e_{\mathbb{G}} & ::= \underline{u} \\
& e_{\mathbb{2}} & ::= \underline{q} \mid \texttt{prev}(e_{\mathbb{M}}) \mid \texttt{fn}(x_{\mathbb{2}}.x_{\mathbb{2}}.e_{\mathbb{2}}) \\
\text{Partial Values} & v & ::= x_{\mathbb{M}} \mid \texttt{<>} \mid \texttt{fn}(x_{\mathbb{M}}.x_{\mathbb{M}}.e_{\mathbb{M}}) \mid \texttt{<}v, v\texttt{>} \mid \texttt{roll}(v) \\
& & \quad \mid \texttt{inl}(v) \mid \texttt{inr}(v) \mid \texttt{next}(y) \mid \texttt{gr}(v) \\
\text{Ground Values} & u & ::= x_{\mathbb{G}} \mid \texttt{<>} \mid \texttt{fn}(x_{\mathbb{G}}.x_{\mathbb{G}}.e_{\mathbb{G}}) \mid \texttt{<}u, u\texttt{>} \mid \texttt{roll}(u) \mid \texttt{inl}(u) \mid \texttt{inr}(u) \\
\text{Residuals} & q & ::= x_{\mathbb{2}} \mid \texttt{<>} \mid \texttt{fn}(x_{\mathbb{2}}.x_{\mathbb{2}}.q) \mid \texttt{app}(q\texttt{;}q) \mid \texttt{<}q, q\texttt{>} \mid \texttt{pi1}(q) \mid \texttt{pi2}(q) \\
& & \quad \mid \texttt{roll}(q) \mid \texttt{unroll}(q) \mid \texttt{inl}(q) \mid \texttt{inr}(q) \mid \texttt{case}(q\texttt{;}x_{\mathbb{2}}.q\texttt{;}x_{\mathbb{2}}.q)
\end{array}
$$

Fig. 3: $\lambda^{\mathbb{1}\mathbb{2}}$ abstract syntax. Subscript-w rules apply at all worlds.

which may not. The distinction between these two first-stage worlds is necessary for the splitting algorithm to produce efficient outputs and will be discussed in Section 4.6.

The abstract syntax of $\lambda^{\mathbb{1}\mathbb{2}}$ is presented as a grammar in Figure 3. To simplify the upcoming translation in Section 4, we have chosen to statically distinguish between values and general computations, using an underline constructor ("$\underline{v}$") which explicitly note the parts of a computation that have been reduced to a value.[1] Moreover, the value/computation distinction interacts non-trivially with the three possible world classifications. As a result, we end up with six classes of term in the grammar.

The key feature of all three forms of value is that they have no remaining work in the first stage. That is, all of the first-stage portions of a value are fully reduced (except the bodies of first-stage functions). In the case of values at $\mathbb{1}\mathbb{G}$, which we call *ground values*, this collapses to just the standard notion of values in a monostage language. In the case of values at $\mathbb{2}$, which we call *residuals*, this collapses to just standard monostage terms. Lastly, values at $\mathbb{1}\mathbb{M}$ end up with more of a mixed character, so we call them *partial values*.

[1] In this presentation, the value/computation distinction is encoded intrinsically at the syntactic level. However, we found it more convenient in the formal Twelf implementation to maintain the distinction with an extrinsic judgement.

$$\frac{\Gamma \vdash v : A \mathbin{@} w}{\Gamma \vdash \underline{v} : A \mathbin{@} w} \qquad \frac{\cdot}{\Gamma \vdash \texttt{<>} : \mathrm{unit} \mathbin{@} w} \qquad \frac{\Gamma, f : A \to B \mathbin{@} w, x : A \mathbin{@} w \vdash e : B \mathbin{@} w}{\Gamma \vdash \texttt{fn(}f\texttt{.}x\texttt{.}e\texttt{)} : A \to B \mathbin{@} w}$$

$$\frac{\Gamma \vdash e_1 : A \to B \mathbin{@} w \quad \Gamma \vdash e_2 : A \mathbin{@} w}{\Gamma \vdash \texttt{app(}e_1\texttt{;}e_2\texttt{)} : B \mathbin{@} w} \qquad \frac{\Gamma \vdash e_1 : A \mathbin{@} w \quad \Gamma \vdash e_2 : B \mathbin{@} w}{\Gamma \vdash \texttt{<}e_1\texttt{,}\ e_2\texttt{>} : A \times B \mathbin{@} w}$$

$$\frac{\Gamma \vdash e : A \times B \mathbin{@} w}{\Gamma \vdash \texttt{pi1(}e\texttt{)} : A \mathbin{@} w} \qquad \frac{\Gamma \vdash e : A \times B \mathbin{@} w}{\Gamma \vdash \texttt{pi2(}e\texttt{)} : B \mathbin{@} w} \qquad \frac{\Gamma \vdash e : \mu\alpha.\tau \mathbin{@} w}{\Gamma \vdash \texttt{unroll(}e\texttt{)} : [\mu\alpha.\tau/\alpha]\tau \mathbin{@} w}$$

$$\frac{\Gamma \vdash e : A \mathbin{@} w}{\Gamma \vdash \texttt{inl(}e\texttt{)} : A + B \mathbin{@} w} \qquad \frac{\Gamma \vdash e : B \mathbin{@} w}{\Gamma \vdash \texttt{inr(}e\texttt{)} : A + B \mathbin{@} w} \qquad \frac{\Gamma \vdash e : [\mu\alpha.\tau/\alpha]\tau \mathbin{@} w}{\Gamma \vdash \texttt{roll(}e\texttt{)} : \mu\alpha.\tau \mathbin{@} w}$$

$$\frac{\Gamma \vdash e_1 : A + B \mathbin{@} w \quad \Gamma, x_2 : A \mathbin{@} w \vdash e_2 : C \mathbin{@} w \quad \Gamma, x_3 : B \mathbin{@} w \vdash e_3 : C \mathbin{@} w}{\Gamma \vdash \texttt{case(}e_1\texttt{;}x_2\texttt{.}e_2\texttt{;}x_3\texttt{.}e_3\texttt{)} : C \mathbin{@} w}$$

$$\frac{\Gamma \vdash e : A \mathbin{@} \mathbb{2}}{\Gamma \vdash \texttt{next(}e\texttt{)} : \bigcirc A \mathbin{@} \mathbb{1}\mathbb{M}} \qquad \frac{\Gamma \vdash e : \bigcirc A \mathbin{@} \mathbb{1}\mathbb{M}}{\Gamma \vdash \texttt{prev(}e\texttt{)} : A \mathbin{@} \mathbb{2}} \qquad \frac{\Gamma \vdash e : A \mathbin{@} \mathbb{1}\mathbb{G}}{\Gamma \vdash \texttt{gr(}e\texttt{)} : \nabla A \mathbin{@} \mathbb{1}\mathbb{M}}$$

$$\frac{\Gamma \vdash e_1 : \nabla A \mathbin{@} \mathbb{1}\mathbb{M} \quad \Gamma, x : A \mathbin{@} \mathbb{1}\mathbb{G} \vdash e_2 : B \mathbin{@} \mathbb{1}\mathbb{M}}{\Gamma \vdash \texttt{letg(}e_1\texttt{;}x\texttt{.}e_2\texttt{)} : B \mathbin{@} \mathbb{1}\mathbb{M}} \qquad \frac{\Gamma \vdash e_1 : \nabla(A + B) \mathbin{@} \mathbb{1}\mathbb{M} \quad \Gamma, x_2 : \nabla A \mathbin{@} \mathbb{1}\mathbb{M} \vdash e_2 : C \mathbin{@} \mathbb{1}\mathbb{M} \quad \Gamma, x_3 : \nabla B \mathbin{@} \mathbb{1}\mathbb{M} \vdash e_3 : C \mathbin{@} \mathbb{1}\mathbb{M}}{\Gamma \vdash \texttt{caseg(}e_1\texttt{;}x_2\texttt{.}e_2\texttt{;}x_3\texttt{.}e_3\texttt{)} : C \mathbin{@} \mathbb{1}\mathbb{M}}$$

Fig. 4: $\lambda^{\mathbb{1}\mathbb{2}}$ statics, split into standard rules and staged rules.

3.1 Statics

The typing judgment $\Gamma \vdash e : A \mathbin{@} w$, defined in Figure 4, means that e has type A at world w, in the context Γ.

All three worlds contain unit, product, function, sum and recursive types defined in the usual fashion. These "standard" features can only be constructed from subterms of the same world, and variables can only be used at the same world where they were introduced. Thus differing worlds (and hence, differing stages of computation) only interact by means of the $\bigcirc$ and ∇ type formers. These modalities are internalizations of worlds $\mathbb{2}$ and $\mathbb{1}\mathbb{G}$, respectively, as types at world $\mathbb{1}\mathbb{M}$.

At the term level, `next` blocks can be used to form future computations: given a term e of type A at world $\mathbb{2}$, `next(`e`)` has type $\bigcirc A$ at $\mathbb{1}\mathbb{M}$. This essentially encapsulates e as a computation that will be evaluated in the future, and it provides a handle (of type $\bigcirc A$) now to that eventual value. Computations at $\mathbb{1}\mathbb{M}$ can shuffle this handle around as a value, but the future result it refers to cannot be accessed. This is because the only way to eliminate a $\bigcirc$ wrapper is by using a `prev`, which yields an A at $\mathbb{2}$. This feature was adapted from linear temporal logic, via [4], and ensures that there can be no flow of information from the second stage to the first.

∇A is a type in world $\mathbb{1}\mathbb{M}$ which classifies purely-first-stage computations of type A. Given a world $\mathbb{1}\mathbb{G}$ term e of type A, $\texttt{gr}(e)$ has type ∇A at world $\mathbb{1}\mathbb{M}$. (e is guaranteed not to contain second-stage computations because $\bigcirc$ types are not available in world $\mathbb{1}\mathbb{G}$.) An e of type ∇A at $\mathbb{1}\mathbb{M}$ can be unwrapped as an A at $\mathbb{1}\mathbb{G}$ using the $\texttt{letg}(e;x.e')$ construct, which binds $x : A @ \mathbb{1}\mathbb{G}$ in a $\mathbb{1}\mathbb{M}$ term e'. This allows us to compute under ∇—for example, given a $p : \nabla(A \times B) @ \mathbb{1}\mathbb{M}$, the term $\texttt{letg}(p;x.\texttt{gr}(\texttt{pi1}(x)))$ computes its first projection, of type ∇A. This elimination form, in contrast to that of $\bigcirc$, does not permit world $\mathbb{1}\mathbb{M}$ subterms within any world $\mathbb{1}\mathbb{G}$ term.

These features are sufficient to ensure that mixed code does not leak into ground code, however they also prevent information from ever escaping a ∇ wrapper. So to allow the latter behavior but not the former, we introduce the $\texttt{caseg}(e;x.e_1;x.e_2)$ construct, whose predicate is of type $\nabla(A + B)$ and whose branches are world $\mathbb{1}\mathbb{M}$ terms open on ∇A and ∇B respectively. This essentially allows code at $\mathbb{1}\mathbb{M}$ to inspect an injection tag within a ∇.

Although products and functions are restricted to types at the same world, $\bigcirc$ allows construction of "mixed-stage" products and functions. For example, `qss` is a function at world $\mathbb{1}\mathbb{M}$ which takes a $\nabla\text{list} \times \bigcirc\text{int}$ (a purely-first-stage list and a second-stage integer) to a $\bigcirc$int (a second-stage computation of an integer).

The example code in this paper uses an extension of the formalized $\lambda^{\mathbb{1}\mathbb{2}}$. In particular, it makes liberal use of ints and various functions on these, as well as a function `hold()` which takes a ∇int to a $\bigcirc$int.[2]

3.2 Dynamics

The central tenet of a staged language is that first-stage code should be evaluated entirely before second-stage code. Accordingly, our dynamics operates in two passes. The first pass takes an input top-level program $e : A @ \mathbb{2}$ and reduces all of its first-stage (worlds $\mathbb{1}\mathbb{M}$ and $\mathbb{1}\mathbb{G}$) subterms in place, eventually resulting in a residual q. The second pass further reduces this residual. Since q is monostage by definition, this second pass is standard unstaged evaluation and is not described in further detail in this paper. Moreover, for the purposes of these dynamics, we consider a top level program to always be typed at world $\mathbb{2}$.

Since $e : A @ \mathbb{2}$ may be constructed out of terms at other worlds, our dynamics requires notions of values and steps that are specialized to each world. The rules for all parts of first-pass evaluation are given in Figure 5. In this and later figures, we extensively use an $\mathcal{S}[\text{-}]$ construction to indicate a *shallow* evaluation context which looks a single level deep.

World $\mathbb{2}$. Steps at world $\mathbb{2}$ are given by the judgment $e \xhookrightarrow{\mathbb{2}} e'$. Since first pass evaluation should not reduce stage two terms, this judgment does nothing but

[2] `hold()` is definable in $\lambda^{\mathbb{1}\mathbb{2}}$ given an inductive definition of ints. In practice, we provide both ints and `hold()` as primitives. It is sensible to extend `hold()` to all base types and to products and sums thereof. This is related to the notion of *mobility* ([21]).

Shallow Reductions, $e \xrightarrow{w} e'$		
e	e'	w
$\texttt{<}\underline{v_1}\texttt{,}\,\underline{v_2}\texttt{>}$	$\underline{\texttt{<}v_1\texttt{,}\,v_2\texttt{>}}$	all
$\texttt{pi1(}\underline{\texttt{<}v_1\texttt{,}\,v_2\texttt{>}}\texttt{)}$	$\underline{v_1}$	$\mathbb{1}\mathbb{G}, \mathbb{1}\mathbb{M}$
$\texttt{pi2(}\underline{\texttt{<}v_1\texttt{,}\,v_2\texttt{>}}\texttt{)}$	$\underline{v_2}$	$\mathbb{1}\mathbb{G}, \mathbb{1}\mathbb{M}$
$\texttt{pi1(}\underline{v}\texttt{)}$	$\underline{\texttt{pi1(}v\texttt{)}}$	$\mathbb{2}$
$\texttt{pi2(}\underline{v}\texttt{)}$	$\underline{\texttt{pi2(}v\texttt{)}}$	$\mathbb{2}$
$\texttt{app(}\underline{\texttt{fn(}f\texttt{.}x\texttt{.}e\texttt{)}}\texttt{;}\underline{v}\texttt{)}$	$[\texttt{fn(}f\texttt{.}x\texttt{.}e\texttt{)}, v/f, x]e$	$\mathbb{1}\mathbb{G}, \mathbb{1}\mathbb{M}$
$\texttt{app(}\underline{v_1}\texttt{;}\underline{v_2}\texttt{)}$	$\underline{\texttt{app(}v_1\texttt{;}v_2\texttt{)}}$	$\mathbb{2}$
$\texttt{inl(}\underline{v}\texttt{)}$	$\underline{\texttt{inl(}v\texttt{)}}$	all
$\texttt{inr(}\underline{v}\texttt{)}$	$\underline{\texttt{inr(}v\texttt{)}}$	all
$\texttt{case(}\underline{\texttt{inl(}v\texttt{)}}\texttt{;}x_2\texttt{.}e_2\texttt{;}\cdot\texttt{)}$	$[v/x_2]e_2$	$\mathbb{1}\mathbb{G}, \mathbb{1}\mathbb{M}$
$\texttt{case(}\underline{\texttt{inr(}v\texttt{)}}\texttt{;}\cdot\texttt{;}x_3\texttt{.}e_3\texttt{)}$	$[v/x_3]e_3$	$\mathbb{1}\mathbb{G}, \mathbb{1}\mathbb{M}$
$\texttt{roll(}\underline{v}\texttt{)}$	$\underline{\texttt{roll(}v\texttt{)}}$	all
$\texttt{unroll(}\underline{\texttt{roll(}v\texttt{)}}\texttt{)}$	$\underline{v}$	$\mathbb{1}\mathbb{G}, \mathbb{1}\mathbb{M}$
$\texttt{unroll(}\underline{v}\texttt{)}$	$\underline{\texttt{unroll(}v\texttt{)}}$	$\mathbb{2}$
$\texttt{case(}\underline{e_1}\texttt{;}x_2\texttt{.}\underline{e_2}\texttt{;}x_3\texttt{.}\underline{e_3}\texttt{)}$	$\underline{\texttt{case(}e_1\texttt{;}x_2\texttt{.}e_2\texttt{;}x_3\texttt{.}e_3\texttt{)}}$	$\mathbb{2}$
$\texttt{next(}\underline{y}\texttt{)}$	$\underline{\texttt{next(}y\texttt{)}}$	$\mathbb{1}\mathbb{M}$
$\texttt{prev(}\underline{\texttt{next(}y\texttt{)}}\texttt{)}$	$\underline{y}$	$\mathbb{2}$
$\texttt{gr(}\underline{v}\texttt{)}$	$\underline{\texttt{gr(}v\texttt{)}}$	$\mathbb{1}\mathbb{M}$
$\texttt{letg(}\underline{\texttt{gr(}v\texttt{)}}\texttt{;}x\texttt{.}e\texttt{)}$	$[v/x]e$	$\mathbb{1}\mathbb{M}$
$\texttt{caseg(}\underline{\texttt{gr(inl(}v\texttt{))}}\texttt{;}x_2\texttt{.}e_2\texttt{;}\cdot\texttt{)}$	$[\texttt{gr(}v\texttt{)}/x_2]e_2$	$\mathbb{1}\mathbb{M}$
$\texttt{caseg(}\underline{\texttt{gr(inr(}v\texttt{))}}\texttt{;}\cdot\texttt{;}x_3\texttt{.}e_3\texttt{)}$	$[\texttt{gr(}v\texttt{)}/x_3]e_3$	$\mathbb{1}\mathbb{M}$

Shallow Contexts, $\mathcal{S}\ \mathsf{vc}_w$	
$\mathcal{S}$[-]	w
$\texttt{<-,}\,e\texttt{>}$	all
$\texttt{<}\underline{v}\texttt{,->}$	all
$\texttt{pi1(-)}$	all
$\texttt{pi2(-)}$	all
$\texttt{fn(}f\texttt{.}x\texttt{.-)}$	$\mathbb{2}$
$\texttt{app(-;}e\texttt{)}$	all
$\texttt{app(}\underline{v}\texttt{;-)}$	all
$\texttt{inl(-)}$	all
$\texttt{inr(-)}$	all
$\texttt{roll(-)}$	all
$\texttt{unroll(-)}$	all
$\texttt{case(-;}x_2\texttt{.}e_2\texttt{;}x_3\texttt{.}e_3\texttt{)}$	all
$\texttt{case(}\underline{v_1}\texttt{;}x_2\texttt{.-;}x_3\texttt{.}e_3\texttt{)}$	$\mathbb{2}$
$\texttt{case(}\underline{v_1}\texttt{;}x_2\texttt{.}\underline{v_2}\texttt{;}x_3\texttt{.-)}$	$\mathbb{2}$
$\texttt{caseg(-;}x_2\texttt{.}e_2\texttt{;}x_3\texttt{.}e_3\texttt{)}$	$\mathbb{1}\mathbb{M}$
$\texttt{letg(-;}x\texttt{.}e\texttt{)}$	$\mathbb{1}\mathbb{M}$

$$\frac{e \xrightarrow{w} e'}{e \xhookrightarrow{w} e'} \qquad \frac{e \xhookrightarrow{w} e' \quad \mathcal{S}\ \mathsf{vc}_w}{\mathcal{S}[e] \xhookrightarrow{w} \mathcal{S}[e']} \qquad \frac{e \xhookrightarrow{\mathbb{1}\mathbb{G}} e'}{\texttt{gr(}e\texttt{)} \xhookrightarrow{\mathbb{1}\mathbb{M}} \texttt{gr(}e'\texttt{)}} \qquad \frac{e \xhookrightarrow{\mathbb{2}} e'}{\texttt{next(}e\texttt{)} \xhookrightarrow{\mathbb{1}\mathbb{M}} \texttt{next(}e'\texttt{)}} \qquad \frac{e \xhookrightarrow{\mathbb{1}\mathbb{M}} e'}{\texttt{prev(}e\texttt{)} \xhookrightarrow{\mathbb{2}} \texttt{prev(}e'\texttt{)}}$$

$$\frac{q \text{ not a variable}}{\texttt{next(}q\texttt{)} \nearrow [\![q/y]\!]\texttt{next(}y\texttt{)}} \qquad \frac{e \nearrow [\![q/y]\!]e'}{\texttt{prev(}e\texttt{)} \xhookrightarrow{\mathbb{2}} \texttt{let(}y\texttt{;}q\texttt{.prev(}e'\texttt{))}} \qquad \frac{e \nearrow [\![q/y]\!]e' \quad \mathcal{S}\ \mathsf{vc}_{\mathbb{1}\mathbb{M}}}{\mathcal{S}[e] \nearrow [\![q/y]\!]\mathcal{S}[e']}$$

Fig. 5: First-pass evaluation of $\lambda^{\mathbb{1}\mathbb{2}}$ dynamics. The judgement $e \xhookrightarrow{w} e'$ indicates that e takes a step to e' in the first pass; $\xrightarrow{w}$ and $\mathcal{S}\ \mathsf{vc}_w$ are helper judgements.

traverse e to find `prev` blocks, under which it performs in-place reductions. A world $\mathbb{2}$ term is done evaluating when it has the form q, where q is a residual. To be a residual, a term must have no first-stage subterms (equivalently, no `prev`s), even within the body of a function or branches of a case. This implies that $\xhookrightarrow{\mathbb{2}}$ must proceed underneath second-stage binders.

World $\mathbb{1}\mathbb{G}$. Since the ground fragment of the language is not dependent on other worlds, the semantics of ground is just that of a monostage language. Thus, $e \xhookrightarrow{\mathbb{1}\mathbb{G}} e'$ traverses into subterms to find the left-most unevaluated code where it performs a reduction. A ground value u comprises only units, injections, tuples, and functions, where the body of the function may be any ground term.

World $\mathbb{1}\mathbb{M}$. Like its ground counterpart, the $\mathbb{1}\mathbb{M}$ step judgment, $e \overset{\mathbb{1}\mathbb{M}}{\hookrightarrow} e'$, finds the left-most unevaluated subterm and performs a reduction. It also descends into `gr` and `next` blocks, using one of the other two step judgements ($\overset{\mathbb{1}\mathbb{G}}{\hookrightarrow}$ or $\overset{\mathbb{2}}{\hookrightarrow}$) there. The value form for $\mathbb{1}\mathbb{M}$, called a partial value, comprises units, tuples, functions, `gr` blocks of ground values, and `next` blocks containing *only a stage two variable*. This strong requirement ensures that second-stage computations are not duplicated when partial values are substituted for a variable. This is a departure from the staged semantics of [30] and [4]. Whereas those semantics interpret values of type $\bigcirc A$ to mean "code of type A that can be executed in the future," ours interprets $\bigcirc A$ to mean "a reference to a value that will be accessible in the future." This contrast stems from differing goals: metaprogramming explicitly intends to model code manipulating code, whereas our applications feel more natural with an eager interpretation of `next`. One consequence of the stronger requirement on partial values is that a new kind of step is necessary to put terms into that form. To illustrate, consider:

```
prev{(fn x : ◯int => e') (next{e})}
```

We could reduce this to `prev(`$[$`next(`e`)`$/x]e'$`)`, but this may potentially duplicate an expensive computation e depending on how many times x appears in e'. Instead, we choose to *hoist* e outside, binding it to a temporary variable y, and substituting that variable instead:

```
let val y = e in prev{[next{y}/x]e'}
```

This behavior is implemented by the $e \nearrow [\![q/y]\!]e'$ judgment, called a *hoisting step*. We read this as saying that somewhere within e there was a subterm q which needs to be hoisted out, yielding the new term e' which has a new variable y where q used to be. These steps occur when a `next` block has contents that are a residual but (to prevent loops) not when those contents are already a variable. In essence, the rules for hoisting steps operate by "bubbling up" a substitution to the innermost containing `prev`, where it is reified into a `let` statement.[3]

Consider the following example, where `P` has type $\nabla(\text{unit} + \text{unit})$,

```
caseg P of _ => next{0} | _ => next{1}
```

Depending on what `P` evaluates to in the first stage, the whole term will step to either `next{0}` or `next{1}`. In this sense, we can see `case` (at world $\mathbb{1}\mathbb{M}$) and `caseg` as the constructs that facilitate all cross-stage communication.

3.3 Type Safety

The statics and dynamics of $\lambda^{\mathbb{1}\mathbb{2}}$ are related by the type safety theorems below, again annotated by world. In all cases, Γ may be any list of variable bindings at world $\mathbb{2}$, representing the second-stage binders under which we are evaluating. Note how the progress theorem for world $\mathbb{1}\mathbb{M}$ states that every well-typed term must take either a standard step or a hoisting step.

[3] Because a program is a term at $\mathbb{2}$, this `prev` always exists. Otherwise, the semantics would need a mechanism to accumulate the bindings that hoisting steps create.

Theorem 1 (Progress).

- *If* $\Gamma \vdash e : A \mathbin{@} \mathbb{1}\mathbb{M}$*, then either* e *has the form* $\underline{v}$*, or* $e \overset{\mathbb{1}\mathbb{M}}{\hookrightarrow} e'$*, or* $e \nearrow [\![q/y]\!]e'$*.*
- *If* $\Gamma \vdash e : A \mathbin{@} \mathbb{1}\mathbb{G}$*, then either* e *has the form* $\underline{u}$*, or* $e \overset{\mathbb{1}\mathbb{G}}{\hookrightarrow} e'$*.*
- *If* $\Gamma \vdash e : A \mathbin{@} \mathbb{2}$*, then either* e *has the form* $\underline{q}$*, or* $e \overset{\mathbb{2}}{\hookrightarrow} e'$*.*

Theorem 2 (Preservation).

- *If* $\Gamma \vdash e : A \mathbin{@} \mathbb{1}\mathbb{M}$ *and* $e \nearrow [\![q/y]\!]e'$*,*
  then $\Gamma \vdash q : B \mathbin{@} \mathbb{2}$ *and* $\Gamma, y : B \mathbin{@} \mathbb{2} \vdash e' : A \mathbin{@} \mathbb{1}\mathbb{M}$*.*
- *If* $\Gamma \vdash e : A \mathbin{@} w$ *and* $e \overset{w}{\hookrightarrow} e'$*, then* $\Gamma \vdash e' : A \mathbin{@} w$*.*

3.4 Evaluating Staged Programs

Multistage functions, such as `qss` from Section 2, can be represented as terms with a type fitting the pattern $A \to \bigcirc(B \to C)$ at $\mathbb{1}\mathbb{M}$.[4] To apply such a function f to arguments $a : A \mathbin{@} \mathbb{1}\mathbb{M}$ and $b : B \mathbin{@} \mathbb{2}$, simply evaluate the program:

```
prev{f a} b
```

Moreover, the reuse of first-stage computations across multiple second-stage computations can even be encoded within $\lambda^{\mathbb{1}\mathbb{2}}$. The following program runs many order statistics queries `k1`, ..., `km` on the same list:

```
prev{ let val list = gr{[7,4,2,5,9,...,3]} in
next{ let fun lookup k = prev{qss (list,next{k})}
      in (lookup k1,...,lookup km)}}
```

Observe that this code evaluates `qss` only once and subsitutes its result into the body of the function `lookup`, which is then called many times in the second stage.

4 Splitting Algorithm

The goal of a stage splitting translation is to send a program $\mathcal{P}$ in a multistage language to an equivalent form $\mathcal{P}'$ where the stages are seperated at the top level. More specifically, $\mathcal{P}$ and $\mathcal{P}'$ should produce the same answer under their respective semantics.

Since $\lambda^{\mathbb{1}\mathbb{2}}$ has three classes of multistage term, our splitting algorithm has three forms: $e \overset{\mathbb{2}}{\rightsquigarrow} \{p \,|\, l \,.\, r\}$ for $\mathbb{2}$-terms, $e \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{c \,|\, l.r\}$ for $\mathbb{1}\mathbb{M}$-terms, and $v \rightsquigarrow \{i \,;\, q\}$ for partial values. In each form the output has two parts, corresponding to the first-stage (p, c, and i) and second-stage ($l.r$, and q) content of the input. Note that there's no need to provide forms of splitting for ground terms or residuals, since those classes of term are already monostage by construction.

The rules of the three splitting judgements are given in Figures 6 and 7. Since the rules are simply recursive on the structure of the term, the splitting algorithm runs in linear time on the size of the input program. Splitting is defined for all well-typed inputs (Theorem 3), and it produces unique results (Theorem 4). That is, each splitting judgement defines a total function.

[4] We can rewrite `qss` in this curried form, or apply a higher-order currying function.

Theorem 3 (Splitting Totality).

- *For term e, if $\Gamma \vdash e : A \mathbin{@} \mathbb{2}$, then $e \overset{\mathbb{2}}{\rightsquigarrow} \{p \mid l \,.\, r\}$.*
- *For term e, if $\Gamma \vdash e : A \mathbin{@} \mathbb{1}\mathbb{M}$, then $e \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{c \mid l.r\}$.*
- *For partial value v, if $\Gamma \vdash v : A \mathbin{@} \mathbb{1}\mathbb{M}$, then $v \hookrightarrow \{i\,;q\}$.*

Theorem 4 (Splitting Uniqueness).

- *If $e \overset{\mathbb{2}}{\rightsquigarrow} \{p \mid l \,.\, r\}$ and $e \overset{\mathbb{2}}{\rightsquigarrow} \{p' \mid l' \,.\, r'\}$, then $p = p'$, and $l.r = l'.r'$.*
- *If $e \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{p \mid l.r\}$ and $e \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{c' \mid l'.r'\}$, then $c = c'$, and $l.r = l'.r'$.*
- *If $v \hookrightarrow \{i\,;q\}$ and $v \hookrightarrow \{i'\,;q'\}$, then $i = i'$ and $q = q'$.*

We prove these theorems by straightforward induction on the typing derivation and simultaneous induction on the splitting derivations, respectively.

4.1 Outputs of Splitting

Splitting a top level program $e : A \mathbin{@} \mathbb{2}$, via $\overset{\mathbb{2}}{\rightsquigarrow}$, yields $\{p \mid l \,.\, r\}$. Like e, this output is evaluated in two passes. The first pass reduces p to the value b and plugs this result in for l to produce $[b/l]r$; the second pass evaluates $[b/l]r$. The relationship between the two stages in this case is thus like a pipeline, which is why we write them with a '|' in between. Since execution of the first pass serves to generate input for the second pass, we say p is a *precomputation* that produces a *boundary value* (b) for the *resumer* ($l.r$).

Splitting a partial value v, via $\hookrightarrow$, yields $\{i\,;q\}$. Since partial values, by definition, have no remaining work in the first pass and since transfer of information between the stages occurs in the first pass of evalution, we know that the second-stage components of v can no longer depend on its first-stage components. Analogously, this must also hold for the output of splitting, which is why q—unlike the resumer of world $\mathbb{2}$ term splitting—is not open on a variable. Thus, i and q are operationally independent, but they represent the complementary portions of v that are relevant to each stage. We call i the *immediate value* and q the residual.

For any $e : A \mathbin{@} \mathbb{1}\mathbb{M}$, splitting e via $\overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow}$ yields the pair of monostage terms $\{c \mid l \,.\, r\}$. This output form is essentially a hybrid of the previous two. Because e is a term, c needs to produces a boundary value b to be passed to the resumer ($l.r$) And since e types at world $\mathbb{1}\mathbb{M}$, it has an eventual result at the first stage as well as the second, and so it must produce an immediate value i. The term c meets both of these responsibilities by reducing to the tuple $\langle i, b \rangle$, and so we call it a *combined term*.

4.2 World $\mathbb{2}$ Term Splitting

The dynamic correctness of the splitting translation requires that the simple evaluate-and-plug semantics on the output produces the same answer as the staged semantics of the previous section. That is, $[b/l]r$ (the *applied resumer*) should be equivalent to the residual q produced by direct evaluation, $e \overset{\mathbb{2}}{\hookrightarrow} \cdots \overset{\mathbb{2}}{\hookrightarrow}$

q. This condition is stated more precisely as Theorem 5, where "$e \Downarrow v$" indicates standard monostage reduction of the term e to the value v, and "$\equiv$" indicates a monostage equivalence, which is defined in Figure 8.

Theorem 5 (End-to-End Correctness). *If* $\cdot \vdash e : A \,@\, \mathbb{2}$, $e \xhookrightarrow{\mathbb{2}} \cdots \xhookrightarrow{\mathbb{2}} q$, *and* $e \overset{\mathbb{2}}{\rightsquigarrow} \{p \mid l \,.\, r\}$, *then* $p \Downarrow b$ *and* $[b/l]r \equiv q$.

We prove Theorem 5 by induction on the steps of evaluation. In the base case, where e is already a residual of the form q, we know $q \overset{\mathbb{2}}{\rightsquigarrow} \{\underline{<>} \mid \_ \,.\, q\}$, so by uniqueness of splitting, $p = <>$ and $r = q$. From here, we can directly derive $\underline{<>} \Downarrow <>$ and $q \equiv q$.

In the recursive case, where the evaluation takes at least one step, we have $e \xhookrightarrow{\mathbb{2}} e' \xhookrightarrow{\mathbb{2}} \cdots \xhookrightarrow{\mathbb{2}} q$ as well as $\cdot \vdash e : A \,@\, \mathbb{2}$ and $e \overset{\mathbb{2}}{\rightsquigarrow} \{p \mid l \,.\, r\}$. By preservation and totality of splitting, we know $\cdot \vdash e' : A \,@\, \mathbb{2}$ and $e' \overset{\mathbb{2}}{\rightsquigarrow} \{p' \mid l' \,.\, r'\}$. From here, the inductive hypothesis yields $p' \Downarrow b'$ and $[b'/l']r' \equiv q$. All that we now require is $p \Downarrow b$ and $[b/l]r \equiv [b'/l']r'$, To close this gap we introduce Lemma 1, which essentially states that any single step is correct, and whose proof will concern the rest of this section.

Lemma 1 (Single Step Correctness).

- *If* $e \xhookrightarrow{\mathbb{2}} e'$, $e \overset{\mathbb{2}}{\rightsquigarrow} \{p \mid l \,.\, r\}$, $e' \overset{\mathbb{2}}{\rightsquigarrow} \{p' \mid l' \,.\, r'\}$, *and* $p' \Downarrow b'$, *then* $p \Downarrow b$ *and* $[b/l]r \equiv [b'/l']r'$.
- *If* $e \xhookrightarrow{\mathbb{1}\mathbb{M}} e'$, $e \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{c \mid l.r\}$, $e' \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{c' \mid l' \,.\, r'\}$, *and* $c' \Downarrow <i, b'>$, *then* $c \Downarrow <i, b>$ *and* $[b/l]r \equiv [b'/l']r'$.

After invocation of that new lemma, we can derive $[b/l]r \equiv [b'/l']r' \equiv q$ directly. In order to prove Lemma 1, we will need to state analogous version for steps at $\mathbb{1}\mathbb{M}$, since the various kinds of multistage term in $\lambda^{\mathbb{12}}$ are mutually dependent. Thus, this section proceeds by covering the definition of splitting at $\mathbb{1}\mathbb{M}$, starting with the value form at that world.

4.3 Partial Value Splitting

To provide intuition about the behavior of partial value splitting, consider the following partial value:

```
(next{y}, (gr{injL 7}, next{y}))
```

To construct the value i representing its first-stage components, splitting first redacts all second-stage (blue) parts, along with the surrounding `next` annotations. The resulting "holes" in the term are replaced with unit values.

```
((), (gr{injL 7},()))
```

Finally, partial value splitting drops `gr` annotations, yielding:

```
((), (injL 7,()))
```

To construct the residual q (corresponding to second-stage computations) partial value splitting redacts all `gr` blocks (replacing them with unit) and `next` annotations:

$$\frac{}{x \rightharpoonup \{x;x\}} \qquad \frac{}{<> \rightharpoonup \{<>;<>\}} \qquad \frac{}{\texttt{gr}(m) \rightharpoonup \{m;<>\}} \qquad \frac{}{\texttt{next}(y) \rightharpoonup \{<>;y\}}$$

$$\frac{v_1 \rightharpoonup \{i_1;q_1\} \quad v_2 \rightharpoonup \{i_2;q_2\}}{<v_1, v_2> \rightharpoonup \{<i_1, i_2>;<q_1, q_2>\}} \qquad \frac{v \rightharpoonup \{i;q\} \quad \mathcal{S}[\text{-}] \in \{\texttt{inl(-)}, \texttt{inr(-)}, \texttt{roll(-)}\}}{\mathcal{S}[v] \rightharpoonup \{\mathcal{S}[i];q\}}$$

$$\frac{e \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{c \,|\, l.r\}}{\texttt{fn}(f.x.e) \rightharpoonup \{\texttt{fn}(f.x.\texttt{let}(c;<x, y>.<x, \texttt{roll}(y)>));\texttt{fn}(f.<x, \texttt{roll}(l)>.r)\}}$$

Fig. 6: Partial value splitting rules.

`(y, ((),y))`

A precise definition of the partial value splitting relation is given in Figure 6. In some regards, the formulation of partial value splitting is arbitrary. For instance, we chose to replace "holes" with unit values, but in fact we could have used any value there and it would make no difference in the end. There are however, at least some parts of the definition that are not arbitrary. Importantly, partial value splitting must not lose any meaningful information, such as injection tags.

4.4 World 𝟙𝕄 Term Splitting

The correctness of 𝟙𝕄 term splitting with respect to a $\overset{\mathbb{1}\mathbb{M}}{\hookrightarrow}$ step is given in Lemma 1. It's very similar to the world 𝟚 version, in that the reduction of the first stage part of e' should imply reduction of the first stage part of e, and that the resulting applied resumers should be equivalent. But split forms at 𝟙𝕄 have one more piece of output than those at 𝟚, namely the immediate value i. Lemma 1 accounts for this by saying that that immediate value must be exactly identical on both sides of the step.

The correctness of 𝟙𝕄 term splitting with respect to hoisting steps is given in Lemma 2. Because hoisting steps are nothing but rearrangement of second-stage code, this lemma can use the strong requirement of identical combined terms.

Lemma 2 (Hoisting Step Correctness). *If* $e \nearrow [\![q/y]\!]e'$, $e \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{c \,|\, l.r\}$, *and* $e' \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{c' \,|\, l'.r'\}$, *then* $c = c'$, *and* $l.r \equiv l'.\texttt{let}(q;y.r')$.

4.5 Example Cases

In this section, we consider a few exemplar cases from the proof of Lemma 1.

Reduction of `pi1`. Define $C = \texttt{let}(\underline{<<i_1, i_2>, <>>};<y, z>.\texttt{<pi1}(y), \underline{z}>)$ and $E = \texttt{pi1}(\underline{<v_1, v_2>})$. We are given $E \overset{\mathbb{1}\mathbb{M}}{\hookrightarrow} \underline{v_1}$, $E \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{C \,|\, l.\texttt{pi1}(\underline{<q_1, q_2>})\}$, and $\underline{v_1} \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{\underline{<i_1, <>>} \,|\, \_.q_1\}$, and we need to show $C \Downarrow <i_1, <>>$ and $\texttt{pi1}(\underline{<q_1, q_2>}) \equiv q_1$.

$$\frac{v \rightharpoonup \{i;q\}}{\underline{v} \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{\underline{<i, <>>} \mid \_.q\}} \qquad \frac{e \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{c \mid l.r\} \quad \mathcal{S}[\text{-}] \in \{\texttt{inl(-)}, \texttt{inr(-)}, \texttt{roll(-)}, \texttt{unroll(-)}\}}{\mathcal{S}[e] \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{\texttt{let}(c;<y, z>.<\mathcal{S}[\underline{y}], \underline{z}>) \mid l.r\}}$$

$$\frac{e \overset{2}{\rightsquigarrow} \{p \mid l.r\}}{\texttt{next}(e) \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{<<>, p> \mid l.r\}} \qquad \frac{e \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{c \mid l.r\} \quad \mathcal{S}[\text{-}] \in \{\texttt{pi1(-)}, \texttt{pi2(-)}\}}{\mathcal{S}[e] \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{\texttt{let}(c;<y, z>.<\mathcal{S}[\underline{y}], \underline{z}>) \mid l.\mathcal{S}[r]\}}$$

$$\frac{e_1 \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{c_1 \mid l_1.r_1\} \quad e_2 \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{c_2 \mid l_2.r_2\}}{<e_1, e_2> \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \left\{ \begin{pmatrix} \texttt{let}(c_1;<y_1, z_1>. \\ \texttt{let}(c_2;<y_2, z_2>. \\ \underline{<<y_1, y_2>, <z_1, z_2>>}) \end{pmatrix} \middle| <l_1, l_2>.<r_1, r_2> \right\}}$$

$$\frac{e_1 \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{c_1 \mid l_1.r_1\} \quad e_2 \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{c_2 \mid l_2.r_2\}}{\texttt{app}(e_1;e_2) \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \left\{ \begin{pmatrix} \texttt{let}(c_1;<y_1, z_1>. \\ \texttt{let}(c_2;<y_2, z_2>. \\ \texttt{let}(\texttt{app}(y_1;y_2);<y_3, z_3>. \\ \underline{<y_3, <z_1, z_2, z_3>>})) \end{pmatrix} \middle| <l_1, l_2, l_3>.\texttt{app}(\underline{r_1};\underline{<r_2, l_3>}) \right\}}$$

$$\frac{\cdot}{\texttt{gr}(e) \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{<e, <>> \mid \_.<>\}} \qquad \frac{e_1 \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{c_1 \mid l_1.r_1\} \quad e_2 \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{c_2 \mid l_2.r_2\}}{\texttt{letg}(e_1;x.e_2) \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \left\{ \begin{matrix} \texttt{let}(c_1;<x, z_1>.\texttt{let}(c_2;<y_2, z_2>.<y_2, <z_1, z_2>>)) \\ \mid <l_1, l_2>.\texttt{let}(r_1;\_.r_2) \end{matrix} \right\}}$$

$$\frac{e_1 \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{c_1 \mid l_1.r_1\} \quad e_2 \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{c_2 \mid l_2.r_2\} \quad e_3 \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{c_3 \mid l_3.r_3\}}{\texttt{caseg}(e_1;x_2.e_2;x_3.e_3) \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \left\{ \begin{pmatrix} \texttt{let}(c_1;<y_1, z_1>. \\ \texttt{case}(y_1; \\ \ x_2.\texttt{let}(c_2;<y_2, z_2>.\underline{<y_2, <z_1, \texttt{inl}(z_2)>>}); \\ \ x_3.\texttt{let}(c_3;<y_3, z_3>.\underline{<y_3, <z_1, \texttt{inr}(z_3)>>})) \end{pmatrix} \middle| <l_1, l_b>.(r_1;\texttt{case}(\underline{l_b};l_2.[<>/x_2]r_2;l_3.[<>/x_3]r_3)) \right\}}$$

$$\frac{\cdot}{\underline{q} \overset{2}{\rightsquigarrow} \{\underline{<>} \mid \_.q\}} \qquad \frac{e \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{c \mid l.r\}}{\texttt{prev}(e) \overset{2}{\rightsquigarrow} \{\texttt{pi2}(c) \mid l.r\}}$$

$$\frac{e \overset{2}{\rightsquigarrow} \{p \mid l.r\} \quad \mathcal{S}[\text{-}] \in \{\texttt{pi1(-)}, \texttt{pi2(-)}, \texttt{inl(-)}, \texttt{inr(-)}, \texttt{roll(-)}, \texttt{unroll(-)}, \texttt{fn}(f\,x.\text{-})\}}{\mathcal{S}[e] \overset{2}{\rightsquigarrow} \{p \mid l.\mathcal{S}[r]\}}$$

$$\frac{e_1 \overset{2}{\rightsquigarrow} \{p_1 \mid l_1.r_1\} \quad e_2 \overset{2}{\rightsquigarrow} \{p_2 \mid l_2.r_2\} \quad \mathcal{S}[\text{-}, \text{-}] \in \{<\text{-}, \text{->}, \texttt{app(-;-)}, \texttt{let(-;}x.\text{-)}\}}{\mathcal{S}[e_1, e_2] \overset{2}{\rightsquigarrow} \{<p_1, p_2> \mid <l_1, l_2>.\mathcal{S}[r_1, r_2]\}}$$

$$\frac{e_1 \overset{2}{\rightsquigarrow} \{p_1 \mid l_1.r_1\} \quad e_2 \overset{2}{\rightsquigarrow} \{p_2 \mid l_2.r_2\} \quad e_3 \overset{2}{\rightsquigarrow} \{p_3 \mid l_3.r_3\}}{\texttt{case}(e_1;x_2.e_2;x_3.e_3) \overset{2}{\rightsquigarrow} \{<p_1, p_2, p_3> \mid <l_1, l_2, l_3>.\texttt{case}(r_1;x_2.r_2;x_3.r_3)\}}$$

Fig. 7: Splitting rules for terms at $\mathbb{1}\mathbb{M}$ and $\mathbb{2}$.

$$\frac{\cdot}{e \equiv e} \qquad \frac{e \equiv e'}{e' \equiv e} \qquad \frac{e \equiv e' \quad e' \equiv e''}{e \equiv e''} \qquad \frac{e \xrightarrow{\mathbb{1}\mathsf{G}} e'}{e \equiv e'} \qquad \frac{\cdot}{\texttt{let}(q;x.\underline{x}) \equiv q}$$

$$\frac{e \equiv e' \quad \mathcal{S}[\text{-}] \in \left\{ \begin{array}{l} \texttt{pi1(-)}, \texttt{pi2(-)}, \texttt{inl(-)}, \texttt{inr(-)}, \texttt{roll(-)}, \texttt{unroll(-)}, \texttt{fn}(f\,.\,x\,.\text{-}), \\ \texttt{<-,}e\texttt{>}, \texttt{<}e\texttt{,->}, \texttt{app(-;}e\texttt{)}, \texttt{app(}e\texttt{;-)}, \texttt{let(}e\texttt{;}x\texttt{.-)}, \texttt{let(-;}x\texttt{.}e\texttt{)}, \\ \texttt{case(-;}x_2\texttt{.}e_2\texttt{;}x_3\texttt{.}e_3\texttt{)}, \texttt{case(}e_1\texttt{;}x_2\texttt{.-;}x_3\texttt{.}e_3\texttt{)}, \texttt{case(}e_1\texttt{;}x_2\texttt{.}e_2\texttt{;}x_3\texttt{.-)}, \end{array} \right\}}{\mathcal{S}[e] \equiv \mathcal{S}[e']}$$

$$\frac{\mathcal{S}[\text{-}] \in \{\texttt{pi1(-)}, \texttt{pi2(-)}, \texttt{<-,}e\texttt{>}, \texttt{<}\underline{v}\texttt{,->}, \texttt{app(-;}e\texttt{)}, \texttt{app(}\underline{v}\texttt{;-)}, \texttt{case(-;}x_2\texttt{.}e_2\texttt{;}x_3\texttt{.}e_3\texttt{)}\}}{\mathcal{S}[\texttt{let}(q;x.e)] \equiv \texttt{let}(q;x.\mathcal{S}[e])}$$

Fig. 8: Monostage equivalence relation, including reduction, congruence, and let-transposition rules. Since the $\mathbb{1}\mathsf{G}$ fragment of $\lambda^{\mathbb{1}\mathbb{2}}$ is monostage, we simply use $e \xrightarrow{\mathbb{1}\mathsf{G}} e'$ to mean any standard reduction from e to e'.

Both of these can be derived directly. This pattern, where the outputs can be directly derived, extends to the `pi2` and `prev` reduction rules and all value promotion rules.

Reduction of Application. Let
$E = \texttt{app(}\underline{\texttt{fn(}f\,.\,x\,.\,e\texttt{)}}\texttt{;}\underline{v}\texttt{)}$,
$E' = [\texttt{fn(}f\,.\,x\,.\,e\texttt{)}, v/f, x]e$,
$I = \texttt{fn(}f\,.\,x\,.\,\texttt{let(}c\texttt{;<}x, y\texttt{>.<}x, \texttt{roll(}y\texttt{)>))}$,
$Q = \texttt{fn(}f\,.\texttt{<}x, \texttt{roll(}l\texttt{)>.}r\texttt{)}$, and
$C = \begin{bmatrix} \texttt{let(}\underline{\texttt{<}I, \texttt{<>>}}\texttt{;<}y_1, z_1\texttt{>.} \\ \texttt{let(}\underline{\texttt{<}i_1, \texttt{<>>}}\texttt{;<}y_2, z_2\texttt{>.} \\ \texttt{let(app(}\underline{y_1}\texttt{;}\underline{y_2}\texttt{);<}y_3, z_3\texttt{>.} \\ \underline{\texttt{<}y_3, \texttt{<}z_1, z_2, z_3\texttt{>>}}\texttt{)))} \end{bmatrix}$.
We are given $E \overset{\mathbb{1}\mathbb{M}}{\hookrightarrow} E'$, $E \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{C\,|\,\texttt{<}\_, \_, l\texttt{>.app(}\underline{Q}\texttt{;}\underline{\texttt{<}q_1, l\texttt{>}}\texttt{)}\}$,
$E' \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{[I, i_1/f, x]c\,|\,l.[Q, q_1/f, x]r\}$, and $[I, i_1/f, x]c \Downarrow \texttt{<}i, b\texttt{>}$. From this, we can derive $C \Downarrow \texttt{<}i, \texttt{<<>,<>,roll(}b\texttt{)>>}$ and $\texttt{app(}\underline{Q}\texttt{;}\underline{\texttt{<}q_1, \texttt{roll(}b\texttt{)>}}\texttt{)} \equiv [Q, q_1, b/f, x, l]r$ directly. This pattern applies to all of the other reduction rules involving substitution, namely those for `caseg`, and `letg`.

Compatibility of `pi1` *at* $\mathbb{2}$. By the case, we are given $\texttt{pi1(}e\texttt{)} \overset{\mathbb{2}}{\hookrightarrow} \texttt{pi1(}e'\texttt{)}$,
$\texttt{pi1(}e\texttt{)} \overset{\mathbb{2}}{\rightsquigarrow} \{p\,|\,l\texttt{.pi1(}r\texttt{)}\}$, $\texttt{pi1(}e'\texttt{)} \overset{\mathbb{2}}{\rightsquigarrow} \{p'\,|\,l'\texttt{.pi1(}r'\texttt{)}\}$, and $p' \Downarrow b'$. Inversion of the first three yields $e \overset{\mathbb{2}}{\hookrightarrow} e'$, $e \overset{\mathbb{2}}{\rightsquigarrow} \{p\,|\,l\,.\,r\}$, and $e' \overset{\mathbb{2}}{\rightsquigarrow} \{p'\,|\,l'\,.\,r'\}$. Using Lemma 1 inductively gives $p \Downarrow b$ and $[b/l]r \equiv [b'/l']r$. From this, $[b/l]\texttt{pi1(}r\texttt{)} \equiv [b'/l']\texttt{pi1(}r\texttt{)}$ can be derived directly. This pattern generalizes to all world $\mathbb{2}$ compatibility rules.

Compatibility of `pi1` *at* $\mathbb{1}\mathbb{M}$. By the case, we are given $\texttt{pi1(}e\texttt{)} \overset{\mathbb{1}\mathbb{M}}{\hookrightarrow} \texttt{pi1(}e'\texttt{)}$,
$\texttt{pi1(}e\texttt{)} \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{\texttt{let(}c\texttt{;<}y, z\texttt{>.<pi1(}\underline{y}\texttt{),}\underline{z}\texttt{>)}\,|\,l\texttt{.pi1(}r\texttt{)}\}$,
$\texttt{pi1(}e'\texttt{)} \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \{\texttt{let(}c'\texttt{;<}y, z\texttt{>.<pi1(}\underline{y}\texttt{),}\underline{z}\texttt{>)}\,|\,l'\texttt{.pi1(}r'\texttt{)}\}$, and
$\texttt{let(}c'\texttt{;<}y, z\texttt{>.<pi1(}\underline{y}\texttt{),}\underline{z}\texttt{>)} \Downarrow \texttt{<}i, b'\texttt{>}$. Inversion of the first three yields $e \overset{\mathbb{1}\mathbb{M}}{\hookrightarrow} e'$,

$e \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \texttt{\{}c\texttt{|}l.r\texttt{\}}$, and $e' \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \texttt{\{}c'\texttt{|}l'.r'\texttt{\}}$, and inversion of the reduction yields, for some i_2, $c' \Downarrow \texttt{<<}i, i_2\texttt{>}, b'\texttt{>}$. Using Lemma 1 inductively gives $c \Downarrow \texttt{<<}i, i_2\texttt{>}, b\texttt{>}$ and $[b/l]r \equiv [b'/l']r$. From this, we can derive $\texttt{let(}c\texttt{;<}y, z\texttt{>.<pi1(}y\texttt{),}\underline{z}\texttt{>)} \Downarrow \texttt{<}i, b\texttt{>}$ and $[b/l]\texttt{pi1(}r\texttt{)} \equiv [b'/l']\texttt{pi1(}r\texttt{)}$ directly. This pattern generalizes to all world $\mathbb{1}\mathbb{M}$ compatibility rules.

Compatibility of `gr`. We are given $\texttt{gr(}e\texttt{)} \overset{\mathbb{1}\mathbb{M}}{\hookrightarrow} \texttt{gr(}e'\texttt{)}$, $\texttt{gr(}e\texttt{)} \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \texttt{\{<}e, \underline{\texttt{<>}}\texttt{>|\_.}\underline{\texttt{<>}}\texttt{\}}$, $\texttt{gr(}e'\texttt{)} \overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow} \texttt{\{<}e', \underline{\texttt{<>}}\texttt{>|\_.}\underline{\texttt{<>}}\texttt{\}}$, and $\texttt{<}e', \underline{\texttt{<>}}\texttt{>} \Downarrow \texttt{<}i, \texttt{<>>}$. Inversion of the step and reduction yield $e \overset{\mathbb{1}\mathbb{G}}{\hookrightarrow} e'$ and $e' \Downarrow i$. As a simple property of monostage reduction (which $\overset{\mathbb{1}\mathbb{G}}{\hookrightarrow}$ is), we know $e \Downarrow i$. From here, we can derive $\texttt{<}e, \underline{\texttt{<>}}\texttt{>} \Downarrow \texttt{<}i, \texttt{<>>}$ and $\underline{\texttt{<>}} \equiv \underline{\texttt{<>}}$ directly.

4.6 Role of World $\mathbb{1}\mathbb{G}$

The splitting algorithm described in the previous subsections operates purely on the local structure of $\lambda^{\mathbb{1}\mathbb{2}}$ terms. One artifact of this design is that splitting $\mathbb{1}\mathbb{M}$ terms may generate resumers containing unnecessary logic. For example, the rule for splitting $\mathbb{1}\mathbb{M}$ `caseg` terms inserts the tag from the `caseg` argument into the boundary value, then decodes this tag in the resumer. This logic occurs regardless of whether the terms forming the branches of the `caseg` contain second-stage computations. Worse, if this `caseg` appeared in the body of a recursive function with no other second-stage computations, splitting would generate a resumer with (useless) recursive calls.

An illustrative example is the `part` function in the quickselect example of Section 6.3. If `part` were defined at $\mathbb{1}\mathbb{M}$ then (like `qs`) it would split into two functions `part1` and `part2`, the latter of which recursively computes the (trivial) second-stage component of `part`'s result. Moreover, `qs2` would call `part2`, just as `qs1` calls `part1`:

```
fun qs2 (p : tree, k : int) : int =
... | Branch (i,h,p1,p2) =>
  let val () = part2 ((),()) in
  case compare k i of ...
```

Rather than attempt global optimization of the outputs of splitting, we instead leverage the type system to indicate when a term contains no second-stage computations by adding a third world $\mathbb{1}\mathbb{G}$ whose terms are purely first-stage. Defining `part` in this world is tantamount to proving it has no second-stage computations, allowing splitting to avoid generating the resumer `part2` and calling it from `qs2`.

Since direct staged term evaluation (Section 3) reduces all first-stage terms to value forms without any remaining stage two work, the distinction between $\mathbb{1}\mathbb{M}$ and $\mathbb{1}\mathbb{G}$ is unnecessary. In contrast, when performing program transformations before the first-stage inputs are known, it is valuable to form a clear distinction between ground and mixed terms. This was similarly observed in prior work seeking to implement self-applicable partial evaluators [20,19]. While this paper

assumes that ground annotations already exist in the input, it may be possible to use binding-time analysis techniques to automatically insert them. Ground is also similar to the validity mechanism in ML5 [21].

4.7 Typing the Boundary Data Structure

One of the central features of our splitting algorithm is that it encodes the control flow behavior of the original staged program into the boundary data structure. For instance, the `case` and `caseg` splitting rules put injection tags on the boundary based on which branch was taken, and the `fn` rule adds a roll tag to the boundary. As a result, the boundary value passed between the two output programs has a (potentially recursive) structure like a tree or list.

This structure can be described with a type; for instance, the staged quickselect yields a binary search tree. Indeed, for most of our examples in Section 6, inferring this type is straight-forward. However, we do not yet have a formal characterization of these boundary types that is defined for all $\lambda^{\mathbb{1}\mathbb{2}}$ programs, though we plan to pursue this in future work.[5]

5 Implementation

We have encoded the type system, dynamic semantics, splitting algorithm, and output semantics in Twelf, as well as all of the theorems of Sections 3 and 4, and their proofs. We also have a Standard ML implementation of the $\lambda^{\mathbb{1}\mathbb{2}}$ language, a staged interpreter, the splitting algorithm, and an interpreter for split programs. This implementation extends the language described in Section 3 with ints, bools, `let` statements, n-ary sums and products, datatype and function declarations, and deep pattern matching (including `next()` and `gr()` patterns). The code snippets in this paper are written in our concrete syntax, using these additional features when convenient. Our expanded syntax allows staging annotations around declarations. For example,

```
@gr{
  datatype list = Empty | Cons of int * list
  fun part (...) = ...
}
fun qss (...) = ...
```

declares the `list` datatype and `part` function at world $\mathbb{1}\mathbb{G}$, by elaborating into:

```
val gr{Empty} = gr{roll (inj ...)}
val gr{Cons} = gr{fn (...) => roll (inj ...)}
val gr{part} = gr{fn (...) => ...}
val qss = fn (...) => ...
```

[5] Observe that in the outputs of splitting we could have omitted the `roll` tag from functions or replaced `inl(`a`)` and `inr(`b`)` with $\texttt{<}0, a\texttt{>}$ and $\texttt{<}1, b\texttt{>}$, and the proof of correctness would still have gone through. We chose the tags, however, in order to keep the typed interpretation more natural, even in absence of a formal result.

In the splitting algorithm, we perform a number of optimizations which drastically improve the readability of split programs. For example, we split patterns directly, instead of first translating them into lower-level constructs. We also take advantage of many local simplifications, most notably, not pairing precomputations when one is known to be `<>`.

6 Examples of Splitting

Now we investigate the behavior of our splitting algorithm on several examples. The split code that follows is the output of our splitting implementation; for clarity, we have performed some minor optimizations and manually added type annotations (including datatype declarations and constructor names), as our algorithm does not type its output.

6.1 Dot Product

Our first example, `dot`, appears in [17]. `dot` is a first-order, non-recursive function—precisely the sort of code studied in prior work on pass separation for imperative languages. `dot` takes the dot product of two three-dimensional vectors, where the first two coordinates are first-stage, and the last coordinate is second-stage.

```
type vec = ∇int * ∇int * ◯int
// dot : vec * vec -> ◯int
fun dot ((gr{x1},gr{y1},next{z1}),(gr{x2},gr{y2},next{z2})) =
  next{ prev{hold gr{(x1*x2) + (y1*y2)}} + (z1*z2) }
```

The body of `dot` is an int term at world $\mathbb{2}$ containing an int computation of `(x1*x2) + (y1*y2)` which is promoted from world $\mathbb{1G}$ to world $\mathbb{2}$. We would expect the first stage of the split program to take the first two coordinates of each vector and perform that first-stage computation; and the second stage to take the final coordinates and the result of the first stage, then multiply and add. Our algorithm splits `dot` into the two functions:

```
fun dot1 ((x1,y1,()), (x2,y2,()))     = ((), (x1*x2)+(y1*y2))
fun dot2 ((((),(),z1),((),(),z2)),l) = l+(z1*z2)
```

As expected, `dot1` returns `(x1*x2)+(y1*y2)` as the precomputation, and `dot2` adds that precomputation to the products of the final coordinates. This is exactly what is done in [17], though they write the precomputation into a mutable cache.

6.2 Exponentiation by Squaring

Our next example, `exp`, is a mainstay of the partial evaluation literature (for example, in [13]). `exp` recursively computes b^e using exponentiation by squaring, where e is known at the first stage, and b is known at the second stage.

```
fun exp (next{b} : ◯int , gr{e} : ∇int) =
 if      gr{e == 0}       then next{1}
 else if gr{e mod 2 == 0} then exp(next{b*b},gr{e/2})
 else                     next{b*prev{exp(next{b*b},gr{(e-1)/2})}}
```

Because `exp` is a recursive function whose conditionals test the parity of the exponent argument, the sequence of branches taken corresponds exactly to the binary representation of e. Partially evaluating `exp` with e eliminates all of the conditionals, selecting and expanding the appropriate branch in each case.

Our algorithm, on the other hand, produces:

```
datatype binnat = Zero
                | Even of binnat
                | Odd of binnat

fun exp1 (b : unit, e : int) =
  if      e == 0       then ((), Zero)
  else if e mod 2 == 0 then ((), Even (#2 (exp1 ((), e/2))))
  else                      ((), Odd  (#2 (exp1 ((), (e-1)/2))))

fun exp2 ((b : int, e : unit), l : binnat) =
    case l of
      Zero => 1
    | Even n => exp2 ((b*b, ()), n)
    | Odd n => b * exp2 ((b*b, ()), n)
```

`exp1` recursively performs parity tests on e, but unlike `exp`, it simply computes a data structure (a `binnat`) recording which branches were taken. `exp2` takes b and a `binnat` l, and uses l to determine how to compute with b.

Of course, the `binnat` computed by `exp1` is precisely the binary representation of e! While partial evaluation realizes `exp`'s control-flow dependency on a fixed e by recursively expanding its branches in place, we explicitly record this dependency generically over all e by creating a boundary data structure. This occurs in the $\overset{\mathbb{1}\mathbb{M}}{\rightsquigarrow}$ rule for `case`, which emits a tag corresponding to the taken branch in the precomputation, and `case`s on it (as l_b) in the residual.

Because splitting `exp` does not eliminate its conditionals, partial evaluation is more useful in this case. (Notice, however, that partially evaluating `exp2` on a `binnat` is essentially the same as partially evaluating `exp` on the corresponding int.) Nevertheless, splitting `exp` still demonstrates how our algorithm finds interesting data structures latent in the structure of recursive functions.

6.3 Quickselect

Let us return to the quickselect algorithm, which we discussed at length in Section 2. (The code is in Figure 1(b).) `qss` finds the kth largest element of a list l by recursively partitioning the list by its first element, then recurring on the side containing the kth largest element. l is first-stage and k is second-stage.

Stage-splitting `qss` produces:

```
datatype tree = Leaf | Branch of int * tree * int * tree
datatype list = Empty | Cons of int * list

fun part ((p,l):int*list) : (int*list*list) =
```

```
  case l of Empty => (0,Empty, Empty)
  | Cons (h,t) =>
      let val (n,left,right) = part (p,t) in
      if h<p then (n+1,Cons(h,left),right)
             else (n,left,Cons(h,right))

fun qs1 (l : list, k : unit) =
  (), case l of Empty => Leaf
       | Cons (h,t) => Branch (
           let val (n,left,right) = part (h,t) in
           (n, #2 (qs1 left k), h, #2 (qs1 right k))))

fun qs2 ((), k : int), p : tree) =
  case p of Leaf => 0
  | Branch (n,left,h,right) =>
      case compare k n of
        LT => qs2 ((), k), left)
      | EQ => h
      | GT => qs2 ((), k-n-1), right)
```

This is nearly identical to the cleaned-up code we presented in Figure 2, except we do not suppress the trivial inputs and outputs of `qs1` and `qs2`.

The function `qs1` partitions l, but since the comparison with k (to determine which half of l to recur on) is at the second stage, it simply recurs on *both* halves, pairing up the results along with h (the head of l) and n (the size of the left half). `qs2` takes k and this tree p, and uses k to determine how to traverse p.

How does our splitting algorithm generate binary search trees and a traversal algorithm? The $\overset{2}{\rightsquigarrow}$ rule for `case` tuples up the precomputations for its branches, and in the residual, selects the residual corresponding to the appropriate branch. The tree is implicit in the structure of the code; ordinarily, the quickselect algorithm only explores a single branch, but the staging annotations force the entire tree to be built.

This is an instance where stage-splitting is more practical than partial evaluation; if l is large, partially evaluating `quickselect` requires runtime generation of a huge amount of code simultaneously encoding the tree and traversal algorithm. (Avoiding the code blowup, by not expanding some calls to `part`, would result in duplicating first-stage computations.)

Note that the recursive `part` function is defined within a `gr` annotation. As explained in Section 4.6, defining `part` at $\mathbb{1}\mathbb{M}$ would cause it to split in a way that incurs extra cost at the second stage. In this case, that cost would be $\Theta(n)$ in the size of the input list, enough to overpower the asymptotic speedup gained elsewhere. With `gr` annotations, however, this can be prevented.

As discussed in Section 2, `qs1` performs $\Theta(n \log n)$ expected work per call, whereas `qs2` performs $\Theta(\log n)$ expected work. This results in a net speedup over standard quickselect if we perform many (specifically, $\omega(\log n)$) queries on the same list—precisely the topic of our next example.

6.4 Mixed-Stage Map Combinator

As a lambda calculus, one of the strengths of $\lambda^{\mathbb{12}}$ is that it can express combinators as higher order functions. In this example, we consider just such a combinator: `tmap`, which turns a function of type ∇`list` $*$ ◯`int` $\to$ ◯`int` into one of type ∇`list` $*$ ◯`list2` $\to$ ◯`list2`, by mapping over the second argument.[6]

```
@next{ datatype list2 = Empty2 | Cons2 of int * list2 }

fun tmap (f : ∇list * ◯int -> ◯int) (l : ∇list, q : ◯list2) =
  next{ let fun m Empty2       = Empty2
              | m (Cons2(h,t)) = Cons2(prev{f (l,next{h})}, m t)
        in m prev{q}}
val mapqss = tmap qss
```

Importantly, `tmap f` performs the first-stage part of `f` once and second-stage part of `f` many times. This was discussed in the context of partial evaluation in Section 3.4, but it is especially clear when we look at the output of splitting:

```
fun tmap1 f = (fn (l,()) => ((),#2 (f (l,()))),())
val (mapqss1,()) = tmap1 qs1

datatype list2 = Empty2 | Cons2 of int * list2
fun tmap2 (f,()) ((l,q), p : tree) =
  let fun m Empty2 = Empty2
        | m (Cons2(h,t)) = Cons2(f ((l,h), p), m t)
  in m q
val mapqss2 = tmap2 (qs2, ())
```

Indeed, observe that the argument `f` (for which `qs1` is later substituted) of `tmap1` is called only once, whereas the corresponding argument `f` (for which `qs2` is later substituted) in `tmap2` is evaluated once per element in the query list `q`.

6.5 Composing Graphics Pipeline Programs

Composable staged programs are particularly important the domain of real-time graphics. This need arises because modern graphics architectures actually *require* that graphics computations be structured as a pipeline of stages which perform increasingly fine-grained computations (e.g., per-object, per-screen region, per-pixel), where computations in later stages use the results of an earlier stage multiple times [28].

The standard way to program these graphics pipelines is to define one program (usually called a *shader*) per stage. In other words, the programmer is expected to write their multi-stage programs in an already split form. This requirement results in complex code where invariants must hold across different stages and local changes to the logic of one stage may require changes to that of

[6] $\lambda^{\mathbb{12}}$ doesn't have a way to "share" datatype declarations between stages, so we define `list2` to be a list of integers at the second stage.

upstream stages. This harms composition and modularity. Graphics researchers therefore have suggested using mechanisms like our stages [24,8,11] to express graphics program logic, including representing entire pipeline as a single multi-stage function [8].

As a language, λ^{12} is well suited for specification of such functions, and we give a simple example below. In it, we consider a graphics pipeline program to be a staged function that takes an object definition in the first stage (`object`) and a pixel coordinate in the second stage (◯`coord`), and emits the color of the object at the specified pixel (◯`color`). Given two such multistage functions, we then define a combinator that multiplies their results pointwise in the second stage.[7]

```
datatype object = ...
@next{  type coord = int * int
        type color = ...        }
type pipeline = object * ◯coord -> ◯color

fun shade (refl : pipeline, albedo : pipeline) : pipeline =
  fn (obj : object, next{xy} : ◯coord) =>
       prev{refl (obj,next{xy})} * prev{albedo (obj,next{xy})}
```

Prior work [8] lacks the ability to define such combinators, because it does not support higher order functions.

7 Related Work

Frequency reduction and precomputation are common techniques for both designing algorithms and performing compiler optimizations [15]. The idea behind precomputation is to identify computations that can be performed earlier (e.g., at compile time) if their inputs are available statically and perform them at that earlier time. Dynamic algorithms, partial evaluation, and incremental computation are all examples of precomputation techniques. The idea behind frequency reduction is to identify computations that are performed multiple times and pull them ahead so that they can be performed once and used later as needed. Dynamic programming, loop hoisting, and splitting (presented here) are all examples of frequency reduction techniques.

Precomputation techniques. Perhaps one of the most studied examples of precomputation is partial evaluation, which distinguishes between static (compile-time) and dynamic (runtime) stages. Given a program and values for all static inputs, partial evaluation generates a specialized program by performing computations that depend only on the static inputs [13]. We refer the reader to the

[7] Although this is a general pointwise multiplication combinator, the variable names suggest a possible interpretation of the inputs and output: the first input function calculates the albedo of an object (a measure of how much light it reflects), the second input calculates the object's incoming light, and so the output (the product of these terms) is a function that calculates the outgoing light from an object.

book by Jones, Gommard, and Sestoft for a comprehensive discussion of partial evaluation work until the early 90's [14].

Early approaches to partial evaluation can be viewed as operating in two stages: binding time analysis and program specialization. For a multivariate program with clearly marked static and dynamic arguments, binding-time analysis identifies all the parts of the program that can be computed by the knowledge of static arguments. Using this information and the values of static arguments, program specialization specializes the original program to a partially-evaluated one that operates on many different dynamic arguments. This approach has been applied to construct partial evaluators for a number of languages such as Scheme [2,3].

Researchers have explored other staging transformations that, like splitting, partition an input two-stage program into two components, one corresponding to each stage. In particular, *binding time separation* [20] (also called *program bifurcation* [6]) has been used as a preprocessor step in partial evaluators, allowing efficient specialization of programs with mixed-stage data structures without changes to the specializer itself. Notably, the grammar-based binding-time specifications used in binding time separation are capable of describing data structures with purely-static, purely-dynamic, and mixed-stage content, much like the type system of $\lambda^{\mathbb{1}2}$ (though this correspondence is less clear without our addition of $\mathbb{1}\mathbb{G}$ and ∇).

However unlike splitting, where the goal is to emit code where first-stage results are computed once and then reused in multiple invocations of second-stage execution, the second (dynamic) function produced by binding time separation does not use the results of the first; instead, it has access to the first-stage inputs and recomputes all required first-stage computations. As noted by Knoblock and Ruf [17], it may be possible to modify the program bifurcation algorithm to cache and reuse the intermediate results, but this was never attempted. Alternatively, our algorithm could potentially be used as the basis of a general bifurcation algorithm in a partial evaluator.

Experience with binding time analysis showed that it can be difficult to control, leading to programs whose performance was difficult to predict. This led to investigations based on type systems for making the stage of computations explicit in programs [10,23] and writing "metaprograms" that, when evaluated at a stage, yield a new program to be evaluated at the next stage. Davies [4] presented a logical construction of binding-time type systems by deriving a type system via the Curry-Howard correspondence applied to temporal logic. Davies and Pfenning proposed a new type system for staged computation based on a particular fragment of modal logic [5]. The work on MetaML extended type-based techniques to a full-scale language by developing a statically typed programming language based on ML that enables the programmer to express programs with multiple stages [31,30]. MetaML's type system is similar to Davies [4] but extends it in several important ways. Nanevksi and Pfenning further extended the these techniques by allowing free variables to occur within staged computations [22].

The type-system of $\lambda^{\mathbb{12}}$ is closely related to this later line of work on metaprogramming and staged computation. The specific extension to the typed lambda calculus that we use here is based on the $\bigcirc$ modality of Davies [5]. Our types differ in the restriction to two stages and the addition of ∇; however, the key difference between our work and this prior work is that we instead focus on the problem of splitting.

Another class of precomputation techniques is incremental computation, where a program can efficiently respond to small changes in its inputs by only recomputing parts of the computation affected by the changes [7,25,26,1]. However, unlike splitting, incremental computation does not require fixing any of its inputs and, in the general case, allows for all program inputs to change. Thus, the benefits of incremental computation depend on what changes to inputs are made. For example, while it is possible to apply incremental computation to the quickselect example in Section 2, techniques would unfold the quickselect function based on the demanded ranks, potentially incurring linear time cost at each step of the algorithm (as opposed to the logarithmic result produced by splitting). Moreover, incremental computation techniques must also maintain sophisticated data structures dynamically at run time to track what computations must be performed.

Stage Splitting. Algorithms for stage splitting have appeared in the literature under the names *pass separation* [15] and *data specialization* [17]. Perhaps the closest work to ours is the algorithm for data specialization given by Knoblock and Ruf [17], which also seeks to statically split an explicitly staged program into two stages. However, they only consider a simple first-order language and straight-line programs. Their work also treats all computations guarded by second-stage conditionals as second-stage computations, which would prevent optimization (via splitting) of programs such as quickselect.

As noted in Section 6.5, splitting algorithms have also been a topic of interest in programming systems for computer graphics, where, to achieve high performance, programs are manually separated into components by frequency of execution corresponding to graphics hardware pipeline stages. The software engineering challenges of modifying multiple per-stage programs have led to suggestions of writing graphics programs in an explicitly-staged programming language [24,8,11] and deferring the task of pass separation to the compiler. However, all prior splitting efforts in computer graphics, like that of Knoblock and Ruf [17], have been limited to simple, imperative languages.

Defunctionalization. Defunctionalization [27] is a program transformation that eliminates high-order functions and replaces them with lower order functions that manipulate a data structure encoding the original control flow. This operation has similarity to our splitting transformation, which eliminates staging in a program by encoding control flow in a data structure passed between the stage one and stage two outputs. It would interesting to explore the possibility of a semantics-preserving transformation to convert a staged program into a higher-order form, and then applying defunctionalization to obtain our results.

8 Conclusion

This paper presents a splitting algorithm for $\lambda^{\mathbb{12}}$, a two-staged typed lambda calculus with support for recursion and first-class functions. The type system of $\lambda^{\mathbb{12}}$ uses three worlds $\mathbb{1M}$, $\mathbb{1G}$, and $\mathbb{2}$ to classify code by its stage, with the modalities $\bigcirc$ and ∇ providing internalizations of second-stage code and ground first-stage code to the mixed world. We present a dynamic semantics that evaluates terms in two passes and provides an eager interpretation of `next` via hoisting steps. We prove the correctness of our splitting algorithm against the dynamic semantics, and implement this proof in Twelf. Finally, we discuss our implementation of $\lambda^{\mathbb{12}}$ and its splitting algorithm and analyze their behavior for a number of staged programs, including those with higher-order and recursive functions.

Looking forward, we are interested in investigating the behavior of splitting in the presence of richer language features such as mutation, polymorphism, parallel constructs, and more than two stages. Also while we have proven the dynamic correctness of our splitting algorithm, we do not yet have a characterization of the types of the output or of the cost behavior of output terms.

References

1. Acar, U.A., Blelloch, G.E., Blume, M., Harper, R., Tangwongsan, K.: An experimental analysis of self-adjusting computation. ACM Trans. Prog. Lang. Sys. 32(1), 3:1–53 (2009)
2. Bondorf, A., Danvy, O.: Automatic autoprojection of recursive equations with global variable and abstract data types. Sci. Comput. Program. 16(2), 151–195 (Sep 1991)
3. Consel, C.: New insights into partial evaluation: the schism experiment. In: Ganzinger, H. (ed.) ESOP '88, Lecture Notes in Computer Science, vol. 300, pp. 236–246 (1988)
4. Davies, R.: A temporal-logic approach to binding-time analysis. In: LICS. pp. 184–195 (1996)
5. Davies, R., Pfenning, F.: A modal analysis of staged computation. J. ACM 48(3), 555–604 (2001)
6. De Niel, A., Bevers, E., De Vlaminck, K.: Program bifurcation for a polymorphically typed functional language. SIGPLAN Not. 26(9), 142–153 (May 1991), http://doi.acm.org/10.1145/115866.115880
7. Demers, A., Reps, T., Teitelbaum, T.: Incremental evaluation of attribute grammars with application to syntax-directed editors. In: Principles of Programming Languages. pp. 105–116 (1981)
8. Foley, T., Hanrahan, P.: Spark: Modular, composable shaders for graphics hardware. ACM Trans. Graph. 30(4), 107:1–107:12 (Jul 2011)
9. Futamura, Y.: Partial evaluation of computation process – an approach to a compiler-compiler. Systems, Computers, Controls 2(5), 45–50 (1971)
10. Gomard, C.K., Jones, N.D.: A partial evaluator for the untyped lambda-calculus. J. Funct. Program. 1(1), 21–69 (1991)
11. He, Y., Gu, Y., Fatahalian, K.: Extending the graphics pipeline with adaptive, multi-rate shading. ACM Trans. Graph. 33(4), 142:1–142:12 (Jul 2014)

12. Hoare, C.A.R.: Algorithm 65: Find. Commun. ACM 4(7), 321–322 (Jul 1961)
13. Jones, N.D.: An introduction to partial evaluation. ACM Comput. Surv. 28(3), 480–503 (Sep 1996)
14. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial evaluation and automatic program generation. Prentice Hall international series in computer science, Prentice Hall (1993)
15. Jørring, U., Scherlis, W.L.: Compilers and staging transformations. In: Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 86–96. POPL '86, ACM, New York, NY, USA (1986)
16. Kleene, S.C.: Introduction to Metamathematics. D. Van Nostrand Co., Inc. (1952)
17. Knoblock, T.B., Ruf, E.: Data specialization. In: Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation. pp. 215–225. PLDI '96, ACM, New York, NY, USA (1996)
18. Krishnaswami, N.R.: Higher-order functional reactive programming without space-time leaks. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. pp. 221–232. ICFP '13, ACM, New York, NY, USA (2013)
19. Mogensen, T.A.: Binding time analysis for polymorphically typed higher order languages. In: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 2: Advanced Seminar on Foundations of Innovative Software Development II and Colloquium on Current Issues in Programming Languages. pp. 298–312. TAPSOFT '89, Springer-Verlag, London, UK, UK (1989), http://dl.acm.org/citation.cfm?id=646625.698197
20. Mogensen, T.A.: Separating binding times in language specifications. In: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture. pp. 12–25. FPCA '89, ACM, New York, NY, USA (1989), http://doi.acm.org/10.1145/99370.99372
21. Murphy, VII, T., Crary, K., Harper, R.: Distributed control flow with classical modal logic. In: Ong, L. (ed.) Computer Science Logic, 19th International Workshop (CSL 2005). Lecture Notes in Computer Science, Springer (August 2005)
22. Nanevski, A., Pfenning, F.: Staged computation with names and necessity. J. Funct. Program. 15(5), 893–939 (2005)
23. Nielson, F., Nielson, H.R.: Two-level Functional Languages. Cambridge University Press, New York, NY, USA (1992)
24. Proudfoot, K., Mark, W.R., Tzvetkov, S., Hanrahan, P.: A real-time procedural shading system for programmable graphics hardware. In: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques. pp. 159–170. SIGGRAPH '01, ACM (2001)
25. Pugh, W., Teitelbaum, T.: Incremental computation via function caching. In: Principles of Programming Languages. pp. 315–328 (1989)
26. Ramalingam, G., Reps, T.: A categorized bibliography on incremental computation. In: Principles of Programming Languages. pp. 502–510 (1993)
27. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: Proceedings of the ACM Annual Conference - Volume 2. pp. 717–740. ACM '72, ACM, New York, NY, USA (1972), http://doi.acm.org/10.1145/800194.805852
28. Segal, M., Akeley, K.: The OpenGL Graphics System: A Specification (Version 4.0). The Khronos Group, Inc. (2010)
29. Seidel, R., Aragon, C.R.: Randomized search trees. Algorithmica 16(4–5), 464–497 (1996)
30. Taha, W.: Multi-Stage Programming: Its Theory and Applications. Ph.D. thesis, Oregon Graduate Institute of Science and Technology (1999)

31. Taha, W., Sheard, T.: Multi-stage programming with explicit annotations. In: Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation. PEPM '97 (1997)