A Complete Refinement System for Substructural SSA

ANONYMOUS AUTHOR(S)

In this paper, we give a type-theoretic presentation of static single assignment (SSA) form, the dominant compiler intermediate representation. Our type theory is very rich one, with substructural types and an effect system, which enables us to give a syntactic presentation of refinement for SSA programs which validates the effect-dependent program transformations that compilers depend upon. We are also able to give a categorical semantics for our calculus, and we prove our calculus both sound and complete with respect to the categorical axiomatization. Completeness ensures that there are no missing refinements from our calculus, which lets compiler writers validate optimizations in a model-free way. On the other side, the fact that our axiomatization is sound gives us a syntax-free way of verifying that complex combinations of features such as undefined behaviour, nondeterminism, and weak memory still validate the program transformations compilers rely upon.

CCS Concepts: • **Theory of computation** → **Denotational semantics**; **Categorical semantics**; **Type theory**.

Additional Key Words and Phrases: SSA, Categorical Semantics, Elgot Structure, Effectful Category

ACM Reference Format:
Anonymous Author(s). 2025. A Complete Refinement System for Substructural SSA. 1, 1 (March 2025), 55 pages.
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Static single assignment form, or SSA, rapidly became the dominant compiler intermediate representation (IR) after its introduction by Alpern et al. [2] and Rosen et al. [26] in the 1980s. The fundamental idea behind SSA is one which is very familiar to functional programmers: if a variable is defined exactly once and never reassigned, then substitution is then a valid program transformation. This both simplifies the implementation and improves the performance of many compiler optimizations.

The correctness of SSA transformations has mostly been handled informally, because it was originally intended to be a simple, first-order imperative programming language. Unfortunately, in the decades since its introduction, computers have become much less simple, and the optimizations that compiler writers want to do have become much more complicated. Modern hardware is highly concurrent, and exhibits user-visible non-sequentially-consistent behaviour: *weak memory* [4]. This means memory can no longer be correctly modelled as a global array of bytes. Furthermore, compilers like LLVM and GCC now perform very aggressive optimizations exploiting knowledge of memory aliasing and undefined behaviour. Because all these optimizations are all done in the presence of nondeterminism, the transformations compilers want to do are not usually program equivalences, but merely refinements: the possible behaviours of the optimized program must be a subset of the possible behaviours of the original program.

As a concrete example, consider the interaction of undefined behaviour and other effects. The division operation div y z exhibits *undefined behavior* (UB) in both C and LLVM if z is zero, but

ACM XXXX-XXXX/2025/3-ART
https://doi.org/XXXXXXX.XXXXXXX

otherwise has no side-effects. Therefore, the program let $x = \mathsf{div}\ y\ z;\ e$ is equivalent to $[\mathsf{div}\ y\ z/x]e$ if z is known to be nonzero, but not otherwise. For example, if $z = 0$ and $e = ()$, let $x = \mathsf{div}\ y\ 0;\ ()$ always exhibits UB, whereas $[\mathsf{div}\ y\ 0/x]() \equiv ()$ simply does nothing! However, one direction of the rewrite – turning let $x = \mathsf{div}\ y\ z;\ e$ into $[\mathsf{div}\ y\ z/x]e$ – is always a safe transformation, since:

- If z is zero, then let $x = \mathsf{div}\ y\ z;\ e$ has UB, so we can rewrite it to anything we want!
- Otherwise, $\mathsf{div}\ y\ z$ is pure, so substitution is unconditionally valid

Note that this substitution moved a potentially-effectful operation *forward* in the program's execution. Depending upon how often x occurs inside of e, this substitution could also duplicate or drop the effectful operation. This observation gives rise to a categorization of effectful terms, based on how they interact with substitution.

Because it is always safe to move UB after any other effect, we call it a *right-mover* with respect to that effect. Furtheremore, since it is also always a refinement both to eliminate and to duplicate UB, we say UB is both an *eliminable* and *duplicable* effect. This is because eliminating UB reduces the set of possible behaviours, and duplicating occurences of UB does not increase the set of possible behaviours. We also have that $(\mathsf{div}\ y\ z, \mathsf{div}\ y\ z) \twoheadrightarrow \mathsf{let}\ x = \mathsf{div}\ y\ z;\ (x, x)$, so UB is also *fusable*. When an effect is both fusable and duplicable, we also call it *relevant*. On the other hand, it is not a refinement to introduce UB, since $() \not\twoheadrightarrow \mathsf{let}\ x = \mathsf{div}\ y\ z;\ ()$. Hence we say it is not *introducible*. We say an effect which is both *eliminable* and *introducible* is *affine*.

We might also want to ask with respect to which effects UB is a *left-mover*, i.e., for which effects of e we have $(e, \mathsf{div}\ y\ z) \twoheadrightarrow \mathsf{let}\ x = \mathsf{div}\ y\ z;\ (e, x)$ (with pairs evaluated left-to-right). With some thought, we can see this refinement holds only when execution is guaranteed to continue after evaluating e, so this rules out effects like nontermination or exceptions, but effects like nondeterminism or memory access are fine. That is, UB is a *left-mover* with respect to nondeterminism and memory access, but not in general. Since it is both a left and right mover w.r.t. these effects, we say UB *commutes* with them. An effect that commutes with everything, like the empty effect $\bot$, is called *central*.

The question of what rewrites are permitted in the presence of effects is one that compiler writers struggle with [1], because the less conservative they are, the faster the code they can generate, but the more dependent the are on having a clear understanding of each effectful operation's semantics and interactions.

Because of this complexity, the old informal techniques are no longer sufficient, and we need to study SSA using more mathematically sophisticated techniques to understand and justify what modern hardware and compilers do. In this paper, we introduce a type-theoretic account of static single assigment form, and equip it with a semantics which explains and can be used to justify many program transformations even in the presence of effects like state, undefined behaviour, and weak memory concurrency. Concretely, our contributions are as follows:

- First, we give a pair of type theories for SSA. The first language, λ_{SSA}, is intended to mimic the structure of traditional presentations of SSA. The second type theory, λ_{iter}, has a syntax very different from ordinary SSA, but which greatly facilitates giving a syntactic presentation of its refinement theory. Both of these theories have a rich type system with both linearity and effect tracking. The extra structure enables us to express many effect-dependent program transformations in a type-directed way. We show that λ_{iter}can be seen as an alternative presentation of ordinary SSA by showing that there are meaning-preserving translations to and from λ_{SSA}.
- We then give a categorical axiomatization of first-order effectful programs with looping and control, and then show that $\lambda_{\mathsf{iter}}$is soundly interpreted in this model, and furthermore

we show completeness of our refinement theory by proving that the syntactic model is the initial model.
- We then give a collection of concrete models validating the categorical axiomatization. We start with a model including nontermination, nondeterminism, and undefined behaviour, and then show how to give another model augmented with state, as well as giving a model for release-acquire concurrency which validates our axioms. Finally, we show how a variety of interesting optimizations are validated by our model.

Finally, many of the results in this paper have been mechanized in the Lean proof assistant.

2 SSA

One of the first IRs to find widespread use was *register transfer level (RTL)* code. RTL programs are composed of a collection of *basic blocks*, defined to be a sequence of instructions of the form $x = f(y, z)$ ending with a *terminator* instruction, which may branch to other basic blocks. The basic blocks, and the jumps between them, form the nodes and edges of that program's *control-flow graph*. Because RTL variables are *mutable*, program analyses have to keep track of the values of every variable *at every program point*, significantly complicating performant implementations.

To avoid this multplicative overhead, the *static single assignment*, or *SSA*, IR enforces the restriction that every variable has precisely one definition. Just as in functional programs, this raises the question of how to handle control-flow dependent variables such as loop induction variables. The answer is the same [3]: each basic block (or tail-recursive function!) takes a list of control-flow dependent variables as *arguments*. Traditionally, these arguments are represented "inside out" using *ϕ-functions*, which are assignments whose value depends on which basic block is their immediate predecessor. Our formalization follows modern practice and uses the *basic blocks with arguments* (BBA) representation of SSA.

In SSA, the property that every variable has a single definition is graph-theoretic: we require that the use point of the variable is *dominated* by its definition in the control-flow graph.[1] Semantics, however, is much easier with lexical scoping. To interconvert between the two, we note that the dominance relation on basic blocks always forms a tree rooted at the entry block: we call this the *dominator tree*. We will call a subtree of the dominator tree, which has a single entry point (the root) and multiple exits (the leaves, all dominated by the root), a *region*. The variables defined in a given basic block β are visible from another block β' if and only if β dominates β', i.e., if β is a child in the dominator tree, contained in the region r rooted at β. Hence, dominance based scoping can be represented as lexical scoping *with respect to the dominator tree*.

This idea underlies the design of λ_{SSA}, what we call *type-theoretic SSA*, which has the grammar given in Figure 2. Rather than give a grammar for basic-blocks and control-flow graphs, we instead give a grammar for *regions* r, s, t, which are composed of a series of let-bindings (each corresponding to an instruction o), followed by a subtree κ, which is composed of a terminator τ wrapped in where-blocks containing the region's dominated subregions.[2]

If we squint, we can see that this is just SSA with where-blocks as an annotation representing the dominator tree: basic blocks correspond to sequences of let-bindings followed by a terminator. Indeed, it is straightforward to verify that any well-scoped SSA program can be converted to a λ_{SSA} program by simply adding in where-blocks corresponding to the dominator tree; similarly, simply erasing the where-blocks from an λ_{SSA} program yields a program in standard basic-blocks with arguments SSA; we see an example of this in Figure 1c. We can also show that any two programs

[1] A node n is dominated by a node N in a directed graph if every path to n passes through N.
[2] We distinguish recursive and non-recursive where-blocks for effect-system bookkeeping, but semantically they are identical.

```
start :  let n = 10;
         br loop
loop :   let i0 = φ(start : 1, body : i1)
         let a0 = φ(start : 1, body : a1)
         if i0 < n { br body }
         else { ret a0 }
body :   let t = i0 + 1
         let a1 = a0 * t
         let i1 = i0 + 1
         br loop
```

(a) ϕ-nodes

```
start :          let n = 10;
                 br loop(1, 1)
loop(i0, a0) :   if i0 < n { br body }
                 else { ret a0 }
body :           let t = i0 + 1
                 let a1 = a0 * t
                 let i1 = i0 + 1
                 br loop(i1, a1)
```

(b) Basic-blocks with arguments

```
let n = 10;
br loop(1, 1)
where loop(i0, a0) : {
  if i0 < n { br body }
  else { ret a0 }
  where body : {
    let t = i0 + 1
    let a1 = a0 * t
    let i1 = i0 + 1
    br loop(i1, a1)
  }
}
```

(c) Lexical scoping

Fig. 1. A program to compute 10! written in standard SSA (using ϕ nodes), like in LLVM [18], and using basic-blocks with arguments, like in MLIR [19] and Cranelift [27], with both implicit (dominance-based) and explicit (lexical) scoping. The arguments i_0, a_0 corresponding to the ϕ-nodes i_0, a_0 are colored in red and blue, respectively.

$$o ::= x \mid f\ x \mid () \mid (x, y) \mid \iota_l\ x \mid \iota_r\ x \mid \text{abort}\ x$$

$$r, s, t ::= \kappa \mid \text{let}\ x = o; t \mid \text{let}\ (x, y) = o; t$$

$$\kappa ::= \tau \mid \kappa\ \text{where}_{\text{nonrec}}\ L \mid \kappa\ \text{where}_{\text{rec}}\ L$$

$$\tau ::= \text{br}\ \ell\ o \mid \text{case}\ o\ \{\iota_l\ y : \tau, \iota_r\ z : \tau'\}$$

$$L ::= \cdot \mid L, \ell(x) : \{t\}$$

Fig. 2. Grammar for λ_{SSA} programs.

```
struct BasicBlock {
  vector<Instruction> instructions;              // unary/binary let-bindings
  Terminator terminator;                         // LHS of where-block
  map<Label, (Argument, BasicBlock)> children;   // RHS of where-block
}
```

Fig. 3. Data encoded by the grammar in Figure 2

which are equivalent up to the placement of where-blocks have equivalent semantics, therefore justifying λ_{SSA} as being simply SSA with additional annotations.

3 λ_{iter}, an expression language for SSA

SSA is a useful IR because it enables complex, whole-procedure transformation of code, but its block-statement-value hierarchy makes it difficult to uniformly formulate the allowed transformations as an (in)equational theory. If we unified all of these levels into a single expression language, then formulating many transformations becomes much simpler. For example, if loops were an expression,

$$A, B, C ::= X \mid A \otimes B \mid \mathbf{1} \mid A + B \mid \mathbf{0}$$

$$a, b, c ::= x \mid f\ a \mid \text{let } x = a;\ b \mid () \mid (a, b) \mid \text{let } (x, y) = a;\ b \mid \iota_l\ a \mid \iota_r\ b \mid \text{case } a\ \{\iota_l\ x : b, \iota_r\ y : c\} \mid \text{abort } a \mid \text{iter } a\ \{\iota_r\ x : b\}$$

$$q ::= 0 \mid 1 \mid \omega^+ \mid 1^? \mid \omega$$

$$\Gamma ::= \cdot \mid \Gamma, x : A$$

$$\mathbf{q} ::= \cdot \mid \mathbf{q}, q$$

Fig. 4. Syntax for λ_{iter} types, expressions, quantities, and contexts.

then control-flow-graph transformations which duplicate or fuse loop bodies are merely instances of substitution (and its dual, CSE/let-introduction).

In this section, we introduce λ_{iter}, which is an expression-oriented variant of SSA. Essentially, it is a simple first-order expression language with support for binding/sequencing, branching, and loops. λ_{iter} looks very different from traditional presentations of SSA, but we later prove in Subsection 5.3 that the two syntaxes are completely equivalent to one another. The main novelty of λ_{iter} is in its type system and equational theory. It has a rich substructural type and effect system, which enables us to give a complete inequational characterization of refinement of λ_{iter} programs. Since λ_{iter} is equivalent to SSA, we get a complete syntactic characterization of refinement for SSA programs.

3.1 Syntax and Typing Rules

As mentioned before, λ_{iter} is a standard first-order expression language with branching and iteration: a functional analogue of While. Its grammar is in Figure 4, and is parametrized by a set of *base types* $X \in \mathcal{X}$ and a set of *instructions* $f \in \mathcal{I}$. To model multiple arguments and control flow, our grammar of types A, B, C includes all *tensor products* $A \otimes B$ and *coproducts* $A + B$ generated by base types X, along with a *unit type* $\mathbf{1}$ and an *empty type* $\mathbf{0}$. Mirroring this, expressions a, b, c consist of

- *Variables* x, y, z
- *Applications* $f\ a$, consisting of instructions $f \in \mathcal{I}$ applied to an expression a
- *Let-bindings* $\text{let } x = a;\ b$ and *destructuring let-bindings* $\text{let } (x, y) = a;\ b$. We write $a; b$ as syntactic sugar for $\text{let } \cdot = a;\ b$ (i.e., a let binding where the bound variable is not used in b).
- *Case-expressions* $\text{case } e\ \{\iota_l\ x : a, \iota_r\ y : b\}$, representing branching control-flow
- *Iteration-expressions* $\text{iter } a\ \{\iota_r\ x : b\}$ representing a variant of *tail-controlled loop*, which:
  - Evaluates an initial value $a : A$
  - Evaluates the loop body $b : B + A$ with $x = a$; if the loop evaluates to a value of type B, we return it, otherwise, we re-evaluate the loop with x having the new value of type A

Our typing judgement is $\Gamma^{\mathbf{q}} \vdash_\epsilon a : A$. This says that in the context Γ, the expression a has type A and *effects* ϵ, and uses Γ's variables according to the quantities in the *quantity vector* $\mathbf{q}$.

3.1.1 Variable Quantities. Our type system tracks how often individual variables are used to reason about rewrites involving effectful expressions. Inspired by substructural logic, which distinguishes linear (exactly once), affine (at most once), and relevant (at least once), we introduce a join semilattice of *quantities* to model these usages. The primitive quantities are:

- 0 – corresponding to being used zero times
- 1 – corresponding to being used exactly once
- ω^+ – corresponding to being used multiple (≥ 1) times

This forms a partial order $\{0, 1 \leq \omega^+\}$. To complete it into a join-semilattice, we add elements

- $0 \sqcup 1$ – written $1^?$, corresponding to being used at most once
- $0 \sqcup \omega^+$ – written ω, corresponding to being used any number of times

We call the set $Q^0 = \{0, 1, \omega^+, 1^?, \omega\}$ the *extended* set of quantities, and $Q = \{1, \omega^+, 1^?, \omega\}$ the set of (*nonzero*) quantities. Q also forms a lattice, with $\omega^+ \sqcap 1^? = 1$ (though Q^0 does not).

Contexts are a list Γ of typed variables $x : A$ along with a list of quantities of equal length $\mathbf{q}$, which we write as $\Gamma^{\mathbf{q}}$. This is equivalent to annotating each variable with a quantity $x : A^q$. We define the syntax sugars $\Gamma^{\mathbf{q}}, x : A^q := (\Gamma, x : A)^{\mathbf{q},q}$ and $\Gamma^{\mathbf{q}}, x : A := (\Gamma, x : A)^{\mathbf{q},\omega}$.

Since we already track the usage of variables, it adds little extra complexity to support substructural types (types whose elements can *only* be used a certain number of times). Assuming each base type $X \in \mathcal{X}$ is equipped with a *linearity* $\mathrm{q}(X) \in Q$, the linearity of any type A is defined as

$$\mathrm{q}(A \otimes B) = \mathrm{q}(A + B) = \mathrm{q}(A) \sqcap \mathrm{q}(B) \qquad \mathrm{q}(\mathbf{1}) = \mathrm{q}(\mathbf{0}) = \omega \tag{1}$$

We now define the linearity of annotated types A^q and contexts Γ as follows:

$$\mathrm{q}(\cdot) = \omega \qquad \mathrm{q}(\Gamma^{\mathbf{q}}, x : A^q) = \mathrm{q}(\Gamma^{\mathbf{q}}) \sqcap \mathrm{q}(A^q) \qquad \mathrm{q}(A^q) = \begin{cases} \mathrm{q}(A) \sqcap q & \text{if } q \neq 0 \\ \omega & \text{if } q = 0 \end{cases} \tag{2}$$

In particular, note that the linearity of an unused variable is always unrestricted. We will enforce our quantity restrictions through the means of *weakening* and *splitting* judgements, which will determine how variables may be left unused or apportioned between subcontexts, respectively. In particular, the judgement $\Gamma^{\mathbf{q}} \mapsto \Delta^{\mathbf{q}'}$, pronounced "$\Gamma^{\mathbf{q}}$ weakens $\Delta^{\mathbf{q}'}$," is defined as follows:

$$\frac{}{\cdot \mapsto \cdot}\ \text{nil} \qquad \frac{\Gamma^{\mathbf{q}} \mapsto \Delta^{\mathbf{q}'} \quad q' * \mathrm{q}(A) \leq q * \mathrm{q}(A)}{\Gamma^{\mathbf{q}}, x : A^q \mapsto \Delta^{\mathbf{q}'}, x : A^{q'}}\ \text{cons} \qquad \frac{\Gamma^{\mathbf{q}} \mapsto \Delta^{\mathbf{q}'} \quad 0 \leq q * \mathrm{q}(A)}{\Gamma^{\mathbf{q}}, x : A^q \mapsto \Delta^{\mathbf{q}'}}\ \text{skip}$$

To define weakening, we extend the meet on Q to a *product* of quantities $q, q' \in Q^0$ as follows:

$$q * q' = \begin{cases} 0 & \text{if } q = 0 \text{ or } q' = 0 \\ q \sqcap q' & \text{if } q, q' \in Q \end{cases} \tag{3}$$

The rule skip says affine variables are discarable, which it encodes via the condition $0 \leq q * \mathrm{q}(A)$. The rule cons says that we may replace a variable quantity q with another quantity q' allowing less permissive usage *relative to the type's quantity*. For example, for an affine type A (usable $1^?$), the quantity ω^+ forces linear usage ($\omega^+ * 1^? = 1$), whereas the quantity $1^?$ still permits deletion ($1^? * 1^? = 1^?$). This induces a *preorder* on contexts, with two contexts equivalent if their component variables can be used the same way. We now define *context splitting* $\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r$ as follows:

$$\frac{}{\cdot \vdash \cdot = \cdot + \cdot}\ \text{nil} \qquad \frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \omega^+ \leq q \sqcap \mathrm{q}(A)}{\Gamma, x : A \vdash (\mathbf{q}, q) = (\mathbf{q}_l, q) + (\mathbf{q}_r, q)}\ \text{both}$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r}{\Gamma, x : A \vdash (\mathbf{q}, q) = (\mathbf{q}_l, q) + (\mathbf{q}_r, 0)}\ \text{left} \qquad \frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r}{\Gamma, x : A \vdash (\mathbf{q}, q) = (\mathbf{q}_l, 0) + (\mathbf{q}_r, q)}\ \text{right}$$

The rules left and right allow us to use a variable, regardless of quantity, in either subexpression, whereas the rule both states that it must be relevant to be used in both branches.

3.1.2 Effects. Our language of effects is a bounded[3] join-semilattice $\mathcal{E}$. We will represent pure expressions as having the bottom effect $\bot$, whereas expressions with effect $\top$ have "arbitrary" effect, and expressions with effect $\epsilon \sqcup \epsilon'$ have both effects ϵ and ϵ'.

Becaise we want to rewrite effectful programs it is not enough to know which effects a program might have: we also need to know how they interact. In particular, we need to know:

[3] has a maximum element $\top$ and minimum element $\bot$

(1) Which effects are *iterable*, i.e., stable under loops.
(2) The *multiplicity* of each effect ϵ : whether it is *duplicable*, *fusable*, *introducible*, or *eliminable*.
(3) If effect ϵ is a *left-mover* or *right-mover* relative to η (i.e. can ϵ be moved before or after η)

Not all effects are stable under arbitrary iteration. For example, a pure, total computation iterated infinitely often can now exhibit the effect of nontermination. Indeed, any total effect (such as reading or writing from a location) repeated infinitely often can gain the nontermination effect. To model stability, we are going to require that there is an upwards-closed subset $\mathcal{E}^\infty \subseteq \mathcal{E}$ of *iterative effects*. The intuition behinds the upwards-closed requirement is that once nontermination is in the effect ϵ, then it will continue to be present in any supereffect.

To represent multiplicity, we take advantage of the fact that we already have a notion of quantitity. Unlike in ordinary linear type systems, we are interested in refinement (a directed notion) rather than pure equations. We split the property of contraction into the pair of properties *duplicability* and *fusability*: contraction in each direction of refinement. Likewise weakening splits into the pair of directed weakening properties of *eliminability* and *introducability*. To model this, we assign a *pair* of quantities $\mathsf{q}^+(e), \mathsf{q}^-(e) \in Q$ to each effect, such that

- An effect can be eliminated if $1^? \leq \mathsf{q}^+(e)$, and introduced if $1^? \leq \mathsf{q}^-(e)$
- An effect can be duplicated if $\omega^+ \leq \mathsf{q}^+(e)$, and fused if $\omega^+ \leq \mathsf{q}^-(e)$

To be compatible with our notion of sub-effect, such a map should be *antitone* and have the property that $\mathsf{q}^p(\bot) = \omega$ for $p \in \{+, -\}$ (i.e., pure computations can be used as often as you like).

Finally, we need to know how effects can be reordered. For example, a memory read and an IO write cannot interfere, and hence can be reordered. To model this, we introduce a *directed commutativity relation* $\rightharpoonup \subseteq \mathcal{E} \times \mathcal{E}$. If $\epsilon \rightharpoonup \epsilon'$, then a computation performing ϵ-effects can be moved past a computation performing ϵ'-effects without introducing new behaviour. We require that $\bot \rightharpoonup \epsilon$ and $\epsilon \rightharpoonup \bot$, to ensure that pure computations can be moved freely, and we also require that the relation is antitone, to model that decreasing the number of effects never loses rewrites. We introduce the syntax sugar $\epsilon \leftharpoonup \eta \iff \eta \rightharpoonup \epsilon, \epsilon \rightleftharpoons \eta \iff \epsilon \rightharpoonup \eta \wedge \epsilon \leftharpoonup \eta$. For polarities $p \in \{+, -\}$, we define $\rightharpoonup^p$ in the obvious manner, with $\epsilon \rightharpoonup^+ \eta \iff \epsilon \rightharpoonup \eta$ and $\epsilon \rightharpoonup^- \eta \iff \epsilon \leftharpoonup \eta$. Putting all this together, we may now define an *effect system* as follows:

Definition 3.1 (Effect system). An *effect system* $\mathcal{E}$ is a bounded join-semilattice $\mathcal{E}$ equipped with:

- an upwards closed subset $\mathcal{E}^\infty$ of *iterative effects* containing $\top$, and
- a pair of antitone maps $\mathsf{q}^+, \mathsf{q}^- : \mathcal{E} \to Q$ which map $\bot$ to ω,
- a *directed commutativity relation* $\rightharpoonup \subseteq \mathcal{E} \times \mathcal{E}$, an antitone relation containing $\bot \rightharpoonup \epsilon$ and $\epsilon \rightharpoonup \bot$ for all ϵ.

3.1.3 *Typing rules.*

We now have all the pieces we need to define a λ_{iter}-*signature*:

Definition 3.2 (λ_{iter}-signature). A λ_{iter}signature $\mathcal{S} = (\mathcal{X}, \mathcal{I}, \mathcal{E})$ consists of a set of *base types* $X \in \mathcal{X}$, a set of *instructions* $f \in \mathcal{I}$, and an iterative effect system $\mathcal{E}$ such that we associate:

- Every base type X type to a *quantity* $\mathsf{q}(X) \in \{1, 1^?, \omega^+, \omega\}$.
- Every instruction to a *source type* $\mathrm{src}(f) = A$, a *target type* $\mathrm{trg}(f) = B$, and an *effect* $\mathrm{eff}(f) = \epsilon$.

All the following definitions in this section will be with respect to an arbitrary λ_{iter}-signature $\mathcal{S}$. We give the typing rules for λ_{iter} in Figure 5. Our rules are syntax directed, with one rule for each production in our grammar. In particular,

- The var rule says that a variable x has type A in the context Γ^{q} if Γ^{q} weakens to the singleton context $x : A^1$, i.e., if x has nonzero quantity and all other (unused) variables are affine. Since accessing a variable is pure, we can give it an arbitrary effect ϵ.

$$\frac{\Gamma^{\mathbf{q}} \mapsto x : A^1}{\Gamma^{\mathbf{q}} \vdash_\epsilon x : A}\ \text{var} \qquad \frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_r} \vdash_\epsilon a : A \quad \Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon b : B}{\Gamma^{\mathbf{q}} \vdash \text{let } x = a;\ b : B}\ \text{let}_1$$

$$\frac{f : A \to_\epsilon B \quad \Gamma^{\mathbf{q}} \vdash_\epsilon a : A}{\Gamma^{\mathbf{q}} \vdash_\epsilon f\ a : B}\ \text{op} \qquad \frac{f \in \mathcal{I} \quad \text{src}(f) = A \quad \text{trg}(f) = B \quad \text{eff}(f) \le \epsilon}{f : A \to_\epsilon B}\ \text{inst}$$

$$\frac{\Gamma^{\mathbf{q}} \mapsto \cdot}{\Gamma^{\mathbf{q}} \vdash_\epsilon () : \mathbf{1}}\ \text{unit} \qquad \frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_l} \vdash_\epsilon a : A \quad \Gamma^{\mathbf{q}_r} \vdash_\epsilon b : B}{\Gamma^{\mathbf{q}} \vdash_\epsilon (a, b) : A \otimes B}\ \text{pair}$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_r} \vdash a : A \otimes B \quad \Gamma^{\mathbf{q}_l}, x : A, y : B \vdash_\epsilon c : C}{\Gamma^{\mathbf{q}} \vdash_\epsilon \text{let } (x, y) = a;\ c : C}\ \text{let}_2$$

$$\frac{\Gamma^{\mathbf{q}} \vdash_\epsilon a : A}{\Gamma^{\mathbf{q}} \vdash_\epsilon \iota_l\ a : A + B}\ \text{inl} \qquad \frac{\Gamma^{\mathbf{q}} \vdash_\epsilon b : B}{\Gamma^{\mathbf{q}} \vdash_\epsilon \iota_r\ b : A + B}\ \text{inr} \qquad \frac{\Gamma^{\mathbf{q}} \vdash_\epsilon a : \mathbf{0}}{\Gamma^{\mathbf{q}} \vdash_\epsilon \text{abort } a : C}\ \text{abort}$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_r} \vdash_\epsilon e : A + B \quad \Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon a : C \quad \Gamma^{\mathbf{q}_l}, y : B \vdash_\epsilon b : C}{\Gamma^{\mathbf{q}} \vdash_\epsilon \text{case } e\ \{\iota_l\ x : a, \iota_r\ y : b\} : C}\ \text{case}$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \mathrm{q}(\Gamma^{\mathbf{q}_l}) = \omega \quad \epsilon \in \mathcal{E}^\infty \quad \Gamma^{\mathbf{q}_r} \vdash_\epsilon a : A \quad \Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon b : B + A}{\Gamma^{\mathbf{q}} \vdash_\epsilon \text{iter } a\ \{\iota_r\ x : b\} : B}\ \text{iter}$$

Fig. 5. Typing rules for λ_{iter}

- To type a let-binding let $x = a;\ e$ in $\Gamma^{\mathbf{q}}$ with let_1 with effect ϵ, we must:
  - Split the context into a left-component $\mathbf{q}_l$ and a right-component $\mathbf{q}_r$, such that:
  - a is well-typed and has effect ϵ in the *right* component $\Gamma^{\mathbf{q}_r}$, and b is well-typed in $\Gamma^{\mathbf{q}_l}$ plus an unrestricted parameter $x : A^\top$.
- case and let_2 are similar, except that let_2's body requires two parameters (one for each component of the tensor product), while both branches of a case-statement share the *same* left-component $\Gamma^{\mathbf{q}_l}$ (since exactly one branch is executed.)
- unit is well-typed and pure (and therefore can be assigned an arbitrary effect ϵ) in any context composed solely of affine variables, i.e. satisfying $\Gamma^{\mathbf{q}} \mapsto \cdot$.
- pair splits the context between the left and right components and checks each one,
- The iter rule is the most complex: we split the context into a component $\mathbf{q}_r$ for the initial value and a component $\mathbf{q}_l$ for the body. The initial value is then evaluated and passed in as an additional parameter of type A to the body. $\mathbf{q}_l$ must be unrestricted, since the loop may execute any number of times. Finally, the effect of the loop body must be an iterative effect.

3.2 Syntactic Metatheory

As a basic sanity check, we can verify that our calculus admits *weakening*:

Lemma 3.3 (Weakening). *If* $\Gamma'^{\mathbf{q}'} \mapsto \Gamma^{\mathbf{q}}$ *and* $\Gamma^{\mathbf{q}} \vdash_\epsilon a : A$ *then* $\Gamma'^{\mathbf{q}'} \vdash_\epsilon a : A$.

We now define substitution for effectful terms. First, we define a judgement $\Gamma^{\mathbf{q}} \vdash_\epsilon \sigma \rhd \Delta^{\mathbf{q}'}$ for substitutions, with rules in Figure 6. This may be read as "σ takes the context $\Gamma^{\mathbf{q}}$ to the context $\Delta^{\mathbf{q}'}$ with effect ϵ." Our rules may be interpreted as follows:

- nil says that the empty substitution $\cdot$ takes any affine context $\Gamma^{\mathbf{q}}$ to the empty context $\cdot$. This holds because a well-typed term in the empty context is also well-typed in any affine context. The effect is an arbitrary ϵ, since the empty substitution is always pure.
- zero rule says that if a variable $x : A^0$ is unused, then we can map it to an arbitrary (even ill-typed) term a. Since x will never appear in a well-typed term, a will never appear in their substitutions, and therefore needs no restrictions. The effect ϵ of σ is unchanged.

$$\frac{\Gamma^{\mathbf{q}} \mapsto \cdot}{\Gamma^{\mathbf{q}} \vdash_{\epsilon} \cdot \rhd \cdot}\ \text{nil} \qquad \frac{\Gamma^{\mathbf{q}} \vdash_{\epsilon} \sigma \rhd \Delta}{\Gamma^{\mathbf{q}} \vdash_{\epsilon} \sigma, x \mapsto a \rhd \Delta^{\mathbf{q}'}, x : A^{0}}\ \text{zero}$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_l} \vdash_{\epsilon_l} \sigma \rhd \Delta^{\mathbf{q}} \quad \Gamma^{\mathbf{q}_r} \vdash_{\epsilon_r} a : A \quad q \le \mathrm{q}(\Gamma^{\mathbf{q}_r}) \quad \epsilon_l \rightleftharpoons \epsilon_r \quad \epsilon_l, \epsilon_r \le \epsilon}{\Gamma^{\mathbf{q}} \vdash_{\epsilon} \sigma, x \mapsto a \rhd \Delta^{\mathbf{q}'}, x : A^{q}}\ \text{cons}$$

Fig. 6. Substitution rules for λ_{iter}

- cons: to type a substitution $\sigma, x \mapsto a$ taking $\Gamma^{\mathbf{q}}$ to $\Delta^{\mathbf{q}'}, x : A^{q}$ with effect ϵ, we split the input context into a context $\Gamma^{\mathbf{q}_l}$, used to type σ with effect $\epsilon_l \le \epsilon$, and $\Gamma^{\mathbf{q}_r}$, used to type a with effect $\epsilon_r \le \epsilon$. $\Gamma^{\mathbf{q}_r}$ must be useable with quantity q; i.e., is relevant if q is relevant and affine if q is affine. Finally, the effect of σ (ϵ_l) and the effect of a (ϵ_r) must commute.

We write $[\sigma]$ for the action of substitution σ on term a. We can now state substitution:

Lemma 3.4 (Substitution). *If* $\Gamma'^{\mathbf{q}'} \vdash_{\epsilon} \sigma \rhd \Gamma^{\mathbf{q}}$ *and* $\Gamma^{\mathbf{q}} \vdash_{\epsilon} a : A$ *then* $\Gamma'^{\mathbf{q}'} \vdash_{\epsilon} [\sigma]a : A$.

3.3 Refinement Theory

We now define our core notion of refinement. The judgment

$$\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} a \twoheadrightarrow b : A$$

means a is refined by b in the context $\Gamma^{\mathbf{q}}$, modulo rewrites $\mathcal{R}$. Intuitively, this expresses that b is at least as defined or deterministic as a, possibly after applying known rewrites in $\mathcal{R}$. We define equivalence of terms as mutual refinement:

$$\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} a \approx b : A \iff \Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} a \twoheadrightarrow b : A \wedge \Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} b \twoheadrightarrow a : A \tag{4}$$

For notational convenience, we also use a polarity-marked notation for refinement relation:

$$\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} a \twoheadrightarrow^{+} b : A \iff \Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} a \twoheadrightarrow b : A \qquad \Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} a \twoheadrightarrow^{-} b : A \iff \Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} b \twoheadrightarrow a : A$$

A *rewrite system* $\mathcal{R}$ consists of a set judgments of the form $\Gamma^{\mathbf{q}} \vdash a \twoheadrightarrow b : A$ closed under pure substitution. That is, given a pure substitution σ, we have that

$$\frac{\Gamma^{\mathbf{q}} \vdash_{\perp} \sigma \rhd \Delta^{\mathbf{q}'} \quad (\Delta^{\mathbf{q}'} \vdash a \twoheadrightarrow b : A) \in \mathcal{R}}{(\Gamma^{\mathbf{q}} \vdash [\sigma]a \twoheadrightarrow [\sigma]b : A) \in \mathcal{R}}$$

We will often describe a rewrite system as that *generated* by a set of equations with free variables; e.g., the system generated by $x : \mathbb{N}, y : \mathbb{N} \vdash \mathrm{add}\ (x, y) \twoheadrightarrow \mathrm{add}\ (y, x) : \mathbb{N}$. Our goal is to construct a refinement relation $\twoheadrightarrow$ satisfying the following properties:

(1) **Inclusion of $\mathcal{R}$:** all given rewrites are valid refinements.
(2) **Congruence:** $\twoheadrightarrow$ is closed under term formers and is a preorder.
(3) **Let-normalization:** $\twoheadrightarrow$ abstracts away syntactic associativity of let-bindings.
(4) **Universal properties:** $\twoheadrightarrow$ validates the β- and η-laws of the language.
(5) **Iteration semantics:** $\twoheadrightarrow$ captures fixpoint and control-flow behavior of iteration.

We guarantree that $\twoheadrightarrow$ contains the (reflexive, transitive closure of) $\mathcal{R}$, and therefore satisfies property 1, by the following rules:

$$\frac{(\Gamma^{\mathbf{q}} \vdash a \twoheadrightarrow b : A) \in \mathcal{R}}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} a \twoheadrightarrow b : A}\ \text{base} \qquad \frac{\Gamma^{\mathbf{q}} \vdash_{\epsilon} a : A}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} a \twoheadrightarrow a : A}\ \text{refl} \qquad \frac{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} a \twoheadrightarrow b : A \quad \Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} b \twoheadrightarrow c : A}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} a \twoheadrightarrow c : A}\ \text{trans}$$

The other congruence rules (in the appendix in Figure 24) correspond one-to-one with our term formers to ensure property 2. Property 2 means that the induced equivalence $\approx$ is also a congruence.

To satisfy property 3, we introduce the binding rules (Figure 7), which are stated as equivalences. These rules express syntactic equivalences up to reassociation and let-floating. For instance, nested let-bindings are rearranged to a canonical form. We do not require binding rules for every construct—rules for pairs and sums, for example, can be derived via β-reduction.

Property 4 is addressed via the reduction rules (Figure 8). Most of these are standard β- and η-equivalences. However, the rule let_1-β^p is *directional*: it expresses that let $x = a;\ b$ refines $[x/a]b$ when the effect ϵ of a is a right-mover with respect to the effect η of b, and when x is used in a way compatible with ϵ and the context. We also require that the context $\Gamma^{\mathbf{q}_r}$ used to type a has a linearity compatible with the usage of x in b, as well as with the effect ϵ (for example, since printing is linear, if a performs printing then x must be used linearly in b even if b is pure). The reverse direction is permitted only under the dual (left-mover) condition. Thus, the usual β-equation:

$$\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{let}\ x = a;\ b \approx [x/a]b : B$$

is derivable under sufficient purity assumptions, and in particular always holds when $\epsilon = \bot$. The rule elim, at first glance, can be viewed as a special case of let_1-β^p (combined with term), but is introduced as a separate rule since it does *not* require $\Gamma^{\mathbf{q}_l}$ to be affine to delete a.

In particular, this means that the following more standard typing rule is *derivable*:

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_l} \vdash_{\epsilon} a : A \quad \Gamma^{\mathbf{q}_r}, x : A^{q} \vdash_{\eta} b : B \quad \epsilon \rightleftharpoons \eta \quad q \leq \mathsf{q}(\Gamma^{\mathbf{q}_r}) \sqcap \mathsf{q}(\epsilon)}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{let}\ x = a;\ b \approx [x/a]b : B}\ \mathsf{let}_1\text{-}\beta \tag{5}$$

In particular, this obviously holds for pure expressions with $\epsilon = \bot$ (modulo linearity of $\Gamma^{\mathbf{q}_r}$), as we would normally expect in an effectful language.

The last thing that remains is to treat iteration. Our rules for iteration, given in Figure 9, are based on the properties of a Conway iteration operator as given in Levy and Goncharov [21]. In particular, we require our operator to satisfy the following properties:

Fixpoint. behavior is encoded by the rule iter-β, which expresses that iteration behaves as a least fixpoint. That is, iter $a\ \{\iota_r\ x : b\}$ evaluates to a, then to $b[a/x]$, and depending on the result of b, either exits with a value in the left branch or continues recursively with a value in the right branch. This unfolds the iteration into a case split let $x = a;$ case $b\ \{\iota_l\ y : y, \iota_r\ z : \mathsf{iter}\ z\ \{\iota_r\ x : b\}\}$ capturing precisely one unfolding of the loop. This gives rise to an inductive account of iteration semantics that meshes naturally with refinement.

Naturality, expressed by the rule let-iter, states that sequencing a computation c after a loop can be interchanged with sequencing c inside the loop's exit branch. This enables reasoning about program structure independent of surface syntax and supports loop-invariant motion of subsequent code.

Codiagonality, via the rule codiag, states that nested iterations of the same body can be collapsed into a single iteration by fusing their recursive branches. Operationally, this justifies flattening nested loops into one with a more complex continuation.

Uniformity, via the rule unif^p, allows us to effectively commute certain effectful operations with the infinite unrolling of a loop body. This is best explained in terms of control-flow graphs: the precondition of the rule is corresponds to the left-hand-side of the diagram in Figure 10, while the postcondition corresponds to the right-hand-side. If we unroll the loops on both sides "infinitely many times," to obtain an infinite tree, we see that unif^p just says that we can apply the rewrite on the left-hand-side "infinitely many times" to convert a tree of b's into a tree of b''s. The name uniformity is by analogy to uniformity in analysis, in which an operation can be used to uniformly transmute each term of an infinite series (rather than only be valid for a finite number of terms).

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad f : A \to_\epsilon B \quad \Gamma^{\mathbf{q}_r} \vdash_\epsilon a : A \quad \Gamma^{\mathbf{q}_l}, y : B \vdash_\epsilon c : C}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{let}\ y = f\ a;\ c \approx \mathsf{let}\ x = a;\ \mathsf{let}\ y = f\ x;\ c : C}\ \text{let-op}$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_c + \mathbf{q}_r \quad \Gamma \vdash \mathbf{q}_c = \mathbf{q}_l + \mathbf{q}_m \quad \Gamma^{\mathbf{q}_r} \vdash_\epsilon a : A \quad \Gamma^{\mathbf{q}_m}, x : A \vdash_\epsilon b : B \quad \Gamma^{\mathbf{q}_l}, y : B \vdash_\epsilon c : C}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{let}\ y = (\mathsf{let}\ x = a;\ b);\ c \approx \mathsf{let}\ x = a;\ \mathsf{let}\ y = b;\ c : C}\ \text{let-let}_1$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathbf{q} = \mathbf{q}_c + \mathbf{q}_r \\ \Gamma \vdash \mathbf{q}_c = \mathbf{q}_l + \mathbf{q}_m \quad \Gamma^{\mathbf{q}_r} \vdash_\epsilon a : A \otimes B \quad \Gamma^{\mathbf{q}_m}, x : A, y : B \vdash_\epsilon c : C \quad \Gamma^{\mathbf{q}_l}, z : C \vdash_\epsilon d : D\end{array}}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{let}\ z = (\mathsf{let}\ (x, y) = a;\ c);\ d \approx \mathsf{let}\ (x, y) = a;\ \mathsf{let}\ z = c;\ d : D}\ \text{let-let}_2$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathbf{q} = \mathbf{q}_c + \mathbf{q}_r \qquad\qquad \Gamma^{\mathbf{q}_l}, x : A \vdash_{\mathcal{R}} a : C \\ \Gamma \vdash \mathbf{q}_c = \mathbf{q}_l + \mathbf{q}_m \quad \Gamma^{\mathbf{q}_m} \vdash_{\mathcal{R}} e : A + B \quad \Gamma^{\mathbf{q}_l}, y : B \vdash_{\mathcal{R}} b : C \quad \Gamma^{\mathbf{q}_r}, z : C \vdash_{\mathcal{R}} d : D\end{array}}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{let}\ z = \mathsf{case}\ e\ \{\iota_l\ x : a, \iota_r\ y : b\};\ d \approx \mathsf{case}\ e\ \{\iota_l\ x : \mathsf{let}\ z = a;\ d, \iota_r\ y : \mathsf{let}\ z = b;\ d\} : D}\ \text{let-case}$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_r} \vdash_\epsilon a : A \otimes B \quad \Gamma^{\mathbf{q}_l}, x : A, y : B \vdash_\epsilon c : C}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{let}\ (x, y) = a;\ c \approx \mathsf{let}\ z = a;\ \mathsf{let}\ (x, y) = z;\ c : C}\ \text{let}_2\text{-bind}$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_r} \vdash_{\mathcal{R}} e : A + B \quad \Gamma^{\mathbf{q}_l}, x : A \vdash_{\mathcal{R}} a : C \quad \Gamma^{\mathbf{q}_l}, y : B \vdash_{\mathcal{R}} b : C}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{case}\ e\ \{\iota_l\ x : a, \iota_r\ y : b\} \approx \mathsf{let}\ z = e;\ \mathsf{case}\ z\ \{\iota_l\ x : a, \iota_r\ y : b\} : C}\ \text{case-bind}$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \mathsf{q}(\Gamma^{\mathbf{q}_l}) = \top \quad \epsilon \in \mathcal{E}^\infty \quad \Gamma^{\mathbf{q}_r} \vdash_\epsilon a : A \quad \Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon b : B + A}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{iter}\ a\ \{\iota_r\ x : b\} \approx \mathsf{let}\ y = a;\ \mathsf{iter}\ y\ \{\iota_r\ x : b\} : B}\ \text{iter-bind}$$

Fig. 7. Binding rules for λ_{iter}

Note that we *cannot* do this for an arbitrary s; we need it to *commute* with the effect of the loop. For example, if the effect of the loop is (potential) nontermination, and s contains a print-statement, uniformity does *not* apply, since while for any given iteration of the loop body we have, where $\mathsf{print} : \mathbf{1}$ and $\mathsf{expr} : \mathbf{1} + \mathbf{1}$ commutative with printing (e.g., reading and writing from memory), $\mathsf{print}; \mathsf{expr} \approx \mathsf{case}\ \mathsf{expr}\ \{\iota_l\ x : \iota_l\ \mathsf{print}, \iota_r\ y : \iota_l\ \mathsf{print}\}$ but

$$\mathsf{iter}\ \mathsf{print}\ \{\iota_r\ \cdot : \mathsf{expr}\} \not\approx \mathsf{iter}\ ()\ \{\iota_r\ \cdot : \mathsf{case}\ \mathsf{expr}\ \{\iota_l\ x : \iota_l\ \mathsf{print}, \iota_r\ y : \iota_l\ y\}\}$$

since the latter may delay the print-statement infinitely far into the future, and hence hang before printing. On the other hand, if we instead have commutative effects (e.g., if s has nondeterminism and the loop has printing and nontermination), we can safely perform the infinite rewrite.

We will denote the set of refinements generated by a set of base refinements $\mathcal{R}$ as $\mathsf{Th}(\mathcal{R})$. In particular, we note that $\mathsf{Th}(\cdot)$ is monotonic, idempotent, and satisfies $\mathcal{R} \subseteq \mathsf{Th}(\mathcal{R})$, making it a closure operator.

4 Semantics

In this section, we will work towards a categorical semantics for λ_{iter}, which we will later prove to be sound and complete w.r.t. the refinement theory in Section 3.3. More precisely, fixing a category C, we wish to map types A and contexts $\Gamma^{\mathbf{q}}$ to objects $[\![A]\!]$ and $[\![\Gamma^{\mathbf{q}}]\!]$ respectively, and then take the semantics $[\![\Gamma^{\mathbf{q}} \vdash_\epsilon a : A]\!]$ of a well-typed term to be a morphism $C([\![\Gamma^{\mathbf{q}}]\!], [\![A]\!])$.

4.1 Models of λ_{iter}

A good categorical semantics is one in which the semantics of a term is constructed in a straightforward, compositional manner from the semantics of its subterms. Furthermore, we would like the equational properties of our term formers to correspond closely to the universal properties of the categorical structure used to interpret them. Thus, our goal is to pick categorical structures which correspond one-to-one to the features of λ_{iter}. In particular, we need to find categorical structures to model our three primary structured control-flow constructs, which are:

$$\frac{\Gamma^{\mathbf{q}} \vdash_{\epsilon} a : \mathbf{1}}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{let}\ x = a;\ ()\approx a : \mathbf{1}}\ \text{term} \qquad \frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q} \quad \Gamma^{\mathbf{q}_l} \vdash_{\epsilon} a : \mathbf{1} \quad 0 \le \mathsf{q}^p(\epsilon) \quad \Gamma^{\mathbf{q}} \vdash_{\eta} b : B}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{let}\ x = a;\ b \twoheadrightarrow^p \mathsf{let}\ x = ();\ b : B}\ \text{elim}$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_r} \vdash_{\epsilon} a : \mathbf{0} \quad \Gamma^{\mathbf{q}_l}, x : A \vdash_{\epsilon} b : B \quad \Gamma^{\mathbf{q}_l}, x : A \vdash_{\epsilon} b' : B}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{let}\ x = \mathsf{abort}\ a;\ b \approx \mathsf{let}\ x = \mathsf{abort}\ a;\ b' : B}\ \text{init}$$

$$\frac{\Gamma^{\mathbf{q}} \vdash_{\epsilon} a : A \otimes B}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{let}\ (x, y) = a;\ (x, y) \approx a : A \otimes B}\ \text{let}_2\text{-}\eta$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma \vdash \mathbf{q}_l = \mathbf{q}_a + \mathbf{q}_b \quad \Gamma^{\mathbf{q}_a} \vdash_{\epsilon} a : A \quad \Gamma^{\mathbf{q}_b} \vdash_{\epsilon} b : B \quad \Gamma^{\mathbf{q}_r}, x : A, y : B \vdash_{\epsilon} c : C}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{let}\ (x, y) = (a, b);\ c \approx \mathsf{let}\ x = a;\ \mathsf{let}\ y = b;\ c : C}\ \text{let}_2\text{-}\beta$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_r} \vdash_{\mathcal{R}} e : A \quad \Gamma^{\mathbf{q}_l}, x : A \vdash_{\mathcal{R}} a : C \quad \Gamma^{\mathbf{q}_l}, y : B \vdash_{\mathcal{R}} b : C}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{case}\ \iota_l\ e\ \{\iota_l\ x : a, \iota_r\ y : b\} \approx \mathsf{let}\ x = e;\ a : C}\ \text{case-}\beta_l$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_r} \vdash_{\mathcal{R}} e : B \quad \Gamma^{\mathbf{q}_l}, x : A \vdash_{\mathcal{R}} a : C \quad \Gamma^{\mathbf{q}_l}, y : B \vdash_{\mathcal{R}} b : C}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{case}\ \iota_r\ e\ \{\iota_l\ x : a, \iota_r\ y : b\} \approx \mathsf{let}\ y = e;\ b : C}\ \text{case-}\beta_r$$

$$\frac{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} e : A + B}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{case}\ e\ \{\iota_l\ x : \iota_l\ x, \iota_r\ y : \iota_r\ y\} \approx e : A + B}\ \text{case-}\eta$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_l} \vdash_{\epsilon} a : A \quad \Gamma^{\mathbf{q}_r}, x : A^{\mathsf{q}} \vdash_{\eta} b : B \quad \epsilon \rightharpoonup \eta \quad q \le \mathsf{q}(\Gamma^{\mathbf{q}_r}) \sqcap \mathsf{q}^p(\epsilon)}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{let}\ x = a;\ b \twoheadrightarrow^p [x/a]b : B}\ \text{let}_1\text{-}\beta^p$$

Fig. 8. Reduction rules for λ_{iter}

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \mathsf{q}(\Gamma^{\mathbf{q}_l}) = \top \quad \epsilon \in \mathcal{E}^{\infty} \quad \Gamma^{\mathbf{q}_r} \vdash_{\epsilon} a : A \quad \Gamma^{\mathbf{q}_l}, x : A \vdash_{\epsilon} b : B + A}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{iter}\ a\ \{\iota_r\ x : b\} \approx \mathsf{let}\ x = a;\ \mathsf{case}\ b\ \{\iota_l\ y : y, \iota_r\ z : \mathsf{iter}\ z\ \{\iota_r\ x : b\}\} : B}\ \text{iter-}\beta$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_c \qquad \mathsf{q}(\Gamma^{\mathbf{q}_l}) = \top \\ \Gamma \vdash \mathbf{q}_c = \mathbf{q}_m + \mathbf{q}_r \quad \mathsf{q}(\Gamma^{\mathbf{q}_m}) = \top \quad \Gamma^{\mathbf{q}_r} \vdash_{\epsilon} a : A \quad \Gamma^{\mathbf{q}_m}, x : A \vdash_{\epsilon} b : B + A \quad \Gamma^{\mathbf{q}_l}, y : B \vdash_{\epsilon} c : C\end{array}}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{let}\ y = \mathsf{iter}\ a\ \{\iota_r\ x : b\};\ c \approx \mathsf{iter}\ a\ \{\iota_r\ x : \mathsf{case}\ b\ \{\iota_l\ y : \iota_l\ c, \iota_r\ z : \iota_r\ z\}\} : C}\ \text{let-iter}$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \mathsf{q}(\Gamma^{\mathbf{q}_l}) = \top \quad \epsilon \in \mathcal{E}^{\infty} \quad \Gamma^{\mathbf{q}_r} \vdash_{\epsilon} a : A \quad \Gamma^{\mathbf{q}_l}, y : A \vdash_{\epsilon} b : (B + A) + A}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{iter}\ a\ \{\iota_r\ x : \mathsf{iter}\ x\ \{\iota_r\ y : b\}\} \approx \mathsf{iter}\ a\ \{\iota_r\ y : \mathsf{case}\ b\ \{\iota_l\ x : x, \iota_r\ y : \iota_r\ y\}\} : B}\ \text{codiag}$$

$$\frac{\eta \rightharpoonup \epsilon \quad \Gamma^{\mathbf{q}_c}, x : A \vdash_{\mathcal{R}} \mathsf{let}\ y = s;\ b \twoheadrightarrow^p \mathsf{case}\ b'\ \{\iota_l\ z : \iota_l\ c, \iota_r\ x : \iota_r\ s\} : C + S}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{let}\ x = a;\ \mathsf{iter}\ s\ \{\iota_r\ y : b\} \twoheadrightarrow^p \mathsf{let}\ z = \mathsf{iter}\ a\ \{\iota_r\ x : b'\};\ c : C}\ \text{unif}^p$$

where $\Gamma \vdash \mathbf{q} = \mathbf{q}_c + \mathbf{q}_r$ $\Gamma \vdash \mathbf{q}_c = \mathbf{q}_l + \mathbf{q}_c$ $\mathsf{q}(\Gamma^{\mathbf{q}_l}) = \top$ $\epsilon \in \mathcal{E}^{\infty}$ $\Gamma^{\mathbf{q}_r} \vdash_{\epsilon} a : A$
$\Gamma^{\mathbf{q}_c}, x : A \vdash_{\eta} s : S$ $\Gamma^{\mathbf{q}_l}, y : S \vdash_{\epsilon} b : C + A$ $\Gamma^{\mathbf{q}_l}, x : A \vdash b' : B + A$ $\Gamma^{\mathbf{q}_c}, z : B \vdash c : C$

Fig. 9. Iteration rules for λ_{iter}

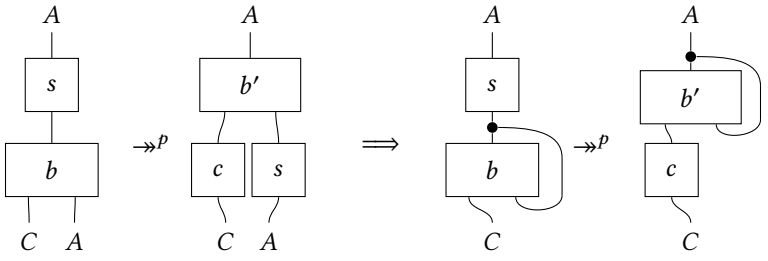

Fig. 10. Control-flow graphs for the uniformity rule in Figure 9

- *Sequencing* and *binding*, which we will do using the structure of a *premonoidal category*
- *Branching*, which we will do using *coproducts*
- *Iteration*, which we will do using a *Conway operator*

To maintain compositionality, we must additionally require that these structures interact properly with each other. Hence, we will also require our category to satisfy

- *Distributivity*, which ensures the premonoidal and coproduct structures are compatible
- *Strength*, which ensures the Conway operator is compatible with the premonoidal structure
- *(Directed) Uniformity*, which ensures the Conway operator is compatible with our effect and refinement systems.

Finally, we'll need to model the features of our type theory; particularly:

- *Refinement*, which we will do using *poset-enrichment*
- *Effects*, which we will do by introducing the new notion of a *substructural effectful category*

In ordinary categorical semantics, we want syntactic equivalence to correspond to equality of morphisms: given $\Gamma^{\mathsf{q}} \vdash_{\mathcal{R}} a \approx b : A$, $[\![\Gamma^{\mathsf{q}} \vdash_{\epsilon} a : A]\!] = [\![\Gamma^{\mathsf{q}} \vdash_{\epsilon} b : A]\!]$.

However, in our calculus, we do not only have a syntactic notion of equivalence, but we also have the order structure arising from refinement. Since terms correpond to morphisms, this means we need to be able to compare morphisms according to an order structure interpreting the refinement relation. Thus, to interpret $\twoheadrightarrow$, we generalize from ordinary categories to *poset-enriched* categories, in which our hom-sets are partially ordered. In particular, we define:

Definition 4.1 (Poset-enriched category). A category $\mathcal{C}$ is *poset-enriched* if each hom-set $\mathcal{C}(A, B)$ is equipped with a partial order $\twoheadrightarrow$ which is compatible with composition, i.e., which satisfies

$$\forall f \twoheadrightarrow f' \in \mathcal{C}(A, B). \forall g \twoheadrightarrow g' \in \mathcal{C}(B, C). (f; g) \twoheadrightarrow (f'; g')$$

A poset-enriched functor between categories $\mathcal{C}$, $\mathcal{D}$ is then simply a functor whose action on morphisms is monotonic.

We can now quite naturally intepret soundness of refinement as follows: given $\Gamma^{\mathsf{q}} \vdash_{\mathcal{R}} a \twoheadrightarrow b : A$, we require that $[\![\Gamma^{\mathsf{q}} \vdash_{\epsilon} a : A]\!] \twoheadrightarrow [\![\Gamma^{\mathsf{q}} \vdash_{\epsilon} b : A]\!]$. Soundness of equivalence follows directly from antisymmetry of partial orders.[4]

We now wish to give poset-enriched versions of the models for our three control-flow primitives, namely, seqencing, branching, and iteration. Premonoidal categories were introduced in Power and Robinson [24] to model side-effectful, sequential computations with first-order binding. A premonoidal category can be viewed as a generalization of a monoidal category, in which *sliding* does not necessarily hold; i.e., we have that, in general, $(f \otimes \mathsf{id}); (\mathsf{id} \otimes g) \neq (\mathsf{id} \otimes g); (f \otimes \mathsf{id})$. To better understand this in terms of programming languages, a premonoidal category may also be viewed as a generalization of the Moggi's [23] monadic semantics: if we consider a strong monad over a Cartesian category, then the Kleisli category is premonoidal with the Cartesian product in the underlying category as tensor product.

The Kleisli category satisfies sliding when the underlying monad is *commutative*: when the sequencing of side-effects does not matter (e.g., the reader monad). We need to generalize monoidal categories to premonoidal categories precisely to support monads for which sequencing does matter, such as printing or state. For example, given a function $\mathsf{print} : \mathsf{str} \to ()$, $(\mathsf{print} \otimes \mathsf{id}); (\mathsf{id} \otimes \mathsf{print}) \neq (\mathsf{id} \otimes \mathsf{print}); (\mathsf{print} \otimes \mathsf{id})$ since, given input ("`hello`", "`world`"), the left-hand side will print `helloworld`, while the right-hand side will print `worldhello`. Note that the sequencing in the induced premonoidal category corresponds precisely to the bind of the underlying monad.

[4] Requiring $\mathcal{C}$ to be a poset-enriched category does not in fact rob us of any generality, since *all* categories are poset-enriched under the identity ordering on hom-sets.

We give a precise definition of a (*poset-enriched*) premonoidal category below:

Definition 4.2 (Symmetric Premonoidal Category). We define a *binoidal category* to be a category C equipped with a binary operation $\otimes : |C| \times |C| \to |C|$ on the objects of C and, for each $A, B \in |C|$, functors $A \otimes -, - \otimes B : C \to C$. We say a morphism $f : A \to A'$ in a binoidal category is *central* if, for all $g : B \to B'$, it satisfies *sliding*:

$$f \otimes B; A' \otimes g = A \otimes g; f \otimes B' \qquad B \otimes f; g \otimes A' = g \otimes A; B' \otimes f$$

in which case we may write these morphisms as $f \otimes g : A \otimes B \to A' \otimes B'$ and $g \otimes f : B \otimes A \to B' \otimes A'$ respectively. In the poset-enriched setting, we also assume that the left and right tensor functors are poset-enriched, i.e. monotonic on morphisms.

A *premonoidal category* is, then, a binoidal category equipped with:

- An *identity* object $I \in |C|$
- For each triple of objects $A, B, C \in |C|$, a central, natural isomorphism $\alpha_{A,B,C} : (A \otimes B) \otimes C \to A \otimes (B \otimes C)$, the *associator*
- For each object A, central, natural isomorphisms $\lambda_A : A \otimes I \to A$ and $\rho_A : I \otimes A \to A$, the *left* and *right unitors*

satisfying the *triangle* and *pentagon identity*

$$\alpha_{A,I,B}; A \otimes \lambda_B = \rho_A \otimes B \qquad \alpha_{A \otimes B,C,D}; \alpha_{A,B,C \otimes D} = \alpha_{A,B,C} \otimes D; \alpha_{A,B \otimes C,D}; A \otimes \alpha_{A,B,C}$$

We say a premonoidal category is *symmetric* if it is also equipped with a central, natural involution $\sigma_{A,B} : A \otimes B \to B \otimes A$, the *symmetry*, satisfying the *hexagon identity*

$$\alpha_{A,B,C}; \sigma_{A,B \otimes C}; \alpha_{B,C,A} = \sigma_{A,B} \otimes C; \alpha_{B,A,C}; B \otimes \sigma_{A,C}$$

One theorem of note about premonoidal categories is *coherence*, which we state as follows:

Theorem 4.3. *For any premonoidal category C, the subcategory $C_{\mathcal{A}}$ generated by associators, unitors, and their tensor products is an equivalence relation, i.e., for all $A, B : |C_{\mathcal{A}}|$, if $f, g : A \to B$ can be constructed using only identity, composition, associators, unitors, and their tensor products, then $f = g$, and f, g are isomorphisms in $C_{\mathcal{A}}$.*

We will hence sometimes abuse notation and write $\alpha_B : C(A, B)$ or simply α for the unique morphism in $C_{\mathcal{A}}$ from A to B, where the composition of associators and unitors is too cumbersome to write out in full. In particular, we note that we always have that $\alpha : C(A, A) := \mathsf{id}_A$.

We model branching control-flow using the coproduct $A + B$ as a primitive, whose definition is unchanged in the poset-enriched setting. Given morphisms $f : A \to C$ and $g : B \to C$, their coproduct $[f, g] : A + B \to C$ quite naturally models a case-statement which executes f given an A and executes g given a B. We can then implement an if-statement as a case-statement on $\mathbf{2} = I + I$.

Since coproducts induce a monoidal structure on a category, we will also write $\alpha^+_B : C(A, B)$ or simply α^+ to denote the unique morphism from A to B with analogy to the α notation described above. Similarly, we will write σ^+ to denote the symmetry for coproducts.

Coproducts on their own, however, cannot interpret variables captured by the branches of a case-statement. For example, given $x : \mathbb{Z}$, $y : \mathbb{Z} + \mathsf{str}$, consider the following expression:

$$\mathsf{case}\ y\ \{\iota_l\ y : \mathsf{print}(\texttt{"add: "}, x + y), \iota_r\ y : \mathsf{print}(y, x)\}$$

While our input context corresponds to the object $\mathbb{Z} \otimes (\mathbb{Z} + \mathsf{str})$, we need to somehow get to $(\mathbb{Z} \otimes \mathbb{Z}) + (\mathbb{Z} \otimes \mathsf{str})$ to be able to evaluate our branches. Categorically, what we require is that our tensor product *distributes* over our coproduct; in which case we say our category is *distributive*:

Definition 4.4 (Distributive Category). We say a premonoidal category is *distributive* if:

- It is equipped with chosen coproducts $A + B$ such that the injections ι_l, ι_r are central
- The obvious morphism $\delta : (A \otimes B) + (A \otimes C) \to A \otimes (B + C)$ is an isomorphism.

The last control-flow construct we need to model is looping. A loop with input A will either exit with output B or recurse with a new input A. Consequently, since we model branching control-flow using coproducts, the *body* of a loop will look like a morphism $A \to B + A$. Therefore, a natural way to model iteration is to posit the existence of a fixpoint operator $(\cdot)^\dagger$ taking morphisms $f : A \to B + A$ to their fixpoints $f^\dagger : A \to B$. $f^\dagger$ being the fixpoint of the body f means that executing $f^\dagger$ on an input A is the same as executing f on an input A and,

- If we get an output B, return it
- If we get an output A, feed it as an input to $f^\dagger$ and return the resulting B

Indeed, in a functional language supporting higher-order functions such as ML or Haskell, we might write `iterate :: (A -> B + A) -> A -> B` with definition

```
iterate f a = case f a of { Left b -> b ; Right a' -> iterate f a' }
```

This corresponds exactly to the notion of a *pre-iterative category* given below:

Definition 4.5 (Pre-iterative Category). Let $\mathcal{C}$ be a category with chosen coproducts. We say $\mathcal{C}$ is *pre-iterative* if it is equipped with a fixpoint operator $(-)^\dagger : \mathcal{C}(A, B + A) \to \mathcal{C}(A, B)$ satisfying the loop unrolling equation $f; [\mathsf{id}, f^\dagger] = f$

Our goal is to have our fixpoint operator's properties correspond precisely to drawing a loop in a control-flow graph, as in the left-hand side of Figure 11a, which corresponds to $f^\dagger$. In particular, we should be able to reconfigure such diagrams up to isotopy (i.e., moving boxes and wires around without changing connectivity) without changing the meaning of our program. To be able to do so soundly, we will need to introduce some additional equations, which correspond to the graphical transformations in Figure 11; a fixpoint satisfying these equations is called a *Conway iteration operator*, as defined below:

Definition 4.6 (Conway Iteration Operator). Given a pre-iterative category $\mathcal{C}$, we say $(-)^\dagger$ is a *Conway iteration operator* if it additionally satisfies

- *Naturality:* given $f : A \to B + A$ and $g : B \to C$, we have $(f; g + \mathsf{id})^\dagger = f^\dagger; g : A \to C$
- *Dinaturality:* given morphisms $g : A \to B + C$ and $h : C \to B + A$, we have that $(g; [\iota_l, h])^\dagger = g; [\mathsf{id}_B, (h; [\iota_l, g])^\dagger]$
- *Codiagonal:* given $f : A \to (B + A) + A$, we have $(f^\dagger)^\dagger = (f; [\mathsf{id}, \iota_r])^\dagger : A \to B$

In particular, we note that naturality and codiagonal correspond directly to our rules let-iter and codiag respectively; we will later see that dinaturality is derivable.

Just like for branching control-flow, we also require an additional condition to ensure that our iteration operator is compatible with our premonoidal structure. Specifically, we would like to be able to "thread" values through our loop bodies; i.e., the following two programs should be equivalent for *pure* c:

$$(\mathsf{iter}\ a\ \{\iota_r\ x : b\}, c) \approx \mathsf{iter}\ (a, c)\ \{\iota_r\ (x, y) : \mathsf{case}\ b\ \{\iota_l\ z : \iota_l\ (z, y), \iota_r\ z : \iota_r\ (z, y)\}\}$$

This corresponds to requiring our Conway iteration operator to be *strong*, defined as follows:

Definition 4.7 (Strong Conway Iteration Operator). If $\mathcal{C}$ is distributive, we say an iteration operator $(\cdot)^\dagger$ is *strong* if

$$\forall f : A \to B + A, (C \otimes f; \delta^{-1})^\dagger = C \otimes f^\dagger$$

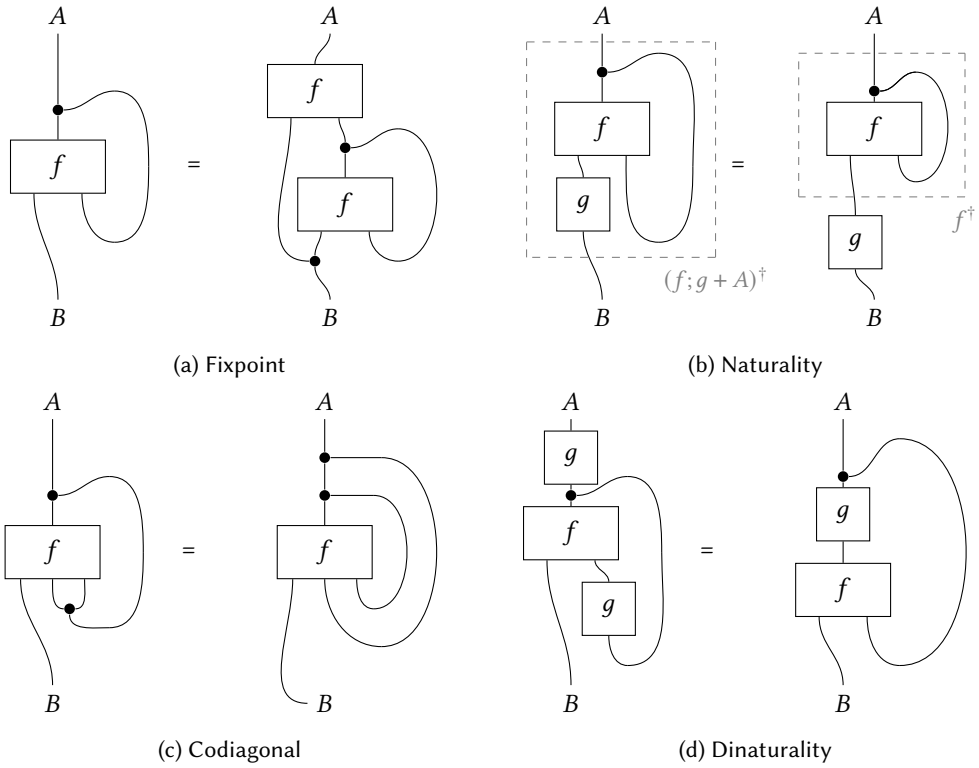

Fig. 11. Representations of the Conway iteration axioms as string diagrams

We've now got almost everything we need to model pure and arbitrarily effectful λ_{iter} programs, but we still need to be able to perform a more fine-grained classification of effects. To do so, we introduce a generalization of the notion of an *effectful category*, which in the literature (e.g. [25]) only distinguishes between pure and arbitrarily effectful morphisms.

Definition 4.8 (Effectful category). An effectful category C over an effect system $\mathcal{E}$ consists of a symmetric premonoidal poset-enriched category C equipped with a monotonic mapping from $\epsilon \in \mathcal{E}$ to wide[5] (symmetric premonoidal) subcategories $C_\epsilon \subseteq \mathcal{E}$ [6] such that, given $\epsilon, \eta \in \mathcal{E}$, $f \in C_\epsilon(A, B)$, and $g \in C_\eta(A', B')$, $\epsilon \rightharpoonup \eta \implies f \ltimes g \twoheadrightarrow f \rtimes g$ and $\epsilon \leftharpoonup \eta \implies f \ltimes g \twoheadleftarrow f \rtimes g$. We call morphisms with effect $\bot$ *pure*. We say an effectful category is *distributive* if the underlying premonoidal category is, and the injections are pure.

The goal of allowing morphisms with compatible effects to commute is to allow proving substitution of effectful programs sound. Unfortunately, we don't have quite enough structure to allow substitution *into* the body of a loop: while commutativity of a and b is enough to justify that let $x = a;\ (b, x) \approx (b, a)$ proving that let $x = a;$ iter $b\ \{\iota_r\ y : c\} \approx$ iter $b\ \{\iota_r\ y :$ let $x = a;\ c\}$ for $x \notin \text{fv}(b)$ requires us to be able to move a morphism *into* the body of a loop. To be able to do that effectively, we need to introduce the concept of a $\mathcal{K}$*-uniform iteration operator* as follows:

[5] A *wide* subcategory is one which has all of the objects of the original category.
[6] Note that $C_\epsilon(A, B)$ is *not* necessarily closed under refinement: we can have $f \in C_\epsilon(A, B)$ and $f \twoheadrightarrow f'$ with $f' \notin C_\epsilon(A, B)$.

Definition 4.9 (Uniformity). Given a wide subcategory $\mathcal{K} \subseteq \mathcal{C}$ of a category equipped with a Conway iteration operator, we say $\mathcal{C}$ is $\mathcal{K}^p$*-uniform* for $p \in \{+,-\}$ if, for all $h : A \to_{\mathcal{K}} B$, $f : B \to C + B$, and $g : A \to C + A$, we have that $h; f \twoheadrightarrow^p g; C + h \implies h; f^\dagger \twoheadrightarrow^p g^\dagger$. We say a category $\mathcal{K}$ is $\mathcal{K}$*-uniform* if it is both $\mathcal{K}^+$- and $\mathcal{K}^-$-uniform, which implies in particular that $h; f = g; C + h \implies h; f^\dagger = g^\dagger$.

We note that this definition corresponds precisely to the ability to perform the rewrites shown in Figure 10 (setting $c = \mathrm{id}_C$) whenever s is in $\mathcal{K}$ and b is in $\mathcal{C}$. Requiring that substitution is compatible with loops is then equivalent to requiring that the subcategories of morphisms having commutative effects are uniform with respect to each other. We call this notion an (effectful) Elgot category:

Definition 4.10 (Elgot category). We say a distributive effectful category $\mathcal{C}$ is *Elgot* if it has an iterative effect system and is equipped with a strong Conway iteration operator, such that, for all effects ϵ, η where $\epsilon \in \mathcal{E}^\infty$, the wide subcategory $\mathcal{C}_\epsilon$ is closed under iteration, and, iff $\epsilon \rightharpoonup^p \eta$, then $\mathcal{C}_\epsilon$ is $\mathcal{C}_\eta^p$-uniform. In particular, we note that $\mathcal{C}$ and hence every $\mathcal{C}_\epsilon$ is $\mathcal{C}_\bot$-uniform.

The final piece of the puzzle is that we need a way to *duplicate* variables of relevant type, as well as *discard* variables of affine type. We'll supply families of morphisms $\Delta : A \to A \otimes A$, $! : A \to I$ for this purpose. We can then handle relevant and affine *effects* by requiring that the appropriate morphism families are *natural* w.r.t. the subcategory corresponding to that effect. Following this idea, we can now define a λ_{iter}-model as follows: λ_{iter}:

Definition 4.11 (λ_{iter}-model). A model $\mathcal{M} = (\mathcal{C}, (\cdot)^\dagger, [\![\cdot]\!], \Delta, !)$ of a λ_{iter}-signature $\mathcal{S} = (\mathcal{X}, \mathcal{I}, \mathcal{E})$ is:

- An effectful Elgot category $(\mathcal{C}, (\cdot)^\dagger)$ over $(\mathcal{E}, \mathcal{E}^\infty)$
- For each base type $X \in \mathcal{X}$, an object $[\![X]\!] \in |\mathcal{C}|$, equipped with
  - For X affine, a *discard morphism* $!_X : \mathcal{C}_\bot([\![X]\!], I)$
  - For X relevant, a *diagonal morphism* $\Delta_X : \mathcal{C}_\bot([\![X]\!], [\![X]\!] \otimes [\![X]\!])$
- For each function $f : \mathsf{Inst}(\mathcal{S})_\epsilon(A, B)$, a morphism $[\![f]\!] : \mathcal{C}_\epsilon([\![A]\!], [\![B]\!])$

such that, for all $A, B \in |\mathcal{S}|$, $f : \mathcal{C}_\epsilon([\![A]\!], [\![B]\!])$ we have

- If A relevant, $\Delta_A; \Delta_A \otimes [\![A]\!]; \alpha = \Delta_A; [\![A]\!] \otimes \Delta_A$ and $\Delta_A; \sigma_{A,A} = \Delta_A$
- If $0 \le \mathrm{q}^p(\epsilon)$ and A, B affine, $f; !_B \twoheadrightarrow^p !_A$
- If $0 \le \mathrm{q}^p(\epsilon)$ and A relevant, B affine, $\Delta_A; (f; !_B) \otimes [\![A]\!] \twoheadrightarrow^p \rho^{-1}$
- If $\omega^+ \le \mathrm{q}^p(\epsilon)$ and A, B relevant, $f; \Delta_B \twoheadrightarrow^p \Delta_A; f \ltimes f = \Delta_A; f \rtimes f$

where

- $[\![\mathbf{1}]\!] = I$, $[\![A \otimes B]\!] = [\![A]\!] \otimes [\![B]\!]$, $!_{\mathbf{1}} = \mathrm{id}_I$, and $\Delta_{\mathbf{1}} = \lambda^{-1} = \rho^{-1}$
- For A, B affine, $!_{A\otimes B} = !_A \otimes !_B; \lambda$, and for A, B relevant, $\Delta_{A\otimes B} = \Delta_A \otimes \Delta_B; \sigma^{\mathrm{mid}}$ where we define $\sigma^{\mathrm{mid}}_{A,B,C,D} = \alpha_{A\otimes(B\otimes C)\otimes D}; A \otimes \sigma \otimes D; \alpha$ having type $\mathcal{C}_\bot((A \otimes B) \otimes (C \otimes D), (A \otimes C) \otimes (B \otimes D))$
- $[\![\mathbf{0}]\!] = \mathbf{0}$, $[\![A + B]\!] = [\![A]\!] + [\![B]\!]$, $!_{\mathbf{0}} = 0_I$, and $\Delta_{\mathbf{0}} = 0_{0+0}$
- For A, B affine, $!_{A+B} = [!_A, !_B]$, and for A, B relevant, $\Delta_{A+B} = [\Delta_A; \iota_l \otimes \iota_l, \Delta_B; \iota_r \otimes \iota_r]$

We define the *effective type* of an annotated type A^q, $[A^q]$ to be $\mathbf{1}$ if $q = 0$, and A otherwise. We can then proceed to define the effective type of an annotated context $\Gamma^{\mathbf{q}}$ to be the tensor product of its variables, i.e., $[\cdot] = \mathbf{1}$ and $[\Gamma, x : A^q] = [\Gamma] \otimes [A^q]$. We will abuse notation slightly and extend $[\![\cdot]\!]$ to quantity annotated types A^q by taking $[\![A^q]\!] = [\![[A^q]]\!]$; likewise, we define $!_{A^q} = !_{[A^q]}$ and $\Delta_{A^q} = \Delta_{[A^q]}$, where the *effective type* of an annotated type A^q, $[A^q]$, is $\mathbf{1}$ if $q = 0$, and A otherwise. Likewise, we proceed to define the semantics of an annotated context $[\![\Gamma^{\mathbf{q}}]\!] : |\mathcal{C}|$ as $[\![[\Gamma^{\mathbf{q}}]]\!]$.

We can now define the semantics of our structural judgements, weakenings and context splitting, in Figure 12. In particular, weakenings simply discard unused variables (which are guaranteed

$$\boxed{[\![\Gamma^{\mathbf{q}} \mapsto \Delta^{\mathbf{q}'}]\!] : C_\perp([\![\Gamma^{\mathbf{q}}]\!], [\![\Delta^{\mathbf{q}'}]\!])}$$

$$[\![\cdot \mapsto \cdot]\!] = \mathrm{id}_I \qquad [\![\Gamma^{\mathbf{q}}, x : A^q_\epsilon \mapsto \Delta^{\mathbf{q}'}]\!] = [\![\Gamma^{\mathbf{q}}]\!] \otimes !_{A^q}; \lambda; [\![\Gamma^{\mathbf{q}} \mapsto \Delta^{\mathbf{q}'}]\!]$$

$$[\![\Gamma^{\mathbf{q}}, x : A^q_\epsilon \mapsto \Delta^{\mathbf{q}'}, x : A^{q'}_{\epsilon'}]\!] = [\![\Gamma^{\mathbf{q}} \mapsto \Delta^{\mathbf{q}'}]\!] \otimes \begin{cases} \mathrm{id}_{[\![A]\!]} & \text{if } q, q' \neq 0 \\ !_{A^q} & \text{otherwise} \end{cases}$$

$$\boxed{[\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!] : C_\perp([\![\Gamma^{\mathbf{q}}]\!], [\![\Gamma^{\mathbf{q}_l}_{\mathbf{e}_l}]\!] \otimes [\![\Gamma^{\mathbf{q}_r}_{\mathbf{e}_r}]\!])}$$

$$[\![\cdot \vdash \cdot = \cdot + \cdot]\!] = \rho^{-1}$$

$$[\![\Gamma, x : A \vdash (\mathbf{q}, q) = (\mathbf{q}_l, q_l) + (\mathbf{q}_r, q_r)]\!] = [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!] \otimes \left(\begin{cases} \lambda^{-1} & \text{if } q_l = 0 \text{ else} \\ \rho^{-1} & \text{if } q_r = 0 \text{ else} \\ \Delta_A & \text{otherwise} \end{cases} \right); \sigma^{\mathrm{mid}}$$

Fig. 12. Denotational semantics for structural λ_{iter} judgements

to be of affine type), while context splitting duplicates variables used in both the left and right component contexts (which are guaranteed to be of relevant type).

4.2 Semantics of λ_{iter} Expressions

We now give the semantics of each of our term formers in Figure 13 by induction on derivations; we write $[\![\Gamma^{\mathbf{q}} \vdash_\epsilon a : A]\!]$ to denote the appropriate derivation/sub-derivation. Each of them corresponds quite closely to the the underlying categorical structure. In particular,

- We intepret a variable x of type A as the weakening $\Gamma^{\mathbf{q}} \mapsto x : A^1$; this discards all other variables from the input environment.
- Let-bindings and pairs are interpreted as sequencing, with, in the former case, the output of the first term being passed as an input to the second. In both cases, we use context-splitting to apportion the variables between the two terms.
- Units are interpreted as the weakening $\Gamma^{\mathbf{q}} \mapsto \cdot$, i.e., discarding all variables.
- Case statements are interpreted by passing the result of the discriminator into the coproduct of the branches, after context-splitting to apportion the variables between the discriminator and branches. We use the distributor to thread the variables into both branches of the coproduct, as discussed.
- Injections and abort are interpreted trivially as the injections and the zero morphism respectively, post-composed with their argument.
- To interpret iteration, after splitting the context, we interpret the initial value, and then feed that into the fixpoint of the body (computed using the Conway iteration operator) along with the remainder of the context.

We use the $[\![\Gamma^{\mathbf{q}} \vdash_\epsilon a : A]\!]$ notation, rather than specifying a particular derivation, because *coherence* holds:

Lemma 4.12 (Coherence). *Given derivations D for $\Gamma^{\mathbf{q}} \vdash_\epsilon a : A$ and D' for $\Gamma^{\mathbf{q}} \vdash_{\epsilon'} a : A$, we have $[\![D]\!] = [\![D']\!]$*

Observe that the coherence theorem allows us to omit the effect ϵ, because the effect is a property of the semantics: the same term with two different effect typings will have the same denotation.

Weakening. As a sanity check on our semantics, we can verify that it satisfies *weakening* by a straightforward induction, stated as follows:

$$\boxed{[\![\Gamma^{\mathbf{q}} \vdash_\epsilon a : A]\!] : C_\epsilon([\![\Gamma^{\mathbf{q}}]\!], [\![A]\!])}$$

$$[\![\Gamma^{\mathbf{q}} \vdash_\epsilon x : A]\!] = [\![\Gamma^{\mathbf{q}} \mapsto x : A^1_\epsilon]\!] \qquad [\![\Gamma^{\mathbf{q}} \vdash_\epsilon f\ a : B]\!] = [\![\Gamma^{\mathbf{q}} \vdash_\epsilon a : A]\!]; [\![f]\!]$$

$$[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \mathsf{let}\ x = a;\ b : B]\!] = [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; [\![\Gamma^{\mathbf{q}_l}]\!] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon a : A]\!]; [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon b : B]\!]$$

$$[\![\Gamma^{\mathbf{q}} \vdash_\epsilon (a, b) : A \otimes B]\!] = [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; [\![\Gamma^{\mathbf{q}_l} \vdash_\epsilon a : A]\!] \ltimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon b : B]\!] \qquad [\![\Gamma^{\mathbf{q}} \vdash_\epsilon () : \mathbf{1}]\!] = [\![\Gamma^{\mathbf{q}} \mapsto \cdot]\!]$$

$$[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \mathsf{let}\ (x, y) = a;\ c : C]\!] = [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; [\![\Gamma^{\mathbf{q}_l}]\!] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon a : A \otimes B]\!]; \alpha; [\![\Gamma^{\mathbf{q}_l}, x : A, y : B \vdash_\epsilon c : C]\!]$$

$$\begin{aligned}[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \mathsf{case}\ e\ \{\iota_l\ x : a, \iota_r\ y : b\} : C]\!] = {} & [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; [\![\Gamma^{\mathbf{q}_l}]\!] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon e : A + B]\!]; \delta^{-1} \\ & ; [[\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon a : C]\!], [\![\Gamma^{\mathbf{q}_l}, y : B \vdash_\epsilon b : C]\!]]\end{aligned}$$

$$[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \iota_l\ a : A + B]\!] = [\![\Gamma^{\mathbf{q}} \vdash_\epsilon a : A]\!]; \iota_l \qquad [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \iota_r\ b : A + B]\!] = [\![\Gamma^{\mathbf{q}} \vdash_\epsilon b : B]\!]; \iota_r$$

$$[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \mathsf{abort}\ a : A]\!] = [\![\Gamma^{\mathbf{q}} \vdash_\epsilon a : \mathbf{0}]\!]; 0_A$$

$$\begin{aligned}[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \mathsf{iter}\ a\ \{\iota_r\ x : b\} : B]\!] = {} & [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; [\![\Gamma^{\mathbf{q}_l}]\!] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon a : A]\!] \\ & ; ([\![\Gamma \vdash \mathbf{q}_l = \mathbf{q}_l + \mathbf{q}_l]\!] \otimes [\![A]\!]; \alpha; [\![\Gamma^{\mathbf{q}_l}]\!] \otimes [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon b : B + A]\!]; \delta^{-1})^\dagger \\ & ; [\![\Gamma^{\mathbf{q}_l} \mapsto \cdot]\!] \otimes [\![B]\!]; \rho\end{aligned}$$

Fig. 13. Denotational semantics for λ_{iter} expressions

$$\boxed{[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \sigma \rhd \Delta^{\mathbf{q}'}]\!] : C_\epsilon([\![\Gamma^{\mathbf{q}}]\!], [\![\Delta^{\mathbf{q}'}]\!])}$$

$$[\![\cdot \vdash_\epsilon \cdot \rhd \cdot]\!] = \mathsf{id}_I$$

$$[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \sigma, x \mapsto a \rhd \Delta^{\mathbf{q}'}, x : A^q]\!] = \begin{cases} [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; [\![\Gamma^{\mathbf{q}_l}_{e_l} \vdash_\epsilon \sigma \rhd \Delta^{\mathbf{q}'}]\!] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon a : A]\!] \text{ if } q \neq 0 \\ [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \sigma \rhd \Delta^{\mathbf{q}'}]\!]; \lambda^{-1} \text{ otherwise} \end{cases}$$

Fig. 14. Semantics of λ_{iter} substitutions

Lemma 4.13 (Weakening). *Given* $\Gamma^{\mathbf{q}} \mapsto \Delta^{\mathbf{q}'}$ *and* $\Delta^{\mathbf{q}'} \vdash_\epsilon a : A$, *we have that*

$$[\![\Gamma^{\mathbf{q}} \mapsto \Delta^{\mathbf{q}'}]\!]; [\![\Delta^{\mathbf{q}'} \vdash_\epsilon a : A]\!] = [\![\Gamma^{\mathbf{q}} \vdash_\epsilon a : A]\!]$$

Substitution. We proceed to give a semantics for substitution in Figure 14. We split up the input context into subcontexts for each *used* variable, which are then simply interpreted using their denotation. Unused variables are simply represented via the left unitor. We can then state soundness of substitution in the following manner, which is standard except that we prove an *inequality* whose direction is determined by the commutativity of the effects of the term and of the substitution.

Theorem 4.14 (Soundness of Substitution). *Given* $\Gamma^{\mathbf{q}} \vdash_\eta \sigma \rhd \Delta^{\mathbf{q}'}$ *and* $\Delta^{\mathbf{q}'} \vdash_\epsilon a : A$, *we have*

$$\eta \rightharpoonup \epsilon \implies [\![\Gamma^{\mathbf{q}} \vdash_\eta \sigma \rhd \Delta^{\mathbf{q}'}]\!]; [\![\Delta^{\mathbf{q}'} \vdash_\epsilon a : A]\!] \twoheadrightarrow [\![\Gamma^{\mathbf{q}} \vdash [\sigma]a : A]\!]$$

$$\eta \leftharpoonup \epsilon \implies [\![\Gamma^{\mathbf{q}} \vdash_\eta \sigma \rhd \Delta^{\mathbf{q}'}]\!]; [\![\Delta^{\mathbf{q}'} \vdash_\epsilon a : A]\!] \geq [\![\Gamma^{\mathbf{q}} \vdash [\sigma]a : A]\!]$$

Soundness. Now that we have given our terms a denotational semantics, we would like to show that our refinement theory is *sound* w.r.t. this semantics. In particular, as we allow a refinement theory to be parametrized by set of base refinements $\mathcal{R}$, we need to be able to express whether a model $\mathcal{M}$ satisfies these refinements. To do so, we introduce the notion of a model *validating* a typed refinement family as follows:

Definition 4.15. We say a model $\mathcal{M}$ *validates* a typed refinement family $\mathcal{R}$, written $\mathcal{M} \models \mathcal{R}$, if, for all $(\Gamma^{\mathbf{q}} \vdash a \twoheadrightarrow b : A) \in \mathcal{R}$ we have that $[\![\Gamma \vdash_\epsilon a : A]\!] \twoheadrightarrow [\![\Gamma \vdash_\epsilon b : A]\!]$

We note that, for every model $\mathcal{M}$, $\mathcal{M} \models \varnothing$. We can then phrase soundness as follows: if $\mathcal{M}$ models $\mathcal{R}$, for every refinement in the *theory* generated by $\mathcal{R}$, $\mathsf{Th}(\mathcal{R})$, $\mathcal{M}$ validates that refinement. Or, more formally,

Theorem 4.16 (Soundness). *We have that* $\mathcal{M} \models \mathcal{R} \iff \mathcal{M} \models \mathsf{Th}(\mathcal{R})$. *That is, given* $\mathcal{M} \models \mathcal{R}$ *and* $\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} a \twoheadrightarrow b : A$, *we have* $[\![\Gamma \vdash_\epsilon a : A]\!]_{\mathcal{M}} \twoheadrightarrow [\![\Gamma \vdash_\epsilon b : A]\!]_{\mathcal{M}}$.

In particular, we hence have that, for every model $\mathcal{M}$, $\mathcal{M} \models \mathsf{Th}(\varnothing)$, validating our equational theory.

Syntactic Models and Completeness. We now wish to show that our equational theory is *complete* with respect to our denotational semantics; that is, if some refinement holds for every $\mathcal{M} \models \mathcal{R}$, then this refinement is in fact contained in $\mathsf{Th}(\mathcal{R})$. We will do this by constructing an *initial* model $\mathsf{Tm}(\mathcal{R})$, the *syntactic model*, such that the following theorem holds:

Theorem 4.17 (Completeness). *If* $[\![\Gamma \vdash a : A]\!]_{\mathsf{Tm}(\mathcal{R})} \twoheadrightarrow [\![\Gamma \vdash b : A]\!]_{\mathsf{Tm}(\mathcal{R})}$, *then* $\Gamma \vdash_{\mathcal{R}} a \twoheadrightarrow b : A$

It then follows from soundness and the existence of $\mathsf{Tm}(\mathcal{R})$ that $\Gamma \vdash_{\mathcal{R}} a \twoheadrightarrow b : A$ *if and only if* for all models $\mathcal{M} \models \mathcal{R}$, $[\![\Gamma \vdash a : A]\!]_{\mathcal{M}} \twoheadrightarrow [\![\Gamma \vdash b : A]\!]_{\mathcal{M}}$, as desired.

We now proceed to give a sketch of the construction of $\mathsf{Tm}(\mathcal{R})$ and the proof of completeness; full details are given in Appendix B. As is standard, our syntactic model $\mathsf{Tm}(\mathcal{R})$ will have types as objects. To construct morphisms, we start with terms with a single free variable. We stratify these terms by type and effect as follows:

$$\mathsf{Term}(\mathcal{R})_\epsilon(A, B) := \{(x, a) \mid x : A^\top \vdash_\epsilon a : B\}$$

We will quotient each $\mathsf{Term}_\epsilon(\mathcal{R})$ by term equivalence $\approx_{\mathcal{R}}$, as well as by renaming the free variable x, to define $\mathsf{Tm}(\mathcal{R})_\epsilon$; we will somewhat suggestively write quotiented pairs (x, a) as lambda-expressions $\lambda x.a$, with $(\lambda x.a)(y) = [y/x]a$ yielding a term up to equivalence. The identity morphism is simply given as $\mathsf{id}_A = (\lambda x.x)$, while composition is given not by substitution (since terms may be impure!) but rather by let-bindings as follows:

$$(\lambda x.a); (\lambda y.b) := (\lambda x.\mathsf{let}\ y = a;\ b)$$

We can equip $\mathsf{Tm}(\mathcal{R})$ with the structure of a poset-enriched category by using the refinement relation as a partial order; it is trivial to see that this is well-defined. The rest of the structure of a λ_{iter}-model is given in Appendix B.1. To prove completeness, it then suffices to show that $[\![\cdot]\!]_{\mathsf{Tm}(\mathcal{R})}$ *reflects* refinement. The details of how to do so are given in Appendix B.2.

5 SSA Typing and Semantics

5.1 Typing Rules

We now turn back to the promises at the end of Section 2 and attempt to give typing rules and denotational semantics for λ_{SSA}. Recall that the primitive syntactic element of an λ_{SSA} program is a *region* r, which can be viewed as a program fragment with a single entry point and multiple exit points. Consequently, our primitive typing judgement will be of the form $\Gamma^{\mathbf{q}} \vdash_\epsilon r \triangleright \mathsf{L}^{\mathbf{Q}}$, which we will read as stating that "*if* the variables in Γ are live on entry, with quantity $\mathbf{q}$, *then* executing the region r jumps to one of the labels in L, with *leftover* quantities $\mathbf{Q}$." Here, L is a list of labels ℓ_i, annotated with a single parameter type (multiple parameters are implemented as a single tuple parameter), and $\mathbf{Q}$ is a list of quantity vectors $\mathbf{q}_i$ which we call the *quantity matrix*.

We may define a weakening judgement on annotated label contexts using the rules in Figure 16; the judgement is with respect to a particular context Γ used to interpret the quantity vectors in $\mathbf{Q}$. In particular, weakening allows us to insert arbitrary labels using skip, as well as weaken the quantities associated with each using cons. As we can see, a label context is interpreted w.r.t. a

$$\mathsf{L} ::= \cdot \mid \mathsf{L}, \ell(A) \qquad \mathbf{Q} ::= \cdot \mid \mathbf{Q}; \mathbf{q}$$

Fig. 15. Grammar for label contexts

$$\frac{}{\Gamma \vdash \cdot \rightsquigarrow \cdot}\ \text{nil} \qquad \frac{\Gamma \vdash \mathsf{L}^{\mathbf{Q}} \rightsquigarrow \mathsf{K}^{\mathbf{Q}} \quad \Gamma^{\mathbf{q}'} \mapsto \Gamma^{\mathbf{q}}}{\Gamma \vdash \mathsf{L}^{\mathbf{Q}}, \ell(A)^{\mathbf{q}} \rightsquigarrow \mathsf{K}^{\mathbf{Q}}, \ell(A)^{\mathbf{q}'}}\ \text{cons} \qquad \frac{\Gamma \vdash \mathsf{L}^{\mathbf{Q}} \rightsquigarrow \mathsf{K}^{\mathbf{Q}} \quad |\Gamma| = |\mathbf{q}|}{\Gamma \vdash \mathsf{L}^{\mathbf{Q}} \rightsquigarrow \mathsf{K}^{\mathbf{Q}}, \ell(A)^{\mathbf{q}}}\ \text{skip}$$

Fig. 16. Rules for weakening label contexts

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_r} \vdash_{\perp} o : A \quad \Gamma \vdash \ell(A)^{\mathbf{q}_l} \rightsquigarrow \mathsf{L}^{\mathbf{Q}}}{\Gamma^{\mathbf{q}} \vdash_{\epsilon} \mathsf{br}\ \ell\ o \rhd \mathsf{L}^{\mathbf{Q}}}\ \text{br}$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_r} \vdash_{\epsilon} o : A \quad \Gamma^{\mathbf{q}_l}, x : A \vdash_{\epsilon} t \rhd \mathsf{L}^{\mathbf{Q}^{\uparrow}}}{\Gamma^{\mathbf{q}} \vdash_{\epsilon} \mathsf{let}\ x = o; t \rhd \mathsf{L}^{\mathbf{Q}}}\ \text{let}_1$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_r} \vdash_{\epsilon} o : A \otimes B \quad \Gamma^{\mathbf{q}_l}, x : A, y : B \vdash_{\epsilon} t \rhd \mathsf{L}^{(\mathbf{Q}^{\uparrow})^{\uparrow}}}{\Gamma^{\mathbf{q}} \vdash_{\epsilon} \mathsf{let}\ (x, y) = o; t \rhd \mathsf{L}^{\mathbf{Q}}}\ \text{let}_2$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_r} \vdash_{\perp} o : A + B \quad \Gamma^{\mathbf{q}_l}, x : A \vdash_{\epsilon} \tau_l \rhd \mathsf{L}^{\mathbf{Q}^{\uparrow}} \quad \Gamma^{\mathbf{q}_l}, y : B \vdash_{\epsilon} \tau_r \rhd \mathsf{L}^{\mathbf{Q}^{\uparrow}}}{\Gamma^{\mathbf{q}} \vdash_{\epsilon} \mathsf{case}\ o\ \{\iota_l\ x : \tau_l, \iota_r\ y : \tau_r\} \rhd \mathsf{L}^{\mathbf{Q}}}\ \text{case}$$

$$\frac{\Gamma^{\mathbf{q}} \vdash_{\epsilon} \kappa \rhd \mathsf{L}^{\mathbf{Q}}, \mathsf{R}^{\mathbf{Q}'} \quad \forall \ell_i(A_i)^{\mathbf{q}_i} \in \mathsf{R}^{\mathbf{Q}'}. \Gamma^{\mathbf{q}_i}, x_i : A_i \vdash_{\epsilon} t_i \rhd \mathsf{L}^{\mathbf{Q}^{\uparrow}}}{\Gamma^{\mathbf{q}} \vdash_{\epsilon} \kappa\ \mathsf{where}_{\mathsf{nonrec}}\ (\ell_i(x_i) : \{t_i\},)_i \rhd \mathsf{L}^{\mathbf{Q}}}\ \text{where}_{\text{nonrec}}$$

$$\frac{\Gamma^{\mathbf{q}} \vdash_{\epsilon} \kappa \rhd \mathsf{L}^{\mathbf{Q}}, \mathsf{R}^{\mathbf{Q}'} \quad \epsilon \in \mathcal{E}^{\infty} \quad \forall \ell_i(A_i)^{\mathbf{q}_i} \in \mathsf{R}^{\mathbf{Q}'}. \Gamma^{\mathbf{q}_i}, x_i : A_i \vdash_{\epsilon} t_i \rhd \mathsf{L}^{\mathbf{Q}^{\uparrow}}, \mathsf{R}^{\mathbf{Q}'^{\uparrow}}}{\Gamma^{\mathbf{q}} \vdash_{\epsilon} \kappa\ \mathsf{where}_{\mathsf{rec}}\ (\ell_i(x_i) : \{t_i\},)_i \rhd \mathsf{L}^{\mathbf{Q}}}\ \text{where}_{\text{rec}}$$

Fig. 17. Typing rules for λ_{SSA}

quantity matrix $\mathbf{Q}$, which also depends on the set of live variables Γ. To extend the set of live variables (e.g. to type a let-binding), we need to zero-pad the quantity matrix to obtain its *lifting* $\mathbf{Q}^{\uparrow}$. In particular, we define this inductively with $\cdot^{\uparrow} = \cdot$ and $(\mathbf{Q}; \mathbf{q})^{\uparrow} = \mathbf{Q}^{\uparrow}; (\mathbf{q}, 0)$. With this, we give typing rules for λ_{SSA}, which we do in Figure 17.

- We begin with the typing rule for branches, br. This states that, if o is a pure expression of type A, and $\ell(A)^{\mathbf{q}_l}$ weakens to the label context $\mathsf{L}^{\mathbf{Q}}$, where $\mathbf{q}_l$ is the quantities left over after typing o, then $\mathsf{br}\ \ell\ o$ is a valid branch into $\mathsf{L}^{\mathbf{Q}}$
- Let-bindings are typed using let_1 and let_2, which are exactly the same as the corresponding rule for expressions, except that we target a label-context, which needs to be lifted (i.e. $\mathbf{Q}$ replaced with $\mathbf{Q}^{\uparrow}$) in the premise to deal with the additional variable in the input context.
- Case terminators are typed using case, which is again the same as the rule for expressions modulo lifting, except for the fact that the discriminator o is required to be pure.
- where-blocks are typed using $\mathsf{where}_{\mathsf{nonrec}}$ and $\mathsf{where}_{\mathsf{rec}}$, which we distinguish since the effect of a $\mathsf{where}_{\mathsf{rec}}$ subtree must be iterative. In particular, a $\mathsf{where}_{\mathsf{rec}}$ subtree is composed of an entry subtree κ, which we take to target the compound label-context $\mathsf{L}^{\mathbf{Q}}, \mathsf{R}^{\mathbf{Q}'}$, and, for each *sublabel* in $\ell_i(A_i)^{\mathbf{q}_i}$ in $\mathsf{R}^{\mathbf{Q}'}$, an associated *subregion* t_i which, with variables $\Gamma^{\mathbf{q}_i}$ plus an argument $x_i : A_i$ live on entry, targets the exit labels $\mathsf{L}^{\mathbf{Q}}$ or makes a recursive call to $\mathsf{R}^{\mathbf{Q}'}$. $\mathsf{where}_{\mathsf{nonrec}}$ simply removes the requirement that ϵ be iterative, in exchange requiring the t_i to only jump to exit labels in $\mathsf{L}^{\mathbf{Q}}$.

$$\boxed{[\![\Gamma \vdash \mathsf{L}^{\mathbf{Q}} \rightsquigarrow \mathsf{K}^{\mathbf{Q}'}]\!] : C_\perp([\![[\Gamma \mapsto \mathsf{L}^{\mathbf{Q}}]]\!], [\![[\Gamma \mapsto \mathsf{K}^{\mathbf{Q}'}]]\!])}$$

$$[\![\Gamma \vdash \cdot \rightsquigarrow \cdot]\!] = \mathsf{id}_{\mathbf{0}} \qquad [\![\Gamma \vdash \mathsf{L}^{\mathbf{Q}}, \ell(A)^{\mathbf{q}} \rightsquigarrow \mathsf{K}^{\mathbf{Q}'}, \ell(A)^{\mathbf{q}'}]\!] = [\![\Gamma \vdash \mathsf{L}^{\mathbf{Q}} \rightsquigarrow \mathsf{K}^{\mathbf{Q}'}]\!] + [\![\Gamma^{\mathbf{q}} \mapsto \Gamma^{\mathbf{q}'}]\!] \otimes [\![A]\!]$$

$$[\![\Gamma \vdash \mathsf{L}^{\mathbf{Q}} \rightsquigarrow \mathsf{K}^{\mathbf{Q}'}, \ell(A)^{\mathbf{q}}]\!] = [\![\Gamma \vdash \mathsf{L}^{\mathbf{Q}} \rightsquigarrow \mathsf{K}^{\mathbf{Q}'}]\!]; \iota_l$$

Fig. 18. Denotational semantics for label contexts and label weakenings

$$\boxed{[\![\Gamma \vdash_\epsilon t \rhd \mathsf{L}^{\mathbf{Q}}]\!] : C_\epsilon([\![\Gamma^{\mathbf{q}}]\!], [\Gamma \mapsto [\![\mathsf{L}^{\mathbf{Q}}]\!]])}$$

$$[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \mathsf{br}\ \ell\ a \rhd \mathsf{L}^{\mathbf{Q}}]\!] = [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; - \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\perp a : A]\!]; \iota_r; [\![\Gamma \vdash \ell(A)^{\mathbf{q}_l} \rightsquigarrow \mathsf{L}^{\mathbf{Q}}]\!]$$

$$[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \mathsf{let}\ x = o; t \rhd \mathsf{L}^{\mathbf{Q}}]\!] = [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; - \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon o : A]\!]; [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon t : \mathsf{L}^{\mathbf{Q}^\uparrow}]\!]; \alpha^\downarrow$$

$$[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \mathsf{let}\ (x, y) = o; t \rhd \mathsf{L}^{\mathbf{Q}}]\!] = [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; - \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon o : A \otimes B]\!]; \alpha$$
$$; [\![\Gamma^{\mathbf{q}_l}, x : A, y : B \vdash_\epsilon t : \mathsf{L}^{(\mathbf{Q}^\uparrow)^\uparrow}]\!]; \alpha^\downarrow; \alpha^\downarrow$$

$$[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \mathsf{case}\ o\ \{\iota_l\ x : \tau_l, \iota_r\ y : \tau_r\} \rhd \mathsf{L}^{\mathbf{Q}}]\!] = [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; - \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\perp o : A + B]\!]; \delta^{-1}$$
$$; [[\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon \tau_l : \mathsf{L}^{\mathbf{Q}^\uparrow}]\!]; \alpha^\downarrow, [\![\Gamma^{\mathbf{q}_l}, y : B \vdash_\epsilon \tau_r : \mathsf{L}^{\mathbf{Q}^\uparrow}]\!]; \alpha^\downarrow]$$

$$[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \kappa\ \mathsf{where}_{\mathsf{nonrec}}\ (\ell_i(x_i) : \{t_i\},)_i \rhd \mathsf{L}^{\mathbf{Q}}]\!] = [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \kappa \rhd \mathsf{L}^{\mathbf{Q}}, \mathsf{R}^{\mathbf{Q}'}]\!]; \alpha^+$$
$$; [\mathsf{id}_{[\![[\Gamma \mapsto \mathsf{L}^{\mathbf{Q}}]]\!]}, [[\![\Gamma^{\mathbf{q}_i}, x_i : A_i \vdash_\epsilon t_i : \mathsf{L}^{\mathbf{Q}^\uparrow}]\!]; \alpha^\downarrow,]_{\ell_i(A_i)^{\mathbf{q}_i} \in \mathsf{R}^{\mathbf{Q}'}}]$$

$$[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \kappa\ \mathsf{where}_{\mathsf{rec}}\ (\ell_i(x_i) : \{t_i\},)_i \rhd \mathsf{L}^{\mathbf{Q}}]\!] = [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \kappa \rhd \mathsf{L}^{\mathbf{Q}}, \mathsf{R}^{\mathbf{Q}'}]\!]; \alpha^+$$
$$; [\mathsf{id}_{[\![[\Gamma \mapsto \mathsf{L}^{\mathbf{Q}}]]\!]}, [[\![\Gamma^{\mathbf{q}_i}, x_i : A_i \vdash_\epsilon t_i : \mathsf{L}^{\mathbf{Q}^\uparrow}, \mathsf{R}^{\mathbf{Q}'^\uparrow}]\!]; \alpha^\downarrow; \alpha^+,]^\dagger_{\ell_i(A_i)^{\mathbf{q}_i} \in \mathsf{R}^{\mathbf{Q}'}}]$$

Fig. 19. Denotational semantics for λ_{SSA}

5.2 Denotational Semantics

We give a denotational semantics for λ_{SSA}, targeting an arbitrary λ_{iter} model. We interpret a derivation $\Gamma^{\mathbf{q}} \vdash_\epsilon r \rhd \mathsf{L}^{\mathbf{Q}}$ as a morphism from the input state, the set of live variables $[\![\Gamma^{\mathbf{q}}]\!]$, to the output state, which consists of a label ℓ_i in L, its argument A_i, and leftover variables $\mathbf{q}_i$ in $\mathbf{Q}$.

To represent this as a type, we can define the (context-dependent) *effective type* $[\Gamma \mapsto \mathsf{L}]$ of an annotated label context with $[\Gamma \mapsto \cdot] = \mathbf{0}$ and $[\Gamma \mapsto \mathsf{L}^{\mathbf{Q}}, \ell(A)^{\mathbf{q}}] = [\Gamma \mapsto \mathsf{L}^{\mathbf{Q}}] + [\Gamma^{\mathbf{q}}] \otimes A$. Here, the label is encoded as the branch of the coproduct we end up in, with each branch carrying the argument and any leftover variables as data. We may now give a denotational semantics for label contexts and label weakenings in Figure 18. It is easy to verify that we can reassociate $\alpha^+ : C_\perp([\![\mathsf{L}^{\mathbf{Q}}, \mathsf{R}^{\mathbf{Q}'}]\!](\Gamma), [\![\mathsf{L}^{\mathbf{Q}}]\!](\Gamma) + [\![\mathsf{R}^{\mathbf{Q}'}]\!](\Gamma))$. We also note that we can apply the associator "pointwise" to reassociate $\alpha^\downarrow : C_\perp([\![[\Gamma, x : A \mapsto \mathsf{L}^{\mathbf{Q}^\uparrow}]]\!], [\![[\Gamma \mapsto \mathsf{L}^{\mathbf{Q}}]]\!])$. The semantics for regions is in Figure 19:

- Branches are interpreted by splitting the context, evaluating the argument, and then passing the remainder of the context and the result into the appropriate label weakening.
- The denotation of let- and case-statements is exactly the same as for expressions, except that we need to re-associate the output object from $[\![\mathsf{L}^{\mathbf{Q}^\uparrow}]\!](\Gamma, -)$ to $[\![\mathsf{L}^{\mathbf{Q}}]\!]$.
- Non-recursive where-subtrees are interpreted by the denotation of their entry subtree, reassociated to the sum of the exit labels $\mathsf{L}^{\mathbf{Q}}$ and sublabels $\mathsf{R}^{\mathbf{Q}'}$. The exit labels are passed through as-is, and the sublabels ℓ_i are piped to the appropriate subregion t_i.
- Recursive where-subtree is as above, except that we take the *fixpoint* of the sum of the denotations of the subregions viewed as morphisms from $[\![\mathsf{R}^{\mathbf{Q}}]\!](\Gamma)$ to $[\![\mathsf{L}^{\mathbf{Q}}]\!](\Gamma) + [\![\mathsf{R}^{\mathbf{Q}}]\!](\Gamma)$, feeding recursive calls back into the where-block's body.

	∞	$\lightning$	?	$\infty\lightning$	∞?	$\lightning$?	$\infty\lightning$?	q^+	q^-
$\mathsf{UB}_\infty(A,B) \approx A \to B \cup \{\infty\}$	$\rightleftharpoons$	$\leftharpoondown$	$\rightleftharpoons$	$\leftharpoondown$	$\rightleftharpoons$	$\leftharpoondown$	$\leftharpoondown$	ω^+	ω^+
$\mathsf{UB}_\lightning(A,B) \approx A \to B \cup \{\lightning\}$	$\rightharpoonup$	$\rightleftharpoons$	$\rightleftharpoons$	$\rightharpoonup$	$\rightharpoonup$	$\rightleftharpoons$	$\rightharpoonup$	ω	ω^+
$\mathsf{UB}_?(A,B) \approx A \to \mathcal{P}^+(B)$	$\rightleftharpoons$	$\rightleftharpoons$	$\rightleftharpoons$	$\rightleftharpoons$	$\rightleftharpoons$	$\rightleftharpoons$	$\rightleftharpoons$	$1^?$	ω
$\mathsf{UB}_{\infty\lightning}(A,B) \approx A \to B \cup \{\infty\} \cup \{\lightning\}$	$\rightharpoonup$	$\leftharpoondown$	$\rightleftharpoons$	$\cdot$	$\rightharpoonup$	$\leftharpoondown$	$\cdot$	ω^+	ω^+
$\mathsf{UB}_{\infty?}(A,B) \approx A \to \mathcal{P}^+(B \cup \{\infty\})$	$\rightleftharpoons$	$\leftharpoondown$	$\rightleftharpoons$	$\leftharpoondown$	$\rightleftharpoons$	$\leftharpoondown$	$\leftharpoondown$	1	ω^+
$\mathsf{UB}_{\lightning?}(A,B) \approx A \to \mathcal{P}^+(B) \cup \{\lightning\}$	$\rightharpoonup$	$\rightleftharpoons$	$\rightleftharpoons$	$\rightharpoonup$	$\rightharpoonup$	$\rightleftharpoons$	$\rightharpoonup$	1	ω^+

Fig. 20. Submonads of UB, along with their commutativity information (i.e. whether they are a left or right mover w.r.t. other effects) and linearity information (given as positive and negative quantities $\mathsf{q}^+, \mathsf{q}^- \in \mathsf{Q}$)

5.3 Interconversion with λ_{iter}

When we say that λ_{SSA} and λ_{iter} are *equivalent*, what we mean is that there are *semantics-preserving* functions SSA, Expr on derivations satisfying

$$[\![\Gamma^{\mathsf{q}} \vdash_\epsilon \mathsf{SSA}_{ell}(\Gamma^{\mathsf{q}} \vdash_\epsilon a : A) \rhd \ell(A)^0]\!]_{\mathcal{M}} = [\![\Gamma^{\mathsf{q}} \vdash_\epsilon a : A]\!]_{\mathcal{M}}; \alpha; \iota_r$$
$$[\![\Gamma^{\mathsf{q}} \vdash_\epsilon \mathsf{Expr}(\Gamma^{\mathsf{q}} \vdash_\epsilon r \rhd \mathsf{L}^{\mathsf{Q}}) : [\Gamma \mapsto \mathsf{L}^{\mathsf{Q}}]]\!]_{\mathcal{M}} = [\![\Gamma^{\mathsf{q}} \vdash_\epsilon r \rhd \mathsf{L}^{\mathsf{Q}}]\!]_{\mathcal{M}}$$

for arbitrary models $\mathcal{M}$. It is easy enough to construct Expr: for each derivation, we simply pick a representative $(x, a) \in [\![\Gamma^{\mathsf{q}} \vdash_\epsilon r \rhd \mathsf{L}^{\mathsf{Q}}]\!]_{\mathsf{Th}(\cdot)}$. Since $[\![\Gamma^{\mathsf{q}}]\!]_{\mathsf{Th}(\cdot)} = [\Gamma^{\mathsf{q}}]$, we may simply define $\mathsf{Expr}(\Gamma^{\mathsf{q}} \vdash_\epsilon r \rhd \mathsf{L}^{\mathsf{Q}}) := \mathsf{let}\ \Gamma = x;\ a$ where the unpacking of a value $c : [\Gamma^{\mathsf{q}}]$ is defined in the obvious recursive manner (see Appendix B.2 for details). On the other hand, the function SSA is just standard expression compilation presented inductively (the details are in Appendix C).

6 Concrete Models

In this section, we sketch some concrete models. All the given models have types with unrestricted linearity, and so we will write our typing rules using *un-annotated* contexts and variables $\Gamma, x : A$, assuming every quantity annotation is ω.

6.1 UB

The simplest model exercising all the features of our refinement calculus is the language with nondeterminism, undefined behavior and nontermination. We represent this using the monad $\mathsf{UB}\ A = \mathcal{P}^+(A \cup \{\infty\}) \cup \{\lightning\}$ over sets, with ∞ a sentinel value for nontermination and $\lightning$ for UB. Its ordering is the inclusion order on nonempty sets $\mathcal{P}^+(A \cup \{\infty\})$ extended with a top element $\lightning$. (The full description of this model is in Appendix D.2.) The monadic bind is:

$$\lightning \gg\!=_{\mathsf{UB}} f = \lightning \qquad X \gg\!=_{\mathsf{UB}} f = \{f\ a \mid a \in X\} \ \text{ if } \forall a \in X, f\ a \neq \lightning \qquad X \gg\!=_{\mathsf{UB}} f = \lightning \ \text{ otherwise}$$

where we define $f\ \infty = \infty$. The Kleisli category of UB has the lattice of effects generated by the nontermination effect ∞, the UB effect $\lightning$, and the nondeterminism effect ?. In Figure 20, we give the submonad for each effect, along with its linearity and commutativity with other effects.

This model lets us interpret instructions $\lightning_A : \mathbf{1} \to_\lightning A$ analogous to LLVM's `unreachable` intrinsic. We write $\lightning$ as shorthand for $\lightning_A\ ()$. This instruction lets us define *unsafe unwrap* operators $\rho_l : A + B \to_\lightning A$ and $\rho_r : A + B \to_\lightning B$ which take the other branch to $\lightning$. These satisfy the usual rewrite rules for UB, some of which are given in Figure 21.

6.2 Heaps

We model a single-threaded heap as a finitely-supported function of type $S := \mathbb{N} \to_{\mathsf{fin}} \mathbb{N}$. Using the state transformer on UB, we can define a heap monad $\mathsf{Heap}\ A = S \to \mathsf{UB}\ (A \times S)$, with the natural pointwise partial order. (The full description of this model is in Appendix D.3.) Our effect system is

$$\frac{\Gamma, x : A \vdash_\epsilon b : B}{\Gamma \vdash \mathsf{let}\ x = \lightning;\ b \approx \lightning : B}\ \text{ub-then} \qquad \frac{\Gamma \vdash_\epsilon a : A + B \quad \Gamma, x : A \vdash_\epsilon c : C}{\Gamma \vdash \mathsf{case}\ a\ \{\iota_l\ x : c, \iota_r\ y : \lightning\} \approx \mathsf{let}\ x = \rho_l\ a;\ c : C}\ \text{ub-left}$$

$$\frac{\Gamma \vdash_\epsilon a : A + B \quad \Gamma, y : B \vdash_\epsilon c : C}{\Gamma \vdash \mathsf{case}\ a\ \{\iota_l\ x : \lightning, \iota_r\ y : c\} \approx \mathsf{let}\ y = \rho_r\ a;\ c : C}\ \text{ub-right}$$

Fig. 21. Some of the usual rewrite rules for undefined behavior.

$$\frac{\Gamma \vdash_\perp p : \mathbb{N} \quad \Gamma \vdash_\perp a : \mathbb{N}}{\Gamma \vdash \mathsf{st}\ (p, a); \mathsf{ld}\ p \approx \mathsf{st}\ (p, a); a : \mathbb{N}}\ \text{st-ld} \qquad \frac{\Gamma \vdash_\perp a : \mathbb{N}}{\Gamma \vdash \mathsf{let}\ p = \mathsf{mk}\ a;\ (p, \mathsf{ld}\ p) \approx (\mathsf{mk}\ a; a) : \mathbb{N} \otimes \mathbb{N}}\ \text{mk-ld}$$

$$\frac{\Gamma \vdash_\perp p : \mathbb{N} \quad \Gamma \vdash_\perp a : \mathbb{N} \quad \Gamma \vdash_\perp b : \mathbb{N}}{\Gamma \vdash \mathsf{st}\ (p, a); \mathsf{st}\ (p, b) \approx \mathsf{st}\ (p, b) : \mathbf{1}}\ \text{st-st} \qquad \frac{\Gamma \vdash_\perp a : \mathbb{N} \quad \Gamma \vdash_\perp b : \mathbb{N}}{\Gamma \vdash \mathsf{let}\ p = \mathsf{mk}\ a;\ \mathsf{st}\ (p, b); p \approx \mathsf{mk\ b} : \mathbb{N}}\ \text{mk-st}$$

$$\frac{\Gamma \vdash_\perp a : \mathbb{N}}{\Gamma \vdash \mathsf{let}\ p = \mathsf{mk}\ a;\ \mathsf{rm}\ p \approx () : \mathbf{1}}\ \text{mk-rm} \qquad \frac{\Gamma \vdash_\perp p : \mathbb{N} \quad \Gamma \vdash_\perp a : \mathbb{N}}{\Gamma \vdash \mathsf{st}\ (p, a); \mathsf{rm}\ p \approx \mathsf{rm}\ p : \mathbf{1}}\ \text{st-rm}$$

Fig. 22. Some properties of heaps

the bounded lattice generated by $\{\infty, \lightning, ?, \mathsf{r}, \mathsf{w}, \mathsf{a}\}$, with r, w, and a corresponding to "read", "write", and "allocate" respectively. The new primitive operations are:

- $\mathsf{ld} : \mathsf{Heap}_{\mathsf{r}\lightning}(\mathbb{N}, \mathbb{N}) := \lambda p\ s.\mathsf{if}\ p \in s\ \mathsf{then}\ \mathsf{pure}_{\mathsf{ND}}(s\ p, s)\ \mathsf{else}\ \lightning$
- $\mathsf{st} : \mathsf{Heap}_{\mathsf{w}\lightning}(\mathbb{N} \otimes \mathbb{N}, \mathbf{1}) := \lambda(p, x)\ s.\mathsf{if}\ p \in s\ \mathsf{then}\ \mathsf{pure}_{\mathsf{ND}}((), [p \mapsto x]s)\ \mathsf{else}\ \lightning$
- $\mathsf{mk} : \mathsf{Heap}_{\mathsf{a}?}(\mathbb{N}, \mathbb{N}) := \lambda x\ s.\{(p, [p \mapsto x]s) \mid p \notin s\}$
- $\mathsf{rm} : \mathsf{Heap}_{\mathsf{w}\lightning}(\mathbb{N}, ()) := \lambda p.\mathsf{if}\ p \in s\ \mathsf{then}\ s \setminus p\ \mathsf{else}\ \lightning$

This model validates the usual properties of heaps (see Figure 22), as well as that allocation is central, reads and writes are relevant, and reads are eliminable and commutative.

6.3 Concurrency

In Brookes's [5] concurrency semantics, a memory effect is a trace consisting of a finite sequence of transitions $\langle \mu, \rho \rangle$ between memory states μ and ρ. Traces represent *interrupted* program executions: a transition $\langle \mu, \rho \rangle$ represents a single step in which the program, *relying on* a state μ from the environment, *guarantees* it will return a state ρ to the environment. A program's denotation is now a *set* of pairs $\xi \therefore r$ of a trace ξ and return value r. Each pair corresponds to a possible execution of the program given potential interference (e.g., an external thread reading and writing).

Brookes [5] observed that to make a denotational model sound with respect to the operational semantics, the traces need to satisfy certain *closure properties*. Trace sets should be closed under "*stuttering*" (adding transitions of the form $\langle \mu, \mu \rangle$, corresponding to the program doing nothing) and "*mumbling*" (merging $\langle \mu, \rho \rangle \langle \rho, \theta \rangle \twoheadrightarrow \langle \mu, \theta \rangle$, corresponding to the environment doing nothing).

By choosing appropriate state types, closure properties, and operations on states, we can model a variety of memory models. In particular, Dvir et al. [7] show how to define a monad $\mathcal{I}\ A = \mathcal{P}(\{\xi \therefore r \mid r \in A\})$, which they use to guve a denotational semantics for release-acquire weak memory. This monad has a poset-enriched Kleisli category under the usual subset order, and can be equipped with an Elgot structure, which we show how to do in Appendix D.4, hence giving us a λ_{iter}-model.

Just like for heaps, we can define an effect lattice $\{\mathsf{r}, \mathsf{w}, \infty\}$, though for trace models reading is *not* commutative; similarly, we have instructions $\mathsf{ld} : \mathbb{N} \to_{\mathsf{r}} \mathbb{N}$ and $\mathsf{st} : \mathbb{N} \otimes \mathbb{N} \to_{\mathsf{w}} \mathbf{1}$, as well as an atomic *fetch-and-add* instruction $\mathsf{faa} : \mathbb{N} \otimes \mathbb{N} \to_{\mathsf{rw}} \mathbb{N}$. We may validate numerous refinements on memory operations in this model, some of which we give in Figure 23. Note that all of these are

$$\frac{\Gamma \vdash_\bot p : \mathbb{N} \quad \Gamma \vdash_\bot a : \mathbb{N}}{\Gamma \vdash \mathsf{st}\ (p,a); \mathsf{ld}\ p \twoheadrightarrow \mathsf{st}\ (p,a); a : \mathbb{N}}\ \text{st-ld} \qquad \frac{\Gamma \vdash_\bot p : \mathbb{N} \quad \Gamma \vdash_\bot a : \mathbb{N} \quad \Gamma \vdash_\bot b : \mathbb{N}}{\Gamma \vdash \mathsf{st}\ (p,a); \mathsf{st}\ (p,b) \twoheadrightarrow \mathsf{st}\ (p,b) : \mathbf{1}}\ \text{st-st}$$

$$\frac{\Gamma \vdash_\bot p : \mathbb{N} \quad \Gamma \vdash_\bot a : \mathbb{N}}{\Gamma \vdash \mathsf{let}\ x = \mathsf{ld}\ p;\ \mathsf{st}\ p\ (x + a); x \twoheadrightarrow \mathsf{faa}\ (p,a) : \mathbb{N}}\ \text{ld-st}$$

Fig. 23. Some refinements valid in the release-acquire λ_{iter}-model.

valid *equalities* in the heap model given above. Both the read and write effects are duplicable in this model, while the read effect is not only eliminable, but also *introducable*, since we do not have UB.

7 Discussion and Related Work

From SSA to FP SSA was introduced as a compiler IR by Alpern et al. [2] and Rosen et al. [26], with the goal of simplifying reasoning about variable values. In three address code, a variable is mutable and can have a different value at each program point, whereas in SSA, each variable has a single abstract value, a fact which greatly simplifies optimizations like global value numbering. Kelsey [16] demonstrated a correspondence between SSA and a fragment of continuation-passing style, and Appel [3] simplified this further and showed that an SSA program can be seen as a group of procedures mutually tail-calling each other. When the dominance tree is further made explicit in the syntax, we recover lexical scoping, and Chakravarty et al. [6] used this to show how to translate SSA programs into ANF. Ghalayini and Krishnaswami [10] relaxed the ANF restriction to permit compound (pure) expressions, which gives a calculus with better substitution principles, but which still has explicit block definitions. In this work, we take the final step and give a fully expression-oriented syntax, which is still completely equivalent to traditional SSA.

Semantics of SSA Zhao et al. [28] is a mechanized semantics for LLVM's SSA dialect. Their syntax closely follows LLVM's, and so their operational semantics must compute ϕ-nodes at each step. Furthermore, they give several such semantics and prove refinements between them. Each semantics is further parameterized over a memory model, and so execution builds a tree of possible executions, resumption-style, which can then be made into concrete execution traces with an "effect handler" or free monad interpreter computing the effect of each memory action. Garbuzov et al. [9] exhibit a correspondence between an operational semantics for both SSA and a fragment of call-by-push-value [20], and then use the normal form bisimulations of Lassen and Levy [17] to derive an equational theory for justifying optimizations. This work considers nontermination as the only effect, and studies equivalence rather than refinement. Ghalayini and Krishnaswami [10] give a denotational semantics similar to this paper's, but their model is not in enriched categories and hence does not model refinement, nor does it model an effect system or linear types.

Effect Systems and Linearity Effect systems were introduced by Gifford and Lucassen [11], which introduced a type system with a lattice of effects to track which effects a program could potentially perform, with the idea of gaining reasoning principles by restricting effects. Führmann [8] introduced the idea of categorizing effects in terms of linearity (i.e., whether effects can be moved, duplicated or dropped), which reverses the conceptual priority: effects are classified not by their intension, but by which equations they validate. Kammar and Plotkin [15] build on this idea, and give a Gifford-style type-and-effect system for a variant of call-by-push-value, an effect-dependent equational, and conditions on the semantics monad to ensure each equation holds. We build on this work by integrating the idea of linearity of effects with an old idea of Lipton [22]. By simply classifying effects by whether they can commute to the left or right of another effect, we gain a very rich (in)equational theory very cheaply.

References

[1] [n. d.]. clang introduces undefined behavior into well-formed C++ program with infinite loop. https://github.com/llvm/llvm-project/issues/62057 Accessed: 11 July 2025.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. 1988. Detecting Equality of Variables in Programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/73560.73561

[3] Andrew W. Appel. 1998. SSA is Functional Programming. *ACM SIGPLAN Notices* 33, 4 (1998), 17–20. https://doi.org/10.1145/278283.278285

[4] Mark Batty. 2017. Compositional relaxed concurrency. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375, 2104 (2017), 20150406. https://doi.org/10.1098/rsta.2015.0406 arXiv:https://royalsocietypublishing.org/doi/pdf/10.1098/rsta.2015.0406

[5] Stephen D. Brookes. 1996. Full Abstraction for a Shared-Variable Parallel Language. *Inf. Comput.* 127, 2 (1996), 145–163. https://doi.org/10.1006/INCO.1996.0056

[6] Manuel M. T. Chakravarty, Gabriele Keller, and Patryk Zadarnowski. 2003. A Functional Perspective on SSA Optimisation Algorithms. In *Compiler Optimization Meets Compiler Verification, COCV@ETAPS 2003, Warsaw, Poland, April 12, 2003 (Electronic Notes in Theoretical Computer Science, Vol. 82)*, Jens Knoop and Wolf Zimmermann (Eds.). Elsevier, Warsaw, Poland, 347–361. https://doi.org/10.1016/S1571-0661(05)82596-4

[7] Yotam Dvir, Ohad Kammar, and Ori Lahav. 2024. A Denotational Approach to Release/Acquire Concurrency. In *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 14577)*, Stephanie Weirich (Ed.). Springer, Luxembourg City, Luxembourg, 121–149. https://doi.org/10.1007/978-3-031-57267-8_5

[8] Carsten Führmann. 1999. Direct Models for the Computational Lambda Calculus. In *Fifteenth Conference on Mathematical Foundations of Progamming Semantics, MFPS 1999, Tulane University, New Orleans, LA, USA, April 28 - May 1, 1999 (Electronic Notes in Theoretical Computer Science, Vol. 20)*, Stephen D. Brookes, Achim Jung, Michael W. Mislove, and Andre Scedrov (Eds.). Elsevier, USA, 245–292. https://doi.org/10.1016/S1571-0661(04)80078-1

[9] Dmitri Garbuzov, William Mansky, Christine Rizkallah, and Steve Zdancewic. 2018. Structural Operational Semantics for Control Flow Graph Machines. *CoRR* abs/1805.05400 (2018), 27 pages. arXiv:1805.05400 http://arxiv.org/abs/1805.05400

[10] Jad Elkhaleq Ghalayini and Neel Krishnaswami. 2024. The Denotational Semantics of SSA. *arXiv e-prints*, Article arXiv:2411.09347 (Nov. 2024), arXiv:2411.09347 pages. https://doi.org/10.48550/arXiv.2411.09347 arXiv:2411.09347 [cs.PL]

[11] David K. Gifford and John M. Lucassen. 1986. Integrating Functional and Imperative Programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming, LFP 1986, Cambridge, Massachusetts, USA, August 4-6, 1986*, William L. Scherlis, John H. Williams, and Richard P. Gabriel (Eds.). ACM, 28–38. https://doi.org/10.1145/319838.319848

[12] Sergey Goncharov, Stefan Milius, and Christoph Rauch. 2016. Complete Elgot Monads and Coalgebraic Resumptions. In *The Thirty-second Conference on the Mathematical Foundations of Programming Semantics, MFPS 2016, Carnegie Mellon University, Pittsburgh, PA, USA, May 23-26, 2016 (Electronic Notes in Theoretical Computer Science, Vol. 325)*, Lars Birkedal (Ed.). Elsevier, Amsterdam, The Netherlands, 147–168. https://doi.org/10.1016/J.ENTCS.2016.09.036

[13] Sergey Goncharov and Lutz Schröder. 2018. Guarded Traced Categories. In *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10803)*, Christel Baier and Ugo Dal Lago (Eds.). Springer, Thessaloniki, Greece, 313–330. https://doi.org/10.1007/978-3-319-89366-2_17

[14] Radha Jagadeesan, Gustavo Petri, and James Riely. 2012. Brookes Is Relaxed, Almost!. In *Foundations of Software Science and Computational Structures - 15th International Conference, FOSSACS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7213)*, Lars Birkedal (Ed.). Springer, Tallinn, Estonia, 180–194. https://doi.org/10.1007/978-3-642-28729-9_12

[15] Ohad Kammar and Gordon D. Plotkin. 2012. Algebraic foundations for effect-dependent optimisations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, 349–360. https://doi.org/10.1145/2103656.2103698

[16] Richard Kelsey. 1995. A Correspondence between Continuation Passing Style and Static Single Assignment Form. In *Proceedings ACM SIGPLAN Workshop on Intermediate Representations (IR'95), San Francisco, CA, USA, January 22, 1995*, Michael D. Ernst (Ed.). ACM, USA, 13–23. https://doi.org/10.1145/202529.202532

[17] Søren B. Lassen and Paul Blain Levy. 2007. Typed Normal Form Bisimulation. In *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4646)*, Jacques Duparc and Thomas A. Henzinger (Eds.). Springer, Lausanne, Switzerland, 283–297. https://doi.org/10.1007/978-3-540-74915-8_23

[18] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75.

[19] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, Piscataway, NJ, USA, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[20] Paul Blain Levy. 2004. *Call-By-Push-Value: A Functional/Imperative Synthesis*. Semantics Structures in Computation, Vol. 2. Springer, USA.

[21] Paul Blain Levy and Sergey Goncharov. 2019. Coinductive Resumption Monads: Guarded Iterative and Guarded Elgot. In *8th Conference on Algebra and Coalgebra in Computer Science, CALCO 2019, June 3-6, 2019, London, United Kingdom (LIPIcs, Vol. 139)*, Markus Roggenbach and Ana Sokolova (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, London, UK, 13:1–13:17. https://doi.org/10.4230/LIPICS.CALCO.2019.13

[22] Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (1975), 717–721. https://doi.org/10.1145/361227.361234

[23] Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. https://doi.org/10.1016/0890-5401(91)90052-4

[24] John Power and Edmund Robinson. 1997. Premonoidal Categories and Notions of Computation. *Math. Struct. Comput. Sci.* 7, 5 (1997), 453–468. https://doi.org/10.1017/S0960129597002375

[25] Mario Román. 2023. Promonads and String Diagrams for Effectful Categories. *Electronic Proceedings in Theoretical Computer Science* 380 (aug 2023), 344–361. https://doi.org/10.4204/eptcs.380.20

[26] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 12–27. https://doi.org/10.1145/73560.73562

[27] Cranelift Development Team. 2023. Cranelift: A Simple, Fast WebAssembly Code Generator. https://github.com/bytecodealliance/wasmtime/tree/main/cranelift

[28] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, Philadelphia, Pennsylvania, 427–440. https://doi.org/10.1145/2103656.2103709

A Refinement Rules and Notation

We begin by giving the congruence rules for λ_{iter} in Figure 24, which completes our presentation of λ_{iter}'s type theory. We now wish to go over some basic derivable refinement rules, which we will make use of throughout the rest of the appendix. We begin with some useful lemmas and notational conventions:

- $\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \iff \Gamma \vdash \mathbf{q} = \mathbf{q}_r + \mathbf{q}_l$
- We will write Γ^0 to mean the variable context Γ with every variable having the zero quantity 0. In particular, we note that $\Gamma \vdash \mathbf{q} = 0 + \mathbf{q}$ and $\Gamma \vdash \mathbf{q} = 0 + \mathbf{q}$ are always derivable.

Recall that we define $a; b := \text{let } x = a;\ b$ for $x \notin \text{fv}(b)$. The following typing rule is hence trivially derivable:

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_l} \vdash_\epsilon a : A \quad 0 \le \text{q}(A) \quad \Gamma^{\mathbf{q}_r} \vdash_\epsilon b : B}{\Gamma^{\mathbf{q}} \vdash_\epsilon a; b : B}\ \text{seq}$$

We can easily convince ourselves that this satisfies some of the basic properties of sequencing; for example, we have

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_c \quad \Gamma \vdash \mathbf{q}_c = \mathbf{q}_r + \mathbf{q}_m \quad \Gamma^{\mathbf{q}_l} \vdash_\epsilon a : A \quad \Gamma^{\mathbf{q}_m} \vdash_\epsilon b : B \quad 0 \le \text{q}(A), \text{q}(B) \quad \Gamma^{\mathbf{q}_r} \vdash_\epsilon c : C}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} (a; b); c \approx a; (b; c) : C}$$

As stated in Section 3.3, binding rules for the rest of our calculus are derivable from the rest of λ_{iter}'s refinement rules. We give these explicitly in Figure 25, along with the η-rule for unary let-bindings, which is similarly derived from let_1-β. We note the commutativity requirement for pair-right-bind, since on the left-hand side a executes before b, while on the right-hand side b executes before a, hence the requirement that their effects commute. We can also derive a simplified rule for uniformity, given below, by simply choosing $q_l = q_c$ and $c = z$ in unif^p:

$$\frac{\eta \rightharpoonup \epsilon \quad \Gamma^{\mathbf{q}_l}, x : A \vdash_{\mathcal{R}} \text{let } y = s;\ b \twoheadrightarrow^p \text{case } b' \ \{\iota_l\ x : \iota_l\ x, \iota_r\ x : \iota_r\ s\} : B + S}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \text{let } x = a;\ \text{iter } s\ \{\iota_r\ y : b\} \twoheadrightarrow^p \text{iter } a\ \{\iota_r\ x : b'\} : B}\ \text{simp-unif}^p$$

$$\text{where} \qquad \Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \qquad \text{q}(\Gamma^{\mathbf{q}_l}) = \top \qquad \epsilon \in \mathcal{E}^\infty \qquad \Gamma^{\mathbf{q}_r} \vdash_\epsilon a : A$$
$$\Gamma^{\mathbf{q}_l}, x : A \vdash_\eta s : S \qquad \Gamma^{\mathbf{q}_l}, y : S \vdash_\epsilon b : B + A \qquad \Gamma^{\mathbf{q}_l}, x : A \vdash b' : B + A$$

We define *pattern binding* $\text{let } P = a;\ b$ of patterns $P ::= x \mid (P, P')$ inductively as follows (with bindings of variables x and pairs (x, y) defined as normally):

$$\text{let } (P, y) = a;\ b := \text{let } (x, y) = a;\ \text{let } P = x;\ b \qquad \text{for } P \notin \text{Var}, x \notin \text{fv}(b)$$
$$\text{let } (x, P) = a;\ b := \text{let } (x, y) = a;\ \text{let } P = y;\ b \qquad \text{for } P \notin \text{Var}, y \notin \text{fv}(b)$$
$$\text{let } (P, P') = a;\ b := \text{let } (x, y) = a;\ \text{let } P = x;\ \text{let } P' = y;\ b \qquad \text{for } P, P' \notin \text{Var}$$

By a straightforward case analysis, we may show that the following rule is admissible:

$$\frac{\Gamma^{\mathbf{q}} \vdash_\epsilon \text{let } (P, P') = x;\ a : B}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \text{let } (P, P') = a;\ b \approx \text{let } (x, y) = a;\ \text{let } P = x;\ \text{let } P' = y;\ b : B}\ \text{pattern-pair'}$$

We extend pattern binding to other binding forms, for $P \notin \text{Var}$, in the obvious manner:

$$\text{case } a\ \{\iota_l\ P : b, \iota_r\ y : c\} := \text{case } a\ \{\iota_l\ x : \text{let } P = x;\ b, \iota_r\ y : c\} \qquad \text{for } x \notin \text{fv}(b)$$
$$\text{case } a\ \{\iota_l\ x : b, \iota_r\ P : c\} := \text{case } a\ \{\iota_l\ x : b, \iota_r\ y : \text{let } P = y;\ c\} \qquad \text{for } y \notin \text{fv}(c)$$
$$\text{iter } a\ \{\iota_r\ P : b\} := \text{iter } a\ \{\iota_r\ x : \text{let } P = x;\ b\} \qquad \text{for } x \notin \text{fv}(b)$$

$$\frac{\Gamma^{\mathbf{q}} \mapsto x : A^{q}_{\epsilon} \quad 1 \le q}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} x \twoheadrightarrow x : A}\ \text{var} \qquad \frac{f : A \to_{\epsilon} B \quad \Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} a \twoheadrightarrow a' : A}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} f\ a \twoheadrightarrow f\ a' : B}\ \text{op}$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_r} \vdash_{\mathcal{R}} a \twoheadrightarrow a' : A \quad \Gamma^{\mathbf{q}_l}, x : A \vdash_{\mathcal{R}} b \twoheadrightarrow b' : B}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{let}\ x = a;\ b \twoheadrightarrow \mathsf{let}\ x = a';\ b' : B}\ \text{let}_1$$

$$\frac{\Gamma^{\mathbf{q}} \mapsto \cdot}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} () \twoheadrightarrow () : \mathbf{1}}\ \text{unit} \qquad \frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_l} \vdash_{\mathcal{R}} a \twoheadrightarrow a' : A \quad \Gamma^{\mathbf{q}_r} \vdash_{\mathcal{R}} b \twoheadrightarrow b' : B}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} (a, b) \twoheadrightarrow (a', b') : A \otimes B}\ \text{pair}$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_r} \vdash_{\mathcal{R}} a \twoheadrightarrow a' : A \otimes B \quad \Gamma^{\mathbf{q}_l}, x : A, y : B \vdash_{\mathcal{R}} c \twoheadrightarrow c' : C}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{let}\ (x, y) = a;\ c \twoheadrightarrow \mathsf{let}\ (x, y) = a';\ c' : C}\ \text{let}_2$$

$$\frac{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} a \twoheadrightarrow a' : A}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \iota_l\ a \twoheadrightarrow \iota_l\ a' : A + B}\ \text{inl} \qquad \frac{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} b \twoheadrightarrow b' : B}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \iota_r\ b \twoheadrightarrow \iota_r\ b' : A + B}\ \text{inr}$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_r} \vdash_{\mathcal{R}} e \twoheadrightarrow e' : A + B \quad \Gamma^{\mathbf{q}_l}, x : A \vdash_{\mathcal{R}} a \twoheadrightarrow a' : C \quad \Gamma^{\mathbf{q}_l}, y : B \vdash_{\mathcal{R}} b \twoheadrightarrow b' : C}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{case}\ e\ \{\iota_l\ x : a, \iota_r\ y : b\} \twoheadrightarrow \mathsf{case}\ e'\ \{\iota_l\ x : a', \iota_r\ y : b'\} : C}\ \text{case}$$

$$\frac{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} a \twoheadrightarrow a' : \mathbf{0}}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{abort}\ a \twoheadrightarrow \mathsf{abort}\ a' : C}\ \text{abort}$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \mathsf{q}(\Gamma^{\mathbf{q}_l}) = \top \quad \Gamma^{\mathbf{q}_r} \vdash_{\mathcal{R}} a \twoheadrightarrow a' : A \quad \Gamma^{\mathbf{q}_l}, x : A \vdash_{\mathcal{R}} b \twoheadrightarrow b' : B + A}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{iter}\ a\ \{\iota_r\ x : b\} \twoheadrightarrow \mathsf{iter}\ a'\ \{\iota_r\ x : b'\} : B}\ \text{iter}$$

Fig. 24. λ_{iter} congruence rules

$$\frac{f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash_{\mathcal{R}} f\ a \approx \mathsf{let}\ x = a;\ f\ x : B}\ \text{op-bind}$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_l} \vdash a : A \quad \Gamma^{\mathbf{q}_r} \vdash b : B}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} (a, b) \approx \mathsf{let}\ x = a;\ \mathsf{let}\ y = b;\ (x, y) : A \otimes B}\ \text{pair-bind}$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_l} \vdash a : A \quad \Gamma^{\mathbf{q}_r} \vdash b : B}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} (a, b) \approx \mathsf{let}\ x = a;\ (x, b) : A \otimes B}\ \text{pair-left-bind}$$

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathbf{q}_l} \vdash_{\epsilon} a : A \quad \Gamma^{\mathbf{q}_r} \vdash_{\eta} b : B \quad \epsilon \rightleftharpoons \eta}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} (a, b) \approx \mathsf{let}\ y = b;\ (a, y) : A \otimes B}\ \text{pair-right-bind}$$

$$\frac{\Gamma^{\mathbf{q}} \vdash a : A}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \iota_l\ a \approx \mathsf{let}\ x = a;\ \iota_l\ x : A + B}\ \text{inl-bind} \qquad \frac{\Gamma^{\mathbf{q}} \vdash b : B}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \iota_r\ b \approx \mathsf{let}\ y = b;\ \iota_r\ y : A + B}\ \text{inr-bind}$$

$$\frac{\Gamma^{\mathbf{q}} \vdash a : \mathbf{0}}{\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{abort}\ a \approx \mathsf{let}\ x = a;\ \mathsf{abort}\ x : C}\ \text{abort-bind}$$

Fig. 25. Derivable binding rules for λ_{iter}

B Completeness

In this section, we give the details of the proof of completeness of our refinement calculus w.r.t. our denotational semantics.

B.1 Syntactic Model

We begin by defining the syntactic category $\mathsf{Tm}(\mathcal{R})$ of λ_{iter}-models of a signature $\mathcal{S}$ quotiented by a set of primitive rewrites $\mathcal{R}$ as follows:

- Objects types $A, B, C \in \mathsf{Ty}(\mathcal{X})$

- Morphisms $\mathsf{Tm}_\epsilon(A,B) := \{(x,a) \mid x : A \vdash_\epsilon a : B\}/\approx$, with quotiented pairs written as λ-expressions $\lambda x.a$, where $(\lambda x.a) \approx (\lambda x.b)$ if and only if $x : A \vdash_{\mathcal{R}} a \approx [x/y]b : B$. Equivalently, morphisms in $\mathsf{Tm}_\epsilon(A,B)$ may be viewed as *functions* $f : \mathsf{Var} \to \mathsf{Term}$, with $(x,a)(y) := [y/x]a$, satisfying the property that, for all $x, y \in \mathsf{Var}$, $x : A \vdash_\epsilon f(x) : B$ $f(y) = [y/x]f(x)$, quotiented up to equivalence $f \approx g \iff x : A \vdash_{\mathcal{R}} f(x) \approx g(x) : B$. The equivalence between these two representations is given by the η-law $f \approx \lambda x.f(x)$.
- Refinement on morphisms $f \twoheadrightarrow g$ if and only if $x : A \vdash_{\mathcal{R}} f(x) \twoheadrightarrow g(x) : B$ (this is well-defined since, by definition, we quotient precisely the equivalence clases of $\twoheadrightarrow_{\mathcal{R}}$)
- Identity morphisms $\mathsf{id}_A := (\lambda x.x)$
- Composition $(\lambda x.a); (\lambda y.b) := (\lambda x.\mathsf{let}\ y = a;\ b)$. We note that, for *a pure*, this reduces to the usual substitution-based definition $(\lambda x.a); (\lambda y.b) = (\lambda x.[a/y]b)$.
- Tensor products $A \otimes B$ with monoidal unit $\mathbf{1}$, and tensor functors

  $$(\lambda x.a) \otimes A := (\lambda z.\mathsf{let}\ (x,y) = z;\ (a,y)) \qquad A \otimes (z,b) := (\lambda x.\mathsf{let}\ (y,z) = x;\ (y,b)))$$

  In general, we extend pattern binding to λ-expressions, writing (for $x \notin \mathsf{fv}(a)$),

  $$(\lambda P.a) := (\lambda x.\mathsf{let}\ P = x;\ a)$$

  in which case we may rewrite the above as

  $$(\lambda x.a) \otimes A := (\lambda(x,y).(a,y)) \qquad A \otimes (\lambda z.b) := (\lambda(y,z).(y,b)))$$

  In partiuclar, we hence have that

$$(\lambda x.a) \ltimes (\lambda y.b) = (\lambda(x,y).(a,b)) \qquad (\lambda x.a) \rtimes (\lambda y.b) = (\lambda(x,y).\mathsf{let}\ y' = b;\ \mathsf{let}\ x' = a;\ (x',y'))$$

- Associators, unitors, and symmetries

$$\alpha := (\lambda((x,y),z).(x,(y,z))) \qquad \alpha^{-1} := (\lambda(x,(y,z)).((x,y),z)) \qquad \sigma := (\lambda(x,y).(y,x))$$
$$\lambda := (\lambda(x,y).x) \qquad \lambda^{-1} := (\lambda x.(x,())) \qquad \rho := (\lambda(x,y).y) \qquad \rho^{-1} := (\lambda y.((),y))$$

- Coproducts $A + B$ with initial object $\mathbf{0}$, zero morphisms $0 := (x, \mathsf{abort}\ x)$, and injections and distributors

$$\iota_l := (\lambda x.\iota_l\ x) \qquad \iota_r := (\lambda x.\iota_r\ x) \qquad [(\lambda x.a),(\lambda y.b)] := (\lambda z.\mathsf{case}\ z\ \{\iota_l\ x : a, \iota_r\ y : b\})$$
$$\delta := [- \otimes \iota_l, - \otimes \iota_r] = (\lambda x.\mathsf{case}\ x\ \{\iota_l\ (y_l,y_r) : (y_l, \iota_l\ y_r), \iota_r\ (z_l,z_r) : (z_l, \iota_r\ z_r)\})$$
$$\delta^{-1} := (\lambda(x_l,x_r).\mathsf{case}\ x_r\ \{\iota_l\ y : \iota_l\ (x_l,y), \iota_r\ z : \iota_r\ (x_r,z)\})$$

  In particular, this means that we have

$$(\lambda x.a); [(\lambda y.b),(\lambda z.c)] = (\lambda x.\mathsf{case}\ a\ \{\iota_l\ y : b, \iota_r\ z : c\})$$
$$(\lambda x.a) + (\lambda y.b) = (\lambda z.\mathsf{case}\ z\ \{\iota_l\ x : \iota_l\ a, \iota_r\ y : \iota_r\ b\})$$
$$(\lambda x.a); (\lambda y.b) + (\lambda z.c) = (\lambda x.\mathsf{case}\ a\ \{\iota_l\ y : \iota_l\ b, \iota_r\ z : \iota_r\ c\})$$

- Iteration operator $(\lambda x.a)^\dagger := (\lambda y.\mathsf{iter}\ y\ \{\iota_r\ x : a\})$
- Discard and diagonal morphisms $!_A := (\lambda x.())$ and $\Delta_A := (\lambda x.(x,x))$

We aim to prove the following lemma:

Lemma B.1. $\mathsf{Tm}(\mathcal{R})$ *is a* λ_{iter}*-model.*

Proof. We begin by noting that

$$(\lambda x.a); (\lambda P.b) = (\lambda x.\mathsf{let}\ P = a;\ b)$$

In particular, this allows us to deduce the following useful identities:

$$(\lambda x.a); \alpha = (\lambda x.\text{let } ((y,z),w) = a;\ (y,(z,w)))$$
$$(\lambda x.a); \alpha^{-1} = (\lambda x.\text{let } (y,(z,w)) = a;\ ((y,z),w))$$
$$(\lambda x.a); \sigma = (\lambda x.\text{let } (y,z) = a;\ (z,y))$$
$$(\lambda x.a); \lambda = (\lambda x.\text{let } (y,z) = a;\ y) \qquad (\lambda x.a); \lambda^{-1} = (\lambda x.(a,()))$$
$$(\lambda x.a); \rho = (\lambda x.\text{let } (y,z) = a;\ z) \qquad (\lambda x.a); \rho^{-1} = (\lambda x.((),a))$$

We also note that

$$(\lambda P.a) \otimes A = (\lambda(x,y).(\text{let } P = x;\ a,y)) = (\lambda(x,y).\text{let } P = x;\ (a,y)) = (\lambda(P,y).(a,y))$$
$$A \otimes (\lambda P.b) = (\lambda(x,y).(x,\text{let } P = y;\ b)) = (\lambda(x,y).\text{let } P = y;\ (x,b)) = (\lambda(x,P).(x,b))$$

We now verify that:

- $\mathsf{Tm}_\epsilon(\mathcal{R})$ is a category: we have that

$$(\lambda x.x); (\lambda y.a) = (\lambda x.\text{let } y = x;\ a) = (\lambda x.[x/y]a) = (\lambda y.a)$$
$$(\lambda x.a); (\lambda y.y) = (\lambda x.\text{let } y = a;\ y) = (\lambda x.a)$$
$$\begin{aligned}((\lambda x.a); (\lambda y.b)); (\lambda z.c) &= (\lambda x.\text{let } z = (\text{let } y = a;\ b);\ c) = (\lambda x.\text{let } y = a;\ \text{let } z = b;\ c)\\ &= (\lambda x.a); (\lambda y.\text{let } z = b;\ c) = (\lambda x.a); ((\lambda y.b); (\lambda z.c))\end{aligned}$$

- $\mathsf{Tm}_\epsilon(\mathcal{R})$ is binoidal: we have that

$$(\lambda x.x) \otimes A = (\lambda z.\text{let } (x,y) = z;\ (x,y)) = (\lambda z.z)$$
$$\begin{aligned}((\lambda x.a); (\lambda y.b)) \otimes A &= (\lambda(x,z).(\text{let } y = a;\ b,z)) = (\lambda(x,z).\text{let } y = a;\ (b,z))\\ &= (\lambda(x,z).\text{let } (y,z') = (a,z);\ (b,z'))\\ &= (\lambda(x,z).(a,z)); (\lambda(y,z').(b,z'))\\ &= ((\lambda x.a) \otimes A); ((\lambda y.b) \otimes A)\end{aligned}$$
$$A \otimes (\lambda y.y) = (\lambda z.\text{let } (x,y) = z;\ (x,y)) = (\lambda z.z)$$
$$\begin{aligned}A \otimes ((\lambda y.a); (\lambda z.b)) &= (\lambda(x,y).(x,\text{let } z = a;\ b)) = (\lambda(x,y).\text{let } z = a;\ (x,b))\\ &= (\lambda(x,y).\text{let } (x',z) = (x,a);\ (x',b))\\ &= (\lambda(x,y).(x,a)); (\lambda(x',z).(x',b))\\ &= (A \otimes (\lambda y.a)); (A \otimes (\lambda z.b))\end{aligned}$$

- $\mathsf{Tm}_\epsilon(\mathcal{R})$ is symmetric premonoidal:

– Pentagon equation: we have that

$$
\begin{aligned}
&\alpha_{(A\otimes B),C,D};\alpha_{A,B,(C\otimes D)}\\
&= (\lambda((x_{12},x_3),x_4).(x_{12},(x_3,x_4)));(\lambda((x_1,x_2),x_{34}).(x_1,(x_2,x_{34})))\\
&= (\lambda((x_{12},x_3),x_4).\mathsf{let}\ ((x_1,x_2),x_{34}) = (x_{12},(x_3,x_4));\ (x_1,(x_2,x_{34})))\\
&= (\lambda((x_{12},x_3),x_4).\mathsf{let}\ (x_1,x_2) = x_{12};\ (x_1,(x_2,(x_3,x_4))))\\
&= (\lambda(((x_1,x_2),x_3),x_4).(x_1,(x_2,(x_3,x_4))))\\
&= (\lambda(((x_1,x_2),x_3),x_4).\mathsf{let}\ (z_1,((z_2,z_3),z_4)) = (x_1,((x_2,x_3),x_4));\ (z_1,(z_2,(z_3,z_4))))\\
&= (\lambda(((x_1,x_2),x_3),x_4).(x_1,((x_2,x_3),x_4)));(\lambda(z_1,((z_2,z_3),z_4)).(z_1,(z_2,(z_3,z_4))))\\
&= (\lambda(((x_1,x_2),x_3),x_4).\mathsf{let}\ ((y_1,y_{23}),y_4) = ((x_1,(x_2,x_3)),x_4);\ (y_1,(y_{23},y_4)));\\
&\qquad (\lambda(z_1,((z_2,z_3),z_4)).(z_1,(z_2,(z_3,z_4))))\\
&= (\lambda(((x_1,x_2),x_3),x_4).((x_1,(x_2,x_3)),x_4));(\lambda((y_1,y_{23}),y_4).(y_1,(y_{23},y_4)));\\
&\qquad (\lambda(z_1,((z_2,z_3),z_4)).(z_1,(z_2,(z_3,z_4))))\\
&= \alpha_{A,B,C}\otimes D;\alpha_{A,B\otimes C,D};A\otimes\alpha_{B,C,D}
\end{aligned}
$$

– Triangle equation: we have that

$$
\begin{aligned}
\alpha_{A,\mathbf{1},B};X\otimes\lambda_B &= (\lambda((x,y),z).(x,(y,z)));(\lambda(x',y').y')\\
&= (\lambda((x,y),z).\mathsf{let}\ (x',y') = (x,(y,z));\ y')\\
&= (\lambda((x,y),z).(y,z)) = (\lambda(x,y).y)\otimes B\\
&= \rho_A\otimes B
\end{aligned}
$$

– Hexagon equation: we have that

$$
\begin{aligned}
&\alpha_{A,B,C};\sigma_{A,B\otimes C};\alpha_{B,C,A}\\
&= (\lambda((x_1,x_2),x_3).(x_1,(x_2,x_3)));(\lambda(y_1,y_{23}).(y_{23},y_1));(\lambda((z_2,z_3),z_1).(z_2,(z_3,z_1)))\\
&= (\lambda((x_1,x_2),x_3).((x_2,x_3),x_1));(\lambda((z_2,z_3),z_1).(z_2,(z_3,z_1)))\\
&= (\lambda((x_1,x_2),x_3).(x_2,(x_3,x_1)))\\
&= (\lambda((x_1,x_2),x_3).(x_2,(x_1,x_3)));(\lambda(z_2,(z_1,z_3)).(z_2,(z_3,z_1)))\\
&= (\lambda((x_1,x_2),x_3).((x_2,x_1),x_3));(\lambda((y_2,y_1),y_3).(y_2,(y_1,y_3)));(\lambda(z_2,(z_1,z_3)).(z_2,(z_3,z_1)))\\
&= \sigma_{A,B}\otimes C;\alpha_{B,A,C};B\otimes\sigma_{A,C}
\end{aligned}
$$

- $\mathsf{Tm}_\epsilon(\mathcal{R})$ has chosen coproducts and an initial object: see formalization.
- $\mathsf{Tm}_\epsilon(\mathcal{R})$ is distributive: we have that

$$
\begin{aligned}
\delta_{X,Y,Z};\delta^{-1}_{X,Y,Z} &= [X\otimes\iota_l, X\otimes\iota_r];(\lambda(x,w).\mathsf{case}\ w\ \{\iota_l\ y:\iota_l\ (x,y),\iota_r\ z:\iota_r\ (x,z)\})\\
&= [(\lambda(x,y).(x,\iota_l\ y));(\lambda(x,w).\mathsf{case}\ w\ \{\iota_l\ y:\iota_l\ (x,y),\iota_r\ z:\iota_r\ (x,z)\}),\\
&\qquad (\lambda(x,z).(x,\iota_r\ z));(\lambda(x,w).\mathsf{case}\ w\ \{\iota_l\ y:\iota_l\ (x,y),\iota_r\ z:\iota_r\ (x,z)\})]\\
&= [(\lambda(x,y).\mathsf{case}\ \iota_l\ y\ \{\iota_l\ y:\iota_l\ (x,y),\iota_r\ z:\iota_r\ (x,z)\}),\\
&\qquad (\lambda(x,z).\mathsf{case}\ \iota_r\ z\ \{\iota_l\ y:\iota_l\ (x,y),\iota_r\ z:\iota_r\ (x,z)\})]\\
&= [(\lambda(x,y).\iota_l\ (x,y)),(\lambda(x,z).\iota_r\ (x,z))] = [\iota_l,\iota_r] = \mathsf{id}_{(X\otimes Y)+(X\otimes Z)}
\end{aligned}
$$

and

$$\begin{aligned}\delta^{-1}_{X,Y,Z};\delta_{X,Y,Z} &= (\lambda(x,w).\mathsf{case}\ w\ \{\iota_l\ y : \iota_l\ (x,y), \iota_r\ z : \iota_r\ (x,z)\});[X\otimes\iota_l, X\otimes\iota_r]\\ &= (\lambda(x,w).\mathsf{let}\ w' = \mathsf{case}\ w\ \{\iota_l\ y : \iota_l\ (x,y), \iota_r\ z : \iota_r\ (x,z)\};\\ &\quad \mathsf{case}\ w'\ \{\iota_l\ (x,y) : (x,\iota_l\ y), \iota_r\ (x,z) : (x,\iota_r\ z)\})\\ &= (\lambda(x,w).\mathsf{case}\ w\ \{\iota_l\ y : \mathsf{case}\ \iota_l\ (x,y)\ \{\iota_l\ (x,y) : (x,\iota_l\ y), \iota_r\ (x,z) : (x,\iota_r\ z)\}\\ &\quad ,\iota_r\ z : \mathsf{case}\ \iota_r\ (x,z)\ \{\iota_l\ (x,y) : (x,\iota_l\ y), \iota_r\ (x,z) : (x,\iota_r\ z)\}\})\\ &= (\lambda(x,w).\mathsf{case}\ w\ \{\iota_l\ y : (x,\iota_l\ y), \iota_r\ z : (x,\iota_r\ z)\})\\ &= (\lambda(x,w).(x,\mathsf{case}\ w\ \{\iota_l\ y : \iota_l\ y, \iota_r\ z : \iota_r\ z\})) = (\lambda(x,w).(x,w)) = \mathsf{id}_{X\otimes(Y+Z)}\end{aligned}$$

- $\mathsf{Tm}_\epsilon(\mathcal{R})$ has an iteration operator: we have that

$$\begin{aligned}(\lambda x.a)^\dagger &= (\lambda y.\mathsf{iter}\ y\ \{\iota_r\ x : a\})\\ &= (\lambda y.\mathsf{let}\ x = y;\ \mathsf{case}\ a\ \{\iota_l\ z : z, \iota_r\ y : \mathsf{iter}\ y\ \{\iota_r\ x : a\}\})\\ &= (\lambda x.\mathsf{case}\ a\ \{\iota_l\ z : z, \iota_r\ y : \mathsf{iter}\ y\ \{\iota_r\ x : a\}\})\\ &= (\lambda x.\mathsf{let}\ w = a;\ \mathsf{case}\ w\ \{\iota_l\ z : z, \iota_r\ y : \mathsf{iter}\ y\ \{\iota_r\ x : a\}\})\\ &= (\lambda x.a);(\lambda w.\mathsf{case}\ w\ \{\iota_l\ z : z, \iota_r\ y : \mathsf{iter}\ y\ \{\iota_r\ x : a\}\})\\ &= (\lambda x.a);[(\lambda z.z),(\lambda x.a)^\dagger] = (\lambda x.a);[\mathsf{id},(\lambda x.a)^\dagger]\end{aligned}$$

- $\mathsf{Tm}_\epsilon(\mathcal{R})$ is an Elgot category: it suffices to show that our iteration operator satisfies:
  - Naturality: we have that

$$\begin{aligned}((\lambda x.a);(\lambda y.b) + (\lambda z.z))^\dagger &= (\lambda x.\mathsf{case}\ a\ \{\iota_l\ y : \iota_l\ b, \iota_r\ z : \iota_r\ z\})^\dagger\\ &= (\lambda w.\mathsf{iter}\ w\ \{\iota_r\ x : \mathsf{case}\ a\ \{\iota_l\ y : \iota_l\ b, \iota_r\ z : \iota_r\ z\}\})\\ &= (\lambda w.\mathsf{iter}\ w\ \{\iota_r\ x : \mathsf{case}\ a\ \{\iota_l\ y : \iota_l\ b, \iota_r\ z : \iota_r\ z\}\})\\ &= (\lambda w.\mathsf{let}\ y = \mathsf{iter}\ w\ \{\iota_r\ x : a\};\ b)\\ &= (\lambda x.a)^\dagger;(\lambda y.b)\end{aligned}$$

  - Codiagonal: we have that

$$\begin{aligned}((\lambda x.a)^\dagger)^\dagger &= (\lambda z.\mathsf{iter}\ z\ \{\iota_r\ y : \mathsf{iter}\ y\ \{\iota_r\ x : a\}\})\\ &= (\lambda z.\mathsf{iter}\ z\ \{\iota_r\ x : \mathsf{case}\ a\ \{\iota_l\ y : y, \iota_r\ w : \iota_r\ w\}\})\\ &= (\lambda x.\mathsf{case}\ a\ \{\iota_l\ y : y, \iota_r\ w : \iota_r\ w\})^\dagger\\ &= ((\lambda x.a);[(\lambda y.y),(\lambda w.\iota_r\ w)])^\dagger = ((\lambda x.a);[\mathsf{id},\iota_r])^\dagger\end{aligned}$$

  - Directed Uniformity: assume that

$$(\lambda x.a);(\lambda y.b) \twoheadrightarrow^p (\lambda y.b');\mathsf{id} + (\lambda x.a)$$

    Unfolding both sides of this refinement, we hence have that

$$(\lambda x.\mathsf{let}\ y = a;\ b) \twoheadrightarrow^p (\lambda y.\mathsf{case}\ b'\ \{\iota_l\ z : \iota_l\ z, \iota_r\ x : \iota_r\ a\})$$

    We therefore, by unif^p, have that

$$\begin{aligned}(\lambda x.a);(\lambda y.b)^\dagger &= (\lambda x.\mathsf{iter}\ a\ \{\iota_r\ y : b\}) = (\lambda z.\mathsf{let}\ x = z;\ \mathsf{iter}\ a\ \{\iota_r\ y : b\})\\ &\twoheadrightarrow^p (\lambda z.\mathsf{iter}\ z\ \{\iota_r\ y : b'\}) = (\lambda y.b')^\dagger\end{aligned}$$

    as desired.

– Dinaturality: we have that

Proposition B.2. *If $(-)^\dagger$ is an iteration operator which satisfies naturality and codiagonal and is $\mathcal{K}$-uniform for $\mathcal{K}$ cocartesian, then it also satisfies dinaturality.*

Proof. See Lemma 31 of Goncharov and Schröder [13] □
Since all $\mathsf{Tm}_\epsilon(\mathcal{R})$ for $\epsilon \in \mathcal{E}^\infty$ are $\mathsf{Tm}_\perp(\mathcal{R})$-uniform, and $\mathsf{Tm}_\perp(\mathcal{R})$ is cocartesian, $\mathsf{Tm}_\epsilon(\mathcal{R})$ must in fact satisfy dinaturality as desired.

- Strength: fix $z : A \vdash a : B + A$. We have that

$$\begin{aligned} x : C, y : A \vdash_{\mathcal{R}} & \ \mathsf{let}\ (x,z) = (x,y);\ \mathsf{case}\ a\ \{\iota_l\ w_l : \iota_l\ (x, w_l), \iota_r\ w_r : \iota_r\ (x, w_r)\} \\ & \approx \mathsf{case}\ (\mathsf{let}\ z = y;\ a)\ \{\iota_l\ w_l : \iota_l\ (x, w_l), \iota_r\ w_r : \iota_r\ (x, w_r)\} \\ & \approx \mathsf{case}\ (\mathsf{let}\ z = y;\ a)\ \{\iota_l\ w_l : \iota_l\ (x, w_l), \iota_r\ y : \iota_r\ (x, y)\} \end{aligned}$$

Hence, by uniformity, as (v_l, v_r') is pure, we have

$$\begin{aligned} & (C \otimes (\lambda z.a); \delta^{-1})^\dagger \\ & = (\lambda(x,z).\mathsf{let}\ (x', w) = (x, a);\ \mathsf{case}\ w\ \{\iota_l\ w_l : \iota_l\ (x', w_l), \iota_r\ w_r : \iota_r\ (x', w_r)\})^\dagger \\ & = (\lambda(x,z).\mathsf{case}\ a\ \{\iota_l\ w_l : \iota_l\ (x, w_l), \iota_r\ w_r : \iota_r\ (x, w_r)\})^\dagger \\ & = (\lambda v.\mathsf{iter}\ v\ \{\iota_r\ (x,z) : \mathsf{case}\ a\ \{\iota_l\ w_l : \iota_l\ (x, w_l), \iota_r\ w_r : \iota_r\ (x, w_r)\}\}) \\ & = (\lambda(x,y).\mathsf{let}\ y = y;\ \mathsf{iter}\ (x,y)\ \{\iota_r\ (x,z) : \mathsf{case}\ a\ \{\iota_l\ w_l : \iota_l\ (x, w_l), \iota_r\ w_r : \iota_r\ (x, w_r)\}\}) \\ & = (\lambda(x,y).\mathsf{let}\ w = \mathsf{iter}\ y\ \{\iota_r\ y : \mathsf{let}\ z = y;\ a\};\ (x,w)) \\ & = (\lambda(x,y).\mathsf{let}\ w = \mathsf{iter}\ y\ \{\iota_r\ z : a\};\ (x,w)) \\ & = (\lambda(x,y).(x, \mathsf{iter}\ y\ \{\iota_r\ z : a\})) \\ & = C \otimes (\lambda z.a)^\dagger \end{aligned}$$

Hence, to show that we have a valid λ_{iter}-model, it suffices to prove that, given $(\lambda x.a) : \mathsf{Tm}_\epsilon(\mathcal{R})(A, B)$

- If A is relevant,

$$\begin{aligned} \Delta_A; \Delta_A \otimes A; \alpha &= (\lambda x.(x,x)); (\lambda(y,z).((y,y),z)); (\lambda((w_1,w_2),w_3).(w_1,(w_2,w_3))) \\ &= (\lambda x.((x,x),x)); (\lambda((w_1,w_2),w_3).(w_1,(w_2,w_3))) \\ &= (\lambda x.(x,(x,x))) = (\lambda x.(x,x)); (\lambda(y,z).(y,(z,z))) \\ &= \Delta_A; A \otimes \Delta_A \end{aligned}$$

and

$$\begin{aligned} \Delta_A; \sigma_{A,A} &= (\lambda x.(x,x)); (\lambda(y,z).(z,y)) \\ &= (\lambda x.\mathsf{let}\ (y,z) = (x,x);\ (z,y)) = (\lambda x.(x,x)) = \Delta_A \end{aligned}$$

as desired

- If $0 \le \mathrm{q}^p(\epsilon)$ and A, B affine, by let_1-β^p,

$$(\lambda x.a); !_B = (\lambda x.\mathsf{let}\ y = a;\ ()) \twoheadrightarrow^p (\lambda x.()) = !_A$$

- If $0 \le \mathrm{q}^p(\epsilon)$ and A relevant, B affine, by elim^p,

$$\begin{aligned} \Delta_A; (f; !_B) \otimes A &= (\lambda x.\mathsf{let}\ (y,z) = (x,x);\ (\mathsf{let}\ w = f(y);\ (),z)) \\ &= (\lambda x.(f(x);(),x)) = (\lambda x.\mathsf{let}\ y = (f(x);());\ ((),x)) \\ &\twoheadrightarrow^p (\lambda x.\mathsf{let}\ y = ();\ ((),x)) = (\lambda x.((),x)) = \rho^{-1} \end{aligned}$$

- If $\omega^+ \leq \mathsf{q}^p(\epsilon)$ and A, B relevant, by let_1-β^p,

$$\begin{aligned}(\lambda x.a); \Delta_B &= (\lambda x.a); (\lambda y.(y,y)) = (\lambda x.\mathsf{let}\ y = a;\ (y,y)) = (\lambda x.(a,a)) \\ &= (\lambda x.(\mathsf{let}\ y = x;\ [y/x]a, \mathsf{let}\ z = x;\ [z/x]a)) \\ &= (\lambda x.\mathsf{let}\ y = x;\ \mathsf{let}\ z = x;\ ([y/x]a, [z/x]a)) \\ &= (\lambda x.\mathsf{let}\ (y,z) = (x,x);\ ([y/x]a, [z/x]a)) \\ &= (\lambda x.(x,x)); (\lambda(y,z).([y/x]a, [z/x]a)) = \Delta_A; (\lambda x.a) \ltimes (\lambda x.a)\end{aligned}$$

□

B.2 Packing and Unpacking

Given an annotated context Γ^{q}, we can recursively define the *packing* $\Gamma^{\mathsf{q}} \vdash_\perp \mathsf{pack}(\Gamma^{\mathsf{q}}) : [\Gamma^{\mathsf{q}}]$ of its variables in the obvious manner; namely

$$\mathsf{pack}(\cdot) := () \qquad \mathsf{pack}(\Gamma^{\mathsf{q}}, x : A^q) = (\mathsf{pack}(\Gamma^{\mathsf{q}}), x^q) \qquad x^q := \begin{cases} x & \text{if } q \neq 0 \\ () & \text{otherwise} \end{cases}$$

We can similarly *unpack* a context's effective type $a : [\Gamma^{\mathsf{q}}]$, which we write $\mathsf{let}\ \Gamma = a;\ b$, inductively in the obvious manner:

$$(\mathsf{let}\ \cdot = a;\ b) := (a; b) \qquad (\mathsf{let}\ \Gamma, x : A = a;\ b) := (\mathsf{let}\ (g, x) = a;\ \mathsf{let}\ \Gamma = g;\ b)$$

We can show this satisfies the following typing rule by induction on $\Delta^{\mathsf{q}'}$.

$$\frac{\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r \quad \Gamma^{\mathsf{q}_l} \vdash_\epsilon a : [\Delta^{\mathsf{q}'}] \quad \Gamma^{\mathsf{q}_r}, \Delta^{\mathsf{q}'} \vdash_\epsilon b : B}{\Gamma^{\mathsf{q}} \vdash_\epsilon \mathsf{let}\ \Gamma = a;\ b : B}\ \mathsf{unpack}$$

In particular, we proceed as follows:

- Given $a : [\cdot]$, this follows directly from the derived rule seq.
- Given $a : [\Delta^{\mathsf{q}'}, x : A^q]$, we have that $\Gamma^{\mathsf{q}_r}, x : A^q, \Delta^{\mathsf{q}'} \vdash_\epsilon b : B$. Hence,
  - If $q \neq 0$, then $[A^q] = A$, so by weakening $\Gamma^{\mathsf{q}}, x : A \vdash_\epsilon \mathsf{let}\ \Gamma = a';\ b : B$ and therefore $\Gamma^{\mathsf{q}}, g : [\Delta^{\mathsf{q}'}]^0, x : [A^q] \vdash_\epsilon \mathsf{let}\ \Gamma = a';\ b : B$
  - If $q = 0$, then x cannot be used in b, so $\Gamma^{\mathsf{q}} \vdash_\epsilon \mathsf{let}\ \Gamma = a';\ b : B$, and hence, as $[A^0] = \mathbf{1}$ has unrestricted linearity, $\Gamma^{\mathsf{q}}, g : [\Delta^{\mathsf{q}'}]^0, x : [A^q] \vdash_\epsilon \mathsf{let}\ \Gamma = a';\ b : B$ by weakening.

  By induction, we may hence derive

$$\frac{\Gamma^0, g : [\Delta^{\mathsf{q}'}], x : [A^q]^0 \vdash_\epsilon g : [\Delta^{\mathsf{q}'}] \quad \Gamma^{\mathsf{q}_l}, g : [\Delta^{\mathsf{q}'}]^0, x : [A^q], \Delta^{\mathsf{q}'} \vdash_\epsilon b : B}{\Gamma^{\mathsf{q}_l}, g : [\Delta^{\mathsf{q}'}], x : [A^q] \vdash_\epsilon \mathsf{let}\ \Gamma = g;\ b : B}\ \mathsf{unpack}$$

  Therefore, since $[\Delta^{\mathsf{q}'}, x : A^q] = [\Delta^{\mathsf{q}'}] \otimes [A^q]$, we may derive

$$\frac{\Gamma^{\mathsf{q}_r} \vdash_\epsilon a : [\Delta^{\mathsf{q}'}, x : A^q] \quad \Gamma^{\mathsf{q}_l}, g : [\Delta^{\mathsf{q}'}], x : [A^q] \vdash_\epsilon \mathsf{let}\ \Gamma = g;\ b : B}{\Gamma^{\mathsf{q}} \vdash_\epsilon \mathsf{let}\ (g, x) = a;\ \mathsf{let}\ \Gamma = g;\ b : B}\ \mathsf{let}_2$$

  as desired.

We prove that packing and unpacking are mutually inverse up to $\approx$ by induction:

Lemma B.3. *The following rules are derivable:*

$$\frac{\Gamma^{\mathfrak{q}}, \Delta^{\mathfrak{q}'} \vdash_{\epsilon} a : A}{\Gamma^{\mathfrak{q}}, \Delta^{\mathfrak{q}'} \vdash_{\mathcal{R}} \mathsf{let}\ \Delta = \mathsf{pack}(\Delta^{\mathfrak{q}'});\ a \approx a : A}\ \textit{unpack-pack}$$

$$\frac{\Gamma^{\mathfrak{q}} \vdash_{\epsilon} b : [\Delta^{\mathfrak{q}'}]}{\Gamma^{\mathfrak{q}} \vdash_{\mathcal{R}} \mathsf{let}\ \Delta = b;\ \mathsf{pack}(\Delta^{\mathfrak{q}'}) \approx b : [\Delta^{\mathfrak{q}'}]}\ \textit{pack-unpack}$$

Proof. We proceed by $\Delta^{\mathfrak{q}'}$

- $(\cdot)$: the results follow by elim and term, respectively.
- $(\Gamma^{\mathfrak{q}}, x : B^{q})$: we have by induction that
  - We have

$$\begin{aligned}\Gamma^{\mathfrak{q}}, \Delta^{\mathfrak{q}'}, x : B^{q} \vdash_{\mathcal{R}} \mathsf{let}\ \Delta^{\mathfrak{q}'}, x : B^{q} = \mathsf{pack}((\Delta^{\mathfrak{q}'}, x : B^{q});\ a \\ \approx \mathsf{let}\ (g, y) = (\mathsf{pack}(\Delta^{\mathfrak{q}'}), x^{q});\ [y/x]a \\ \approx \mathsf{let}\ g = \mathsf{pack}(\Delta^{\mathfrak{q}'});\ \mathsf{let}\ y = x^{q};\ [y/x]a \\ \approx \mathsf{let}\ y = x^{q};\ [y/x]a \approx a : A\end{aligned}$$

    since if $q \neq 0$, $\mathsf{let}\ y = x^{q};\ [y/x]a \approx \mathsf{let}\ y = x;\ [y/x]a \approx a$, whereas if $q = 0$ $x \notin \mathsf{fv}(a)$ so $\mathsf{let}\ y = x^{q};\ [y/x]a \approx \mathsf{let}\ y = x;\ a \approx a$.
  - We have

$$\begin{aligned}&\Gamma^{\mathfrak{q}} \vdash_{\mathcal{R}} \mathsf{let}\ \Delta^{\mathfrak{q}'}, x : A^{q} = b;\ \mathsf{pack}(\Delta^{\mathfrak{q}'}, x : B^{q}) \\ &\approx \mathsf{let}\ (g, x) = b;\ \mathsf{let}\ \Delta^{\mathfrak{q}'} = g;\ (\mathsf{pack}(\Delta^{\mathfrak{q}'}), x^{q}) \\ &\approx \mathsf{let}\ (g, x) = b;\ \mathsf{let}\ w = (\mathsf{let}\ \Delta^{\mathfrak{q}'} = g;\ \mathsf{pack}(\Delta^{\mathfrak{q}'}));\ (w, x^{q}) \\ &\approx \mathsf{let}\ (g, x) = b;\ \mathsf{let}\ w = g;\ (w, x^{q}) \approx \mathsf{let}\ (g, x) = b;\ (g, x^{q}) \qquad \approx (g, x)\end{aligned}$$

    since if $q \neq 0$ $(g, x^{q}) := (g, x)$, whereas if $q = 0$ since x is pure and of type $\mathbf{1}$, by **term**, $(g, x^{q}) := (g, ()) \approx (g, x)$.

□

To show completeness, it hence suffices to prove the following lemma:

Lemma B.4. *Given* $\Gamma^{\mathfrak{q}} \vdash a : A$, *we have that, for all* $(\lambda x.a_x) \in [\![\Gamma^{\mathfrak{q}} \vdash a : A]\!]_{\mathsf{Tm}(\mathcal{R})}$,

$$\Gamma^{\mathfrak{q}} \vdash_{\mathcal{R}} \mathsf{let}\ x = \mathsf{pack}(\Gamma^{\mathfrak{q}});\ a_x \approx a : A$$

Proof. Throughout this proof, given morphism $f \in \mathsf{Tm}(\mathcal{R})(A, B)$, we will use $f(x)$ as a stand-in for an arbitrary member of the appropriate equivalence class. We note that (for arbitrary types)

$$\mathsf{let}\ x = a;\ (f; g)(x) \approx \mathsf{let}\ y = (\mathsf{let}\ x = a;\ f(x));\ g(y)$$

and hence that, if $\mathsf{let}\ x = a;\ f(x) \approx a'$, $\mathsf{let}\ x = a;\ (f; g)(x) = \mathsf{let}\ x = a';\ g(x)$. In particular, we hence have that

- Given b pure and $\mathsf{let}\ x = a;\ f(x) = a'$, $\mathsf{let}\ y = b;\ g(y) = b'$, we have

$$\mathsf{let}\ z = (a, b);\ ((f \otimes g); h)(z) \approx \mathsf{let}\ z = (a', b');\ h(z)$$

- Given a pure,

$$\begin{aligned}&\mathsf{let}\ z = (a, b);\ (\delta^{-1}; f)(z) \\ &\approx \mathsf{case}\ b\ \{\iota_l\ y_l : \mathsf{let}\ z_l = \iota_l\ (a, y_l);\ f(z_l), \iota_r\ y_r : \mathsf{let}\ z_r = \iota_r\ (a, y_r);\ f(z_r)\} \\ &\approx \mathsf{case}\ b\ \{\iota_l\ y_l : \mathsf{let}\ z_l = (a, y_l);\ \iota_l; f(z_l), \iota_r\ y_r : \mathsf{let}\ z_r = (a, y_r);\ \iota_r; f(z_r)\}\end{aligned}$$

In particular, this implies that

$$\begin{aligned}&\mathsf{let}\ z = (a, b);\ (\delta^{-1}; [f, g])(z)\\ &\approx \mathsf{case}\ b\ \{\iota_l\ y_l : \mathsf{let}\ z_l = (a, y_l);\ f(z_l), \iota_r\ y_r : \mathsf{let}\ z_r = (a, y_r);\ g(z_r)\}\end{aligned}$$

We begin by showing that

$$[\![\Gamma^{\mathbf{q}} \mapsto \cdot]\!] \approx (\lambda x.())$$

which we do by a straightforward induction on the derivation $\Gamma^{\mathbf{q}} \mapsto \cdot$:

- $(\cdot \mapsto \cdot)$: we have $[\![\cdot \mapsto \cdot]\!] = \mathsf{id}_{\mathbf{1}} = (\lambda x : \mathbf{1}.x) = (\lambda x.())$ as desired.
- $(\Gamma^{\mathbf{q}}, x : A^q \mapsto \cdot)$: we have that

$$\begin{aligned}[\![\Gamma^{\mathbf{q}}, x : A^q \mapsto \cdot]\!] &= [\Gamma^{\mathbf{q}}] \otimes !_{A^q}; \rho; [\![\Gamma^{\mathbf{q}} \mapsto \cdot]\!] = (\lambda(x, y).((), ())); (\lambda(z, w).z); (\lambda u.())\\ &= (\lambda(x, y).()); (\lambda u.()) = (\lambda(x, y).()) = (\lambda u.())\end{aligned}$$

It follows immediately that

$$\Gamma^{\mathbf{q}} \vdash_{\perp} \mathsf{let}\ x = \mathsf{pack}(\Gamma^{\mathbf{q}});\ [\![\Gamma^{\mathbf{q}} \mapsto \cdot]\!](x) \approx \mathsf{let}\ x = \mathsf{pack}(\Gamma^{\mathbf{q}});\ () \approx () : \mathbf{1}$$

We may therefore show that

$$\Gamma^{\mathbf{q}} \vdash_{\perp} \mathsf{let}\ x = \mathsf{pack}(\Gamma^{\mathbf{q}});\ [\![\Gamma^{\mathbf{q}} \mapsto y : A^1]\!](x) \approx ((), y)) : [y : A^1]$$

We proceed by induction on the derivation $\Gamma^{\mathbf{q}} \mapsto y : A^1$:

- $(\Gamma^{\mathbf{q}}, y : A^q \mapsto y : A^1)$: we have that

$$[\![\Gamma^{\mathbf{q}}, y : A^q \mapsto y : A^1]\!] = [\![\Gamma^{\mathbf{q}} \mapsto \cdot]\!] \otimes \mathsf{id} = (\lambda x.()) \otimes (\lambda y.y) = (\lambda(x, y).((), y))$$

It follows that

$$\begin{aligned}\Gamma^{\mathbf{q}}, y : A^q \vdash_{\perp} &\mathsf{let}\ x = \mathsf{pack}(\Gamma^{\mathbf{q}}, y : A^q);\ [\![\Gamma^{\mathbf{q}}, y : A^q \mapsto y : A^1]\!](x)\\ &\approx \mathsf{let}\ x = (\mathsf{pack}(\Gamma^{\mathbf{q}}, y));\ (\lambda(z, w).((), w))(x)\\ &\approx \mathsf{let}\ (z, w) = (\mathsf{pack}(\Gamma^{\mathbf{q}}, y));\ ((), w)\\ &\approx ((), y) : [y : A^1]\end{aligned}$$

- $(\Gamma^{\mathbf{q}}, z : B^q \mapsto y : A^1)$: we have that

$$\begin{aligned}[\![\Gamma^{\mathbf{q}}, z : B^q \mapsto y : A^1]\!] &= [\Gamma^{\mathbf{q}}] \otimes !_{B^q}; \rho; [\![\Gamma^{\mathbf{q}} \mapsto y : A^1]\!]\\ &= (\lambda(x, y).(x, ())); (\lambda(z, w).z); [\![\Gamma^{\mathbf{q}} \mapsto y : A^1]\!]\\ &= (\lambda(x, y).x); [\![\Gamma^{\mathbf{q}} \mapsto y : A^1]\!]\\ &= (\lambda(x, y).[\![\Gamma^{\mathbf{q}} \mapsto y : A^1]\!](x))\end{aligned}$$

It follows by induction that

$$\begin{aligned}\Gamma^{\mathbf{q}}, z : B^q \vdash_{\perp} &\mathsf{let}\ x = \mathsf{pack}(\Gamma^{\mathbf{q}}, z : B^q);\ [\![\Gamma^{\mathbf{q}}, y : A^q \mapsto y : A^1]\!](x)\\ &\approx \mathsf{let}\ x = \mathsf{pack}(\Gamma^{\mathbf{q}}, z^q);\ \lambda(w, u).[\![\Gamma^{\mathbf{q}} \mapsto y : A^1]\!](w)\\ &\approx \mathsf{let}\ w = \mathsf{pack}(\Gamma^{\mathbf{q}});\ [\![\Gamma^{\mathbf{q}} \mapsto y : A^1]\!](w) \approx ((), y) : [y : A^1]\end{aligned}$$

Similarly, we have that

$$\Gamma^{\mathbf{q}} \vdash_{\perp} \mathsf{let}\ x = \mathsf{pack}(\Gamma^{\mathbf{q}});\ [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q_r}]\!](x) \approx (\mathsf{pack}(\Gamma^{\mathbf{q}_l}), \mathsf{pack}(\Gamma^{\mathbf{q}_r})) : [\Gamma^{\mathbf{q}_l}] \otimes [\Gamma^{\mathbf{q}_r}]$$

We proceed by induction on the derivation $\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q_r}$:

- $(\cdot \vdash \cdot = \cdot + \cdot)$: we have that

$$\begin{aligned}
&\cdot \vdash_\perp \text{let } x = \mathsf{pack}(\cdot);\ [\![\cdot \vdash \cdot = \cdot + \cdot]\!](x) \approx \text{let } x = ();\ \rho^{-1}(x) \\
&\quad \approx \text{let } x = ();\ (x, ()) \approx ((), ()) \approx (\mathsf{pack}(\cdot), \mathsf{pack}(\cdot)) : [\mathsf{pack}(\cdot)] \otimes [\mathsf{pack}(\cdot)]
\end{aligned}$$

  as desired.
- $(\Gamma, x : A \vdash \mathbf{q}, q = \mathbf{q}_l, q + \mathbf{q}_r, 0)$: we have that

$$\begin{aligned}
\Gamma^{\mathbf{q}}, x : A^q \vdash_\perp &\ \text{let } w = \mathsf{pack}(\Gamma^{\mathbf{q}}, x : A^q);\ [\![\Gamma, x : A \vdash \mathbf{q}, q = \mathbf{q}_l, q + \mathbf{q}_r, 0]\!](w) \\
&\approx \text{let } w = (\mathsf{pack}(\Gamma^{\mathbf{q}}), x^q);\ ([\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!] \otimes \rho^{-1}; \sigma^{\mathsf{mid}})(w) \\
&\approx \text{let } w = (\mathsf{pack}(\Gamma^{\mathbf{q}}), x^q);\ ((\lambda(y, z).([\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!](y), (z, ()))); \sigma^{\mathsf{mid}})(w) \\
&\approx \text{let } y = \mathsf{pack}(\Gamma^{\mathbf{q}});\ \text{let } z = x^q;\ \text{let } u = ([\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!](y), (z, ()));\ \sigma^{\mathsf{mid}}(u) \\
&\approx \text{let } y = \mathsf{pack}(\Gamma^{\mathbf{q}});\ \text{let } u = ([\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!](y), (x^q, ()));\ \sigma^{\mathsf{mid}}(u) \\
&\approx \text{let } u = ((\text{let } y = \mathsf{pack}(\Gamma^{\mathbf{q}});\ ([\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!](y)), (x^q, ()));\ \sigma^{\mathsf{mid}}(u) \\
&\approx \text{let } ((u_1, u_2), (u_3, u_4)) = ((\mathsf{pack}(\Gamma^{\mathbf{q}_l}), \mathsf{pack}(\Gamma^{\mathbf{q}_r})), (x^q, ()));\ ((u_1, u_3), (u_2, u_4)) \\
&\approx (\mathsf{pack}(\Gamma^{\mathbf{q}_l}), x^q), (\mathsf{pack}(\Gamma^{\mathbf{q}_r}), ())) \\
&\approx (\mathsf{pack}(\Gamma^{\mathbf{q}_l}, x : A^q), \mathsf{pack}(\Gamma^{\mathbf{q}_r}, x : A^0)) : [\Gamma^{\mathbf{q}_l}, x : A^q] \otimes [\Gamma^{\mathbf{q}_r}, x : A^0]
\end{aligned}$$

  as desired.
- $(\Gamma, x : A \vdash \mathbf{q}, q = \mathbf{q}_l, 0 + \mathbf{q}_r, q)$: we have that

$$\begin{aligned}
\Gamma^{\mathbf{q}}, x : A^q \vdash_\perp &\ \text{let } w = \mathsf{pack}(\Gamma^{\mathbf{q}}, x : A^q);\ [\![\Gamma, x : A \vdash \mathbf{q}, q = \mathbf{q}_l, 0 + \mathbf{q}_r, q]\!](w) \\
&\approx \text{let } w = (\mathsf{pack}(\Gamma^{\mathbf{q}}), x^q);\ ([\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!] \otimes \lambda^{-1}; \sigma^{\mathsf{mid}})(w) \\
&\approx \text{let } w = (\mathsf{pack}(\Gamma^{\mathbf{q}}), x^q);\ ((\lambda(y, z).([\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!](y), ((), z))); \sigma^{\mathsf{mid}})(w) \\
&\approx \text{let } y = \mathsf{pack}(\Gamma^{\mathbf{q}});\ \text{let } z = x^q;\ \text{let } u = ([\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!](y), ((), z));\ \sigma^{\mathsf{mid}}(u) \\
&\approx \text{let } y = \mathsf{pack}(\Gamma^{\mathbf{q}});\ \text{let } u = ([\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!](y), ((), x^q));\ \sigma^{\mathsf{mid}}(u) \\
&\approx \text{let } u = ((\text{let } y = \mathsf{pack}(\Gamma^{\mathbf{q}});\ ([\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!](y)), ((), x^q));\ \sigma^{\mathsf{mid}}(u) \\
&\approx \text{let } ((u_1, u_2), (u_3, u_4)) = ((\mathsf{pack}(\Gamma^{\mathbf{q}_l}), \mathsf{pack}(\Gamma^{\mathbf{q}_r})), ((), x^q));\ ((u_1, u_3), (u_2, u_4)) \\
&\approx (\mathsf{pack}(\Gamma^{\mathbf{q}_l}), ()), (\mathsf{pack}(\Gamma^{\mathbf{q}_r}), x^q)) \\
&\approx (\mathsf{pack}(\Gamma^{\mathbf{q}_l}, x : A^0), \mathsf{pack}(\Gamma^{\mathbf{q}_r}, x : A^q)) : [\Gamma^{\mathbf{q}_l}, x : A^0] \otimes [\Gamma^{\mathbf{q}_r}, x : A^q]
\end{aligned}$$

  as desired.

- $(\Gamma, x : A \vdash \mathbf{q}, q = \mathbf{q}_l, q + \mathbf{q}_r, q)$: we have that

$$\begin{aligned}
\Gamma^{\mathbf{q}}, x : A^q \vdash_\perp & \text{ let } w = \mathsf{pack}(\Gamma^{\mathbf{q}}, x : A^q);\ [\![\Gamma, x : A \vdash \mathbf{q}, q = \mathbf{q}_l, q + \mathbf{q}_r, q]\!](w) \\
&\approx \text{let } w = (\mathsf{pack}(\Gamma^{\mathbf{q}}), x^q);\ ([\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!] \otimes \Delta; \sigma^{\mathsf{mid}})(w) \\
&\approx \text{let } w = (\mathsf{pack}(\Gamma^{\mathbf{q}}), x^q);\ ((\lambda(y,z).([\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!](y), (z,z))); \sigma^{\mathsf{mid}})(w) \\
&\approx \text{let } y = \mathsf{pack}(\Gamma^{\mathbf{q}});\ \text{let } z = x^q;\ \text{let } u = ([\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!](y), (z,z));\ \sigma^{\mathsf{mid}}(u) \\
&\approx \text{let } y = \mathsf{pack}(\Gamma^{\mathbf{q}});\ \text{let } u = ([\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!](y), (x^q, x^q));\ \sigma^{\mathsf{mid}}(u) \\
&\approx \text{let } u = ((\text{let } y = \mathsf{pack}(\Gamma^{\mathbf{q}});\ ([\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!](y)), (x^q, x^q));\ \sigma^{\mathsf{mid}}(u) \\
&\approx \text{let } ((u_1,u_2),(u_3,u_4)) = ((\mathsf{pack}(\Gamma^{\mathbf{q}_l}), \mathsf{pack}(\Gamma^{\mathbf{q}_r})), (x^q, x^q));\ ((u_1,u_3),(u_2,u_4)) \\
&\approx (\mathsf{pack}(\Gamma^{\mathbf{q}_l}), x^q), (\mathsf{pack}(\Gamma^{\mathbf{q}_r}), x^q)) \\
&\approx (\mathsf{pack}(\Gamma^{\mathbf{q}_l}, x : A^q), \mathsf{pack}(\Gamma^{\mathbf{q}_r}, x : A^q)) : [\Gamma^{\mathbf{q}_l}, x : A^q] \otimes [\Gamma^{\mathbf{q}_r}, x : A^q]
\end{aligned}$$

  as desired.

We now wish to show that, given $f \in \mathsf{Tm}(\mathcal{R})(A, B)$,

$$\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \text{let } x = \mathsf{pack}(\Gamma^{\mathbf{q}});\ [\![\Gamma^{\mathbf{q}} \vdash a : A]\!]_{\mathsf{Tm}(\mathcal{R})}(x) \approx a : A$$

We proceed by induction on the derivation $\Gamma^{\mathbf{q}} \vdash a : A$:

- $(\Gamma^{\mathbf{q}} \vdash x : A)$: we have that

$$\begin{aligned}
\Gamma^{\mathbf{q}} \vdash_\perp & \text{ let } y = \mathsf{pack}(\Gamma^{\mathbf{q}});\ [\![\Gamma^{\mathbf{q}} \vdash x : A]\!](y) \\
&\approx \text{let } y = \mathsf{pack}(\Gamma)^{\mathbf{q}};\ ([\![\Gamma^{\mathbf{q}} \mapsto x : A^1]\!]; \lambda)(y) \\
&\approx \text{let } y = (\text{let } z = \mathsf{pack}(\Gamma^{\mathbf{q}});\ [\![\Gamma^{\mathbf{q}} \mapsto x : A^1]\!](z));\ (\lambda(w,u).u)(y) \\
&\approx \text{let } (w,u) = (x, ());\ u \approx x : A
\end{aligned}$$

  as desired.
- $(\Gamma^{\mathbf{q}} \vdash_\epsilon f\ a : B)$: we have that

$$\begin{aligned}
\Gamma^{\mathbf{q}} \vdash_\perp & \text{ let } x = \mathsf{pack}(\Gamma^{\mathbf{q}});\ [\![\Gamma^{\mathbf{q}} \vdash f\ a : B]\!](x) \\
&\approx \text{let } x = \mathsf{pack}(\Gamma^{\mathbf{q}});\ ([\![\Gamma^{\mathbf{q}} \vdash a : A]\!]; (\lambda y. f\ y))(x) \\
&\approx \text{let } x = (\text{let } y = \mathsf{pack}(\Gamma^{\mathbf{q}});\ [\![\Gamma^{\mathbf{q}} \vdash a : A]\!]);\ f\ x \\
&\approx \text{let } x = a;\ f\ x \approx f\ a : B
\end{aligned}$$

- $(\Gamma^{\mathbf{q}} \vdash \text{let } x = a;\ b : B)$: we have

$$\begin{aligned}
\Gamma^{\mathbf{q}} \vdash_\perp & \text{ let } y = \mathsf{pack}(\Gamma^{\mathbf{q}});\ [\![\Gamma^{\mathbf{q}} \vdash \text{let } x = a;\ b : B]\!](y) \\
&\approx \text{let } y = \mathsf{pack}(\Gamma^{\mathbf{q}});\ ([\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; [\Gamma^{\mathbf{q}_l}] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon a : A]\!]; [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon b : B]\!])(y) \\
&\approx \text{let } y = (\mathsf{pack}(\Gamma^{\mathbf{q}_l}), \mathsf{pack}(\Gamma^{\mathbf{q}_r}));\ ([\Gamma^{\mathbf{q}_l}] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon a : A]\!]; [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon b : B]\!])(y) \\
&\approx \text{let } y = (\mathsf{pack}(\Gamma^{\mathbf{q}_l}), a));\ ([\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon b : B]\!])(y) \\
&\approx \text{let } x = a;\ \text{let } y = (\mathsf{pack}(\Gamma^{\mathbf{q}_l}), x));\ ([\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon b : B]\!])(y) \\
&\approx \text{let } x = a;\ \text{let } y = (\mathsf{pack}(\Gamma^{\mathbf{q}_l}, x : A)));\ ([\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon b : B]\!])(y) \\
&\approx \text{let } x = a;\ b : B
\end{aligned}$$

- $(\Gamma^{\mathbf{q}} \vdash () : \mathbf{1})$: we have

$$\begin{aligned}
&\Gamma^{\mathbf{q}} \vdash_{\perp} \text{let } x = \text{pack}(\Gamma^{\mathbf{q}});\ [\![\Gamma^{\mathbf{q}} \vdash () : \mathbf{1}]\!](x) \\
&\quad \approx \text{let } x = \text{pack}(\Gamma^{\mathbf{q}});\ ([\![\Gamma^{\mathbf{q}} \mapsto \cdot]\!])(x) \\
&\quad \approx \text{let } x = \text{pack}(\Gamma^{\mathbf{q}});\ () \approx ()
\end{aligned}$$

- $(\Gamma^{\mathbf{q}} \vdash (a, b) : A \otimes B)$: we have that

$$\begin{aligned}
&\Gamma^{\mathbf{q}} \vdash_{\perp} \text{let } x = \text{pack}(\Gamma^{\mathbf{q}});\ [\![\Gamma^{\mathbf{q}} \vdash (a, b) : A \otimes B]\!](x) \\
&\quad \approx \text{let } x = \text{pack}(\Gamma^{\mathbf{q}});\ ([\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; [\![\Gamma^{\mathbf{q}_l} \vdash a : A]\!] \ltimes [\![\Gamma^{\mathbf{q}_r} \vdash b : B]\!])(x) \\
&\quad \approx \text{let } x = (\text{let } y = \text{pack}(\Gamma^{\mathbf{q}});\ [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]);\ ([\![\Gamma^{\mathbf{q}_l} \vdash a : A]\!] \ltimes [\![\Gamma^{\mathbf{q}_r} \vdash b : B]\!])(x) \\
&\quad \approx \text{let } x = (\text{pack}(\Gamma^{\mathbf{q}_l}), \text{pack}(\Gamma^{\mathbf{q}_r}));\ ([\![\Gamma^{\mathbf{q}_l} \vdash a : A]\!] \ltimes [\![\Gamma^{\mathbf{q}_r} \vdash b : B]\!])(x) \\
&\quad \approx (\text{let } x = \text{pack}(\Gamma^{\mathbf{q}_l});\ [\![\Gamma^{\mathbf{q}_l} \vdash a : A]\!](x), \text{let } y = \text{pack}(\Gamma^{\mathbf{q}_r});\ [\![\Gamma^{\mathbf{q}_r} \vdash b : B]\!](y)) \approx (a, b) : A \otimes B
\end{aligned}$$

- $(\Gamma^{\mathbf{q}} \vdash \text{let } (x, y) = a;\ c : C)$: we have that

$$\begin{aligned}
&\Gamma^{\mathbf{q}} \vdash_{\perp} \text{let } z = \text{pack}(\Gamma^{\mathbf{q}});\ [\![\Gamma^{\mathbf{q}} \vdash \text{let } (x, y) = a;\ c : C]\!](z) \\
&\quad \approx \text{let } z = \text{pack}(\Gamma^{\mathbf{q}}); \\
&\qquad ([\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; [\![\Gamma^{\mathbf{q}_l}]\!] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_{\epsilon} a : A \otimes B]\!]; \alpha; [\![\Gamma^{\mathbf{q}_l}, x : A, y : B \vdash_{\epsilon} c : C]\!])(z) \\
&\quad \approx \text{let } z = (\text{pack}(\Gamma^{\mathbf{q}_l}), \text{pack}(\Gamma^{\mathbf{q}_r})); \\
&\qquad ([\![\Gamma^{\mathbf{q}_l}]\!] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_{\epsilon} a : A \otimes B]\!]; \alpha; [\![\Gamma^{\mathbf{q}_l}, x : A, y : B \vdash_{\epsilon} c : C]\!])(z) \\
&\quad \approx \text{let } z = (\text{pack}(\Gamma^{\mathbf{q}_l}), \text{let } w = \text{pack}(\Gamma^{\mathbf{q}_r});\ [\![\Gamma^{\mathbf{q}_r} \vdash_{\epsilon} a : A \otimes B(w)]\!]); \\
&\qquad (\alpha; [\![\Gamma^{\mathbf{q}_l}, x : A, y : B \vdash_{\epsilon} c : C]\!])(z) \\
&\quad \approx \text{let } z = (\text{pack}(\Gamma^{\mathbf{q}_l}), a);\ (\alpha; [\![\Gamma^{\mathbf{q}_l}, x : A, y : B \vdash_{\epsilon} c : C]\!])(z) \\
&\quad \approx \text{let } (z_1, (z_2, z_3)) = (\text{pack}(\Gamma^{\mathbf{q}_l}), a);\ \text{let } z = ((z_1, z_2), z_3);\ [\![\Gamma^{\mathbf{q}_l}, x : A, y : B \vdash_{\epsilon} c : C]\!](z) \\
&\quad \approx \text{let } (z_2, z_3) = a;\ \text{let } z = ((\text{pack}(\Gamma^{\mathbf{q}_l}), z_2), z_3);\ [\![\Gamma^{\mathbf{q}_l}, x : A, y : B \vdash_{\epsilon} c : C]\!](z) \\
&\quad \approx \text{let } (z_2, z_3) = a;\ \text{let } z = \text{pack}(\Gamma^{\mathbf{q}_l}, z_2 : A, z_3 : B);\ [\![\Gamma^{\mathbf{q}_l}, x : A, y : B \vdash_{\epsilon} c : C]\!](z) \\
&\quad \approx \text{let } (z_2, z_3) = a;\ [z_3/y][z_2/x]c \approx \text{let } (x, y) = a;\ c : C
\end{aligned}$$

- $(\Gamma^{\mathbf{q}} \vdash \iota_l\ a : A + B)$: we have that

$$\begin{aligned}
&\Gamma^{\mathbf{q}} \vdash_{\perp} \text{let } x = \text{pack}(\Gamma^{\mathbf{q}});\ [\![\Gamma^{\mathbf{q}} \vdash \iota_l\ a : A + B]\!](x) \\
&\quad \approx \text{let } x = \text{pack}(\Gamma^{\mathbf{q}});\ ([\![\Gamma^{\mathbf{q}} \vdash a : A]\!]; \iota_l)(x) \\
&\quad \approx \text{let } x = (\text{let } y = \text{pack}(\Gamma^{\mathbf{q}});\ [\![\Gamma^{\mathbf{q}} \vdash a : A]\!]);\ \iota_l\ x \\
&\quad \approx \text{let } x = a;\ \iota_l\ x \approx \iota_l\ a : A + B
\end{aligned}$$

- $(\Gamma^{\mathbf{q}} \vdash \iota_r\ b : A + B)$: we have that

$$\begin{aligned}
&\Gamma^{\mathbf{q}} \vdash_{\perp} \text{let } x = \text{pack}(\Gamma^{\mathbf{q}});\ [\![\Gamma^{\mathbf{q}} \vdash \iota_r\ b : A + B]\!](x) \\
&\quad \approx \text{let } x = \text{pack}(\Gamma^{\mathbf{q}});\ ([\![\Gamma^{\mathbf{q}} \vdash b : B]\!]; \iota_r)(x) \\
&\quad \approx \text{let } x = (\text{let } y = \text{pack}(\Gamma^{\mathbf{q}});\ [\![\Gamma^{\mathbf{q}} \vdash b : B]\!]);\ \iota_r\ x \\
&\quad \approx \text{let } x = b;\ \iota_r\ x \approx \iota_r\ b : A + B
\end{aligned}$$

- $(\Gamma^{\mathbf{q}} \vdash \text{case } e\ \{\iota_l\ x : a, \iota_r\ y : b\} : C)$: we have that

$$\begin{aligned}
&\Gamma^{\mathbf{q}} \vdash_\perp \text{let } z = \text{pack}(\Gamma^{\mathbf{q}});\ [\![\Gamma^{\mathbf{q}} \vdash \text{case } e\ \{\iota_l\ x : a, \iota_r\ y : b\} : C]\!](z) \\
&\quad\approx \text{let } z = \text{pack}(\Gamma^{\mathbf{q}});\ ([\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; [\![\Gamma^{\mathbf{q}_l}]\!] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon e : A + B]\!]; \delta^{-1} \\
&\qquad ; [[\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon a : C]\!], [\![\Gamma^{\mathbf{q}_l}, y : B \vdash_\epsilon b : C]\!]])(z) \\
&\quad\approx \text{let } z = (\text{pack}(\Gamma^{\mathbf{q}_l}), \text{pack}(\Gamma^{\mathbf{q}_r}));\ ([\![\Gamma^{\mathbf{q}_l}]\!] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon e : A + B]\!]; \delta^{-1} \\
&\qquad ; [[\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon a : C]\!], [\![\Gamma^{\mathbf{q}_l}, y : B \vdash_\epsilon b : C]\!]])(z) \\
&\quad\approx \text{let } z = (\text{pack}(\Gamma^{\mathbf{q}_l}), \text{let } w = \text{pack}(\Gamma^{\mathbf{q}_r});\ [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon e : A + B]\!](w));\ (\\
&\qquad \delta^{-1}; [[\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon a : C]\!], [\![\Gamma^{\mathbf{q}_l}, y : B \vdash_\epsilon b : C]\!]])(z) \\
&\quad\approx \text{let } z = (\text{pack}(\Gamma^{\mathbf{q}_l}), e);\ (\delta^{-1}; [[\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon a : C]\!], [\![\Gamma^{\mathbf{q}_l}, y : B \vdash_\epsilon b : C]\!]])(z) \\
&\quad\approx \text{case } e \\
&\qquad \{\iota_l\ x : \text{let } w = (\text{pack}(\Gamma^{\mathbf{q}_l}), x);\ [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon a : C]\!](w) \\
&\qquad , \iota_r\ y : \text{let } \ = (\text{pack}(\Gamma^{\mathbf{q}_l}), y);\ [\![\Gamma^{\mathbf{q}_l}, y : B \vdash_\epsilon b : C]\!]\} \\
&\quad\approx \text{case } e \\
&\qquad \{\iota_l\ x : \text{let } w = \text{pack}(\Gamma^{\mathbf{q}_l}, x : A));\ [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon a : C]\!](w) \\
&\qquad , \iota_r\ y : \text{let } \ = \text{pack}(\Gamma^{\mathbf{q}_l}), y : B);\ [\![\Gamma^{\mathbf{q}_l}, y : B \vdash_\epsilon b : C]\!]\} \\
&\quad\approx \text{case } e\ \{\iota_l\ x : a, \iota_r\ y : b\} : C
\end{aligned}$$

- $(\Gamma^{\mathbf{q}} \vdash \text{abort } a : A)$: we have that

$$\begin{aligned}
&\Gamma^{\mathbf{q}} \vdash_\perp \text{let } x = \text{pack}(\Gamma^{\mathbf{q}});\ [\![\Gamma^{\mathbf{q}} \vdash \text{abort } a : A]\!](x) \\
&\quad\approx \text{let } x = \text{pack}(\Gamma^{\mathbf{q}});\ ([\![\Gamma^{\mathbf{q}} \vdash a : \mathbf{0}]\!]; 0_A)(x) \\
&\quad\approx \text{let } x = (\text{let } y = \text{pack}(\Gamma^{\mathbf{q}});\ [\![\Gamma^{\mathbf{q}} \vdash a : \mathbf{0}]\!]);\ \text{abort } x \\
&\quad\approx \text{let } x = a;\ \text{abort } x \approx \text{abort } a : A
\end{aligned}$$

- $(\Gamma^{\mathbf{q}} \vdash \text{iter } a\ \{\iota_r\ x : b\} : B)$: we begin by noting that

$$\begin{aligned}
&\Gamma^{\mathbf{q}_l}, y : A \vdash_\perp \text{let } w = (\text{pack}(\Gamma^{\mathbf{q}_l}), y); \\
&\qquad ([\![\Gamma \vdash \mathbf{q}_l = \mathbf{q}_l + \mathbf{q}_l]\!] \otimes A; \alpha; [\Gamma^{\mathbf{q}_l}] \otimes [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon b : B + A]\!]; \delta^{-1})(w) \\
&\quad\approx \text{let } w = ((\text{pack}(\Gamma^{\mathbf{q}_l}), \text{pack}(\Gamma^{\mathbf{q}_l})), y);\ (\alpha; [\Gamma^{\mathbf{q}_l}] \otimes [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon b : B + A]\!]; \delta^{-1})(w) \\
&\quad\approx \text{let } w = (\text{pack}(\Gamma^{\mathbf{q}_l}), (\text{pack}(\Gamma^{\mathbf{q}_l}), y));\ ([\Gamma^{\mathbf{q}_l}] \otimes [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon b : B + A]\!]; \delta^{-1})(w) \\
&\quad\approx \text{let } w = (\text{pack}(\Gamma^{\mathbf{q}_l}), \text{let } z = (\text{pack}(\Gamma^{\mathbf{q}_l}), y);\ [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon b : B + A]\!](z));\ \delta^{-1}(w) \\
&\quad\approx \text{case } (\text{let } z = (\text{pack}(\Gamma^{\mathbf{q}_l}), y);\ [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon b : B + A]\!](z))) \\
&\qquad \{\iota_l\ w_l : \iota_l\ (\text{pack}(\Gamma^{\mathbf{q}_l}), w_l), \iota_r\ w_r : (\text{pack}(\Gamma^{\mathbf{q}_l}), w_r)\} \\
&\quad\approx \text{case } (\text{let } z = (\text{pack}(\Gamma^{\mathbf{q}_l}, y : A^q);\ [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon b : B + A]\!](z))) \\
&\qquad \{\iota_l\ w_l : \iota_l\ (\text{pack}(\Gamma^{\mathbf{q}_l}), w_l), \iota_r\ w_r : (\text{pack}(\Gamma^{\mathbf{q}_l}), w_r)\} \\
&\quad\approx \text{case } [y/x]b\ \{\iota_l\ w_l : \iota_l\ (\text{pack}(\Gamma^{\mathbf{q}_l}), w_l), \iota_r\ w_r : (\text{pack}(\Gamma^{\mathbf{q}_l}), w_r)\}
\end{aligned}$$

  It follows by uniformity, since $(\text{pack}(\Gamma^{\mathbf{q}_l}), y)$ is pure, that

$$\begin{aligned}
&\Gamma^{\mathbf{q}} \vdash_\perp \text{let } y = a;\ \text{iter } (\text{pack}(\Gamma^{\mathbf{q}_l}), y)\ \{\iota_r\ w : \\
&\qquad ([\![\Gamma \vdash \mathbf{q}_l = \mathbf{q}_l + \mathbf{q}_l]\!] \otimes A; \alpha; [\Gamma^{\mathbf{q}_l}] \otimes [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon b : B + A]\!]; \delta^{-1})(w)\} \\
&\quad\approx \text{let } u = \text{iter } a\ \{\iota_r\ x : b\};\ (\text{pack}(\Gamma^{\mathbf{q}_l}), u)
\end{aligned}$$

and therefore that

$$\begin{aligned}
&\Gamma^{\mathbf{q}} \vdash_{\perp} \text{let } x = \text{pack}(\Gamma^{\mathbf{q}});\ [\![\Gamma^{\mathbf{q}} \vdash \text{iter } a\ \{\iota_r\ x : b\} : B]\!](x) \\
&\quad \approx \text{let } z = \text{pack}(\Gamma^{\mathbf{q}});\ ([\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; [\Gamma^{\mathbf{q}_l}] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon a : A]\!]; \\
&\qquad ([\![\Gamma \vdash \mathbf{q}_l = \mathbf{q}_l + \mathbf{q}_l]\!] \otimes A; \alpha; [\Gamma^{\mathbf{q}_l}] \otimes [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon b : B + A]\!]; \delta^{-1})^\dagger; \\
&\qquad [\![\Gamma^{\mathbf{q}_l} \mapsto \cdot]\!] \otimes B; \rho)(z) \\
&\quad \approx \text{let } z = (\text{pack}(\Gamma^{\mathbf{q}_l}), \text{pack}(\Gamma^{\mathbf{q}_r}));\ ([\Gamma^{\mathbf{q}_l}] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon a : A]\!]; \\
&\qquad ([\![\Gamma \vdash \mathbf{q}_l = \mathbf{q}_l + \mathbf{q}_l]\!] \otimes A; \alpha; [\Gamma^{\mathbf{q}_l}] \otimes [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon b : B + A]\!]; \delta^{-1})^\dagger; \\
&\qquad [\![\Gamma^{\mathbf{q}_l} \mapsto \cdot]\!] \otimes B; \rho)(z) \\
&\quad \approx \text{let } z = (\text{pack}(\Gamma^{\mathbf{q}_l}), a);\ (\\
&\qquad ([\![\Gamma \vdash \mathbf{q}_l = \mathbf{q}_l + \mathbf{q}_l]\!] \otimes A; \alpha; [\Gamma^{\mathbf{q}_l}] \otimes [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon b : B + A]\!]; \delta^{-1})^\dagger; \\
&\qquad [\![\Gamma^{\mathbf{q}_l} \mapsto \cdot]\!] \otimes B; \rho)(z) \\
&\quad \approx \text{let } z = (\text{let } y = (\text{pack}(\Gamma^{\mathbf{q}_l}), a); \\
&\qquad\quad ([\![\Gamma \vdash \mathbf{q}_l = \mathbf{q}_l + \mathbf{q}_l]\!] \otimes A; \alpha; [\Gamma^{\mathbf{q}_l}] \otimes [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon b : B + A]\!]; \delta^{-1})^\dagger(y)); \\
&\qquad ([\![\Gamma^{\mathbf{q}_l} \mapsto \cdot]\!] \otimes B; \rho)(z) \\
&\quad \approx \text{let } z = (\text{let } y = a;\ \text{iter } (\text{pack}(\Gamma^{\mathbf{q}_l}), y)\ \{\iota_r\ w : \\
&\qquad\quad ([\![\Gamma \vdash \mathbf{q}_l = \mathbf{q}_l + \mathbf{q}_l]\!] \otimes A; \alpha; [\Gamma^{\mathbf{q}_l}] \otimes [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon b : B + A]\!]; \delta^{-1})(w)\}); \\
&\qquad ([\![\Gamma^{\mathbf{q}_l} \mapsto \cdot]\!] \otimes B; \rho)(z) \\
&\quad \approx \text{let } z = (\text{let } u = \text{iter } a\ \{\iota_r\ x : b\};\ (\text{pack}(\Gamma^{\mathbf{q}_l}), u));\ ([\![\Gamma^{\mathbf{q}_l} \mapsto \cdot]\!] \otimes B; \rho)(z) \\
&\quad \approx \text{let } u = \text{iter } a\ \{\iota_r\ x : b\};\ \text{let } z = (\text{pack}(\Gamma^{\mathbf{q}_l}), u);\ ([\![\Gamma^{\mathbf{q}_l} \mapsto \cdot]\!] \otimes B; \rho)(z) \\
&\quad \approx \text{let } u = \text{iter } a\ \{\iota_r\ x : b\};\ \text{let } z = ((), u);\ \rho(z) \\
&\quad \approx \text{let } u = \text{iter } a\ \{\iota_r\ x : b\};\ u \approx \text{iter } a\ \{\iota_r\ x : b\} : B
\end{aligned}$$

as desired.

□

Completeness follows directly from the above lemma, since, given

$$\begin{aligned}
&[\![\Gamma^{\mathbf{q}} \vdash a : A]\!]_{\mathsf{Tm}(\mathcal{R})} \twoheadrightarrow [\![\Gamma^{\mathbf{q}} \vdash b : A]\!]_{\mathsf{Tm}(\mathcal{R})} \\
&\quad \Longrightarrow \forall (\lambda x.a_x) \in [\![\Gamma^{\mathbf{q}} \vdash a : A]\!]_{\mathsf{Tm}(\mathcal{R})}, (\lambda x.b_x) \in [\![\Gamma^{\mathbf{q}} \vdash b : A]\!]_{\mathsf{Tm}(\mathcal{R})}. x : [\Gamma^{\mathbf{q}}] \vdash_{\mathcal{R}} a_x \twoheadrightarrow b_x : A \\
&\quad \Longrightarrow \Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \text{let } x = \text{pack}(\Gamma^{\mathbf{q}});\ a_x \twoheadrightarrow \text{let } x = \text{pack}(\Gamma^{\mathbf{q}});\ b_x : A \\
&\quad \Longrightarrow \Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} a \twoheadrightarrow b : A
\end{aligned}$$

as desired.

C Compiling Expressions to SSA

In this section, we give a compilation function SSA_ℓ that compiles a λ_{iter} terms to an SSA program returning its result as an argument to output label ℓ. We will then show that this function is *semantics-preserving*, i.e. that it satisfies the following property:

$$[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \text{SSA}_\ell(\Gamma^{\mathbf{q}} \vdash_\epsilon a : A) \triangleright \ell(A)^{\mathbf{0}}]\!] = [\![\Gamma^{\mathbf{q}} \vdash_\epsilon a : A]\!]; \alpha; \iota_r$$

Here, the associator α takes $[\![A]\!]$ to $[\![\Gamma^{\mathbf{0}}]\!] \otimes [\![A]\!]$ (the latter being a tensor product of monoidal units), which ι_r then takes to $[\![[\Gamma \mapsto \ell(A)^{\mathbf{0}}]]\!] = \mathbf{0} + [\![\Gamma^{\mathbf{0}}]\!] \otimes [\![A]\!]$. We will do so by requiring the slightly

stronger property that, for all $\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r$, we have

$$[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \mathsf{SSA}_\ell(\Gamma^{\mathbf{q}_r} \vdash_\epsilon a : A) \rhd \ell(A)^{\mathbf{q}_l}]\!] = [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; [\![\Gamma^{\mathbf{q}_l}]\!] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon a : A]\!]; \iota_r$$

As this transformation is complicated to state and verify for general expressions, we will instead define *ANF*, and, after showing how every expression can be compiled to an equivalent expression in ANF, define SSA_ℓ by induction over such expressions.

C.1 ANF Expressions

We begin by defining ANF expressions P, Q, R using the grammar in Figure 26; these are treated as a subset of λ_{iter} expressions. In particular, an ANF program consists of a sequence of instructions of the form $\text{let } x = I;\ P$ (along with destructurings $\text{let } (x, y) = o;\ P$), where each instruction I either calls into a sub-program or returns the value of an operation o. Our goal is to define a function $\mathsf{ANF}(a)$ on terms such that, for all well-typed terms a,

$$\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \mathsf{ANF}(a) \approx a : A$$

Rather than do so directly, we will instead define a transformation $\text{let}_{\mathsf{ANF}}\ x = a; P$ which, given an ANF program P and a term a, returns an ANF program such that

$$\Gamma^{\mathbf{q}} \vdash_{\mathcal{R}} \text{let}_{\mathsf{ANF}}\ x = a; P \approx \text{let } x = a;\ P : A$$

We can then define $\mathsf{ANF}(a) := \text{let}_{\mathsf{ANF}}\ x = a; x$; note that we trivially have that, for any a, $\text{let}_{\mathsf{ANF}}\ x = a; x \approx \text{let } x = a;\ x \approx a \implies \mathsf{ANF}(x) \approx a$ by definition. We can now proceed to define $\text{let}_{\mathsf{ANF}}\ x = a; P$ by induction on terms a as follows:

$$\begin{aligned}
(\text{let}_{\mathsf{ANF}}\ x = o; P) &:= (\text{let } x = o;\ P) \text{ where } o \text{ is a valid instruction} \\
(\text{let}_{\mathsf{ANF}}\ x = f\ a; P) &:= (\text{let}_{\mathsf{ANF}}\ x_a = a; \text{let } x = f\ x_a;\ P) \\
(\text{let}_{\mathsf{ANF}}\ x = \text{let } y = a;\ b; P) &:= (\text{let}_{\mathsf{ANF}}\ y = a; \text{let}_{\mathsf{ANF}}\ x = b; P) \\
(\text{let}_{\mathsf{ANF}}\ x = (a, b); P) &:= (\text{let}_{\mathsf{ANF}}\ x_a = a; \text{let}_{\mathsf{ANF}}\ x_b = b; \text{let } x = (a, b);\ P) \\
(\text{let}_{\mathsf{ANF}}\ x = \iota_l\ a; P) &:= (\text{let}_{\mathsf{ANF}}\ x_a = a; \text{let } x = \iota_l\ x_a;\ P) \\
(\text{let}_{\mathsf{ANF}}\ x = \iota_r\ a; P) &:= (\text{let}_{\mathsf{ANF}}\ x_a = a; \text{let } x = \iota_r\ x_a;\ P) \\
(\text{let}_{\mathsf{ANF}}\ x = \text{case } e\ \{\iota_l\ y : a, \iota_r\ z : b\}; P) &:= (\text{let}_{\mathsf{ANF}}\ x_e = e; \text{let } x = \text{case } x_e \\
&\qquad \{\iota_l\ y : \mathsf{ANF}(a), \iota_r\ z : \mathsf{ANF}(b)\};\ P) \\
(\text{let}_{\mathsf{ANF}}\ x = \text{abort } a; P) &:= (\text{let}_{\mathsf{ANF}}\ x_a = a; \text{let } x = \text{abort } x_a;\ P) \\
(\text{let}_{\mathsf{ANF}}\ x = \text{iter } a\ \{\iota_r\ y : b\}; P) &:= (\text{let}_{\mathsf{ANF}}\ x_a = a; \text{let } x = \text{iter } x_a\ \{\iota_r\ y : \mathsf{ANF}(b)\};\ P)
\end{aligned}$$

We can trivially verify each case by simply applying the binding rule for each term-former, followed by the inductive hypothesis. For example, we have that

$$\begin{aligned}
\Gamma^{\mathbf{q}} \vdash_\perp \text{let } x = \text{iter } a\ \{\iota_r\ y : b\};\ P &\approx \text{let } x = (\text{let } x_a = a;\ \text{iter } x_a\ \{\iota_r\ y : b\});\ P && \text{(iter-bind)} \\
&\approx \text{let } x_a = a;\ \text{let } x = \text{iter } x_a\ \{\iota_r\ y : b\};\ P && \text{(let-let}_1\text{)} \\
&\approx \text{let}_{\mathsf{ANF}}\ x_a = a; \text{let } x = \text{iter } x_a\ \{\iota_r\ y : b\};\ P && \text{(by induction)} \\
&\approx \text{let}_{\mathsf{ANF}}\ x_a = a; \text{let } x = \text{iter } x_a\ \{\iota_r\ y : \mathsf{ANF}(b)\};\ P && \text{(by induction)} \\
&\approx \text{let}_{\mathsf{ANF}}\ x = \text{iter } a\ \{\iota_r\ y : b\}; P && \text{(by definition)}
\end{aligned}$$

$$I, J ::= o \mid \mathsf{case}\ o\ \{\iota_l\ x : P, \iota_r\ y : Q\} \mid \mathsf{iter}\ o\ \{\iota_r\ x : P\}$$

$$P, Q, R ::= o \mid \mathsf{let}\ x = I;\ P \mid \mathsf{let}\ (x, y) = o;\ P$$

$$q ::= 0 \mid 1 \mid \omega^+ \mid 1^? \mid \omega$$

$$\Gamma ::= \cdot \mid \Gamma, x : A$$

$$\mathbf{q} ::= \cdot \mid \mathbf{q}, q$$

Fig. 26. Syntax for λ_{iter} terms in ANF

C.2 ANF to SSA

We begin by stating a few metatheoretic results:

Lemma C.1 (Label Weakening). *Given a label-weakening* $\Gamma \vdash \mathsf{L}^{\mathbf{Q}} \rightsquigarrow \mathsf{L}'^{\mathbf{Q}'}$,

- *If* $\Gamma \vdash \mathsf{L}'^{\mathbf{Q}'} \rightsquigarrow \mathsf{L}''^{\mathbf{Q}''}$, *then* $\Gamma \vdash \mathsf{L}^{\mathbf{Q}} \rightsquigarrow \mathsf{L}''^{\mathbf{Q}''}$ *with*
$$[\![\Gamma \vdash \mathsf{L}^{\mathbf{Q}} \rightsquigarrow \mathsf{L}''^{\mathbf{Q}''}]\!] = [\![\Gamma \vdash \mathsf{L}^{\mathbf{Q}} \rightsquigarrow \mathsf{L}'^{\mathbf{Q}'}]\!]; [\![lwk\Gamma \mathsf{L}'^{\mathbf{Q}'} \mathsf{L}''^{\mathbf{Q}''}]\!]$$
- *If* $\Gamma^{\mathbf{q}} \vdash_\epsilon s \rhd \mathsf{L}^{\mathbf{Q}}$ *and , we have that* $\Gamma^{\mathbf{q}} \vdash_\epsilon s \rhd \mathsf{L}'^{\mathbf{Q}'}$ *with*
$$[\![\Gamma^{\mathbf{q}} \vdash_\epsilon s \rhd \mathsf{L}'^{\mathbf{Q}'}]\!] = [\![\Gamma^{\mathbf{q}} \vdash_\epsilon s \rhd \mathsf{L}^{\mathbf{Q}}]\!]; [\![\Gamma \vdash \mathsf{L}^{\mathbf{Q}} \rightsquigarrow \mathsf{L}'^{\mathbf{Q}'}]\!]$$

Proof. We begin by noting that, via a straightforward induction, we have that

$$[\![\Gamma \vdash \mathsf{L}^{\mathbf{Q}} \rightsquigarrow \mathsf{L}''^{\mathbf{Q}''}]\!] = [\![\Gamma \vdash \mathsf{L}^{\mathbf{Q}} \rightsquigarrow \mathsf{L}'^{\mathbf{Q}'}]\!]; [\![lwk\Gamma \mathsf{L}'^{\mathbf{Q}'} \mathsf{L}''^{\mathbf{Q}''}]\!]$$

and

$$[\![\Gamma, x : A \vdash \mathsf{L}^{\mathbf{Q}^\uparrow} \rightsquigarrow \mathsf{L}'^{\mathbf{Q}'^\uparrow}]\!]; \alpha^\downarrow = \alpha^\downarrow; [\![\Gamma \vdash \mathsf{L}^{\mathbf{Q}} \rightsquigarrow \mathsf{L}'^{\mathbf{Q}'}]\!]$$

We now proceed by induction on the derivation of $\Gamma^{\mathbf{q}} \vdash_\epsilon s \rhd \mathsf{L}^{\mathbf{Q}}$:

- $(\Gamma^{\mathbf{q}} \vdash_\epsilon \mathsf{br}\ \ell\ a \rhd \mathsf{L}^{\mathbf{Q}})$: we have
$$\begin{aligned}
&[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \mathsf{br}\ \ell\ a \rhd \mathsf{L}'^{\mathbf{Q}'}]\!] \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; [\![\Gamma^{\mathbf{q}_l}]\!] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon a : A]\!]; \iota_r; [\![\Gamma \vdash \ell(A)^{\mathbf{q}_l} \rightsquigarrow \mathsf{L}'^{\mathbf{q}'}]\!] \\
&\approx [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; [\![\Gamma^{\mathbf{q}_l}]\!] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon a : A]\!]; \iota_r; [\![\Gamma \vdash \ell(A)^{\mathbf{q}_l} \rightsquigarrow \mathsf{L}^{\mathbf{q}}]\!]; [\![\Gamma \vdash \mathsf{L}^{\mathbf{Q}} \rightsquigarrow \mathsf{L}'^{\mathbf{Q}'}]\!] \\
&\approx [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \mathsf{br}\ \ell\ a \rhd \mathsf{L}^{\mathbf{Q}}]\!]; [\![\Gamma \vdash \mathsf{L}^{\mathbf{Q}} \rightsquigarrow \mathsf{L}'^{\mathbf{Q}'}]\!]
\end{aligned}$$
- $(\Gamma^{\mathbf{q}} \vdash_\epsilon \mathsf{let}\ x = o; t \rhd \mathsf{L}^{\mathbf{Q}})$: we have by induction that
$$\begin{aligned}
&[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \mathsf{let}\ x = o; t \rhd \mathsf{L}'^{\mathbf{Q}'}]\!] \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; [\![\Gamma^{\mathbf{q}_l}]\!] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon o : A]\!]; [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon t \rhd \mathsf{L}'^{\mathbf{Q}'}]\!] \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; [\![\Gamma^{\mathbf{q}_l}]\!] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon o : A]\!]; [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon t \rhd \mathsf{L}^{\mathbf{Q}}]\!]; [\![\Gamma \vdash \mathsf{L}^{\mathbf{Q}} \rightsquigarrow \mathsf{L}'^{\mathbf{Q}'}]\!] \\
&= [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \mathsf{let}\ x = o; t \rhd \mathsf{L}^{\mathbf{Q}}]\!]; [\![\Gamma \vdash \mathsf{L}^{\mathbf{Q}} \rightsquigarrow \mathsf{L}'^{\mathbf{Q}'}]\!]
\end{aligned}$$
- $(\Gamma^{\mathbf{q}} \vdash_\epsilon \mathsf{let}\ (x, y) = o; t \rhd \mathsf{L}^{\mathbf{Q}})$: we have by induction that

$$\begin{aligned}
&[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \mathsf{let}\ x = o; t \rhd \mathsf{L}'^{\mathbf{Q}'}]\!] \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; [\![\Gamma^{\mathbf{q}_l}]\!] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon o : A \otimes B]\!]; \alpha; [\![\Gamma^{\mathbf{q}_l}, x : A, y : B \vdash_\epsilon t \rhd \mathsf{L}'^{\mathbf{Q}'}]\!] \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; [\![\Gamma^{\mathbf{q}_l}]\!] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon o : A \otimes B]\!]; \alpha; [\![\Gamma^{\mathbf{q}_l}, x : A, y : B \vdash_\epsilon t \rhd \mathsf{L}^{\mathbf{Q}}]\!]; [\![\Gamma \vdash \mathsf{L}^{\mathbf{Q}} \rightsquigarrow \mathsf{L}'^{\mathbf{Q}'}]\!] \\
&= [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \mathsf{let}\ (x, y) = o; t \rhd \mathsf{L}^{\mathbf{Q}}]\!]; [\![\Gamma \vdash \mathsf{L}^{\mathbf{Q}} \rightsquigarrow \mathsf{L}'^{\mathbf{Q}'}]\!]
\end{aligned}$$

- $(\Gamma^{\mathbf{q}} \vdash_\epsilon \text{case } o\ \{\iota_l\ x : \tau_l, \iota_r\ y : \tau_r\} \triangleright \mathsf{L}^{\mathsf{Q}})$: we have by induction that

$$
\begin{aligned}
&[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \text{case } o\ \{\iota_l\ x : \tau_l, \iota_r\ y : \tau_r\} \triangleright \mathsf{L}'^{\mathsf{Q}'}]\!] \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; [\![\Gamma^{\mathbf{q}_l}]\!] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon o : A + B]\!]; \\
&\quad [[\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon \tau_l : \mathsf{L}'^{\mathsf{Q}'^{\uparrow}}]\!]; \alpha^{\downarrow}, [\![\Gamma^{\mathbf{q}_l}, y : B \vdash_\epsilon \tau_r : \mathsf{L}'^{\mathsf{Q}'^{\uparrow}}]\!]; \alpha^{\downarrow}] \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; [\![\Gamma^{\mathbf{q}_l}]\!] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon o : A + B]\!]; [\\
&\quad [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon \tau_l : \mathsf{L}^{\mathsf{Q}^{\uparrow}}]\!]; [\![\Gamma, x : A \vdash \mathsf{L}^{\mathsf{Q}^{\uparrow}} \rightsquigarrow \mathsf{L}'^{\mathsf{Q}'^{\uparrow}}]\!]; \alpha^{\downarrow}, \\
&\quad [\![\Gamma^{\mathbf{q}_l}, y : B \vdash_\epsilon \tau_r : \mathsf{L}^{\mathsf{Q}^{\uparrow}}]\!]; [\![\Gamma, y : B \vdash \mathsf{L}^{\mathsf{Q}^{\uparrow}} \rightsquigarrow \mathsf{L}'^{\mathsf{Q}'^{\uparrow}}]\!]; \alpha^{\downarrow}] \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; [\![\Gamma^{\mathbf{q}_l}]\!] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon o : A + B]\!]; [\\
&\quad [\![\Gamma^{\mathbf{q}_l}, x : A \vdash_\epsilon \tau_l : \mathsf{L}^{\mathsf{Q}^{\uparrow}}]\!]; \alpha^{\downarrow}; [\![\Gamma \vdash \mathsf{L}^{\mathsf{Q}^{\uparrow}} \rightsquigarrow \mathsf{L}^{\mathsf{Q}'^{\uparrow}}]\!], [\![\Gamma^{\mathbf{q}_l}, y : B \vdash_\epsilon \tau_r : \mathsf{L}^{\mathsf{Q}^{\uparrow}}]\!]; \alpha^{\downarrow}; [\![\Gamma \vdash \mathsf{L}^{\mathsf{Q}} \rightsquigarrow \mathsf{L}'^{\mathsf{Q}'}]\!]] \\
&= [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \text{case } o\ \{\iota_l\ x : \tau_l, \iota_r\ y : \tau_r\} \triangleright \mathsf{L}^{\mathsf{Q}}]\!]; [\![\Gamma \vdash \mathsf{L}^{\mathsf{Q}} \rightsquigarrow \mathsf{L}'^{\mathsf{Q}'}]\!]
\end{aligned}
$$

- $(\Gamma^{\mathbf{q}} \vdash_\epsilon \kappa\ \text{where}_{\text{nonrec}}\ (\ell_i(x_i) : \{t_i\},)_i \triangleright \mathsf{L}^{\mathsf{Q}})$: we have

$$
\begin{aligned}
&[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \kappa\ \text{where}_{\text{nonrec}}\ (\ell_i(x_i) : \{t_i\},)_i \triangleright \mathsf{L}'^{\mathsf{Q}'}]\!] \\
&= [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \kappa \triangleright \mathsf{L}'^{\mathsf{Q}'}, \mathsf{R}^{\mathsf{Q}''}]\!]; \alpha^{+}; [\mathrm{id}_{[\![[\Gamma \mapsto \mathsf{L}'^{\mathsf{Q}'}]]\!]}, [[\![\Gamma^{\mathbf{q}_i}, x_i : A_i \vdash_\epsilon t_i : \mathsf{L}'^{\mathsf{Q}'^{\uparrow}}]\!]; \alpha^{\downarrow},]_{\ell_i(A_i)^{\mathbf{q}_i} \in \mathsf{R}^{\mathsf{Q}''}}] \\
&= [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \kappa \triangleright \mathsf{L}'^{\mathsf{Q}'}, \mathsf{R}^{\mathsf{Q}''}]\!]; [\![\Gamma \vdash \mathsf{L}^{\mathsf{Q}}, \mathsf{R}^{\mathsf{Q}''} \rightsquigarrow \mathsf{L}'^{\mathsf{Q}'}, \mathsf{R}^{\mathsf{Q}''}]\!]; \alpha^{+}; \\
&\quad [\mathrm{id}_{[\![[\Gamma \mapsto \mathsf{L}'^{\mathsf{Q}'}]]\!]}, [[\![\Gamma^{\mathbf{q}_i}, x_i : A_i \vdash_\epsilon t_i : \mathsf{L}^{\mathsf{Q}^{\uparrow}}]\!]; [\![\Gamma, x_i : A_i \vdash \mathsf{L}^{\mathsf{Q}^{\uparrow}} \rightsquigarrow \mathsf{L}'^{\mathsf{Q}'^{\uparrow}}]\!]; \alpha^{\downarrow},]_{\ell_i(A_i)^{\mathbf{q}_i} \in \mathsf{R}^{\mathsf{Q}''}}] \\
&= [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \kappa \triangleright \mathsf{L}'^{\mathsf{Q}'}, \mathsf{R}^{\mathsf{Q}''}]\!]; \alpha^{+}; [\![\Gamma \vdash \mathsf{L}^{\mathsf{Q}} \rightsquigarrow \mathsf{L}'^{\mathsf{Q}'}]\!] + [\![[\Gamma \mapsto \mathsf{R}^{\mathsf{Q}''}]]\!]; \\
&\quad [\mathrm{id}_{[\![[\Gamma \mapsto \mathsf{L}'^{\mathsf{Q}'}]]\!]}, [[\![\Gamma^{\mathbf{q}_i}, x_i : A_i \vdash_\epsilon t_i : \mathsf{L}^{\mathsf{Q}^{\uparrow}}]\!]; \alpha^{\downarrow}; [\![\Gamma \vdash \mathsf{L}^{\mathsf{Q}} \rightsquigarrow \mathsf{L}'^{\mathsf{Q}'}]\!],]_{\ell_i(A_i)^{\mathbf{q}_i} \in \mathsf{R}^{\mathsf{Q}''}}] \\
&= [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \kappa \triangleright \mathsf{L}'^{\mathsf{Q}'}, \mathsf{R}^{\mathsf{Q}''}]\!]; \alpha^{+}; \\
&\quad [[\![\Gamma \vdash \mathsf{L}^{\mathsf{Q}} \rightsquigarrow \mathsf{L}'^{\mathsf{Q}'}]\!], [[\![\Gamma^{\mathbf{q}_i}, x_i : A_i \vdash_\epsilon t_i : \mathsf{L}^{\mathsf{Q}^{\uparrow}}]\!]; \alpha^{\downarrow},]_{\ell_i(A_i)^{\mathbf{q}_i} \in \mathsf{R}^{\mathsf{Q}''}}; [\![\Gamma \vdash \mathsf{L}^{\mathsf{Q}} \rightsquigarrow \mathsf{L}'^{\mathsf{Q}'}]\!]] \\
&= [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \kappa \triangleright \mathsf{L}'^{\mathsf{Q}'}, \mathsf{R}^{\mathsf{Q}''}]\!]; \alpha^{+}; [[\![\mathrm{id}, [[\![\Gamma^{\mathbf{q}_i}, x_i : A_i \vdash_\epsilon t_i : \mathsf{L}^{\mathsf{Q}^{\uparrow}}]\!]; \alpha^{\downarrow},]_{\ell_i(A_i)^{\mathbf{q}_i} \in \mathsf{R}^{\mathsf{Q}''}}]; \Gamma \vdash \mathsf{L}^{\mathsf{Q}} \rightsquigarrow \mathsf{L}'^{\mathsf{Q}'}]\!] \\
&= [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \kappa\ \text{where}_{\text{nonrec}}\ (\ell_i(x_i) : \{t_i\},)_i \triangleright \mathsf{L}^{\mathsf{Q}}]\!]; [\![\Gamma \vdash \mathsf{L}^{\mathsf{Q}} \rightsquigarrow \mathsf{L}'^{\mathsf{Q}'}]\!]
\end{aligned}
$$

as desired.

- $(\Gamma^{\mathbf{q}} \vdash_\epsilon \kappa \text{ where}_{\text{rec}}\ (\ell_i(x_i) : \{t_i\},)_i \rhd \mathsf{L}^{\mathbb{Q}})$: we have

$$
\begin{aligned}
&[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \kappa \text{ where}_{\text{rec}}\ (\ell_i(x_i) : \{t_i\},)_i \rhd \mathsf{L}'^{\mathbb{Q}'}]\!] \\
&= [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \kappa \rhd \mathsf{L}'^{\mathbb{Q}'}, \mathsf{R}^{\mathbb{Q}''}]\!]; \alpha^+; [\mathsf{id}_{[\![[\Gamma \mapsto \mathsf{L}'^{\mathbb{Q}'}]]\!]}, [[\![\Gamma^{\mathbf{q}_i}, x_i : A_i \vdash_\epsilon t_i : \mathsf{L}'^{\mathbb{Q}'^\uparrow}, \mathsf{R}^{\mathbb{Q}''^\uparrow}]\!]; \alpha^\downarrow; \alpha^+,]^\dagger_{\ell_i(A_i)^{\mathbf{q}_i} \in \mathsf{R}^{\mathbb{Q}'}}] \\
&= [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \kappa \rhd \mathsf{L}'^{\mathbb{Q}'}, \mathsf{R}^{\mathbb{Q}''}]\!]; [\![\Gamma \vdash \mathsf{L}^{\mathbb{Q}}, \mathsf{R}^{\mathbb{Q}''} \rightsquigarrow \mathsf{L}'^{\mathbb{Q}'}, \mathsf{R}^{\mathbb{Q}''}]\!]; \alpha^+; \\
&\quad [\mathsf{id}_{[\![[\Gamma \mapsto \mathsf{L}'^{\mathbb{Q}'}]]\!]}, [[\![\Gamma^{\mathbf{q}_i}, x_i : A_i \vdash_\epsilon t_i : \mathsf{L}^{\mathbb{Q}^\uparrow}, \mathsf{R}^{\mathbb{Q}''^\uparrow}]\!]; [\![\Gamma \vdash \mathsf{L}^{\mathbb{Q}}, \mathsf{R}^{\mathbb{Q}''} \rightsquigarrow \mathsf{L}'^{\mathbb{Q}'}, \mathsf{R}^{\mathbb{Q}''}]\!]; \alpha^\downarrow; \alpha^+,]^\dagger_{\ell_i(A_i)^{\mathbf{q}_i} \in \mathsf{R}^{\mathbb{Q}'}}] \\
&= [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \kappa \rhd \mathsf{L}'^{\mathbb{Q}'}, \mathsf{R}^{\mathbb{Q}''}]\!]; [\![\Gamma \vdash \mathsf{L}^{\mathbb{Q}}, \mathsf{R}^{\mathbb{Q}''} \rightsquigarrow \mathsf{L}'^{\mathbb{Q}'}, \mathsf{R}^{\mathbb{Q}''}]\!]; \alpha^+; \\
&\quad [\mathsf{id}_{[\![[\Gamma \mapsto \mathsf{L}'^{\mathbb{Q}'}]]\!]}, [[\![\Gamma^{\mathbf{q}_i}, x_i : A_i \vdash_\epsilon t_i : \mathsf{L}^{\mathbb{Q}^\uparrow}, \mathsf{R}^{\mathbb{Q}''^\uparrow}]\!]; [\![\Gamma \vdash \mathsf{L}^{\mathbb{Q}}, \mathsf{R}^{\mathbb{Q}''} \rightsquigarrow \mathsf{L}'^{\mathbb{Q}'}, \mathsf{R}^{\mathbb{Q}''}]\!]; \alpha^\downarrow; \alpha^+,]^\dagger_{\ell_i(A_i)^{\mathbf{q}_i} \in \mathsf{R}^{\mathbb{Q}'}}] \\
&= [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \kappa \rhd \mathsf{L}'^{\mathbb{Q}'}, \mathsf{R}^{\mathbb{Q}''}]\!]; \alpha^+; [\![\Gamma \vdash \mathsf{L}^{\mathbb{Q}} \rightsquigarrow \mathsf{L}'^{\mathbb{Q}'}]\!] + [\![[\Gamma \mapsto \mathsf{R}^{\mathbb{Q}''}]]\!]; [\mathsf{id}_{[\![[\Gamma \mapsto \mathsf{L}'^{\mathbb{Q}'}]]\!]}, \\
&\quad [[\![\Gamma^{\mathbf{q}_i}, x_i : A_i \vdash_\epsilon t_i : \mathsf{L}^{\mathbb{Q}^\uparrow}, \mathsf{R}^{\mathbb{Q}''^\uparrow}]\!]; \alpha^\downarrow; \alpha^+, ; [\![\Gamma \vdash \mathsf{L}^{\mathbb{Q}} \rightsquigarrow \mathsf{L}'^{\mathbb{Q}'}]\!] + [\![[\Gamma \mapsto \mathsf{R}^{\mathbb{Q}''}]]\!]]^\dagger_{\ell_i(A_i)^{\mathbf{q}_i} \in \mathsf{R}^{\mathbb{Q}'}}] \\
&= [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \kappa \rhd \mathsf{L}'^{\mathbb{Q}'}, \mathsf{R}^{\mathbb{Q}''}]\!]; \alpha^+; [[\![\Gamma \vdash \mathsf{L}^{\mathbb{Q}} \rightsquigarrow \mathsf{L}'^{\mathbb{Q}'}]\!], \\
&\quad ([[\![\Gamma^{\mathbf{q}_i}, x_i : A_i \vdash_\epsilon t_i : \mathsf{L}^{\mathbb{Q}^\uparrow}, \mathsf{R}^{\mathbb{Q}''^\uparrow}]\!]; \alpha^\downarrow; \alpha^+,]_{\ell_i(A_i)^{\mathbf{q}_i} \in \mathsf{R}^{\mathbb{Q}'}}; [\![\Gamma \vdash \mathsf{L}^{\mathbb{Q}} \rightsquigarrow \mathsf{L}'^{\mathbb{Q}'}]\!] + [\![[\Gamma \mapsto \mathsf{R}^{\mathbb{Q}''}]]\!])^\dagger] \\
&= [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \kappa \rhd \mathsf{L}'^{\mathbb{Q}'}, \mathsf{R}^{\mathbb{Q}''}]\!]; \alpha^+; [[\![\Gamma \vdash \mathsf{L}^{\mathbb{Q}} \rightsquigarrow \mathsf{L}'^{\mathbb{Q}'}]\!], \\
&\quad [[\![\Gamma^{\mathbf{q}_i}, x_i : A_i \vdash_\epsilon t_i : \mathsf{L}^{\mathbb{Q}^\uparrow}, \mathsf{R}^{\mathbb{Q}''^\uparrow}]\!]; \alpha^\downarrow; \alpha^+,]^\dagger_{\ell_i(A_i)^{\mathbf{q}_i} \in \mathsf{R}^{\mathbb{Q}'}}; [\![\Gamma \vdash \mathsf{L}^{\mathbb{Q}} \rightsquigarrow \mathsf{L}'^{\mathbb{Q}'}]\!]] \\
&= [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \kappa \rhd \mathsf{L}'^{\mathbb{Q}'}, \mathsf{R}^{\mathbb{Q}''}]\!]; \alpha^+; [\mathsf{id}, [[\![\Gamma^{\mathbf{q}_i}, x_i : A_i \vdash_\epsilon t_i : \mathsf{L}^{\mathbb{Q}^\uparrow}, \mathsf{R}^{\mathbb{Q}''^\uparrow}]\!]; \alpha^\downarrow; \alpha^+,]^\dagger_{\ell_i(A_i)^{\mathbf{q}_i} \in \mathsf{R}^{\mathbb{Q}'}}]; \\
&\quad [\![\Gamma \vdash \mathsf{L}^{\mathbb{Q}} \rightsquigarrow \mathsf{L}'^{\mathbb{Q}'}]\!] \\
&= [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \kappa \text{ where}_{\text{rec}}\ (\ell_i(x_i) : \{t_i\},)_i \rhd \mathsf{L}^{\mathbb{Q}}]\!]; [\![\Gamma \vdash \mathsf{L}^{\mathbb{Q}} \rightsquigarrow \mathsf{L}'^{\mathbb{Q}'}]\!]
\end{aligned}
$$

as desired.

□

We may now define $\mathrm{SSA}_\ell(P)$ by induction on ANF programs P as follows:

- (Valid operations o): we define $(\mathrm{SSA}_\ell(o)) := \text{br}\ \ell\ o$. This has the desired semantics, since

$$[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \text{br}\ \ell\ o \rhd \ell(A)^{\mathbf{0}}]\!] = [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_l + \mathbf{q}_r]\!]; [\![\Gamma^{\mathbf{q}_l}]\!] \otimes [\![\Gamma^{\mathbf{q}_r} \vdash_\epsilon o : A]\!]; \iota_r; \cancel{[\![\Gamma \vdash \ell(A)^{\mathbf{q}_l} \rightsquigarrow \ell(A)^{\mathbf{q}_l}]\!]}$$

- (let $x = o;\ P$): we define $(\mathrm{SSA}_\ell(\text{let}\ x = o;\ P)) := \text{let}\ x = o; \mathrm{SSA}_\ell(P)$; we verify this has the correct semantics by induction as follows:

$$
\begin{aligned}
&[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \mathrm{SSA}_\ell(\text{let}\ x = o;\ P) \rhd \ell(B)^{\mathbf{q}_1}]\!] \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_{12} + \mathbf{q}_3]\!]; [\![\Gamma^{\mathbf{q}_{12}}]\!] \otimes [\![\Gamma^{\mathbf{q}_3} \vdash_\epsilon o : A]\!]; [\![\Gamma^{\mathbf{q}_{12}}, x : A \vdash_\epsilon \mathrm{SSA}_\ell(P) \rhd \ell(B)^{\mathbf{q}_1, 0}]\!]; \alpha^\downarrow \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_{12} + \mathbf{q}_3]\!]; [\![\Gamma^{\mathbf{q}_{12}}]\!] \otimes [\![\Gamma^{\mathbf{q}_3} \vdash_\epsilon o : A]\!]\ ; [\![\Gamma \vdash \mathbf{q}_{12}, x : A = \mathbf{q}_1, 0 + \mathbf{q}_2, \omega]\!] \\
&\quad ; ([\![\Gamma^{\mathbf{q}_1}]\!] \otimes I) \otimes [\![\Gamma^{\mathbf{q}_2}, x : A \vdash_\epsilon P : B]\!]; \iota_r; \alpha^\downarrow \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_{12} + \mathbf{q}_3]\!]; [\![\Gamma \vdash \mathbf{q}_{12} = \mathbf{q}_1 + \mathbf{q}_2]\!] \otimes [\![\Gamma^{\mathbf{q}_3} \vdash_\epsilon o : A]\!]; \alpha; [\![\Gamma^{\mathbf{q}_1}]\!] \otimes [\![\Gamma^{\mathbf{q}_2}, x : A \vdash_\epsilon P : B]\!]; \iota_r \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_1 + \mathbf{q}_{23}]\!]; [\![\Gamma^{\mathbf{q}_1}]\!] \otimes ([\![\Gamma \vdash \mathbf{q}_{23} = \mathbf{q}_2 + \mathbf{q}_3]\!]; [\![\Gamma^{\mathbf{q}_3}]\!] \otimes [\![\Gamma^{\mathbf{q}_3} \vdash_\epsilon o : A]\!]; [\![\Gamma^{\mathbf{q}_2}, x : A \vdash_\epsilon P : B]\!]); \iota_r \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_1 + \mathbf{q}_{23}]\!]; [\![\Gamma^{\mathbf{q}_1}]\!] \otimes ([\![\Gamma^{\mathbf{q}_{23}} \vdash_\epsilon \text{let}\ x = o;\ P : B]\!]); \iota_r
\end{aligned}
$$

as desired.

- $(\text{let } (x, y) = o;\ P)$: we define $(\text{SSA}_\ell(\text{let } (x, y) = o;\ P)) := \text{let } (x, y) = o; \text{SSA}_\ell(P)$; we verify this has the correct semantics by induction as follows:

$$
\begin{aligned}
&[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \text{SSA}_\ell(\text{let } (x, y) = o;\ P) \rhd \ell(B)^{\mathbf{q}_1}]\!] \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_{12} + \mathbf{q}_3]\!]; [\![\Gamma^{\mathbf{q}_{12}}]\!] \otimes [\![\Gamma^{\mathbf{q}_3} \vdash_\epsilon o : A \otimes B]\!]; [\![\Gamma^{\mathbf{q}_{12}}, x : A, y : B \vdash_\epsilon \text{SSA}_\ell(P) \rhd \ell(C)^{\mathbf{q}_1,0,0}]\!]; \alpha^{\downarrow}; \alpha^{\downarrow} \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_{12} + \mathbf{q}_3]\!]; [\![\Gamma^{\mathbf{q}_{12}}]\!] \otimes [\![\Gamma^{\mathbf{q}_3} \vdash_\epsilon o : A \otimes B]\!]\ ; [\![\Gamma \vdash \mathbf{q}_{12}, x : A, y : B = \mathbf{q}_1, 0, 0 + \mathbf{q}_2, \omega, \omega]\!] \\
&\quad ; ([\![\Gamma^{\mathbf{q}_1}]\!] \otimes I \otimes I) \otimes [\![\Gamma^{\mathbf{q}_2}, x : A, y : B \vdash_\epsilon P : C]\!]; \iota_r; \alpha^{\downarrow}; \alpha^{\downarrow} \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_{12} + \mathbf{q}_3]\!]; [\![\Gamma \vdash \mathbf{q}_{12} = \mathbf{q}_1 + \mathbf{q}_2]\!] \otimes [\![\Gamma^{\mathbf{q}_3} \vdash_\epsilon o : A \otimes B]\!]; \alpha \\
&\quad ; [\![\Gamma^{\mathbf{q}_1}]\!] \otimes [\![\Gamma^{\mathbf{q}_2}, x : A, y : B \vdash_\epsilon P : C]\!]; \iota_r \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_1 + \mathbf{q}_{23}]\!]; [\![\Gamma^{\mathbf{q}_1}]\!] \otimes \\
&\qquad ([\![\Gamma \vdash \mathbf{q}_{23} = \mathbf{q}_2 + \mathbf{q}_3]\!]; [\![\Gamma^{\mathbf{q}_3}]\!] \otimes [\![\Gamma^{\mathbf{q}_3} \vdash_\epsilon o : A \otimes B]\!]; \alpha; [\![\Gamma^{\mathbf{q}_2}, x : A, y : B \vdash_\epsilon P : C]\!]); \iota_r \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_1 + \mathbf{q}_{23}]\!]; [\![\Gamma^{\mathbf{q}_1}]\!] \otimes ([\![\Gamma^{\mathbf{q}_{23}} \vdash_\epsilon \text{let } x = o;\ P : C]\!]); \iota_r
\end{aligned}
$$

- $(\Gamma^{\mathbf{q}} \vdash_\epsilon \text{let } x = \text{case } o\ \{\iota_l\ y : P, \iota_r\ z : Q\};\ R : D)$: we define

$$
\begin{aligned}
(\text{SSA}_\ell(\text{let } x = \text{case } o\ \{\iota_l\ y : P, \iota_r\ z : Q\};\ R)) := (&\text{case } o\ \{\iota_l\ y : \text{br } \ell_l\ y, \iota_r\ z : \text{br } \ell_r\ z\} \\
&\text{where}_{\text{nonrec}} \\
&\quad \ell_l(y) : \{\text{SSA}_{\ell_o}(P)\}, \\
&\quad \ell_r(z) : \{\text{SSA}_{\ell_o}(Q)\}) \\
&\text{where}_{\text{nonrec}} \\
&\quad \ell_o(x) : \{\text{SSA}_\ell(R)\}
\end{aligned}
$$

We verify the semantics by induction as follows:

$$
\begin{aligned}
&[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \mathsf{SSA}_\ell(\mathsf{let}\ x = \mathsf{case}\ o\ \{\iota_l\ y : P, \iota_r\ z : Q\};\ R) \rhd \ell(D)^{\mathbf{q}_1}]\!] \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_{123} + \mathbf{q}_4]\!]; [\![\Gamma^{\mathbf{q}_{123}}]\!] \otimes [\![\Gamma^{\mathbf{q}_4} \vdash_\epsilon o : A + B]\!]; \delta^{-1}; [\\
&\qquad [\![\Gamma^{\mathbf{q}_{123}}, y : A \vdash_\epsilon \mathsf{br}\ \ell_l\ y \rhd \ell(D)^{\mathbf{q}_1,0}, \ell_o(C)^{\mathbf{q}_{12},0}, \ell_l(A)^{\mathbf{q}_{123},0}, \ell_r(B)^{\mathbf{q}_{123},0}]\!]; \alpha^{\downarrow}, \\
&\qquad [\![\Gamma^{\mathbf{q}_{123}}, z : B \vdash_\epsilon \mathsf{br}\ \ell_r\ z \rhd \ell(D)^{\mathbf{q}_1,0}, \ell_o(C)^{\mathbf{q}_{12},0}, \ell_l(A)^{\mathbf{q}_{123},0}, \ell_r(B)^{\mathbf{q}_{123},0}]\!]; \alpha^{\downarrow} \\
&\quad]; [\mathsf{id}, \\
&\qquad [\![\Gamma^{\mathbf{q}_{123}}, y : A \vdash_\epsilon \mathsf{SSA}_{\ell_o}(P) \rhd \ell(D)^{\mathbf{q}_1,0}, \ell_o(C)^{\mathbf{q}_{12},0}]\!]; \alpha^{\downarrow}, \\
&\qquad [\![\Gamma^{\mathbf{q}_{123}}, z : B \vdash_\epsilon \mathsf{SSA}_{\ell_o}(Q) \rhd \ell(D)^{\mathbf{q}_1,0}, \ell_o(C)^{\mathbf{q}_{12},0}]\!]; \alpha^{\downarrow} \\
&\quad]; [\mathsf{id}, [\![\Gamma^{\mathbf{q}_{12}}, x : C \vdash_\epsilon \mathsf{SSA}_\ell(R) \rhd \ell(D)^{\mathbf{q}_1,0}]\!]; \alpha^{\downarrow}] \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_{123} + \mathbf{q}_4]\!]; [\![\Gamma^{\mathbf{q}_{123}}]\!] \otimes [\![\Gamma^{\mathbf{q}_4} \vdash_\epsilon o : A + B]\!]; \delta^{-1}; [\iota_l; \iota_r, \iota_r]; [\mathsf{id}, \\
&\qquad [\![\Gamma^{\mathbf{q}_{123}}, y : A \vdash_\epsilon \mathsf{SSA}_{\ell_o}(P) \rhd \ell_o(C)^{\mathbf{q}_{12},0}]\!]; [\![\Gamma \vdash \ell_o(C)^{\mathbf{q}_{12},0} \rightsquigarrow \ell(D)^{\mathbf{q}_1,0}, \ell_o(C)^{\mathbf{q}_{12},0}]\!]; \alpha^{\downarrow}, \\
&\qquad [\![\Gamma^{\mathbf{q}_{123}}, z : B \vdash_\epsilon \mathsf{SSA}_{\ell_o}(Q) \rhd \ell_o(C)^{\mathbf{q}_{12},0}]\!]; [\![\Gamma \vdash \ell_o(C)^{\mathbf{q}_{12},0} \rightsquigarrow \ell(D)^{\mathbf{q}_1,0}, \ell_o(C)^{\mathbf{q}_{12},0}]\!]; \alpha^{\downarrow} \\
&\quad]; [\mathsf{id}, [\![\Gamma^{\mathbf{q}_{12}}, x : C \vdash \mathbf{q}_1, \omega = \mathbf{q}_1, 0 + \mathbf{q}_2, \omega]\!]; ([\![\Gamma^{\mathbf{q}_1}]\!] \otimes I) \otimes [\![\Gamma^{\mathbf{q}_2}, x : C \vdash_\epsilon R : D]\!]; \iota_r; \alpha^{\downarrow}] \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_{123} + \mathbf{q}_4]\!]; [\![\Gamma^{\mathbf{q}_{123}}]\!] \otimes [\![\Gamma^{\mathbf{q}_4} \vdash_\epsilon o : A + B]\!]; \delta^{-1}; [\\
&\qquad [\![\Gamma, y : A \vdash \mathbf{q}_{123}, \omega = \mathbf{q}_{12}, 0 + \mathbf{q}_3, \omega]\!]; ([\![\Gamma^{\mathbf{q}_{12}}]\!] \otimes I) \otimes [\![\Gamma^{\mathbf{q}_3}, y : A \vdash_\epsilon P : C]\!]; \iota_r; \alpha^{\downarrow}, \\
&\qquad [\![\Gamma, z : B \vdash \mathbf{q}_{123}, \omega = \mathbf{q}_{12}, 0 + \mathbf{q}_3, \omega]\!]; ([\![\Gamma^{\mathbf{q}_{12}}]\!] \otimes I) \otimes [\![\Gamma^{\mathbf{q}_3}, z : B \vdash_\epsilon Q : C]\!]; \iota_r; \alpha^{\downarrow} \\
&\quad]; [\mathsf{id}, [\![\Gamma \vdash \mathbf{q}_{12} = \mathbf{q}_1 + \mathbf{q}_2]\!] \otimes [\![C]\!]; \alpha; [\![\Gamma^{\mathbf{q}_1}]\!] \otimes [\![\Gamma^{\mathbf{q}_2}, x : C \vdash_\epsilon R : D]\!]; \iota_r] \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_{123} + \mathbf{q}_4]\!]; [\![\Gamma^{\mathbf{q}_{123}}]\!] \otimes [\![\Gamma^{\mathbf{q}_4} \vdash_\epsilon o : A + B]\!]; \delta^{-1}; [\\
&\qquad [\![\Gamma \vdash \mathbf{q}_{123} = \mathbf{q}_{12} + \mathbf{q}_3]\!] \otimes [\![A]\!]; \alpha; [\![\Gamma^{\mathbf{q}_{12}}]\!] \otimes [\![\Gamma^{\mathbf{q}_3}, y : A \vdash_\epsilon P : C]\!], \\
&\qquad [\![\Gamma \vdash \mathbf{q}_{123} = \mathbf{q}_{12} + \mathbf{q}_3]\!] \otimes [\![B]\!]; \alpha; [\![\Gamma^{\mathbf{q}_{12}}]\!] \otimes [\![\Gamma^{\mathbf{q}_3}, z : B \vdash_\epsilon Q : C]\!] \\
&\quad]; [\![\Gamma \vdash \mathbf{q}_{12} = \mathbf{q}_1 + \mathbf{q}_2]\!] \otimes [\![C]\!]; \alpha; [\![\Gamma^{\mathbf{q}_1}]\!] \otimes [\![\Gamma^{\mathbf{q}_2}, x : C \vdash_\epsilon R : D]\!]; \iota_r \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_{123} + \mathbf{q}_4]\!]; [\![\Gamma \vdash \mathbf{q}_{123} = \mathbf{q}_{12} + \mathbf{q}_3]\!] \otimes [\![\Gamma^{\mathbf{q}_4}]\!]; \alpha; [\![\Gamma^{\mathbf{q}_{12}}]\!] \otimes (\\
&\qquad [\![\Gamma^{\mathbf{q}_3}]\!] \otimes [\![\Gamma^{\mathbf{q}_4} \vdash_\epsilon o : A + B]\!]; \delta^{-1}; [[\![\Gamma^{\mathbf{q}_3}, y : A \vdash_\epsilon P : C]\!], [\![\Gamma^{\mathbf{q}_3}, z : B \vdash_\epsilon Q : C]\!] \\
&\quad]); [\![\Gamma^{\mathbf{q}_{12}} \vdash \mathbf{q}_1 = \mathbf{q}_1 + \mathbf{q}_2]\!] \otimes [\![C]\!]; \alpha; [\![\Gamma^{\mathbf{q}_1}]\!] \otimes [\![\Gamma^{\mathbf{q}_2}, x : C \vdash_\epsilon R : D]\!]; \iota_r \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_{12} + \mathbf{q}_{34}]\!]; [\![\Gamma \vdash \mathbf{q}_{12} = \mathbf{q}_1 + \mathbf{q}_2]\!] \otimes ([\![\Gamma \vdash \mathbf{q}_{34} = \mathbf{q}_3 + \mathbf{q}_4]\!]; [\![\Gamma^{\mathbf{q}_3}]\!] \otimes [\![\Gamma^{\mathbf{q}_4} \vdash_\epsilon o : A + B]\!]; \\
&\qquad \delta^{-1}; [[\![\Gamma^{\mathbf{q}_3}, y : A \vdash_\epsilon P : C]\!], [\![\Gamma^{\mathbf{q}_3}, z : B \vdash_\epsilon Q : C]\!]]); \alpha; [\![\Gamma^{\mathbf{q}_1}]\!] \otimes [\![\Gamma^{\mathbf{q}_2}, x : C \vdash_\epsilon R : D]\!]; \iota_r \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_1 + \mathbf{q}_{234}]\!]; ([\![\Gamma^{\mathbf{q}_1}]\!] \otimes ([\![\Gamma \vdash \mathbf{q}_{234} = \mathbf{q}_2 + \mathbf{q}_{34}]\!]; \\
&\qquad [\![\Gamma^{\mathbf{q}_2}]\!] \otimes ([\![\Gamma^{\mathbf{q}_{34}} \vdash_\epsilon \mathsf{case}\ o\ \{\iota_l\ y : P, \iota_r\ z : Q\} : C]\!]; [\![\Gamma^{\mathbf{q}_2}, x : C \vdash_\epsilon R : D]\!]))); \iota_r \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_1 + \mathbf{q}_{234}]\!]; [\![\Gamma^{\mathbf{q}_1}]\!] \otimes [\![\Gamma^{\mathbf{q}_{234}} \vdash_\epsilon \mathsf{let}\ x = \mathsf{case}\ o\ \{\iota_l\ y : P, \iota_r\ z : Q\};\ R : D]\!]; \iota_r
\end{aligned}
$$

as desired.

- $(\mathsf{let}\ x = \mathsf{iter}\ o\ \{\iota_r\ x : P\};\ Q)$:

$$
\begin{aligned}
(\mathsf{SSA}_\ell(\mathsf{let}\ x = \mathsf{iter}\ o\ \{\iota_r\ y : P\};\ Q)) := \mathsf{br}\ \ell_b\ o\ \mathsf{where}_{\mathsf{rec}} \\
\ell_b(y) : \{\mathsf{SSA}_{\ell_h}(P)\}, \\
\ell_h(w) : \{\mathsf{case}\ w\ \{\iota_l\ x : \mathsf{br}\ \ell_o\ x, \iota_r\ y : \mathsf{br}\ \ell_b\ y\}\}, \\
\ell_o(x) : \{\mathsf{SSA}_\ell(Q)\}
\end{aligned}
$$

We verify the semantics by induction as follows, where $\mathsf{R}^{\mathbf{Q}} := \ell_r(A)^{\mathbf{q}_{12}}, \ell_h(B+A)^{\mathbf{q}_{12}}, \ell_o(B)^{\mathbf{q}_{12}}$:

$$
\begin{aligned}
&[\![\Gamma^{\mathbf{q}} \vdash_\epsilon \mathsf{SSA}_\ell(\mathsf{let}\ x = \mathsf{iter}\ o\ \{\iota_r\ y : P\};\ Q) \rhd \ell(C)^{\mathbf{q}_1}]\!] \\
&= [\![\Gamma^{\mathbf{q}} \vdash_\epsilon \mathsf{br}\ \ell_r\ o \rhd \ell(C)^{\mathbf{q}_1}, \mathsf{R}^{\mathbf{Q}^\uparrow}]\!]; [\mathsf{id}_{[\![\Gamma^{\mathbf{q}_1}]\!] \otimes [\![C]\!]}, [\\
&\quad [\![\Gamma^{\mathbf{q}_{12}}, y : A \vdash_\epsilon \mathsf{SSA}_{\ell_h}(P) \rhd \ell(C)^{\mathbf{q}_{1,0}}, \mathsf{R}^{\mathbf{Q}^\uparrow}]\!]; \alpha^\downarrow; \alpha^+, \\
&\quad [\![\Gamma^{\mathbf{q}_{12}}, w : A + B \vdash_\epsilon \mathsf{case}\ w\ \{\iota_l\ x : \mathsf{br}\ \ell_o\ x, \iota_r\ y : \mathsf{br}\ \ell_b\ y\} \rhd \ell(C)^{\mathbf{q}_{1,0}}, \mathsf{R}^{\mathbf{Q}^\uparrow}]\!]; \alpha^\downarrow; \alpha^+, \\
&\quad [\![\Gamma^{\mathbf{q}_{12}}, x : B \vdash_\epsilon \mathsf{SSA}_\ell(Q) \rhd \ell(C)^{\mathbf{q}_{1,0}}, \mathsf{R}^{\mathbf{Q}^\uparrow}]\!]; \alpha^\downarrow; \alpha^+]^\dagger] \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_{12} + \mathbf{q}_3]\!]; [\![\Gamma^{\mathbf{q}_{12}}]\!] \otimes [\![\Gamma^{\mathbf{q}_3} \vdash_\epsilon o : A]\!]; \iota_l; \iota_l; [\mathsf{id}_{[\![\Gamma^{\mathbf{q}_1}]\!] \otimes [\![C]\!]}, [\\
&\quad [\![\Gamma^{\mathbf{q}_{12}}, y : A \vdash_\epsilon \mathsf{SSA}_{\ell_h}(P) \rhd \ell(C)^{\mathbf{q}_{1,0}}, \mathsf{R}^{\mathbf{Q}}]\!]; \alpha^\downarrow; \alpha^+, \\
&\quad \delta^{-1}; [\iota_r, \iota_r; \iota_l]; \iota_r, \\
&\quad [\![\Gamma^{\mathbf{q}_{12}}, x : B \vdash_\epsilon \mathsf{SSA}_\ell(Q) \rhd \ell(C)^{\mathbf{q}_{1,0}}, \mathsf{R}^{\mathbf{Q}^\uparrow}]\!]; \alpha^\downarrow; \alpha^+]^\dagger] \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_{12} + \mathbf{q}_3]\!]; [\![\Gamma^{\mathbf{q}_{12}}]\!] \otimes [\![\Gamma^{\mathbf{q}_3} \vdash_\epsilon o : A]\!]; \iota_l; \iota_l; [\\
&\quad [\![\Gamma^{\mathbf{q}_{12}}, y : A \vdash_\epsilon \mathsf{SSA}_{\ell_h}(P) \rhd \ell_h(B+A)^{\mathbf{q}_{12},0}]\!]; [\![\Gamma \vdash \ell_h(B+A)^{\mathbf{q}_{12},0} \rightsquigarrow \ell(C)^{\mathbf{q}_{1,0}}, \mathsf{R}^{\mathbf{Q}^\uparrow}]\!]; \alpha^\downarrow; \alpha^+, \\
&\quad \delta^{-1}; [\iota_r, \iota_r; \iota_l]; \iota_r, \\
&\quad [\![\Gamma^{\mathbf{q}_{12}}, z : B \vdash_\epsilon \mathsf{SSA}_{\ell_h}(Q) \rhd \ell(C)^{\mathbf{q}_{1,0}}]\!]; [\![\Gamma \vdash \ell(C)^{\mathbf{q}_{1,0}} \rightsquigarrow \ell(C)^{\mathbf{q}_{1,0}}, \mathsf{R}^{\mathbf{Q}^\uparrow}]\!]; \alpha^\downarrow; \alpha^+]^\dagger \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_{12} + \mathbf{q}_3]\!]; [\![\Gamma^{\mathbf{q}_{12}}]\!] \otimes [\![\Gamma^{\mathbf{q}_3} \vdash_\epsilon o : A]\!]; \iota_l; \iota_l; [\\
&\quad [\![\Gamma \vdash \mathbf{q}_{12} = \mathbf{q}_{12} + \mathbf{q}_r]\!] \otimes [\![A]\!]; \alpha; [\![\Gamma^{\mathbf{q}_{12}}]\!] \otimes [\![\Gamma^{\mathbf{q}_r}, y : A \vdash_\epsilon P : B + A]\!]; \iota_r; \iota_r, \\
&\quad \delta^{-1}; [\iota_r, \iota_r; \iota_l]; \iota_r, \\
&\quad [\![\Gamma \vdash \mathbf{q}_{12} = \mathbf{q}_1 + \mathbf{q}_2]\!] \otimes [\![B]\!]; \alpha; [\![\Gamma^{\mathbf{q}_1}]\!] \otimes [\![\Gamma^{\mathbf{q}_2}, x : B \vdash_\epsilon Q : C]\!]; \iota_r; \iota_l]^\dagger \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_{12} + \mathbf{q}_3]\!]; [\![\Gamma^{\mathbf{q}_{12}}]\!] \otimes [\![\Gamma^{\mathbf{q}_3} \vdash_\epsilon o : A]\!]; (\\
&\quad [\![\Gamma \vdash \mathbf{q}_{12} = \mathbf{q}_{12} + \mathbf{q}_r]\!] \otimes [\![A]\!]; \alpha; [\![\Gamma^{\mathbf{q}_{12}}]\!] \otimes [\![\Gamma^{\mathbf{q}_r}, y : A \vdash_\epsilon P : B + A]\!]; \delta^{-1}; \\
&\quad ([\![\Gamma \vdash \mathbf{q}_{12} = \mathbf{q}_1 + \mathbf{q}_2]\!] \otimes [\![B]\!]; \alpha; [\![\Gamma^{\mathbf{q}_1}]\!] \otimes [\![\Gamma^{\mathbf{q}_2}, x : B \vdash_\epsilon Q : C; \iota_r]\!]) + ([\![\Gamma^{\mathbf{q}_{12}}]\!] \otimes [\![A]\!]))^\dagger \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_{12} + \mathbf{q}_3]\!]; [\![\Gamma^{\mathbf{q}_{12}}]\!] \otimes [\![\Gamma^{\mathbf{q}_3} \vdash_\epsilon o : A]\!]; (\\
&\quad [\![\Gamma \vdash \mathbf{q}_{12} = \mathbf{q}_{12} + \mathbf{q}_r]\!] \otimes [\![A]\!]; \alpha; [\![\Gamma^{\mathbf{q}_{12}}]\!] \otimes [\![\Gamma^{\mathbf{q}_r}, y : A \vdash_\epsilon P : B + A]\!]; \delta^{-1} \\
&\quad)^\dagger; [\![\Gamma \vdash \mathbf{q}_{12} = \mathbf{q}_1 + \mathbf{q}_2]\!] \otimes [\![B]\!]; \alpha; [\![\Gamma^{\mathbf{q}_1}]\!] \otimes [\![\Gamma^{\mathbf{q}_2}, x : B \vdash_\epsilon Q : C]\!]; \iota_r \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_{12} + \mathbf{q}_3]\!]; [\![\Gamma \vdash \mathbf{q}_{12} = \mathbf{q}_{12} + \mathbf{q}_r]\!] \otimes [\![\Gamma^{\mathbf{q}_3} \vdash_\epsilon o : A]\!]; \alpha; ([\![\Gamma^{\mathbf{q}_{12}}]\!] \otimes (\\
&\quad [\![\Gamma \vdash \mathbf{q}_r = \mathbf{q}_r + \mathbf{q}_r]\!] \otimes [\![A]\!]; \alpha; [\![\Gamma^{\mathbf{q}_r}]\!] \otimes [\![\Gamma^{\mathbf{q}_r}, y : A \vdash_\epsilon P : B + A]\!]; \\
&\quad \delta^{-1}; ([\![\Gamma^{\mathbf{q}_r} \mapsto \cdot]\!] \otimes [\![B]\!]; \lambda) + [\![\Gamma^{\mathbf{q}_r}]\!] \otimes [\![A]\!] \\
&\quad)^\dagger); \alpha; [\![\Gamma \vdash \mathbf{q}_{12} = \mathbf{q}_1 + \mathbf{q}_2]\!] \otimes [\![B]\!]; \alpha; [\![\Gamma^{\mathbf{q}_1}]\!] \otimes [\![\Gamma^{\mathbf{q}_2}, x : B \vdash_\epsilon Q : C]\!]; \iota_r \\
&= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_{12} + \mathbf{q}_{3r}]\!]; [\![\Gamma \vdash \mathbf{q}_{12} = \mathbf{q}_1 + \mathbf{q}_2]\!] \otimes ([\![\Gamma \vdash \mathbf{q}_{3r} = \mathbf{q}_r + \mathbf{q}_3]\!]; [\![\Gamma^{\mathbf{q}_r}]\!] \otimes [\![\Gamma^{\mathbf{q}_3} \vdash_\epsilon o : A]\!]; (\\
&\quad [\![\Gamma \vdash \mathbf{q}_r = \mathbf{q}_r + \mathbf{q}_r]\!] \otimes [\![A]\!]; \alpha; [\![\Gamma^{\mathbf{q}_r}]\!] \otimes [\![\Gamma^{\mathbf{q}_r}, y : A \vdash_\epsilon P : B + A]\!])^\dagger; [\![\Gamma^{\mathbf{q}_r} \mapsto \cdot]\!] \otimes [\![B]\!]; \lambda \\
&\quad); \alpha; [\![\Gamma^{\mathbf{q}_1}]\!] \otimes [\![\Gamma^{\mathbf{q}_2}, x : B \vdash_\epsilon Q : C]\!]; \iota_r
\end{aligned}
$$

$$= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_{12} + \mathbf{q}_{3r}]\!]; [\![\Gamma \vdash \mathbf{q}_{12} = \mathbf{q}_1 + \mathbf{q}_2]\!] \otimes [\![\Gamma^{\mathbf{q}_{3r}} \vdash_\epsilon \mathsf{iter}\ o\ \{\iota_r\ y : P\} : B]\!];$$
$$\alpha; [\![\Gamma^{\mathbf{q}_1}]\!] \otimes [\![\Gamma^{\mathbf{q}_2}, x : B \vdash_\epsilon Q : C]\!]; \iota_r$$
$$= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_{12} + \mathbf{q}_{3r}]\!]; [\![\Gamma \vdash \mathbf{q}_{12} = \mathbf{q}_1 + \mathbf{q}_2]\!] \otimes [\![\Gamma^{\mathbf{q}_{3r}} \vdash_\epsilon \mathsf{iter}\ o\ \{\iota_r\ y : P\} : B]\!];$$
$$\alpha; [\![\Gamma^{\mathbf{q}_1}]\!] \otimes [\![\Gamma^{\mathbf{q}_2}, x : B \vdash_\epsilon Q : C]\!]; \iota_r$$
$$= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_1 + \mathbf{q}_{23r}]\!]; [\![\Gamma^{\mathbf{q}_1}]\!] \otimes (\ [\![\Gamma \vdash \mathbf{q}_{23r} = \mathbf{q}_2 + \mathbf{q}_{3r}]\!]; [\![\Gamma^{\mathbf{q}_2}]\!] \otimes [\![\Gamma^{\mathbf{q}_{3r}} \vdash_\epsilon \mathsf{iter}\ o\ \{\iota_r\ y : P\} : B]\!]; [\![\Gamma^{\mathbf{q}_2}, x : B \vdash_\epsilon Q : C]\!]); \iota_r$$
$$= [\![\Gamma \vdash \mathbf{q} = \mathbf{q}_1 + \mathbf{q}_{23r}]\!]; [\![\Gamma^{\mathbf{q}_1}]\!] \otimes [\![\Gamma^{\mathbf{q}_{23r}} \vdash_\epsilon \mathsf{let}\ x = \mathsf{iter}\ o\ \{\iota_r\ y : P\};\ Q : C]\!]; \iota_r$$

as desired.

D Models

In this section, we give a few simple families of λ_{iter}-models highlighting some of the features of our formalization.

D.1 Poset-Enriched Elgot Monads

A wide variety of λ_{iter}-models can be derived from *poset-enriched Elgot monads* over **Set**, which are just standard monads equipped with a notion of refinement and iteration. We begin by defining a *poset-enriched monad* as follows:

Definition D.1 (Poset-Enriched Monad). We say a monad T over **Set** is *poset-enriched* if each $T(A)$ is equipped with a partial order $\twoheadrightarrow$ compatible with $\cdot \gg\!=_T \cdot$, i.e., such that, for $a \twoheadrightarrow b$ and $f \twoheadrightarrow g$, we have $a \gg\!=_T f \twoheadrightarrow b \gg\!=_T g$, where morphisms in the Kleisli category $f, g : A \to T(B)$ are given the pointwise partial order.

Note that, just like for poset-enriched categories, *every* monad is poset-enriched with the identity order on $T\ A$. Probably the simplest nontrivial example of a poset-enriched monad is given by adjoining a top element to every set as follows $\top(A) := A \cup \{\top\}$, with pure and bind defined as for the option monad; we can symmetrically instead adjoin a bottom element $\bot(A) := A \cup \{\bot\}$. A more complex example is given by the power-set monad $\mathcal{P}$ equipped with the inclusion order, with

$$\eta_{\mathcal{P}}(a) := \{a\} \qquad X \gg\!=_{\mathcal{P}} f := \bigcup_{a \in X} f(a) \tag{6}$$

This has numerous natural submonads which are also poset-enriched, including:

- Nonempty sets $\mathcal{P}^+$
- Finite sets $\mathcal{P}_{\mathsf{fin}}$
- Countable sets $\mathcal{P}_{\mathbb{N}}$
- Sets of cardinality at most one, which is isomorphic to $\bot$.
- Sets of cardinality exactly one, corresponding to pure morphisms.

It is easy to see that the Kleisli category of a poset-enriched monad is always a poset-enriched distributive category; in particular, tensor products and coproducts are simply given by the product and coproduct in **Set**, with

$$f \otimes A := (\lambda(x, y).f\ x \gg\!= \lambda z.(z, y)) \qquad A \otimes f := (\lambda(x, y).f\ y \gg\!= \lambda z.(x, z))$$

and (inverse) associators, unitors, symmetries, distributors, and injections given by the lifts of the underlying morphisms in **Set**. We can always equip the Kleisli category with the structure of a distributive effectful category with pure morphisms

$$\mathrm{Set}_{T\bot}(A, B) := \{\eta_B \circ f \mid f : A \to B\}$$

We can define $\mathrm{Set}_{T\epsilon}$ for other effects $\epsilon \in \mathcal{E}$ in a given effect system as appropriate to the effect system and monad; submonads are often a good place to look for these.

We would now like to equip the Kleisli category of T with an iteration structure. The notion of an Elgot monad [12] generalizes directly to the poset-enriched setting:

Definition D.2 (Poset-Enriched Elgot Monad). We say a poset-enriched monad T is a (strong) Elgot monad if its Kleisli category is equipped with a monotone iteration operator $(\cdot)^\dagger$ satisfying the following properties:

- *Naturality*: for $f : A \to T(B + A)$ and $g : B \to T(C)$, we have $(f; g + A)^\dagger = f^\dagger; g$
- *Codiagonal*: for $f : A \to T((B + A) + A)$, we have $f^{\dagger\dagger} = (f; [\mathrm{id}, \iota_r])^\dagger$
- *Directed uniformity*: given $f : A \to T(B + A)$, $g : X \to T(B + X)$, and *pure* $h : X \to A$, we have
  - $h^\uparrow\ f \twoheadrightarrow g; B + h^\uparrow \implies h^\uparrow; f^\dagger \twoheadrightarrow g^\dagger$
  - $h^\uparrow\ f \twoheadleftarrow g; B + h^\uparrow \implies h^\uparrow; f^\dagger \twoheadleftarrow g^\dagger$

  where $h^\uparrow = \eta_A \circ h : X \to T(A)$.

Strengh: given $f : A \to T(B + A)$, we have $X \otimes f^\dagger = (X \otimes f; \delta^{-1})^\dagger$

We can easily verify that the Kleisli category of a strong Elgot monad is an Elgot category, which in particular is $\mathrm{Set}_{T\bot}$-uniform by uniformity. Probably the simplest example of an Elgot monad is the power-set $\mathcal{P}$, with

$$f^\dagger := \bigcup_i f_i \qquad \text{where} \qquad f_0 = \varnothing \qquad f_{i+1} = f; [\mathrm{id}, f_i]$$

Analysis of semantics in this monad corresponds to *partial correctness*, with any infinite executions discarded. This isn't always what we want, especially since every program can be safely refined to (and hence "optimized to") the infinite loop $0 := \varnothing : A \to B$. Note for example that

- The pure effect is not closed under refinement, since any pure morphism can be refined to 0
- The pure effect, as well as the effect of being finite and the effect of being nonempty, are *not* iterative, but the effect of being countable and the effect of being deterministic but potentially nonterminating *are*.

D.2 Undefined Behavior

In this section, we demonstrate how to build a more complex poset-enriched Elgot monad, by constructing a monad UB which supports

- *Undefined behavior* (UB), represented as a top element $\lightning \twoheadrightarrow a$ for all a.
- *Nondeterministic behavior*
- *Total correctness*, with nontermination represented as a separate possible outcome ∞ (allowing us to distinguish between a program which always terminates, never terminates, and is nondeterministically nonterminating)

As described in Section 6.1, we define $\mathsf{UB}(A) := \mathcal{P}^+(A \cup \{\infty\}) \cup \{\lightning\}$, with

$$\lightning \gg\!=_{\mathsf{UB}} f = \lightning \qquad X \gg\!=_{\mathsf{UB}} f = \{f\ a \mid a \in X\} \ \text{ if } \forall a \in X, f\ a \neq \lightning \qquad X \gg\!=_{\mathsf{UB}} f = \lightning \ \text{ otherwise}$$

We can equip this with iteration structure $(\cdot)^\dagger$ given by, given $f : A \to \mathsf{UB}(B + A)$,

$$f^\dagger(a) := \begin{cases} f_\infty(a) \cup \bigcup_{i \in \mathbb{N}} f_i(a) & \text{if } \forall i, f_i(a) \neq \lightning \\ \lightning & \text{otherwise} \end{cases}$$

where

$$f_0 := \varnothing \qquad f_{i+1} := f; [\mathrm{id}, f_i] \qquad f_\infty(a_0) := \begin{cases} \{\infty\} & \text{if } \exists a_i, \forall i, a_{i+1} \in f(a_i) \\ \varnothing & \text{otherwise} \end{cases}$$

Note that the f_i are in the slightly larger monad $A \mapsto \mathcal{P}(A \cup \{\infty\}) \cup \{\lightning\}$; but it is straightforward to show that $f^\dagger$ must be nonempty and therefore lie in the submonad UB (since if all f_i are empty, $f_\infty = \{\infty\}$). It is straightforward to verify that this indeed gives us an Elgot structure.

D.3 Monad Transformers

In functional programming, we often use a stack of *monad transformers* to build up complex effects from simple building blocks. It turns out that many common monad transformers are compatible with both poset-enrichment and Elgot structure; we give some important examples below:

- The *reader transformer* $\mathrm{Rd}_R\ T\ A := R \to T\ A$ allows us to read from an environment R, with

  $$\eta_{\mathrm{Rd}_R\ T}\ a := \lambda r.\eta_T\ a \qquad a \gg\!=_{\mathrm{Rd}_R\ T} f := \lambda r.a\ r \gg\!=_T \lambda a'.f\ a'\ r$$

  Given $f : A \to \mathrm{Rd}_R\ T\ (B + A) = A \to R \to T(B + A)$, we define

  $$f^{\dagger_{\mathrm{Rd}_R\ T}} := \lambda a, r.((\lambda(r, a).(r, f\ a\ r)); \delta^{-1})^\dagger(r, a) \gg\!=_T \pi_2$$

- The *writer transformer* $\mathrm{Wr}_W\ T\ A := T\ (W \times A)$ for a monoid $(W, \cdot, 1)$ allows us to write to a W-typed log, with

  $$\eta_{\mathrm{Wr}_W\ T}\ a := \eta_T\ (1, a) \qquad a \gg\!=_{\mathrm{Wr}_W\ T} f := a \gg\!=_T \lambda(w, a').f\ a \gg\!=_T \lambda(w', b).(w \cdot w', b)$$

  Given $f : A \to \mathrm{Wr}_W\ T\ (B + A) = A \to T(W \times (B + A))$, we define

  $$f^{\dagger_{\mathrm{Wr}_W\ T}} := \lambda a.(W \otimes f); (\lambda(w, (w', c)).\eta_T(w \cdot w', c)); \delta^{-1})^\dagger(1, a)$$

- The *state transformer* $\mathrm{St}_S\ T\ A := S \to T\ (S \times A)$ allows us to access mutable state of type S, with

  $$\eta_{\mathrm{St}_S\ T}\ a := \lambda s.\eta_T\ (s, a) \qquad a \gg\!=_{\mathrm{St}_S\ T} f := \lambda s.a\ s \gg\!=_T \lambda(s', a').f\ a'\ s'$$

  Given $f : A \to \mathrm{St}_S\ T\ (B + A) = A \to S \to T(S \times (B + A))$, we define

  $$f^{\dagger_{\mathrm{St}_S\ T}} := \lambda a, s.((\lambda(s, a).f\ a\ s); \delta^{-1})^\dagger(s, a)$$

As a concrete example, we can build the heap monad from Section 6.2 by simply applying the state transformer to the UB monad from Appendix D.2, with state $S := \mathbb{N} \rightharpoonup_{\mathrm{fin}} \mathbb{N}$.

D.4 Brookes-Style Concurrency

In this section, we show how to construct a concurrency monad from a Brookes [5]-style model of concurrency, which we axiomatize. We will then explicitly show how to instantiate the model for sequentially consistent execution from Brookes [5].

We begin by defining a *countable closure operator*, given below:

Definition D.3 (Countable Closure Operator). We define a *(countable) closure operator* $\mathrm{c} : \mathcal{P}(T) \to \mathcal{P}(T)$ on T to be a function on sets of T which:

- *is extensive*: $X \subseteq \mathrm{c}(X)$
- *is idempotent*: $\mathrm{c}(\mathrm{c}(X)) = \mathrm{c}(X)$
- *distributes over countable unions*: $\mathrm{c}(\bigcup_i X_i) = \bigcup_i \mathrm{c}(X_i)$

We say a set X such that $\mathrm{c}(X) = X$ is *closed* under c.

Note that this differs from the standard definition of a Kuratowski topological closure operator, which only requires us to distributive over *finite* unions. We define the *lift* of a closure operator $\mathrm{c}_A : \mathcal{P}(T \times A) \to \mathcal{P}(T \times A)$ to be given by

$$\mathrm{c}_A(X) := \bigcup_{a \in A} \{(t', a) \mid t' \in \mathrm{c}(\{t \mid (t, a) \in X\})\}$$

Identifying $\mathcal{P}(T \times A)$ with $A \to \mathcal{P}(T)$, this is equivalent to stating that $\mathrm{c}_A(X) := \mathrm{c} \circ X$. It is hence easy to check that this gives a closure operator on $T \times A$.

Definition D.4 (Brookes Monad). Given a monoid of *traces* T and a closure operator $\mathrm{c} : \mathcal{P}(T) \to \mathcal{P}(T)$, we define the *Brookes monad* B_c over c to be given by closed sets of pairs $t \therefore a$ of *traces* $t \in T$ and *results* $a \in A$. In particular, we define

$$\mathrm{B}_\mathrm{c}(A) := \mathrm{c}_A(\mathcal{P}(T \times A)) \simeq A \to \mathrm{c}(\mathcal{P}(T))$$

with

$$\eta_{\mathrm{B}_\mathrm{c}}\ a := \mathrm{c}_A(\{1 \therefore a\}) \qquad X \gg\!=_{\mathrm{B}_\mathrm{c}} f := \mathrm{c}_A(\{t \cdot t' \therefore b \mid \exists a.t \therefore a \in X, t' \therefore b \in X'\})$$

It is easy to see that the Brookes monad is in fact poset-enriched under the inclusion order. We say a Brookes monad is *standard* if $T = \langle S, S \rangle^*$ is the monoid of finite sequences of *rely-guarantee pairs* of *states* S.

We note that this definition gives us a Kleisli category which is not only poset-enriched, but in fact compatible with countable unions: with the usual definition $\cup_i f_i := \lambda a.\, \cup_i f_i(a)$, we have

$$\begin{aligned}
((\bigcup_i f_i); g)(a) &:= \mathrm{c}_C(\{t \cdot t' \therefore c \mid \exists b.t \therefore b \in \bigcup_i f_i(a), t' \therefore c \in g(b)\}) \\
&= \mathrm{c}(\bigcup_i \{t \cdot t' \therefore c \mid \exists b.t \therefore b \in f_i(a), t' \therefore c \in g(b)\}) \\
&= \bigcup_i \mathrm{c}(\{t \cdot t' \therefore c \mid \exists b.t \therefore b \in f_i(a), t' \therefore c \in g(b)\}) \\
&= \bigcup_i (f_i; g)(a) \\
f; (\bigcup_i g_i)(a) &:= \mathrm{c}(\{t \cdot t' \therefore c \mid \exists b.t \therefore b \in f(a), t' \therefore c \in \bigcup_i g_i(b)\}) \\
&= \mathrm{c}(\bigcup_i \{t \cdot t' \therefore c \mid \exists b.t \therefore b \in f(a), t' \therefore c \in g_i(b)\}) \\
&= \bigcup_i \mathrm{c}(\{t \cdot t' \therefore c \mid \exists b.t \therefore b \in f(a), t' \therefore c \in g_i(b)\}) \\
&= \bigcup_i (f; g_i)(a)
\end{aligned}$$

and likewise for coproducts.

We can always equip B_c with a (poset-enriched) Elgot structure $(\cdot)^\dagger$ by, for $f : A \to \mathrm{B}_\mathrm{c}(B + A)$, defining

$$f^\dagger := \bigcup_{i \in \mathbb{N}} f_i$$

where $f_i : A \to \mathrm{B}_C(B)$ are the *iterates* of f, defined inductively as follows

$$f_0 := 0_{A,B} = (\lambda x.\varnothing) \qquad f_{i+1} := f; [\mathrm{id}_B, f_i]$$

We note in particular that $\forall i. f_i \subseteq f_{i+1}$; similarly, if $g \subseteq f$, we have that $g_i \subseteq f_i$. We can think of analysis using this structure as corresponding to *partial correctness*, since any infinite traces are simply discarded (a program which always diverges will have denotation $\varnothing$). It is straightforward to check that this is indeed an Elgot structure, since it satisfies:

- *Fixpoint*: we have, since $f_0 \cup g = g$,
$$f^\dagger = \bigcup_{i\in\mathbb{N}} f_i = \bigcup_{i\in\mathbb{N}} f_{i+1} = \bigcup_{i\in\mathbb{N}} (f;[\mathrm{id}_B, f_i]) = f;[\mathrm{id}_B, \bigcup_{i\in\mathbb{N}} f_i] = f;[\mathrm{id}_B, f^\dagger]$$
as desired.
- *Naturality*: by induction, we show that $(f; g+A)_i = f_i; g$, since $(f; g+A)_0 = 0_{A,C} = 0_{A,B}; g$ and
$$(f; g+A)_{i+1} = f; g+A; [\mathrm{id}_C, (f; g+A)_i] = f; [g, f_i; g] = f; [\mathrm{id}_B, f_i]; g = f_{i+1}; g$$
and therefore
$$(f; g+A)^\dagger = \bigcup_{i\in\mathbb{N}} (f; g+A)_i = \bigcup_{i\in\mathbb{N}} (f_i; g) = (\bigcup_{i\in\mathbb{N}} f_i); g = f^\dagger; g$$
- *Codiagonal*: given $f : A \to (B+A)+A$, define $g = f; [\mathrm{id}_{B+A}, \iota_r]$ and $h = f^\dagger$. We have
$$\begin{aligned} h_{i+1} &= f^\dagger; [\mathrm{id}, h_i] = f; [\mathrm{id}, f^\dagger]; [\mathrm{id}, h_i] \\ &= f; [[\mathrm{id}, h_i], f^\dagger; [\mathrm{id}, h_i]] = f; [[\mathrm{id}, h_i], h_{i+1}] \\ g_{i+1} &= f; [\mathrm{id}_B, \iota_r]; [\mathrm{id}, g_i] = f; [[\mathrm{id}, g_i], g_i] \end{aligned}$$
It follows by induction that $g_i \subseteq h_i$, since $g_0 = h_0$, and
$$g_{i+1} = f; [[\mathrm{id}, g_i], g_i] \subseteq f; [[\mathrm{id}, h_i], h_i] \subseteq f; [[\mathrm{id}, h_i], h_{i+1}] = h_{i+1}$$
We therefore have that $g^\dagger \subseteq h^\dagger$. On the other hand, we may show by induction on j that $f_j; [\mathrm{id}, g^\dagger] \subseteq g^\dagger$, since $0 \subseteq g^\dagger$ and
$$f_{j+1}; [\mathrm{id}, g^\dagger] = f; [\mathrm{id}, f_j]; [\mathrm{id}, g^\dagger] = f; [[\mathrm{id}, g^\dagger], f_j; [\mathrm{id}, g^\dagger]] \subseteq f; [[\mathrm{id}, g^\dagger], g^\dagger] = g^\dagger$$
We now show by induction that $h_i \subseteq g^\dagger$: noting that $h_0 = 0 \subseteq g^\dagger$, it suffices to show that
$$\begin{aligned} h_{i+1} &= f; [[\mathrm{id}, h_i], f^\dagger; [\mathrm{id}, h_i]] \subseteq f; [[\mathrm{id}, g^\dagger], f_j; [\mathrm{id}, g^\dagger]] \\ &= \bigcup_{j\in\mathbb{N}} (f; [[\mathrm{id}, g^\dagger], f_j; [\mathrm{id}, g^\dagger]]) \subseteq \bigcup_{j\in\mathbb{N}} (f; [[\mathrm{id}, g^\dagger], g^\dagger]) = g^\dagger \end{aligned}$$
It follows that $h^\dagger \subseteq g^\dagger$ and hence that $h^\dagger = g^\dagger$, as desired.
- *Directed Uniformity*: given arbitrary $f : A \to B+A$, $g : X \to B+X$ and $h : X \to A$, we have that
  - Given $h; f \subseteq g; B+h$, for all f_i, we show by induction that $h; f_i \subseteq g_i$ since $h; f_0 = 0 \subseteq g_0$ and
  $$h; f_{i+1} = h; f; [\mathrm{id}, f_i] \subseteq g; B+h; [\mathrm{id}, f_i] = g; [\mathrm{id}, h; f_i] \subseteq g; [\mathrm{id}, g_i] = g_{i+1}$$
  and therefore we have that $h; f^\dagger = \bigcup_i h; f_i \subseteq \bigcup_i g_i = g^\dagger$
  - Given $h; f \supseteq g; B+h$, for all f_i, we show by induction that $h; f_i \supseteq g_i$ since $h; f_0 = 0 = g_0$ and
  $$h; f_{i+1} = h; f; [\mathrm{id}, f_i] \supseteq g; B+h; [\mathrm{id}, f_i] = g; [\mathrm{id}, h; f_i] \supseteq g; [\mathrm{id}, g_i] = g^\dagger$$
  and therefore we have that $h; f^\dagger = \bigcup_i h; f_i \supseteq \bigcup_i g_i = g^\dagger$

Following Brookes [5], we can build up a model of *sequentially consistent* concurrent computation by taking a standard Brookes monad with

- States maps from locations to values $S := \mathsf{Loc} \to \mathsf{Val}$
- Closure operator generated by:
  - *Stuttering* $\forall \mu \in S. \cdot \twoheadrightarrow \langle \mu, \mu \rangle$
  - *Mumbling* $\forall \mu, \rho, \theta \in S. \langle \mu, \rho \rangle \langle \rho, \theta \rangle \twoheadrightarrow \langle \mu, \theta \rangle$

The semantics of a write, for example, is then given by

$$\mathsf{write} : \mathsf{Loc} \times \mathsf{Val} \to \mathsf{B}_{\mathsf{c}}(\mathbf{1}) := \lambda(\ell, v).\mathsf{c}_1(\{\langle \mu, [\ell \mapsto v]\mu \rangle \therefore () \mid \mu \in S\})$$

More complex states (e.g. involving per-thread buffers) and closure operators can allow us to model weak memory models. In particular, Jagadeesan et al. [14] gives a Brookes model of TSO, while Dvir et al. [7] gives a Brookes model of release-acquire.