The Denotational Semantics of SSA

JAD GHALAYINI and NEEL KRISHNASWAMI

Static single assignment form, or SSA, has been the dominant compiler intermediate representation for decades. In this paper, we give a type theory for a variant of SSA, including its equational theory, which are strong enough to validate a variety of control and data flow transformations. We also give a categorical semantics for SSA, and show that the type theory is sound and complete with respect to the categorical axiomatization. We demonstrate the utility of our model by exhibiting a variety of concrete models satisfying our axioms, including in particular a model of TSO weak memory. The correctness of the syntactic metatheory, as well as the completeness proof has been mechanized in the Lean proof assistant.

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:
Jad Ghalayini and Neel Krishnaswami. 2024. The Denotational Semantics of SSA. 1, 1 (March 2024), 100 pages. https://doi.org/XXXXXXX.XXXXXXX

This paper is dedicated to the memory of Alan Jeffrey, who taught us about both premonoidal categories and the semantics of weak memory, and who never shied away from either theory or implementation.

1 Introduction

Static single assignment form, or SSA form, has been the dominant compiler intermediate representation since its introduction by Alpern et al. [4] and Rosen et al. [58] in the late 1980s. Most major compilers – GCC, Clang, MLIR, Cranelift – use this representation, because it makes many optimizations much easier to do than traditional 3-address code IRs.

The key idea behind SSA is to adapt an idea from functional programming: namely, every variable is defined only once. This means that substitution is unconditionally valid, without first requiring a dataflow analysis to compute where definitions reach. Furthermore, because variables are immutable, they can have at most one value, which means that a program analysis only needs to store an abstract value once per variable, rather than once per variable per progam point, reducing memory overheads from quadratic to linear. Unlike in functional programming, though, scoping of definitions in SSA is traditionally not lexical. Instead, scoping is defined by *dominance*: every variable occurrence must be dominated by a single assignment in the control flow graph.

Traditionally, the semantics of SSA has been handled informally. Since it was conceived of as a simple first-order imperative programming language, whether a rewrite is sound or not was usually obvious, without needing any complex correctness arguments. Unfortunately, all of computers, languages and compilers have become more complex since the late 1980s.

Essentially all modern computers are multicore and feature many levels of caching. As a result, the semantics of memory can no longer be correctly modelled as a big array of bytes, and the

Authors' Contact Information: Jad Ghalayini, jeg74@cl.cam.ac.uk; Neel Krishnaswami, nk480@cl.cam.ac.uk.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 ACM.
ACM XXXX-XXXX/2024/3-ART
https://doi.org/XXXXXXX.XXXXXXX

execution of a multithreaded program can no longer be viewed as an interleaving of its threads' sequential execution traces. Finding good semantics for modern weak memory systems remains an ongoing challenge. Furthermore, each programming language's semantics must also have its own model of weak memory which both abstracts over the differing weak memory models of all the architectures the language gets compiled to, and also validates the optimizations which compiler writers wish to perform. Finally, modern compilers exploit semantic properties like undefined behaviour to transform programs much more aggressively than they did in the previous millenium.

As a result, it is no longer correct to justify compiler optimizations in terms of the simple imperative model, and it is an open question which equations should hold of an SSA program. Note that the pressure on SSA as an intermediate representation comes from all directions: the machine semantics, the language semantics, and the compiler optimizations have all become more complicated. Once all of these concerns are tangled together, it is very hard to decide if a particular transformation should hold or not.

To resolve this issue, we propose studying the equational theory of SSA. Having a well-defined equational theory for SSA will let us disentangle these concerns, because the equational theory can serve as an interface which both compiler writers and hardware designers can use. The compiler writers can rely upon the equational theory of SSA when justifying optimizations, without needing to know all the details of the memory model at all times. Conversely, memory models could be validated by seeing if they satisfy the equations of SSA, without needing to study every possible compiler optimization.

Defining an equational theory for SSA is a nontrivial problem, for both minor and major reasons. Traditionally, SSA is presented as a collection of basic blocks augmented with ϕ-functions to handle control-flow merges. This is a good representation for compiler engineering, but creates a large number of papercuts when defining an operational semantics or equational theory. Operational semantics for SSA have to keep track of the execution history to correctly interpret ϕ-nodes, and the lack of compound expressions makes it awkward to define and use primitives like parallel substitution. In addition, when all one has is an operational semantics, the only natural notion of program equivalence is contextual equivalence. Unfortunately, contextual equivalence is extremely sensitive to the effects available in the language (and SSA often has to deal with very complex effects like weak memory), and this makes it difficult to formulate a criterion for when the equational theory is complete.

If we had a class of denotational models for SSA, then it would be possible to give an equational theory, and show that it is complete (or not) relative to this class of models. But this is precisely the classical methodology of type theory and categorical logic! To fully understand a programming language, in general one wants a type theory equipped with an equational theory, a proof that it is sound and complete with respect to a categorical axiomatization, and a variety of interesting concrete models satisfying those axioms. Moreover, it would be best if the categorical axiomatization was constructed from "standard parts": the less novel the categorical axioms, the better, because standard constructions are better-studied and have more theorems already proved about them. This is exactly what we provide in this paper. Concretely, our contributions are as follows:

- First, we give a type-theoretic presentation of SSA, with both typing rules (in Section 3) and an equational theory (in Section 4) for well-typed terms. We also prove the correctness of suitable substitution properties for this calculus.
- Next, in Section 5, we give a categorical semantics for this type theory, in terms of distributive Elgot categories, which are Freyd categories equipped with distributive coproducts and a strong Conway iteration operator. This demonstrates that a categorical semantics for SSA can be constructed from well-known structures. We use Freyd categories to model

sequencing of imperative computations, we use coproducts to model conditionals, and we use Conway iteration to model looping.
We show that any category with this structure is a model of SSA. This establishes that all of the equations we give are sound with respect to the categorical structure.

- We also show, in Section 5.5, that syntax quotiented by the equational theory yields the initial distributive Elgot category. This establishes that our set of syntactic equations is complete, and that there are no equations which the denotational semantics validates, but which cannot be proved syntactically.
  Theoretically, this leads us to the surprising discovery that SSA is actually a syntactic presentation of distributive Elgot categories, thereby establishing a surprising connection between compiler IRs and category theory. (In fact, our calculus is also the first syntactic presentation of distributive Elgot categories.)
  Practically, it means that we can freely mix equational and semantic reasoning depending on what is convenient for the problem at hand: a semantic proof which holds in all λ_{SSA}-models guarantees the existence of a corresponding syntactic sequence of rewrites.
- We proceed in Section 6 to show that this denotational axiomatization is useful in practice. We give a model of TSO weak memory based on Kavanagh and Brookes [38] in Section 6.3, We also give a family of concrete models based on Brookes-style traces [12], which can be instantiated to support the release/acquire model of Dvir et al. [18]. This demonstrates that it is possible to give realistic weak memory models, in a variety of semantic styles, which do not disturb the structure of SSA in fundamental ways. Furthermore, our results constitute the first proof that SSA-based loop transformations are compatible with weak memory.
- Finally, we have substantially mechanized our proofs using the Lean 4 proof assistant. We have mechanized proofs of substitution for our type theory, as well as proofs that the syntax forms the initial model, and that the SPARC TSO semantics forms a valid model of SSA. The denotational semantics and its proof of the soundness of substitution are done on paper.

2 Static Single Assignment Form

2.1 From Three Address Code to SSA

Directly optimizing a source language can be difficult, because surface languages are often very large and have features (such as type inference and overloading) which make it difficult to express sound program equivalences. Elaborating a surface language to a simpler intermediate representation makes it easier to write program analyses and optimizations. One of the earliest compiler IRs introduced is *three-address code* [3] (also known as *register transfer language (RTL)*).

3-address programs consists of a *control-flow graph* (CFG) G with a distinguished, nameless entry block. Each node of the CFG corresponds to a *basic block* β, which is a straight-line sequence of *instructions* $x = f(y, z)$ (hence the name *3-address code*, referring to the typical three variables x, y, z) followed by a *terminator* τ, which can be a (conditional) branch to another basic block. We give a grammar for 3-address code in Figure 1, with some slight adjustments to the usual presentation:

- *Constants* c are interpreted as nullary instructions $c()$.
- We only support nullary tuples $()$ and binary tuples (v, v') as composite values; n-ary tuples (x, y, z) can be encoded as nested binary tuples $((x, y), z)$.
- We allow conditional branches $\mathsf{if}\ o\ \{\tau\}\ \mathsf{else}\ \{\tau'\}$ to be *nested*. This allows us to emulate switch-statements $\mathsf{switch}(o)\{c_1 : \tau_1, ..., c_n : \tau_n\}$ as nested if-statements without introducing spurious basic blocks. Likewise, for uniformity, a return-statement may appear in the branch of a conditional branch.

```
let n = 10;
let mut i = 1;
let mut a = 1;
while i < n {
  a = a * (i + 1)
  i = i + 1;
}
ret a
```

(a) As an imperative program

```
    n = 10;
    i = 1;
    a = 1;
    br loop;
loop : if i < n { br body } else { ret a };
body : t = i + 1;
    a = a * t;
    i = i + 1;
    br loop
```

(b) As 3-address code

Fig. 2. A simple, slightly suboptimal program to compute 10! via multiplication in a loop, represented as typical imperative code and in 3-address code.

$$v ::= x \mid (v, v') \mid ()$$

$$o ::= v \mid f\ v \mid \iota_l\ v \mid \iota_r\ v \mid \text{abort}\ v$$

$$\beta ::= x = o; \beta \mid (x, y) = o; \beta \mid \tau$$

$$\tau ::= \text{br}\ \ell \mid \text{ret}\ v \mid \text{if}\ o\ \{\tau\}\ \text{else}\ \{\tau'\}$$

$$G ::= \beta \mid G;\ \ell : \beta$$

Fig. 1. Grammar for 3-address code

As a concrete example, consider the simple imperative program to compute 10! given in Figure 2. We can normalize our code into 3-address code, as in Figure 2b, by:

- Converting structured control flow (e.g., while) into unstructured jumps between basic blocks.
- Converting composite expressions like $a * (i + 1)$ into a sequence of definitions naming each subexpression. Here, expressions like $a + b$ are syntactic sugar for primitive operations $+(a, b)$.

While functional languages typically rely on *lexical scoping*, where the scope of a variable is determined by its position within the code's nested structure, 3-address code uses a different scoping mechanism based on *dominance*. In particular, a variable x is considered to be in scope at a specific point P if and only if all execution paths from the program's entry point to P pass through a definition D for x. In this case, we say that the definition D *dominates* P. The relation on basic blocks "A dominates B" can in fact be viewed as a tree rooted at the entry block: every pair of basic blocks A, B have a least common ancestor C which dominates them both; we call this tree the *dominator tree* [16].

Even though three address code was designed to simplify flow analysis, many optimizations remain difficult to express in this format. Because a variable's value may be set by multiple definitions throughout the program's execution, variables do not have stable values, and so it is not in general safe to substitute a definition for a variable. To improve our ability to reason about programs, we

introduce the *static single assignment* restriction, originally proposed by Alpern et al. [4], which states that every variable must be defined at exactly one point in the program. Because there is a unique definition for each variable, substitution is valid.

We can we intuitively think of each variable as being defined by an immutable let-binding, and a variable x is in scope at a program point P, if and only if it sunique definition site D_x strictly dominates P.

A given basic block can be converted to SSA form by numbering each definition of a variable, effectively changing references to x to references to x_t, i.e. "x at time t." For example, we could rewrite

$$\begin{array}{l} x = 3y + 5; \\ x = 3x + 2; \\ \text{ret } (3x + 1) \end{array} \quad \approx \quad \begin{array}{l} \text{let } x_0 = 3y + 5; \\ \text{let } x_1 = 3x_0 + 2; \\ \text{ret } (3x_1 + 1) \end{array}$$

This transformation enables algebraic reasoning about expressions involving each x_t. However, since we can only define a variable once in SSA form, expressing programs with loops and branches becomes challenging. For example, naïvely trying to lower the program in Figure 2b into SSA form would not work, since the reference to i in the right-hand-side of the statement $i = i + 1$ can refer to *either* the previous value of i from the last iteration of the loop *or* the original value $i = 1$. The classical solution is to introduce *ϕ-nodes*, which select a value based on the predecessor block from which control arrived. We give the lowering of our program into SSA with ϕ-nodes in Figure 3b.

Cytron et al. [16] introduced the first efficient algorithm to lower a program in 3-address code to valid SSA while introducing a minimum number of ϕ-nodes, making SSA practical for widespread use as an intermediate representation. Unfortunately, ϕ-nodes do not have an obvious operational semantics.

Additionally, they require us to adopt more complex scoping rules than simple dominance-based scoping. For example, in the basic block loop in Figure 3b, i_0 evaluates to 1 if we came from the entry block and to i_1 if we came from body. Similarly, a_0 evaluates to either 1 or a_1 based on the predecessor block. This does not obey dominance-based scoping, since i_0 and i_1 are defined *after* the ϕ-nodes i_0, a_0 that reference them, which seems counterintuitive – after all, variables are typically used after they are defined. In fact, since the value of a ϕ-node is determined by which basic block is our immediate predecessor, we instead need to use the rule that expressions in ϕ-node branches with source S can use any variable y defined at the *end* of S. Note that this is a strict superset of the variables visible for a normal instruction x, which can only use variables y which *dominate* x – i.e., such that *every* path from the entry block to the definition of x goes through y, rather than only those paths which also go through S.

While this rule can be quite confusing, and in particular makes it non-obvious how to assign an operational semantics to ϕ-nodes, the fact that the scoping for ϕ-node branches is based on the source block, rather than the block in which the ϕ-node itself appears, hints at a possible solution. By *moving* the expression in each branch to the *call-site*, we can transition to an isomorphic syntax called basic blocks with arguments (BBA), as illustrated in Figure 5b. In this approach, each ϕ-node – since it lacks side effects and has scoping rules independent of its position in the basic block, depending only on the source of each branch – can be moved to the top of the block. This reorganization allows us to treat each ϕ-node as equivalent to an argument for the basic block, with the corresponding values passed at the jump site. Converting a program from BBA format back to standard SSA form with ϕ-nodes is straightforward: introduce a ϕ-node for each argument of a basic block, and for each branch corresponding to the ϕ-node, add an argument to the jump instruction from the appropriate source block.

```
        n = 10;
        i = 1;
        a = 1;
        br loop;
loop :  if i < n { br body } else { ret a };
body :  let t = i + 1;
        a = a * t;
        i = i + 1;
        br loop
```

(a) 3-address code

```
        let n = 10;
        br loop
loop :  let i0 = φ(entry : 1, body : i1);
        let a0 = φ(entry : 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
```

(b) Converted to SSA form

Fig. 3. Conversion of three address code for the program in Figure 2 to SSA form, requring the insertion of ϕ-nodes for i and a due to control-flow dependent updates. Note how SSA-form can be viewed as "three address code in which all let-bindings are immutable."

We give a formal grammar for basic blocks-with-arguments SSA in Figure 4 [1]. One of the other changes we make is replacing conditional branches if o $\{\tau\}$ else $\{\tau'\}$ with *case-statements* case o $\{\iota_l\ y : \tau, \iota_r\ z : \tau'\}$. In particular, a case-statement if o $\{\tau\}$ else $\{\tau'\}$ may be desugared into a case-statement on a Boolean value $o : \mathbf{1} + \mathbf{1}$.

Note that this grammar no longer needs a separate terminator for returns: we can treat the return point as a distinguished label (with argument) that a program can jump to.

$$v ::= x \mid (v, v') \mid ()$$

$$o ::= v \mid f\ v \mid \iota_l\ v \mid \iota_r\ v \mid \text{abort}\ v$$

$$\beta ::= \text{let}\ x = o; \beta \mid \text{let}\ (x, y) = o; \beta \mid \tau$$

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

$$G ::= G\ \ \ell(x) : \beta \mid \beta$$

Fig. 4. Grammar for basic blocks-with-arguments SSA

This allows us to use dominance-based scoping without any special cases for ϕ-nodes. When considering basic blocks, this means that a variable is visible within the block D where it is defined, starting from the point of its definition. It continues to be visible in all subsequent blocks P that are strictly dominated by D in the control-flow graph (CFG). For example, in Figure 5b:

- The entry block strictly dominates all other blocks by definition; thus, the variable n is visible in loop and body.
- loop strictly dominates body; therefore, the parameters i_0, a_0 to loop are visible in body without the need to pass them as parameters.

[1] Many variants of SSA do not allow variables to appear alone on the right-hand side of assignments, such as $x = y; \beta$. We do not incorporate this restriction, though we could by normalizing even further and substituting $[y/x]\beta$ instead.

```
        let n = 10;
        br loop;
loop :  let i0 = φ(entry : 1, body : i1);
        let a0 = φ(entry : 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) With ϕ-nodes

```
        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

Fig. 5. The program in Figure 2 written in standard SSA (using ϕ nodes), like in LLVM [41], and in basic-blocks with arguments SSA, like in MLIR [42] and Cranelift [59]. The arguments i_0, a_0 corresponding to the ϕ-nodes i_0, a_0 are colored in red and blue, respectively.

- body does *not* strictly dominate loop, since there is a path from the entry block to loop that does not pass through body.

2.2 Type-theoretic SSA

An important insight provided by the BBA format, as discussed by Appel [5] and Kelsey [39], is that a program in SSA form can be interpreted as a collection of tail-recursive functions, where each basic block and branch correspond to a function and tail call, respectively. This yields a natural framework for defining the semantics of SSA and reasoning about optimizations.

A program in BBA is not quite a functional program, because scoping is dominance-based rather than lexically scoped. However, it turns out to be very easy to convert dominance-based scoping into lexical scoping. Observe that the function corresponding to a given basic block B can only be called by other blocks B' having that basic block's parent $P = \mathsf{parent}(B)$ as an ancestor in the dominance tree (as, otherwise, the parent would not actually dominate the block, since we could get to B through B' without passing through P). Moreover, the variables *visible* in B are exactly the variables visible at the end of P; i.e., the variables visible in P and those defined in P.

So if we make the dominance tree explicit in the syntax and tie the binding of variables to this tree structure, then lexical and dominance-based scoping become one and the same. We use this observation to introduce *lexical SSA* in Figure 6. The key idea of this syntax is to, rather than treating the control-flow graph G as a flat collection of basic blocks (with a distinguished block), to instead consider (subtrees of) the dominance tree r, with the root of the tree implicitly being the entry block. We call such subtrees *regions*: we note that they have a single entry (the root) and multiple exits (the leaves), and so generalize the more standard concept of a single-entry-single-exit region in a CFG.

In particular, a *region* r generalizes a basic block β by annotating the terminator τ with a list L of *labeled branches* "$\ell_i(x_i) : \{t_i\}$," yielding a where-*block* "τ where L". Each ℓ_i can only be branched to by τ and the regions t_i, thus syntactically enforcing that the basic block at the root of r (made up of its instructions and terminators) *dominates* all the basic blocks in the subregions t_i (which can only be reached through r). The data of a region r is thus exactly the data contained in a basic block

```
struct BasicBlock {
  // unary/binary let-bindings, collected into a list
  vector<Instruction> instructions;
  Terminator terminator;
  // Dominated basic blocks, forming a subtree of the dominance tree
  // Note that only `terminator` is allowed to jump to blocks in `children`
  map<Label, (Argument, BasicBlock)> children;
}
```

Fig. 7. Data encoded by the grammar in Figure 6

β (its instructions and terminator) together with a set of subregions dominated by r; in C++-like pseudocode, we might represent a region as in Figure 7.

Regions allow us to enforce dominance-based scoping simply by making the variables defined in r visible only in the t_i, which, as previously stated, *must* be dominated by r; i.e., dominance based scoping becomes lexical scoping of where-blocks. It is easy to see (we demonstrate this more rigorously in Section 4.4) that, given a CFG G, there exists some way to annotate its topological sort w.r.t. the dominance relation with where-blocks to obtain a region r which is lexically well-scoped if and only if C is a valid SSA program; we illustrate this process on our running example in Figure 8. Conversely, erasing the where-blocks from a region r and giving the root a name trivially yields a (topologically sorted!) SSA program, establishing an isomorphism between lexical SSA and standard SSA.

$$v ::= x \mid (v, v') \mid ()$$

$$o ::= v \mid f\ v \mid \iota_l\ v \mid \iota_r\ v \mid \text{abort}\ v$$

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

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

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

Fig. 6. Grammar for lexically-scoped SSA

Lexical scoping allows us to apply many of techniques developed in type theory and functional programming for reasoning about program transformations. Indeed, the result of our conversion to lexical scoping looks a lot like the correspondence between SSA and CPS described in Kelsey [39]. We can use this correspondence to guide us in developing an *equational theory* for SSA programs, with the goal of enabling compositional reasoning about program transformations such as:

- *Control-flow rewrites*, such as jump-threading or fusing two identical branches of an if-statement
- *Algebraic rewrites*, such as simplifying arithmetic expressions
- Combinations of the two, such as rewriting "if $x > 0$ then $0 - x$ else x" to "abs(x)".

To help achieve this, we will slightly generalize our syntax by:

(1) Fusing the syntactic categories o, v of operations and values into the syntactic category a of *expressions*
(2) Fusing the syntactic category τ of terminators into the syntactic category of regions r.

```
        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)
```

(a) Dominance-based scoping

```
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)
  }
}
```

(b) Lexical scoping

Fig. 8. Conversion of an SSA program from dominance-based scoping to explicit lexical scoping

(3) Extending expressions a to allow *let-expressions* "let $x = a;\ b$" and *case-expressions* "case $a\ \{\iota_l\ x : b, \iota_r\ y : c\}$"

This leaves us with our final language, λ_{SSA}, the resulting grammar for which is given in Figure 9. It is easy to see that these changes add no expressive power to lexical SSA: we can desugar 1 by introducing names for anonymous sub-expressions, 2 by introducing names for anonymous sub-regions, and 3 by floating out let-bindings and case-statements in the obvious manner, introducing labels as necessary; we discuss this in more detail in Section 4.4.

Change 1 allows us to effectively reason about *substitution*: replacing the value of a variable (which is a value v) with its definition (which is an instruction o). This can be used as a building block for optimizations such as common subexpression elimination and global value numbering; combined with change 3, we can also reason algebraically about "branching" operations like conditional move and absolute value.

On the other hand, 2 lets us replace an unconditional branch $\text{br}\ \ell\ a$ (which is a terminator τ) with the code *pointed to* by the label ℓ (which is a region r), allowing us to perform the jump-threading optimization

$$\text{let } x = a; \text{br } \ell\ b \text{ where } \ell(y) : \{r\} \approx \text{let } x = a; \text{let } y = b; r \text{ where } \ell(y) : \{r\}$$

While both sides of this equation are valid lexical SSA programs, by loosening our syntax slightly, we can *unconditionally* replace jumps with regions, without worrying about jumps nested in case statements or fusing where-blocks. This, especially combined with change 3, makes it much easier to verify optimizations such as

$$\begin{aligned}
&\text{case } a\ \{\iota_l\ x : \text{br } \ell\ (\iota_r\ x), \iota_r\ x : \text{br } \ell\ (\iota_l\ x)\} \text{ where } \ell(y) : \{\text{case } y\ \{\iota_l\ z : \text{ret } (\iota_r\ z), \iota_r\ z : \text{ret } (\iota_l\ z)\}\} \\
&\approx \text{case } a\ \{\iota_l\ x : \text{case } \iota_r\ x\ \{\iota_l\ z : \text{ret } (\iota_r\ z), \iota_r\ z : \text{ret } (\iota_l\ z)\} \\
&\qquad , \iota_r\ x : \text{case } \iota_l\ x\ \{\iota_l\ z : \text{ret } (\iota_r\ z), \iota_r\ z : \text{ret } (\iota_l\ z)\}\} \\
&\approx \text{case } a\ \{\iota_l\ x : \text{ret}(\iota_l\ x), \iota_r\ x : \text{ret}(\iota_r\ x)\} \approx \text{ret } (\text{case } a\ \{\iota_l\ x : \iota_l\ x, \iota_r\ x : \iota_r\ x\}) \approx \text{ret } a
\end{aligned}$$

by repeatedly applying a set of known-good rules, and, moreover, dramatically simplifies the form of the rules themselves.

$$
\begin{aligned}
a, b, c, e ::= {} & x \mid f\ a \mid \text{let } x = a;\ e \\
\mid {} & () \mid (a, b) \mid \text{let } (x, y) = a;\ e \\
\mid {} & \iota_l\ a \mid \iota_r\ a \mid \text{abort } a \mid \text{case } e\ \{\iota_l\ x : a, \iota_r\ y : b\}
\end{aligned}
$$

$$
r, s, t ::= \text{let } x = a; t \mid \text{let } (x, y) = a; t \mid t \text{ where } L \mid \text{br } \ell\ a \mid \text{case } e\ \{\iota_l\ x : s, \iota_r\ y : t\}
$$

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

Fig. 9. Grammar for λ_{SSA}

3 Type Theory

We now give a formal account of λ_{SSA}, starting with the types. Our types are first order, and consists of binary sums $A + B$, products $A \otimes B$, the unit type $\mathbf{1}$, and the empty type $\mathbf{0}$, all parameterised over a set of base types $X \in \mathcal{T}$. We write our set of types as $\text{Ty}(X)$.

A (variable) *context* Γ is a list of *typing hypotheses* $x : A$, where x is a variable name and A is the type of that variable. Similarly, we define a *label-context* to be a list of *labels* $\ell(A)$, where A is the parameter type that must be passed on a jump to the label ℓ. The grammar for types, contexts, and label-contexts is given in Figure 10.

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

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

$$
\mathsf{L} ::= \cdot \mid \mathsf{L}, \ell(A)
$$

Fig. 10. Grammar for λ_{SSA} types, contexts, and label-contexts

Our grammar in Figure 9 was implicitly parameterised over a set of *primitive instructions* $f \in \mathcal{I}$. In particular, for each pair $A, B \in \text{Ty}(X)$ we specify a set of primitive instructions $f \in \mathcal{I}(A, B)$, with a subset of *pure instructions* $\mathcal{I}_\bot(A, B)$. To allow us to write $\mathcal{I}_\epsilon$ for an *effect* $\epsilon \in \{\top, \bot\}$, we denote $\mathcal{I}_\top(A, B) := \mathcal{I}(A, B)$.. In general, we define $\mathcal{I}_\epsilon = \bigcup_{A,B} \mathcal{I}_\epsilon(A, B)$, and $\mathcal{I} = \bigcup_\epsilon \mathcal{I}_\epsilon$.

We'll call a tuple $Sg = (\mathcal{T}, \mathcal{I})$ of types and instructions over these types an λ_{SSA}*-signature*, and, for the rest of this section, work over a fixed signature.

As shown in Figure 9, λ_{SSA} terms are divided into two syntactic categories, each associated with a judgement:

- *Expressions* a, b, c, e, which are typed with the judgement $\Gamma \vdash_\epsilon a : A$, which says that under the typing context Γ, the expression a has type A and effect ϵ. We say a term is *pure* if it has effect $\bot$; note that whether an expression is pure or not depends both on the expression itself and on the purity of the variables used in the expression; this is to allow reasoning about impure substitutions.
- *Regions* r, s, t, which recursively define a lexically-scoped SSA program with a single entry and (potentially) multiple exits. This is typed with the judgement $\Gamma \vdash r \rhd \mathsf{L}$, which states that given that Γ is live at the unique entry point, r will either loop forever or branch to one of the exit labels in $\ell(A) \in \mathsf{L}$ with an argument of type A.

The typing rules for expressions are given in Figure 11. In particular, expressions may be built up from the following fairly standard primitives:

$$\boxed{\Gamma \vdash_\epsilon a : A}$$

$$\frac{\Gamma(x) = A}{\Gamma \vdash_\epsilon x : A}\,\text{var} \qquad \frac{f \in \mathcal{I}_\epsilon(A,B) \quad \Gamma \vdash_\epsilon a : A}{\Gamma \vdash_\epsilon f\ a : B}\,\text{op} \qquad \frac{\Gamma \vdash_\epsilon a : A \quad \Gamma, x : A \vdash_\epsilon b : B}{\Gamma \vdash_\epsilon \text{let } x = a;\ b : B}\,\text{let}_1$$

$$\frac{}{\Gamma \vdash_\epsilon () : \mathbf{1}}\,\text{unit} \qquad \frac{\Gamma \vdash_\epsilon a : A \quad \Gamma \vdash_\epsilon b : B}{\Gamma \vdash_\epsilon (a,b) : A \otimes B}\,\text{pair}$$

$$\frac{\Gamma \vdash_\epsilon e : A \otimes B \quad \Gamma, x : A, y : B \vdash_\epsilon c : C}{\Gamma \vdash_\epsilon \text{let } (x,y) = e;\ c : C}\,\text{let}_2$$

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

$$\frac{\Gamma \vdash_\epsilon e : A + B \quad \Gamma, x : A \vdash_\epsilon a : C \quad \Gamma, y : A \vdash_\epsilon b : C}{\Gamma \vdash_\epsilon \text{case } e\ \{\iota_l\ x : a, \iota_r\ y : b\} : C}\,\text{case}$$

Fig. 11. Rules for typing λ_{SSA} expressions

- A variable x in the context Γ, as typed by var.
- A *primitive instruction* $f \in \mathcal{I}_\epsilon(A,B)$ applied to an expression $\Gamma \vdash_\epsilon a : A$, typed by op
- Unary and binary *let-bindings*, typed by let_1 and let_2 respectively
- A *pair* of expressions $\Gamma \vdash_\epsilon a : A$, $\Gamma \vdash_\epsilon b : B$, typed by pair. Operationally, we interpret this as executing a, and then b, and returning the pair of their values.
- An empty tuple $()$, which types in any context by unit
- Injections, typed by inl and inr
- Pattern matching on sum types, typed by case. Operationally, we interpret this as executing e, and then, if e is a left injection $\iota_l\ x$, executing a with its value (x), otherwise executing b.
- An operator abort e allowing us to abort execution if given a value of the empty type. Since the empty type is a 0-ary sum type, abort can be seen as a case with no branches. Since the empty type is uninhabited, execution can never reach an abort. This can be viewed as a typesafe version of the `unreachable` instruction in LLVM IR.

Traditional presentations of SSA use a boolean type instead of sum types. Naturally, booleans can be encoded with sum types as $\mathbf{1} + \mathbf{1}$. If-then-else is then a case which ignores the unit payloads, so that if $e_1\ \{e_2\}$ else $\{e_3\} := \text{case } e_1\ \{\iota_l\ () : e_2, \iota_r\ () : e_3\}$.

We now move on to *regions*, which can be typed as follows:

- A branch to a label ℓ with pure argument a, typed with br.
- Unary and binary *let-bindings*, typed by let_1 and let_2 respectively
- Pattern matching on sum types, typed by case. Operationally, we interpret this as executing the expression e, and then, if e is a left injection $\iota_l\ x$, executing r with its value (x), otherwise executing s.
- where-*blocks* of the form "r where $(\ell_i(x_i) : \{t_i\})_i$", which consist of a collection of mutually recursive regions $\ell_i(x_i) : \{t_i\}$ and a *terminator region* r which may branch to one of ℓ_i or an exit label.

3.1 Metatheory

We can now begin to state the syntactic metatheory of λ_{SSA}. One of the most important metatheorems, and a basic sanity check of our type theory, is *weakening*; essentially, if something typechecks in a context Δ, and Γ contains all the variables of Δ (written $\Gamma \leq \Delta$, pronounced "Γ *weakens* Δ"),

$$\boxed{\Gamma \vdash r \rhd \mathsf{L}}$$

$$\frac{\Gamma \vdash_{\perp} a : A \quad \mathsf{L}\ \ell = A}{\Gamma \vdash \mathsf{br}\ \ell\ a \rhd \mathsf{L}}\ \mathsf{br} \qquad \frac{\Gamma \vdash_{\epsilon} a : A \quad \Gamma, x : A \vdash r \rhd \mathsf{L}}{\Gamma \vdash \mathsf{let}\ x = a; r \rhd \mathsf{L}}\ \mathsf{let}_1\text{-r}$$

$$\frac{\Gamma \vdash_{\epsilon} e : A \otimes B \quad \Gamma, x : A, y : B \vdash r \rhd \mathsf{L}}{\Gamma \vdash \mathsf{let}\ (x, y) = e; r \rhd \mathsf{L}}\ \mathsf{let}_2\text{-r}$$

$$\frac{\Gamma \vdash_{\epsilon} e : A + B \quad \Gamma, x : A \vdash r \rhd \mathsf{L} \quad \Gamma, y : B \vdash s \rhd \mathsf{L}}{\Gamma \vdash \mathsf{case}\ e\ \{\iota_l\ x : r, \iota_r\ y : s\} \rhd \mathsf{L}}\ \mathsf{case}\text{-r}$$

$$\frac{\Gamma \vdash r \rhd \mathsf{L}, (\ell_i(A_i),)_{i \in I} \quad \forall i \in I. \Gamma, x_i : A_i \vdash t_i \rhd \mathsf{L}, (\ell_j(A_j),)_{j \in I}}{\Gamma \vdash r\ \mathsf{where}\ (\ell_i(x_i) : \{t_i\},)_{i \in I} \rhd \mathsf{L}}\ \mathsf{cfg}$$

Fig. 12. Rules for typing λ_{SSA} regions

then it should typecheck in the context Γ as well. Here, the context with fewer variables appears on the *right*, allowing us to compose typing judgements likeso

$$\Gamma \leq \Delta \implies \Delta \vdash r \rhd \mathsf{L} \implies \Gamma \vdash r \rhd \mathsf{L}$$

As our theory has two types of context; we'd also like to define *label-weakening* $\mathsf{L} \leq \mathsf{K}$, which we should be able to apply in the same manner:

$$\Gamma \vdash r \rhd \mathsf{L} \implies \mathsf{L} \leq \mathsf{K} \implies \Gamma \vdash r \rhd \mathsf{K}$$

If a region r typechecks with exit labels L, and K contains every label in L, then r should obviously also typecheck in K. It follows that in the judgement $\mathsf{L} \leq \mathsf{K}$ the context with fewer labels appears on the *left*-hand side of the judgement: this corresponds precisely to the fact that label-weakening (injection into a coproduct) is semantically dual to variable-weakening (projection from a product), and hence the order is flipped.

We give the (standard) formal rules for weakening $\Gamma \leq \Delta$, and their duals, in the first part of Figure 13.

- wk-nil and lwk-nil say that the empty (label) context weakens itself,
- wk-skip says that if Γ weakens Δ, then $\Gamma, x : A$ also weakens Δ for arbitrary (fresh) x. Dually, lwk-skip says that if L weakens K, then L also weakens $\mathsf{K}, \ell(A)$ for arbitrary (fresh) ℓ.
- wk-cons says that if Γ weakens Δ, then Γ with $x : A$ added weakens $\Delta, x : A$. Likewise, lwk-cons says that if L weakens K, then L with $\ell(A)$ added weakens $\mathsf{K}, \ell(A)$.

It is easy to see that (label) weakening defined in this manner induces a partial order on (label) contexts. Our weakening lemma is then as follows:

Lemma 3.1 (Weakening). *Given $\Gamma \leq \Delta$, $\epsilon \leq \epsilon'$, we have that:*

(a) If $\Delta \vdash_{\epsilon} a : A$, then $\Gamma \vdash_{\epsilon'} a : A$
(b) If $\mathsf{L} \leq \mathsf{K}$ and $\Delta \vdash r \rhd \mathsf{L}$, then $\Gamma \vdash r \rhd \mathsf{K}$
(c) If $\gamma : \Delta \mapsto \Xi$, then $\gamma : \Gamma \mapsto \Xi$
(d) If $\Delta \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}$, then $\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}$

Proof. These are formalized as:

(a) `Term.Wf.wk` in `Typing/Term/Basic.lean`
(b) `Region.Wf.wk` in `Typing/Region/Basic.lean`
(c) Follows from `Term.Subst.Wf.comp` in `Typing/Term/Subst.lean`
(d) Follows from `Region.Subst.Wf.vsubst` in `Typing/Region/LSubst.lean`

$$\boxed{\Gamma \le \Delta}$$

$$\frac{}{\cdot \le \cdot}\ \text{wk-nil} \qquad \frac{\Gamma \le \Delta}{\Gamma, x : A \le \Delta}\ \text{wk-skip} \qquad \frac{\Gamma \le \Delta}{\Gamma, x : A \le \Delta, x : A}\ \text{wk-cons}$$

$$\boxed{\mathsf{L} \le \mathsf{K}}$$

$$\frac{}{\cdot \le \cdot}\ \text{lwk-nil} \qquad \frac{\mathsf{L} \le \mathsf{K}}{\mathsf{L} \le \mathsf{K}, \ell(A)}\ \text{lwk-skip} \qquad \frac{\mathsf{L} \le \mathsf{K}}{\mathsf{L}, \ell(A) \le \mathsf{K}, \ell(A)}\ \text{lwk-cons}$$

$$\boxed{\gamma : \Gamma \mapsto \Delta}$$

$$\frac{}{\cdot : \Gamma \mapsto \cdot}\ \text{sb-nil} \qquad \frac{\gamma : \Gamma \mapsto \Delta \quad \Gamma \vdash_\perp e : A}{\gamma, x \mapsto e : \Gamma \mapsto \Delta, x : A}\ \text{sb-cons}$$

$$\boxed{\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}}$$

$$\frac{}{\Gamma \vdash \cdot : \cdot \rightsquigarrow \mathsf{K}}\ \text{ls-nil}$$

$$\frac{\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K} \quad \Gamma, x : A \vdash r \rhd \mathsf{K} \quad \Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}}{\Gamma \vdash \sigma, \ell(x) \mapsto r : \mathsf{L}, \ell(A) \rightsquigarrow \mathsf{K}}\ \text{ls-cons}$$

Fig. 13. Rules for typing λ_{SSA} weakening and substitution

□

The validity of variable weakening hinges on the fact that all the variables in Δ are also available with the same type in Γ, i.e., if $\Delta \vdash_\epsilon x : A \implies \Gamma \vdash_\epsilon x : A$, then anything which can be typed in Δ can be typed in Γ. So while weakening on *terms* is just the identity, weakening on *derivations* is essentially replacing "variables from Δ" with "variables from Γ." Since none of our typing rules, other than var, make use of variable names, we might ask whether we can repeat essentially the same reasoning to reason about the well-typedness of replacing variables in Γ with arbitrary pure expressions of the same type (i.e., perform a substitution). An assignment of such variables $\gamma : x \mapsto \gamma_x$ is called a *substitution*, which we can type with the judgement $\gamma : \Gamma \mapsto \Delta$ as per the rules given in Figure 13. In particular,

- sb-nil says that the empty substitution takes every context to the empty context.
- sb-cons says that if γ takes Γ to Δ and $\Gamma \vdash_\perp e : A$, then γ with the additional substitution $x \mapsto e$ adjoined takes Γ to $\Delta, x : A$

To *use* a substitution, we simply need to perform standard capture-avoiding substitution (see Figure 37 in the appendix). Substitution satisfies the *substitution lemma* as follows:

Lemma 3.2 (Substitution). *Given $\gamma : \Gamma \mapsto \Delta$, we have that:*

(a) $\Delta \vdash_\epsilon a : A \implies \Gamma \vdash_\epsilon [\gamma]a : A$
(b) $\Delta \vdash r \rhd \mathsf{L} \implies \Gamma \vdash [\gamma]r \rhd \mathsf{L}$
(c) $\gamma_2 : \Delta \mapsto \Xi \implies [\gamma]\gamma_2 : \Gamma \mapsto \Xi$
(d) $\sigma \vdash \Gamma : \mathsf{L} \rightsquigarrow \mathsf{K} \implies [\gamma]\sigma \vdash \Delta : \mathsf{L} \rightsquigarrow \mathsf{K}$

Proof. These are formalized as:

(a) `Term.Wf.subst` in `Typing/Term/Subst.lean`
(b) `Region.Wf.vsubst` in `Typing/Region/VSubst.lean`
(c) `Term.Subst.Wf.comp` in `Typing/Term/Subst.lean`

(d) `Region.Subst.Wf.vsubst` in `Typing/Region/LSubst.lean`

□

Note in particular that this allows us to take the *composition* $[\gamma']\gamma : \Gamma' \mapsto \Delta$ of substitutions $\gamma' : \Gamma' \mapsto \Gamma$ and $\gamma : \Gamma \mapsto \Delta$; the composition associates as expected: $[[\gamma_1]\gamma_2]\gamma_3 = [\gamma_1]([\gamma_2]\gamma_3)$, and has identity $[\mathrm{id}]\gamma = \gamma$, yielding a category of substitutions with variable contexts Γ as objects.

Given a substitution $\gamma : \Gamma \mapsto \Delta$ and context Ξ disjoint from Γ and Δ, we may define a "left extension" operation $\cdot^{\downarrow}_{\Xi}$ yielding $\gamma^{\downarrow}_{\Xi} : \Xi, \Gamma \mapsto \Xi, \Delta$ which appends the identity substitution for each variable in Ξ in the obvious manner:

$$\gamma^{\downarrow}_{\cdot} = \gamma \qquad \gamma^{\downarrow}_{\Xi, x:A} = x \mapsto x, \gamma^{\downarrow}_{\Xi} \tag{1}$$

We may similarly define a "right extension" operation $\cdot^{\uparrow}_{\Xi}$ yielding $\gamma^{\uparrow}_{\Xi} : \Gamma, \Xi \mapsto \Delta, \Xi$ as follows:

$$\gamma^{\uparrow}_{\cdot} = \gamma \qquad \gamma^{\uparrow}_{\Xi, x:A} = \gamma^{\uparrow}_{\Xi}, x \mapsto x \tag{2}$$

In particular, we note that the identity substitution on Γ can be written as $\cdot^{\uparrow}_{\Gamma}$; in general, we have $[\gamma]a = [\Gamma^{\downarrow}_{\Xi}]a = [\gamma^{\uparrow}_{\Xi}]a$. We will usually infer Ξ from context.

One other particularly important form of substitution is that of substituting an expression a for an individual variable x, which we will write $[a/x] := (x \mapsto a)^{\downarrow}$.

Finally, just as we can generalize weakening by substituting expressions for variables via substitution, we can generalize label weakening by substituting *labels* for *(parametrized) regions* via *label substitution*. In particular, a label-substitution $\sigma \vdash \Gamma : \mathsf{L} \rightsquigarrow \mathsf{K}$ maps every label $\ell(A) \in \mathsf{L}$ to a region $\Gamma, x : A \vdash r \rhd \mathsf{K}$ parametrized by $x : A$. As shown in Figure 14, we may then define label-substitution recursively in the obvious manner, mapping $\mathrm{br}\ \ell\ a$ to $[a/x]r$ as a base case. Composition of label-substitutions is pointwise. This allows us to state *label substitution* as follows:

Lemma 3.3 (Label substitution). *Given* $\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}$, *we have that*

(a) $\Gamma \vdash r \rhd \mathsf{L} \implies \Gamma \vdash [\sigma]r \rhd \mathsf{K}$
(b) $\Gamma \vdash \kappa : \mathsf{L} \rightsquigarrow \mathsf{J} \implies \Gamma \vdash [\sigma]\kappa : \mathsf{K} \rightsquigarrow \mathsf{J}$

Proof. These are formalized as:

(a) `Region.Wf.lsubst` in `Typing/Region/LSubst.lean`
(b) `Region.Subst.Wf.comp` in `Typing/Region/LSubst.lean`

□

We may similarly define left and right extensions $\Gamma \vdash \sigma^{\downarrow}_{\mathsf{K}} : \mathsf{L}, \mathsf{J} \rightsquigarrow \mathsf{K}, \mathsf{J}$ and $\Gamma \vdash \sigma^{\uparrow}_{\mathsf{K}} : \mathsf{L}, \mathsf{J} \rightsquigarrow \mathsf{K}, \mathsf{J}$ and for label substitutions $\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}$ in the obvious manner:

$$\sigma^{\uparrow}_{\cdot} = \sigma \qquad \sigma^{\uparrow}_{\mathsf{K}, \ell(A)} = \sigma^{\uparrow}_{\mathsf{K}}, \ell(x) \mapsto \mathrm{br}\ \ell\ x \tag{3}$$

$$\sigma^{\downarrow}_{\cdot} = \sigma \qquad \sigma^{\downarrow}_{\mathsf{K}, \ell(A)} = \ell(x) \mapsto \mathrm{br}\ \ell\ x, \sigma^{\downarrow}_{\mathsf{K}} \tag{4}$$

As for variable substitutions, we will often omit L when it is clear from the context. We also define the shorthand $[\ell/\kappa] = [(\kappa(x) \mapsto \mathrm{br}\ \ell\ x)^{\downarrow}]$ for single-label substitutions.

4 Equational Theory

4.1 Expressions

We can now give an equational theory for λ_{SSA} expressions. In particular, we will inductively define an equivalence relation $\Gamma \vdash_{\epsilon} a \approx a' : A$ on terms a, a' for each context Γ, effect ϵ, and type A. For each of the rules we will present, we assume the rule is valid if and only if *both sides* of the rule are

$$(\sigma, \ell(x) \mapsto r)(\ell, a) = [a/x]r \qquad (\sigma, \kappa(x) \mapsto r)(\ell, a) = \sigma(\ell, a) \qquad (\cdot)(\ell, a) = \text{br } \ell\ a$$

$$[\sigma](\text{br } \ell\ a) = \sigma(\ell, a) \qquad [\sigma](\text{let } x = a; r) = \text{let } x = a; [\sigma]r$$
$$[\sigma](\text{let } (x, y) = e; r) = \text{let } (x, y) = e; [\sigma]r$$
$$[\sigma](\text{case } e\ \{\iota_l\ x : r, \iota_r\ y : s\}) = \text{case } e\ \{\iota_l\ x : [\sigma]r, \iota_r\ y : [\sigma]s\}$$
$$[\sigma](r \text{ where } (\ell_i(x_i) : \{t_i\},)_i) = ([\sigma]r) \text{ where } (\ell_i(x_i) : \{[\sigma]t_i\},)_i$$

$$[\sigma](\cdot) = \cdot \qquad [\sigma](\sigma', \ell(x) \mapsto r) = ([\sigma]\sigma', \ell(x) \mapsto [\sigma]r)$$

Fig. 14. Capture-avoiding label substititon for λ_{SSA} regions and label substitutions; in particular, we assume bound variables and labels are α-converted so as not to appear in σ.

well-typed. We also assume that variables are α-converted as appropriate to avoid shadowing; our formalization uses de Bruijn indices, but we stick with names in this exposition for simplicity.

The rules for this relation can be roughly split into *rewriting rules*, which denote when two particular expressions have equivalent semantics, and *congruence rules*, which govern how rewrites can be composed to enable equational reasoning. In particular, our congruence rules, given in Figure 15, consist of:

- refl, symm, trans, which state that $\Gamma \vdash_\epsilon \cdot \approx \cdot : A$ is reflexive, transitive, and symmetric respectively for each choice of Γ, ϵ, A, and therefore an equivalence relation.
- let_1, let_2, pair, inl, inr, case, and abort, which state that $\Gamma \vdash_\epsilon \cdot \approx \cdot : A$ is a *congruence* with respect to the corresponding expression constructor, and, in particular, that the expression constructors are well-defined functions on the quotient of expressions up to $\approx$.

We also include the following *type-directed* rules as part of our congruence relation:

- initial, which equates *all* terms in a context containing the empty type $\mathbf{0}$, since we will deem any such context to be *unreachable* by control flow. In particular, any instruction or function call returning $\mathbf{0}$ is assumed to diverge.
- terminal, which equates all *pure* terms of unit type $\mathbf{1}$. Note that *impure* terms may be disequal, since while their result values are the same, their side effects may differ!

We may group the rest of our rules according to the relevant constructor, i.e. let (unary and binary) and case. In particular, for unary let, we have the following rules, summarized in Figure 16:

- let_1-β, which allows us to substitute the bound variable in x the let-statement $\text{let } x = a;\ b$ with its definition a, yielding $[a/x]b$. Note that we require $\Gamma \vdash_\perp a : A$; i.e., a must be *pure*.
- let_1-η, which is the standard η-rule for let. This is included as a separate rule since, while it follows trivially from β for pure a, we also want to consider *impure* expressions.
- Rules let_1-op, let_1-let_1, let_1-let_2, let_1-abort, and let_1-case which allow us to "pull" a let-statement out of any of the other expression constructors; operationally, this is saying that the bound expression we pull out is evaluated before the rest of the let-binding.
  For example, let_1-case says that, if both $\text{let } z = \text{case } e\ \{\iota_l\ x : a, \iota_r\ y : b\};\ d$ and $\text{case } e\ \{\iota_l\ x : \text{let } z = a;\ d, \iota_r\ y : \text{let } z = b;\ d\}y$, are well typed, then both must have the same behaviour:
  (1) Compute e
  (2) If $e = \iota_l\ e_l$, compute $[e_l/x]a$, else, if $e = \iota_r\ e_r$, compute $[e_r/y]b$; store this value as z
  (3) Compute d given our value for z
  Note in particular that, since both sides are well-typed, d cannot depend on either x or y.

$$\frac{\Gamma \vdash_\epsilon a : A}{\Gamma \vdash_\epsilon a \approx a : A}\ \text{refl} \qquad \frac{\Gamma \vdash_\epsilon a \approx b : A \quad \Gamma \vdash_\epsilon b \approx c : A}{\Gamma \vdash_\epsilon a \approx c : A}\ \text{trans} \qquad \frac{\Gamma \vdash_\epsilon a \approx b : A}{\Gamma \vdash_\epsilon b \approx a : A}\ \text{symm}$$

$$\frac{\Gamma \vdash_\epsilon a \approx a' : A \quad \Gamma, x : A \vdash_\epsilon b \approx b' : B}{\Gamma \vdash_\epsilon \text{let } x = a;\ b \approx \text{let } x = a';\ b' : B}\ \text{let}_1$$

$$\frac{\Gamma \vdash_\epsilon a \approx a' : A \quad \Gamma \vdash_\epsilon b \approx b' : B}{\Gamma \vdash_\epsilon (a, b) \approx (a', b) : A \otimes B}\ \text{pair}$$

$$\frac{\Gamma \vdash_\epsilon e \approx e' : A \otimes B \quad \Gamma, x : A, y : B \vdash_\epsilon c \approx c' : C}{\Gamma \vdash_\epsilon \text{let } (x, y) = e;\ c \approx \text{let } (x, y) = e';\ c' : C}\ \text{let}_2$$

$$\frac{\Gamma \vdash_\epsilon a \approx a' : A}{\Gamma \vdash_\epsilon \iota_l\ a \approx \iota_l\ a' : A + B}\ \text{inl} \qquad \frac{\Gamma \vdash_\epsilon b \approx b' : B}{\Gamma \vdash_\epsilon \iota_r\ b \approx \iota_r\ b' : A + B}\ \text{inr}$$

$$\frac{\Gamma \vdash_\epsilon e \approx e' : A + B \quad \Gamma, x : A \vdash_\epsilon a \approx a' : C \quad \Gamma, y : B \vdash_\epsilon b \approx b' : C}{\Gamma \vdash_\epsilon \text{case } e\ \{\iota_l\ x : a, \iota_r\ y : b\} \approx \text{case } e'\ \{\iota_l\ x : a', \iota_r\ y : b'\} : C}\ \text{case}$$

$$\frac{\Gamma \vdash_\epsilon a \approx a' : \mathbf{0}}{\Gamma \vdash_\epsilon \text{abort } a \approx \text{abort } a' : A}\ \text{abort}$$

$$\frac{\Gamma \vdash_\epsilon a : A \quad \Gamma \vdash_\epsilon a' : A \quad \Gamma \vdash_\bot e : \mathbf{0}}{\Gamma \vdash_\epsilon a \approx a' : A}\ \text{initial} \qquad \frac{\Gamma \vdash_\bot a : \mathbf{1} \quad \Gamma \vdash_\bot a' : \mathbf{1}}{\Gamma \vdash_\epsilon a \approx a' : \mathbf{1}}\ \text{terminal}$$

Fig. 15. Congruence rules for λ_{SSA} expressions

$$\frac{\Gamma \vdash_\bot a : A \quad \Gamma, x : A \vdash_\epsilon b : B}{\Gamma \vdash_\epsilon \text{let } x = a;\ b \approx [b/x]a : B}\ \text{let}_1\text{-}\beta \qquad \frac{\Gamma \vdash_\epsilon a : A}{\Gamma \vdash_\epsilon \text{let } x = a;\ x \approx a : A}\ \text{let}_1\text{-}\eta$$

$$\frac{f \in \mathcal{I}_\epsilon(A, B) \quad \Gamma \vdash_\epsilon a : A \quad \Gamma, y : B \vdash_\epsilon c : C}{\Gamma \vdash_\epsilon \text{let } y = f\ a;\ c \approx \text{let } x = a;\ \text{let } y = f\ x;\ c : C}\ \text{let}_1\text{-op}$$

$$\frac{\Gamma \vdash_\epsilon a : A \quad \Gamma, x : A \vdash_\epsilon b : B \quad \Gamma, y : B \vdash_\epsilon c : C}{\Gamma \vdash_\epsilon \text{let } y = (\text{let } x = a;\ b);\ c \approx \text{let } x = a;\ \text{let } y = b;\ c : C}\ \text{let}_1\text{-let}_1$$

$$\frac{\Gamma \vdash_\epsilon e : A \times B \quad \Gamma, x : A, y : C \vdash_\epsilon c : C \quad \Gamma, z : C \vdash_\epsilon d : D}{\Gamma \vdash_\epsilon \text{let } z = (\text{let } (x, y) = e;\ c);\ d \approx \text{let } (x, y) = e;\ \text{let } z = c;\ d : D}\ \text{let}_1\text{-let}_2$$

$$\frac{\Gamma \vdash_\epsilon a : \mathbf{0} \quad \Gamma, y : A \vdash_\epsilon b : B}{\Gamma \vdash_\epsilon \text{let } y = \text{abort } b;\ b \approx \text{let } x = a;\ \text{let } y = \text{abort } x;\ b : B}\ \text{let}_1\text{-abort}$$

$$\frac{\Gamma \vdash_\epsilon e : A + B \quad \Gamma, x : A \vdash_\epsilon a : C \quad \Gamma, y : B \vdash_\epsilon b : C \quad \Gamma, z : C \vdash_\epsilon d : D}{\begin{array}{l}\Gamma \vdash_\epsilon \text{let } z = (\text{case } e\ \{\iota_l\ x : a, \iota_r\ y : b\});\ d \\ \quad \approx \text{case } e\ \{\iota_l\ x : \text{let } z = a;\ d, \iota_r\ y : \text{let } z = b;\ d\} : D\end{array}}\ \text{let}_1\text{-case}$$

Fig. 16. Rewriting rules for λ_{SSA} unary let expressions

Handling the other type constructors is a little simpler: by providing a "binding" rule, we generally only need to specify how to interact with let_1, as well as an η and β rule; interactions with the other constructors can then be derived. For example, consider the rules for let_2 given in 17; we have:

- let_2-η, which is the standard η-rule for binary let-bindings

$$\frac{\Gamma \vdash_\epsilon a : A \quad \Gamma \vdash_\epsilon b : B \quad \Gamma, x : A, y : B \vdash_\epsilon c : C}{\Gamma \vdash_\epsilon \text{let } (x,y) = (a,b);\ c \approx \text{let } x = a;\ \text{let } y = b;\ c : C}\ \text{let}_2\text{-pair}$$

$$\frac{\Gamma \vdash_\epsilon e : A \otimes B}{\Gamma \vdash_\epsilon \text{let } (x,y) = e;\ (x,y) \approx e : A \otimes B}\ \text{let}_2\text{-}\eta$$

$$\frac{\Gamma \vdash_\epsilon e : A \otimes B \quad \Gamma, x : A, y : B \vdash_\epsilon c : C}{\Gamma \vdash_\epsilon \text{let } (x,y) = e;\ c \approx \text{let } z = e;\ \text{let } (x,y) = z;\ c : C}\ \text{let}_2\text{-bind}$$

$$\frac{\Gamma \vdash_\epsilon a : A \quad \Gamma, x : A \vdash_\epsilon c : C \quad \Gamma, y : B \vdash_\epsilon d : C}{\Gamma \vdash_\epsilon \text{case } \iota_l\ a\ \{\iota_l\ x : c, \iota_r\ y : d\} \approx \text{let } x = a;\ c : C}\ \text{case-inl}$$

$$\frac{\Gamma \vdash_\epsilon b : B \quad \Gamma, x : A \vdash_\epsilon c : C \quad \Gamma, y : B \vdash_\epsilon d : C}{\Gamma \vdash_\epsilon \text{case } \iota_r\ b\ \{\iota_l\ x : c, \iota_r\ y : d\} \approx \text{let } y = b;\ d : C}\ \text{case-inr}$$

$$\frac{\Gamma \vdash_\epsilon e : A + B}{\Gamma \vdash_\epsilon \text{case } e\ \{\iota_l\ x : \iota_l\ x, \iota_r\ y : \iota_r\ y\} \approx e : A + B}\ \text{case-}\eta$$

$$\frac{\Gamma \vdash_\epsilon e : A + B \quad \Gamma, x : A \vdash_\epsilon c : C \quad \Gamma, y : B \vdash_\epsilon d : C}{\Gamma \vdash_\epsilon \text{case } e\ \{\iota_l\ x : c, \iota_r\ y : d\} \approx \text{let } z = e;\ \text{case } z\ \{\iota_l\ x : c, \iota_r\ y : d\} : C}\ \text{case-bind}$$

Fig. 17. Rewriting rules for λ_{SSA} binary let and case expressions

- let_2-pair, which acts like a slightly generalized β-rule, since we can derive β reduction as follows: given pure $\Gamma \vdash_\perp a : A$ and $\Gamma \vdash_\perp b : B$, we have

$$(\text{let } (x,y) = (a,b);\ c) \approx (\text{let } x = a;\ \text{let } y = b;\ c) \approx ([a/x](\text{let } y = b;\ c)) \approx ([a/x][b/y]c)$$

  We state the rule in a more general form to allow for impure a and b, as well as to simplify certain proofs.
- let_2-bind, which allows us to "pull" out the bound value of a binary let-expression into its own unary let-expression; operationally, this just says that we execute the bound value before executing the binding itself.

This is enough to allow us to define our interactions with the other expression constructors: for example, to show that we can lift an operation f out of a binary let-binding, rather than adding a separate rule, we can instead derive (types omitted for simplicity) it from let_2-bind and let_1-op as follows:

$$\begin{aligned}(\text{let } (x,y) = f\ a;\ b) &\approx (\text{let } z_f = f\ a;\ \text{let } (x,y) = z;\ b)\\ &\approx (\text{let } z_a = a;\ \text{let } z_f = f\ z_a;\ \text{let } (x,y) = z;\ b)\\ &\approx (\text{let } z_a = a;\ \text{let } (x,y) = f\ z_a;\ b)\end{aligned}$$

Similarly, it is enough to give η, β, and binding rules for case expressions. In particular, we have that

- case-inl and case-inr serve as β-reduction rules, telling us that case-expressions given an injection as an argument have the expected operational behaviour. Note that we reduce to a let-expression rather than perform a substitution to allow for impure discriminants.
- case-η is the standard η-rule for case-expressions.
- case-bind allows us to "pull" out the bound value of the discriminant into it's own let-expression; again, operationally, this just says that we need to evaluate the discriminant before executing the case-expression.

It's interesting that this is enough, along with the let-case rule and friends, to derive the distributivity properties we would expect well-behaved case-expressions to have. For example, we have that

$$\begin{aligned} f(\mathsf{case}\ e\ \{\iota_l\ x : a, \iota_r\ y : b\}) &\approx (\mathsf{let}\ z = \mathsf{case}\ e\ \{\iota_l\ x : a, \iota_r\ y : b\};\ f\ z) \\ &\approx \mathsf{case}\ e\ \{\iota_l\ x : \mathsf{let}\ z = a;\ f\ z, \iota_r\ y : \mathsf{let}\ z = b;\ f\ z\} \\ &\approx \mathsf{case}\ e\ \{\iota_l\ x : f\ a, \iota_r\ y : f\ b\} \end{aligned}$$

and likewise for more complicated distributivity properties involving, e.g., let-bindings.

The case for the other constructors is even more convenient: no additional rules are required at all to handle operations, pairs, and injections. For example, we can derive the expected bind-rule for operations as follows:

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

This completes the equational theory for λ_{SSA} terms; in Section 5.5, we will show that this is enough to state a completeness theorem.

4.2 Regions

We now come to the equational theory for regions, which is similar to that for terms, except that we also need to support control-flow graphs. As before, we will split our rules into a set of *congruence rules* and, for each region constructor, *rewriting rules* based on that constructor's semantics. Our congruence rules, given in Figure 18, are quite standard; we have:

- As for terms, refl, trans, and symm state that $\Gamma \vdash \cdot \approx \cdot \rhd \mathsf{L}$ is an equivalence relation for all Γ, L.
- Similarly, let_1, let_2, case, and cfg state that $\Gamma \vdash \cdot \approx \cdot \rhd \mathsf{L}$ is a congruence over the respective region constructors; *as well as* the equivalence relation on terms $\Gamma \vdash_\epsilon \cdot \approx \cdot : A$.
- initial states that any context containing the empty type $\mathbf{0}$ equates all regions, by a similar reasoning to the rules for terms. Note that we do not require an analogue to the terminal rule (for example, for regions targeting $\mathsf{L} = \ell(\mathbf{1})$), since it will follow from the version for terms; this is good, since the concept of a "pure" region has not yet been defined.

Our rewriting rules for unary let-statements, given in Figure 19, are analogous to those for unary let-expressions:

- let_1-β allows us to perform β-reduction of *pure* expressions into regions; unlike for terms, we do not need an η-rule
- Exactly like for let-expressions, let_1-op, let_1-let_1, let_1-let_2, let_1-abort, and let_1-case allow us to pull out nested subexpressions of the bound value of a let-statement into their own unary let-statement

Similarly to expressions, binary let-statements and case-statements need only the obvious β rule and binding rule, with all the interactions with other constructors derivable; these rules are given in Figure 20. Note in particular that η-rules are not necessary, as these are derivable from binding and the η-rules for expressions.

Dealing with where-blocks, on the other hand, is a little bit more complicated, as shown by the number of rules in Figure 21. One difficulty is that, unlike the other region constructors, we will need an η-rule as well as *two* β-rules. The latter are simple enough to state:

- For ℓ_k defined in a where-block, cfg-β_1 says that we can replace a branch to ℓ_k with argument a with a let-statement binding a to the corresponding body t_k's argument x_k.
- For κ *not* defined in a where-block, cfg-β_2 says that a branch to κ within the where-block has the same semantics as if the where-block was not there; hence, it can be removed.

$$\frac{\Gamma \vdash r \rhd \mathsf{L}}{\Gamma \vdash r \approx r \rhd \mathsf{L}}\ \text{refl} \qquad \frac{\Gamma \vdash r \approx s \rhd \mathsf{L} \quad \Gamma \vdash s \approx t \rhd \mathsf{L}}{\Gamma \vdash r \approx t \rhd \mathsf{L}}\ \text{trans} \qquad \frac{\Gamma \vdash r \approx s \rhd \mathsf{L}}{\Gamma \vdash s \approx r \rhd \mathsf{L}}\ \text{symm}$$

$$\frac{\Gamma \vdash_\epsilon a \approx a' : A \quad \Gamma, x : A \vdash r \approx r' \rhd \mathsf{L}}{\Gamma \vdash \text{let } x = a; r \approx \text{let } x = a'; r' \rhd \mathsf{L}}\ \text{let}_1 \qquad \frac{\Gamma \vdash_\epsilon e \approx e' : A \otimes B \quad \Gamma, x : A, y : B \vdash r \approx r' \rhd \mathsf{L}}{\Gamma \vdash \text{let } (x, y) = e; r \approx \text{let } (x, y) = e'; r' \rhd \mathsf{L}}\ \text{let}_2$$

$$\frac{\Gamma \vdash_\epsilon e \approx e' : A + B \quad \Gamma, x : A \vdash r \approx r' \rhd \mathsf{L} \quad \Gamma, y : B \vdash s \approx s' \rhd \mathsf{L}}{\Gamma \vdash \text{case } e\ \{\iota_l\ x : r, \iota_r\ y : s\} \approx \text{case } e'\ \{\iota_l\ x : r', \iota_r\ y : s'\} \rhd \mathsf{L}}\ \text{case}$$

$$\frac{\Gamma \vdash r \approx r' \rhd \mathsf{L}, (\ell_i(A_i),)_{i \in I} \quad \forall i \in I. \Gamma, x_i : A_i \vdash t_i \approx t_i' \rhd \mathsf{L}, (\ell_j(A_j),)_{j \in I}}{\Gamma \vdash r \text{ where } (\ell_i(x_i : A_i) : \{t_i\},)_{i \in I} \approx r' \text{ where } (\ell_i(x_i : A_i) : \{t_i'\},)_{i \in I} \rhd \mathsf{L}}\ \text{cfg}$$

$$\frac{\Gamma \vdash r \rhd \mathsf{L} \quad \Gamma \vdash s \rhd \mathsf{L} \quad \exists x, \Gamma\ x = \mathbf{0}}{\Gamma \vdash r \approx s \rhd \mathsf{L}}\ \text{initial}$$

Fig. 18. Congruence rules for λ_{SSA} regions

$$\frac{\Gamma \vdash_\perp a : A \quad \Gamma, x : A \vdash r \rhd \mathsf{L}}{\Gamma \vdash \text{let } x = a; r \approx [a/x]r \rhd \mathsf{L}}\ \text{let}_1\text{-}\beta$$

$$\frac{f \in \mathcal{I}_\epsilon(A, B) \quad \Gamma \vdash_\epsilon a : A \quad \Gamma, y : B \vdash r \rhd \mathsf{L}}{\Gamma \vdash \text{let } y = f\ a; r \approx \text{let } x = a; \text{let } y = f\ x; r \rhd \mathsf{L}}\ \text{let}_1\text{-op}$$

$$\frac{\Gamma \vdash_\epsilon a : A \quad \Gamma, x : A \vdash_\epsilon b : B \quad \Gamma, y : B \vdash r \rhd \mathsf{L}}{\Gamma \vdash \text{let } y = (\text{let } x = a;\ b); r \approx \text{let } x = a; \text{let } y = b; r \rhd \mathsf{L}}\ \text{let}_1\text{-let}_1$$

$$\frac{\Gamma \vdash_\epsilon e : A \otimes B \quad \Gamma, x : A, y : B \vdash_\epsilon c : C \quad \Gamma, z : C \vdash r \rhd \mathsf{L}}{\Gamma \vdash \text{let } z = (\text{let } (x, y) = e;\ c); r \approx \text{let } (x, y) = e; \text{let } z = c; r \rhd \mathsf{L}}\ \text{let}_1\text{-let}_2$$

$$\frac{\Gamma \vdash_\epsilon e : A + B \quad \Gamma, x : A \vdash_\epsilon a : C \quad \Gamma, y : B \vdash_\epsilon b : C \quad \Gamma, z : C \vdash r \rhd \mathsf{L}}{\begin{array}{c}\Gamma \vdash \text{let } z = (\text{case } e\ \{\iota_l\ x : a, \iota_r\ y : b\}); r \\ \approx \text{case } e\ \{\iota_l\ x : \text{let } z = a; r, \iota_r\ y : \text{let } z = b; r\} \rhd \mathsf{L}\end{array}}\ \text{let}_1\text{-case}$$

$$\frac{\Gamma \vdash_\epsilon a : \mathbf{0} \quad \Gamma, y : A \vdash r \rhd \mathsf{L}}{\Gamma \vdash \text{let } y = \text{abort } a; r \approx \text{let } x = a; \text{let } y = \text{abort } x; r \rhd \mathsf{L}}\ \text{let}_1\text{-abort}$$

Fig. 19. Rewriting rules for λ_{SSA} unary let-statements

To state our η-rule, however, we will need to introduce some more machinery. Given a mapping from a set of labels ℓ_i to associated regions t_i, we may define the *control-flow graph substitution* cfgs $\{(\ell_i(x_i) : \{t_i\},)_i\}$ pointwise as follows:

$$\text{cfgs } \{(\ell_i(x_i) : \{t_i\},)_i\}\ \kappa\ a := (\text{br } \kappa\ a \text{ where } (\ell_i(x_i) : \{t_i\},)_i) \tag{5}$$

In general, we may derive, for any label-context L (assuming cfgs $\{\cdot\}$ acts uniformly on the labels κ in L as described above), the following rule:

$$\frac{\forall i \in I. \Gamma, x_i : A_i \vdash t_i \rhd \mathsf{L}, (\ell_j(A_j),)_{j \in I}}{\Gamma \vdash \text{cfgs } \{(\ell_i(x_i) : \{t_i\},)_{i \in I}\} : \mathsf{L}, (\ell_j(A_j),)_{j \in I} \rightsquigarrow \mathsf{L}}\ \text{cfgs} \tag{6}$$

$$\frac{\Gamma \vdash_\epsilon a : A \quad \Gamma \vdash_\epsilon b : B \quad \Gamma, x : A, y : B \vdash r \rhd \mathsf{L}}{\Gamma \vdash \mathsf{let}\ (x, y) = (a, b); r \approx \mathsf{let}\ x = a; \mathsf{let}\ y = b; r \rhd \mathsf{L}}\ \text{let}_2\text{-pair}$$

$$\frac{\Gamma \vdash_\epsilon e : A \otimes B \quad \Gamma, x : A, y : B \vdash r \rhd \mathsf{L}}{\Gamma \vdash \mathsf{let}\ (x, y) = e; r \approx \mathsf{let}\ z = e; \mathsf{let}\ (x, y) = z; r \rhd \mathsf{L}}\ \text{let}_2\text{-bind}$$

$$\frac{\Gamma \vdash_\epsilon a : A \quad \Gamma, x : A \vdash r \rhd \mathsf{L} \quad \Gamma, y : B \vdash s \rhd \mathsf{L}}{\Gamma \vdash \mathsf{case}\ \iota_l\ a\ \{\iota_l\ x : r, \iota_r\ y : s\} \approx \mathsf{let}\ x = a; r \rhd \mathsf{L}}\ \text{case-inl}$$

$$\frac{\Gamma \vdash_\epsilon b : B \quad \Gamma, x : A \vdash r \rhd \mathsf{L} \quad \Gamma, y : B \vdash s \rhd \mathsf{L}}{\Gamma \vdash \mathsf{case}\ \iota_r\ b\ \{\iota_l\ x : r, \iota_r\ y : s\} \approx \mathsf{let}\ y = b; s \rhd \mathsf{L}}\ \text{case-inr}$$

$$\frac{\Gamma \vdash_\epsilon e : A + B \quad \Gamma, x : A \vdash r \rhd \mathsf{L} \quad \Gamma, y : B \vdash s \rhd \mathsf{L}}{\Gamma \vdash \mathsf{case}\ e\ \{\iota_l\ x : r, \iota_r\ y : s\} \approx \mathsf{let}\ z = e; \mathsf{case}\ z\ \{\iota_l\ x : r, \iota_r\ y : s\} \rhd \mathsf{L}}\ \text{case-bind}$$

Fig. 20. Rewriting rules for λ_{SSA} binary let-statements and case-statements

Our η-rule, cfg-η, says that any where-block of the form r where $(\ell_i(x_i) : \{t_i\},)_i$ has the same semantics as the label-substitution $[\mathsf{cfgs}\ \{(\ell_i(x_i) : \{t_i\},)_i\}]r$, which in effect propagates the where-block to the branches of r, if any. While we called this rule cfg-η, it also functions similarly to a binding rule in that it allows us to derive many of the expected commutativity properties of where; for example, we have that

$$\begin{aligned}\mathsf{let}\ y = a;\ r\ \mathsf{where}\ (\ell_i(x_i) : \{\mathsf{br}\ \ell_j\ a_j\},)_i &\approx [\mathsf{cfgs}\ \{(\ell_i(x_i) : \{\mathsf{br}\ \ell_j\ a_j\},)_i\}](\mathsf{let}\ y = a;\ r)\\ &\approx \mathsf{let}\ y = a;\ [\mathsf{cfgs}\ \{(\ell_i(x_i) : \{\mathsf{br}\ \ell_j\ a_j\},)_i\}]r\\ &\approx \mathsf{let}\ y = a;\ r\ \mathsf{where}\ (\ell_i(x_i) : \{\mathsf{br}\ \ell_j\ a_j\},)_i\end{aligned}$$

One particularly important application of the η-rule for control-flow graphs is in validating the rewrite

$$\frac{\Gamma \vdash_\epsilon a : A + B \quad \Gamma, x : A \vdash s \rhd \mathsf{L} \quad \Gamma, y : B \vdash t \rhd \mathsf{L}}{\begin{aligned}\Gamma \vdash \mathsf{case}\ a\ \{\iota_l\ x : s, \iota_r\ y : t\} \approx (\mathsf{case}\ a\ \{\iota_l\ x : \mathsf{br}\ \ell\ x, \iota_r\ y : \mathsf{br}\ \ell'\ y\})\\ \mathsf{where}\ \ell(x) : \{s\}, \ell'(y) : \{t\} \rhd \mathsf{L}\end{aligned}}\ \text{case2cfg} \tag{7}$$

In addition, we also add as an axiom the ability to get rid of a single, trivially nested where-block; this is given as the rule codiag.

To be able to soundly perform equational rewriting, we will need the *uniformity* property, which is described by the rule uni. In essence, this lets us commute pure expressions with loop bodies, enabling rewrites (in imperative style) like

$$\mathsf{loop}\ \{x = x + 1; \mathsf{if}\ p\ 3x\ \{\mathsf{ret}\ 3x\}\} \qquad \approx \qquad y = 3x; \mathsf{loop}\ \{y = y + 3; \mathsf{if}\ p\ y\ \{\mathsf{ret}\ y\}\} \tag{8}$$

Note that substitution alone would not allow us to derive Equation 8 above, since x and y change each iteration, and hence, in SSA, would need to become parameters as follows:

$$\begin{aligned}&\mathsf{br}\ \ell\ x\ \mathsf{where}\ \ell(y) : \{\mathsf{let}\ x' = y + 1; \mathsf{if}\ p\ 3x'\ \{\mathsf{ret}\ 3x'\}\ \mathsf{else}\ \{\mathsf{br}\ \ell\ x'\}\}\\ &\qquad \approx \mathsf{let}\ y = 3x; \mathsf{br}\ \kappa\ y\ \mathsf{where}\ \kappa(y) : \{\mathsf{let}\ y' = y + 3; \mathsf{if}\ p\ y'\ \{\mathsf{ret}\ y'\}\ \mathsf{else}\ \{\mathsf{br}\ \kappa\ y'\}\}\end{aligned} \tag{9}$$

The actual rule is quite complicated, so let's break it down point by point. Assume we are given:

- A region $\Gamma, y : B \vdash s \rhd \mathsf{L}, \kappa(B)$ taking "input" y of type B and, as "output," jumping to a label κ with an argument of type B. We'll interpret branches to any other label (i.e. any label in L) as a (divergent) "side effect."

- A region $\Gamma, x : A \vdash t \rhd \mathsf{L}, \ell(A)$ taking "input" x of type A and, as "output," jumping to a label ℓ with an argument of type A.
- A *pure* expression $\Gamma, x : A \vdash_\bot e : B$ parameterised by a value x of type A

Suppose further that the following condition holds:

$$\Gamma, x : A \vdash [e/y]s \approx t \text{ where } \ell(x) : \{\mathsf{br}\ \kappa\ e\} \rhd \mathsf{L}, \kappa(B)$$

That is, the following two programs are equivalent:

(a) Given input x, evaluate e and, taking it's output to be input y, evaluate s, (implicitly) yielding as output a new value of y. In imperative pseudocode,

$$y = e; y = s$$

(b) Given input x, evaluate t and, taking it's output to be the *new* value of x, evaluate e, (implicitly) yielding as output a new value y. In imperative pseudocode,

$$x = t; y = e$$

Then, for any well-typed entry block $\Gamma \vdash r \rhd \mathsf{L}, \ell(A)$ (which can produce an appropriate input $x : A$ at label ℓ), we have that

$$\Gamma \vdash (r \text{ where } \ell(x) : \{\mathsf{br}\ \kappa\ e\}) \text{ where } \kappa(y) : \{s\} \approx r \text{ where } t \rhd \mathsf{L}$$

i.e., in imperative pseudocode,

$$x = r; y = e; \mathsf{loop}\ \{y = s\} \approx x = r; \mathsf{loop}\ \{x = t\} \tag{10}$$

since

$$y = e; y = s; y = s; \ldots \quad \approx \quad x = t; y = e; y = s; \ldots \quad \approx \quad x = t; x = t; y = e; \ldots \quad \approx \quad \cdots \tag{11}$$

where s and t may branch out of the loop. Note that, due to let_1-β, cfg-η, and cfg-β_1, this is equivalent to the rule

$$\frac{\Gamma, x : A \vdash [e/y]s \approx [\ell(x) \mapsto \mathsf{br}\ \kappa\ e]t \rhd \mathsf{L}, \kappa(B)}{\Gamma \vdash (([\ell(x) \mapsto \mathsf{br}\ \kappa\ e]r) \text{ where } \kappa(y) : \{s\}) \approx (r \text{ where } \ell(x) : \{t\}) \rhd \mathsf{L}}\ \text{uni'}$$

$$\text{where} \qquad \Gamma \vdash r \rhd \mathsf{L}, \ell(A) \qquad \Gamma, x : A \vdash_\bot e : B \qquad \Gamma, y : B \vdash s \rhd \mathsf{L}, \kappa(B) \qquad \Gamma, x : A \vdash t \rhd \mathsf{L}, \ell(A) \tag{12}$$

Going back to our concrete example from Equation 9, if we first substitute the let-binding $y = 3x$ on the RHS, we get

$$\begin{aligned} &\mathsf{br}\ \ell\ x \text{ where } \ell(y) : \{\mathsf{let}\ x' = y + 1; \mathsf{if}\ p\ 3x'\ \{\mathsf{ret}\ 3x'\}\ \mathsf{else}\ \{\mathsf{br}\ \ell\ x'\}\} \\ &\qquad \approx \mathsf{br}\ \kappa\ 3x \text{ where } \kappa(y) : \{\mathsf{let}\ y' = y + 3; \mathsf{if}\ p\ y'\ \{\mathsf{ret}\ y'\}\ \mathsf{else}\ \{\mathsf{br}\ \kappa\ y'\}\} \end{aligned} \tag{13}$$

Now, instantiate uni′ (Equation 12) by taking:

- $s = \mathsf{let}\ y' = y + 3;\ \mathsf{if}\ p\ y'\ \{\mathsf{ret}\ y'\}\ \mathsf{else}\ \{\mathsf{br}\ \kappa\ y'\}$ to be the loop body on the RHS
- $e = 3x$
- $r = \mathsf{br}\ \ell\ x$
- $t = \mathsf{let}\ x' = y + 1; \mathsf{if}\ p\ 3x'\ \{\mathsf{ret}\ 3x'\}\ \mathsf{else}\ \{\mathsf{br}\ \ell\ x'\}$ to be the loop body on the LHS

$$\frac{\Gamma \vdash_\bot a : A_k \quad \forall i \in I. \Gamma, x_i : A_i \vdash t_i \rhd \mathsf{L}, (\ell_j(A_j),)_{j\in I}}{\Gamma \vdash \mathsf{br}\ \ell_k\ a \ \mathsf{where}\ (\ell_i(x_i) : \{t_i\},)_{i\in I} \approx (\mathsf{let}\ x_k = a; t_k)\ \mathsf{where}\ (\ell_i(x_i) : \{t_i\},)_{i\in I} \rhd \mathsf{L}}\ \text{cfg-}\beta_1$$

$$\frac{\Gamma \vdash_\bot b : B \quad \forall i \in I. \Gamma, x_i : A_i \vdash t_i \rhd \mathsf{L}, (\ell_j(A_j),)_{j\in I} \quad \mathsf{L}\ \kappa = B \quad \kappa \notin \{\ell_i \mid i \in I\}}{\Gamma \vdash \mathsf{br}\ \kappa\ b \ \mathsf{where}\ (\ell_i(x_i) : \{t_i\},)_{i\in I} \approx \mathsf{br}\ \kappa\ b \rhd \mathsf{L}}\ \text{cfg-}\beta_2$$

$$\frac{\Gamma \vdash r \rhd \mathsf{L}, (\ell_i(A_i),)_{i\in I} \quad \forall i \in I. \Gamma, x_i : A_i \vdash t_i \rhd \mathsf{L}, (\ell_j(A_j),)_{j\in I}}{\Gamma \vdash r \ \mathsf{where}\ (\ell_i(x_i) : \{t_i\},)_{i\in I} \approx [\mathsf{cfgs}\ \{(\ell_i(x_i) : \{t_i\},)_{i\in I}\}]r \rhd \mathsf{L}}\ \text{cfg-}\eta$$

$$\frac{\Gamma \vdash r \rhd \mathsf{L}, \ell(A) \quad \Gamma, y : A \vdash s \rhd \mathsf{L}, \ell(A), \kappa(A)}{\Gamma \vdash r \ \mathsf{where}\ \ell(x) : \{\mathsf{br}\ \kappa\ x \ \mathsf{where}\ \kappa(y) : \{s\}\} \approx r \ \mathsf{where}\ \ell(y) : \{[\ell/\kappa]s\} \rhd \mathsf{L}}\ \text{codiag}$$

$$\frac{\Gamma \vdash r \rhd \mathsf{L}, \ell(A) \quad \Gamma, x : A \vdash \mathsf{let}\ y = e;\ s \approx t \ \mathsf{where}\ \ell(x) : \{\mathsf{br}\ \kappa\ e\} \rhd \mathsf{L}, \kappa(B)}{\Gamma \vdash (r \ \mathsf{where}\ \ell(x) : \{\mathsf{br}\ \kappa\ e\}) \ \mathsf{where}\ \kappa(y) : \{s\} \approx r \ \mathsf{where}\ t \rhd \mathsf{L}}\ \text{uni}$$

where $\Gamma, x : A \vdash_\bot e : B$, $\Gamma, y : B \vdash s \rhd \mathsf{L}, \kappa(B)$, and $\Gamma, x : A \vdash t \rhd \mathsf{L}, \ell(A)$

$$\frac{\begin{array}{c}\Gamma \vdash r \rhd \mathsf{L}, (\ell_i(A_i),)_{i\in I} \\ \Gamma \vdash \sigma : (\ell_i(A_i),)_{i\in I} \rightsquigarrow (\kappa_j(B_j),)_{j\in J} \quad \forall j \in J. \Gamma, x_j : B_j \vdash t_j \rhd \mathsf{L}, (\ell_i(A_i),)_{i\in I}\end{array}}{\begin{array}{l}\Gamma \vdash ([\sigma^\dagger]r) \ \mathsf{where}\ (\kappa_j(x_j) : \{[\sigma^\dagger]t_j\},)_{j\in J} \\ \quad \approx r \ \mathsf{where}\ (\ell_i(x_i) : \{[(\kappa_j(x_j) \mapsto t_j,)_{j\in J}{}^\dagger](\sigma\ \ell_i\ x_i)\},)_{i\in I}\end{array}}\ \text{dinat}$$

Fig. 21. Rewriting rules for λ_{SSA} where-blocks

It's easy to see that $(([\ell(x) \mapsto \mathsf{br}\ \kappa\ e]r) \ \mathsf{where}\ \kappa(y) : \{s\})$ and $(r \ \mathsf{where}\ t)$ are syntactically equal to the *RHS* and *LHS* of our desired result (Equation 13). So, it suffices to verify that

$$\begin{aligned}\Gamma, x : A \vdash [e/y]s &\approx \mathsf{let}\ y' = 3x + 3;\ \mathsf{if}\ p\ y'\ \{\mathsf{ret}\ y'\}\ \mathsf{else}\ \{\mathsf{br}\ \kappa\ y'\} \\ &\approx \mathsf{let}\ y' = 3(x + 1);\ \mathsf{if}\ p\ y'\ \{\mathsf{ret}\ y'\}\ \mathsf{else}\ \{\mathsf{br}\ \kappa\ y'\} \\ &\approx \mathsf{let}\ x' = x + 1;\ \mathsf{let}\ y' = 3x';\ \mathsf{if}\ p\ y'\ \{\mathsf{ret}\ y'\}\ \mathsf{else}\ \{\mathsf{br}\ \kappa\ y'\} \\ &\approx \mathsf{let}\ x' = x + 1;\ \mathsf{if}\ p\ 3x'\ \{\mathsf{ret}\ 3x'\}\ \mathsf{else}\ \{\mathsf{br}\ \kappa\ 3x'\} \\ &\approx [\ell(x) \mapsto \mathsf{br}\ \kappa\ e]t\end{aligned}$$

as desired. The reason why we require e to be *pure* in the uniformity rule is that impure expressions do not necessarily commute with infinite loops, even if they commute with any finite number of iterations of the loop. For example, if hi is some effectful operation (say, printing "hello"), it is quite obvious that,

$$\mathsf{hi}; x = x + 1; \mathsf{if}\ x = y\ \{\mathsf{ret}\ y\} \approx x = x + 1; \mathsf{if}\ x = y\ \{\mathsf{hi}; \mathsf{ret}\ y\}; \mathsf{hi} \tag{14}$$

whereas

$$\mathsf{hi}; \mathsf{loop}\{x = x + 1; \mathsf{if}\ x = y\ \{\mathsf{ret}\ y\}\} \not\approx \mathsf{loop}\{x = x + 1; \mathsf{if}\ x = y\ \{\mathsf{hi}; \mathsf{ret}\ y\}\}; \mathsf{hi} \tag{15}$$

since, in particular, we may have $y \leq x$, in which case the loop will never exit and hence hi will never be executed.

The derivable rule uni' (Equation 12) illuminates a very important potential use for uniformity; namely, formalizing rewrites like those in Figure 22. In particular, consider a program of the form

$$\Gamma \vdash ([\ell(x) \mapsto \mathsf{br}\ \kappa\ e]r) \ \mathsf{where}\ \kappa(y) : \{[\ell(x) \mapsto \mathsf{br}\ \kappa\ e]s\} \rhd \mathsf{L}$$

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

(a) Program from Figure 8 after substituting lets.

```
let n = 10;
br loop (0, 1)
where loop(x, y) : {
  let (i₀, a₀) = (x + 1, y * (x + 1));
  if i₀ < n {
    br loop(i₀, a₀)
  } else {
    ret a₀
  }
}
```

(b) Equivalent to Figure 22c by *dinaturality*

```
let n = 10;
br loop(
  let (x, y) = (0, 1);
  (x + 1, y * (x + 1))
)
where loop(i₀, a₀) : {
  if i₀ < n {
    br loop(
      let (x, y) = (i₀, a₀);
      (x + 1, y * (x + 1))
    )
  } else {
    ret a₀
  }
}
```

(c) Equivalent to Figure 22a by congruence

Fig. 22. Decomposing multi-block rewrites (from 22c to 22a, and therefore to the more optimal program 22b) into simple algebraic steps. By verifying each step, we can verify complex optimizations through decomposition.

where

- $\Gamma \vdash r \rhd \mathsf{L}, \ell(A)$
- $\Gamma, y : B \vdash s \rhd \mathsf{L}, \ell(A)$
- $\Gamma, x : A \vdash_{\perp} e : B$ is pure

Then we have that

$$[e/y][\ell(x) \mapsto \mathsf{br}\ \kappa\ (e)]s \approx [\ell(x) \mapsto \mathsf{br}\ \kappa\ [e/y](e)][e/y]s \approx [\ell(x) \mapsto \mathsf{br}\ \kappa\ (e)][e/y]s$$

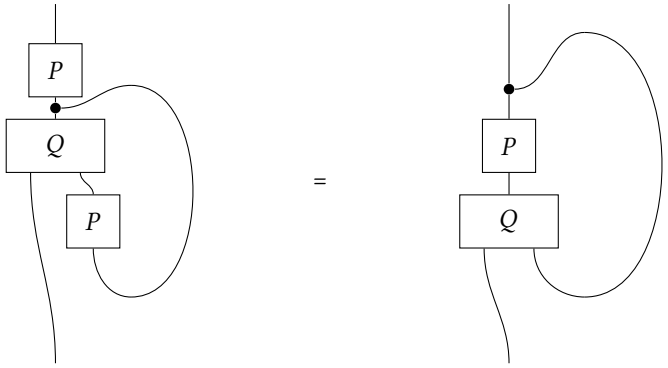


Fig. 23. Dinaturality on a structured loop

and therefore that

$$\Gamma \vdash ([\ell(x) \mapsto \mathsf{br}\ \kappa\ e]r)\ \mathsf{where}\ \kappa(z) : \{[\ell(x) \mapsto \mathsf{br}\ \kappa\ e]s\} \approx r\ \mathsf{where}\ \ell(x) : \{[e/y]s\} \approx r\ \mathsf{where}\ \ell(x) : \{\mathsf{let}\ y = e; s\}$$

In particular, for example, we can then easily derive the rewrite from Figure 22b to Figure 22c by noting the *equalities* (an equivalence would be enough, of course)

$$\begin{array}{l} \mathsf{if}\ i_0 < n\ \{ \\ \quad \mathsf{br}\ \mathsf{loop}(\mathsf{let}\ (x,y) = (i_0,a_0);\ (x+1, y*x+1)) \\ \}\ \mathsf{else}\ \{\mathsf{ret}(a_0)\} \\ = \\ [\mathsf{loop}(i_0,a_0) \mapsto \mathsf{let}\ (x,y) = (i_0,a_0);\ (x+1,y*x+1)](\mathsf{if}\ i_0 < n\ \{ \\ \quad\quad \mathsf{br}\ \mathsf{loop}(i_0,a_0) \\ \quad \}\ \mathsf{else}\ \{\mathsf{ret}(a_0)\}) \end{array}$$

and

$$\mathsf{let}\ n = 10; \mathsf{br}\ \mathsf{loop}(\mathsf{let}\ (x,y) = (0,1); (x+1, y*(x+1))) = [\mathsf{loop}(i_0,a_0) \mapsto \mathsf{let}\ (x,y) = (i_0,a_0);\ (x+1,y*x+1)](\mathsf{let}\ n = 10; \mathsf{br}\ \mathsf{loop}(0,1))$$

Rewrites like this are an instance of the principle we call *dinaturality*, which, for structured control-flow, can be best expressed as an equivalence between the control-flow graphs in Figure 23. Unlike in the case of uniformity, however, this is true even when the program fragment P is *impure*, since, unlike in the case of general uniformity, we do not commute P over an infinite number of iterations. Our final rewriting rule, dinat, generalises the above rewrite from sequential composition on a structured control-flow graph to label substitution on an arbitrary control-flow graph.

We require a separate rule for impure dinaturality as it allows us to relate unary and n-ary where-blocks and, in particular, use this relationship to interconvert between data-flow and control-flow.

This means we now have enough equations to derive the flattening of nested where-blocks:

$$\frac{\Gamma \vdash r \rhd \mathsf{L}, (\ell_i(A_i),)_{i\in I}, (\kappa_j(B_j),)_{j\in I} \quad \forall i \in I.\Gamma, y_i : B_i \vdash s_i \rhd \mathsf{L}, (\ell_j(A_j),)_{j\in J}, (\kappa_k(B_k),)_{k\in K}}{\begin{array}{r}\Gamma \vdash (r \text{ where } (\kappa_k(y_k) : \{s_k\}),)_{k\in K} \text{ where } (\ell_i(x_i) : \{t_i\}),)_{i\in I} \\ \approx r \text{ where } (\kappa_k(y_k) : \{s_k\},)_{k\in K}, (\ell_i(x_i) : \{t_i\}),)_{i\in I} \rhd \mathsf{L}\end{array}} \text{cfg-fuse} \quad \forall i \in I.\Gamma, x_i : A_i \vdash t_i \rhd \mathsf{L}, (\ell_j(A_j),)_{j\in I} \tag{16}$$

Rather than directly giving derivation trees for such auxilliary rules, it is more convenient to give a denotational proof. However, the completeness of our equational theory (proved in Section 5.5) means that the semantic equality implies the existence of the requisite derivation tree. A proof can be found in Lemma C.7 in the appendix. This is one of the benefits of having a completeness result: it lets us switch freely between equational and denotational modes of reasoning.

There are some other basic rules we may want to use which turn out to be derivable from our existing set. For example, while re-ordering labels in a where-block looks like a no-op in our named syntax, to rigorously justify the following rule actually requires dinaturality (with the permutation done via a label-substitution):

$$\frac{\Gamma \vdash r \rhd \mathsf{L}, (\ell_i(A_i),)_{i\in I} \quad \forall i \in I.\Gamma, x_i : A_i \vdash t_i \rhd \mathsf{L}, (\ell_j(A_j),)_{j\in I} \quad \sigma \text{ permutation}}{\Gamma \vdash r \text{ where } (\ell_i(x_i) : \{t_i\},)_{i\in I} \approx r \text{ where } (\ell_{\sigma_i}(x_{\sigma_i}) : \{t_{\sigma_i}\},)_{i\in I} \rhd \mathsf{L}} \text{perm-cfg} \tag{17}$$

Note the implicit use of the fact that if some region r typechecks in some label-context L, then it typechecks in any permutation of L, which is again proven by label-substitution.

4.3 Metatheory

We can now begin to investigate the metatheoretic properties of our equational theory. As a first sanity check, we can verify that weakening, label-weakening, and loosening of effects all respect our equivalence relation, as stated in the following lemma:

Lemma 4.1 (Weakening (Rewriting)). *Given $\Gamma \leq \Delta$, $\mathsf{L} \leq \mathsf{K}$, and $\epsilon \leq \epsilon'$, we have that*

(a) $\Delta \vdash_\epsilon a \approx a' : A \implies \Gamma \vdash_{\epsilon'} a \approx a' : A$
(b) $\Delta \vdash r \approx r' \rhd \mathsf{L} \implies \Gamma \vdash r \approx r' \rhd \mathsf{K}$

Proof. These are formalized as:

(a) `Term.InS.wk_congr` and `Term.InS.wk_eff_congr` in `Rewrite/Term/Setoid.lean`
(b) `Region.InS.vwk_congr` and `Region.InS.lwk_congr` in `Rewrite/Region/Setoid.lean`

□

It is straightforward to verify that these are indeed equivalence relations. In fact, it turns out that substitution and label-substitution both respect these equivalences, in the following precise sense:

Lemma 4.2 (Congruence (Substitution)). *Given $\gamma \approx \gamma' : \Gamma \mapsto \Delta$, we have that*

(a) $\Delta \vdash_\epsilon a \approx a' : A \implies \Gamma \vdash_\epsilon [\gamma]a \approx [\gamma']a' : A$
(b) $\Delta \vdash r \approx r' \rhd \mathsf{L} \implies \Gamma \vdash [\gamma]r \approx [\gamma']r' \rhd \mathsf{L}$
(c) $\rho \approx \rho' : \Delta \mapsto \Xi \implies [\gamma]\rho \approx [\gamma']\rho' : \Gamma \mapsto \Xi$
(d) $\sigma \approx \sigma' \vdash \Delta : \mathsf{L} \rightsquigarrow \mathsf{K} \implies [\gamma]\sigma \approx [\gamma']\sigma' \vdash \Gamma : \mathsf{L} \rightsquigarrow \mathsf{K}$

Proof. These are formalized as:

(a) `Term.InS.subst_congr` in `Rewrite/Term/Setoid.lean`
(b) `Region.InS.vsubst_congr` in `Rewrite/Region/Setoid.lean`
(c) `Term.Subst.InS.comp_congr` in `Rewrite/Term/Setoid.lean`
(d) `Region.Subst.InS.vsubst_congr` in `Rewrite/Region/LSubst.lean`

$$\frac{}{\cdot \approx \cdot : \Gamma \mapsto \cdot}\ \text{sb-nil} \qquad \frac{\Gamma \vdash_\perp a \approx a' : A \quad \gamma \approx \gamma' : \Gamma \mapsto \Delta}{\gamma, x \mapsto a \approx \gamma', x \mapsto a' : \Gamma \mapsto \Delta, x : A}\ \text{sb-cons}$$

$$\frac{\gamma \approx \gamma' : \Gamma \mapsto \Delta}{\gamma, x \mapsto a \approx \gamma' : \Gamma \mapsto \Delta}\ \text{sb-skip-l} \qquad \frac{\gamma \approx \gamma' : \Gamma \mapsto \Delta}{\gamma \approx \gamma', x \mapsto a' : \Gamma \mapsto \Delta}\ \text{sb-skip-r}$$

$$\frac{}{\cdot \approx \cdot \vdash \cdot : \mathsf{K} \rightsquigarrow}\ \text{ls-nil} \qquad \frac{\Gamma, x : A \vdash r \approx r' \rhd \mathsf{K} \quad \sigma \approx \sigma' \vdash \Gamma : \mathsf{L} \rightsquigarrow \mathsf{K}}{\sigma, \ell(x) \mapsto r \approx \sigma', \ell(x) \mapsto r' \vdash \Gamma : \mathsf{L}, \ell(A) \rightsquigarrow \mathsf{K}}\ \text{ls-cons}$$

$$\frac{\sigma \approx \sigma' \vdash \Gamma : \mathsf{L} \rightsquigarrow \mathsf{K}}{\sigma, \ell(x) \mapsto r \approx \sigma' \vdash \Gamma : \mathsf{L} \rightsquigarrow \mathsf{K}}\ \text{ls-skip-l} \qquad \frac{\sigma \approx \sigma' \vdash \Gamma : \mathsf{L} \rightsquigarrow \mathsf{K}}{\sigma \approx \sigma', \ell(x) \mapsto r' \vdash \Gamma : \mathsf{L} \rightsquigarrow \mathsf{K}}\ \text{ls-skip-r}$$

$$\frac{\gamma \approx \gamma' : \Gamma, x : A \mapsto \Delta, x : A}{\gamma \approx \gamma' : \Gamma \mapsto \Delta}\ \text{sb-id} \qquad \frac{\sigma \approx \sigma' \vdash \Gamma : \mathsf{L}, \ell(A) \rightsquigarrow \mathsf{K}, \ell(A)}{\sigma \approx \sigma' \vdash \Gamma : \mathsf{L} \rightsquigarrow \mathsf{K}}\ \text{ls-id}$$

Fig. 24. Rules for the equivalence relation on λ_{SSA} substitutions and label-substitutions

□

In particular, note that this lemma uses an equivalence relation on substitutions and label-substitutions: this is just the obvious pointwise extension of the equivalence relation on terms and regions respectively. We give the rules for this relation in Figure 24 in the interests of explicitness.

Lemma 4.3 (Congruence (Label Substitution)). *Given $\sigma \approx \sigma' \vdash \Gamma : \mathsf{L} \rightsquigarrow \mathsf{K}$, we have that*

(a) $\Gamma \vdash r \approx r' \rhd \mathsf{L} \implies \Gamma \vdash [\sigma]r \approx [\sigma']r' \rhd \mathsf{K}$
(b) $\kappa \approx \kappa' \vdash \Gamma : \mathsf{L} \rightsquigarrow \mathsf{J} \implies [\sigma]\kappa \approx [\sigma']\kappa' \vdash \Gamma : \mathsf{K} \rightsquigarrow \mathsf{J}$

Proof. These are formalized as:

(a) `Region.InS.lsubst_congr` in `Rewrite/Region/LSubst.lean`
(b) `Region.LSubst.InS.comp_congr` in `Rewrite/Region/LSubst.lean`

□

This means, in particular, that, substitution and label-substitution are well-defined operators on equivalence classes of terms, which will come in handy later as we set out to prove completeness in Section 5.5.

4.4 Standard SSA

The relaxation of SSA to λ_{SSA} allows us to state our equational theory and handle substitution more conveniently. However, this approach may lead readers to question whether we have truly provided a type-theoretic presentation of SSA as it is used in practice. The argument given in the introduction for why this is the case can be re-stated as the following series of explicit claims:

(1) Every λ_{SSA} region can be converted to an equivalent lexical SSA region
(2) Every lexical SSA region can be erased to a well-formed SSA program by removing "where"
(3) Every well-formed SSA program can be typed as a lexical SSA region purely by adding "where" . Moreover, a way to do this can be found in nearly linear time.
(4) Two lexical SSA regions which erase to the same SSA program are equivalent

Claim 4 implies that the mappings given in Claim 2 and Claim 3 establish an equivalence (up to our equational theory) between lexical SSA and traditional SSA. This means that the choice of where-bracketing that converts an SSA program into a lexical SSA program is semantically

irrelevant. In other words, where-bracketing only makes dominance syntactically apparent, letting us give syntax-directed rules for typing SSA programs.

We establish this claim in two phases. First, we will show that λ_{SSA} can be transformed (in linear space) into an equivalent lexical SSA program. Next, we will show that ordinary and lexical SSA can be transformed into one another by adding and removing where-blocks.

Converting λ_{SSA} directly to lexical SSA can be unwieldy. Therefore, we break the transformation down into two lowering passes:

(1) We first convert λ_{SSA} into to a subset corresponding to A-normal form (ANF) extended with mutually recursive where-bindings
(2) We then convert the resulting ANF regions into lexical SSA

4.4.1 From λ_{SSA} to A-Normal Form. Our first step is to extend the work of Chakravarty et al. [14], by providing an algorithm for converting between λ_{SSA} and ANF in an equivalence-preserving way. We define the ANF regions, whose grammar is given in Figure 25, to be λ_{SSA} regions with expressions restricted to operations o; equivalently, we can view ANF regions as lexical SSA regions with the syntactic category of regions r and terminators τ fused. We may now introduce the following predicates on syntax:

- $\text{IsANF}(r)$ means that the region r is in ANF
- $\text{IsSSA}(r)$ means that the region r is a lexical SSA region. We note that $\text{IsSSA}(r) \implies \text{IsANF}(r)$.
- $\text{IsVal}(a)$ means that the expression a can be parsed as a value v
- $\text{IsOp}(a)$ means that the expression a can be parsed as an instruction o. We note that $\text{IsVal}(a) \implies \text{IsOp}(a)$.

We can define a syntactic function to convert an λ_{SSA} region to ANF inductively as follows:

$$\begin{aligned}
\text{ANF}(\text{br } \ell\ a) &= \text{ANF}_{\text{let}}(x, a, \text{br } \ell\ x) \\
\text{ANF}(\text{let } x = a;\ r) &= \text{ANF}_{\text{let}}(x, a, \text{ANF}(r)) \\
\text{ANF}(\text{let } (x, y) = a;\ r) &= \text{ANF}_{\text{let}}(z, a, \text{let } (x, y) = z; \text{ANF}(r)) \\
\text{ANF}(\text{case } a\ \{\iota_l\ x : r, \iota_r\ y : s\}) &= \text{ANF}_{\text{let}}(z, a, \text{case } z\ \{\iota_l\ x : \text{ANF}(r), \iota_r\ y : \text{ANF}(s)\}) \\
\text{ANF}(r \text{ where } (\ell_i(x_i) : \{t_i\},)_i) &= \text{ANF}(r) \text{ where } (\ell_i(x_i) : \{\text{ANF}(t_i)\},)_i
\end{aligned} \tag{18}$$

where we define $\text{ANF}_{\text{let}}(x, a, r)$ by induction on expressions a as follows

$$\begin{aligned}
\text{ANF}_{\text{let}}(x, a, r) &= (\text{let } x = a; r) && \text{if IsOp}(a) \\
\text{ANF}_{\text{let}}(x, f\ e, r) &= \text{ANF}_{\text{let}}(y, e, \text{let } x = f\ y; r) \\
\text{ANF}_{\text{let}}(x, (\text{let } y = e;\ a), r) &= \text{ANF}_{\text{let}}(y, e, \text{ANF}_{\text{let}}(x, a, r)) \\
\text{ANF}_{\text{let}}(x, (e_1, e_2), r) &= \text{ANF}_{\text{let}}(y_1, e_1, \text{ANF}_{\text{let}}(y_2, e_2, \text{ANF}_{\text{let}}(x, (y_1, y_2), r))) \\
\text{ANF}_{\text{let}}(x, (\text{let } (y, z) = e;\ a), r) &= \text{ANF}_{\text{let}}(w, e, (\text{let } (y, z) = w; \text{ANF}_{\text{let}}(x, a, r))) \\
\text{ANF}_{\text{let}}(x, \iota_l\ e, r) &= \text{ANF}_{\text{let}}(y, e, (\text{let } x = \iota_l\ y; r)) \\
\text{ANF}_{\text{let}}(x, \iota_r\ e, r) &= \text{ANF}_{\text{let}}(y, e, (\text{let } x = \iota_r\ y; r)) \\
\text{ANF}_{\text{let}}(x, \text{case } e\ \{\iota_l\ y : a, \iota_r\ z : b\}, r) &= \text{ANF}_{\text{let}}(w, e, \text{case } w \\
&\qquad \{\iota_l\ y : \text{ANF}_{\text{let}}(x, a, \text{br } \ell\ x), \iota_r\ z : \text{ANF}_{\text{let}}(x, b, \text{br } \ell\ x)\} \\
&\qquad \text{where } \ell(x) : \{\text{ANF}(r)\}) \\
\text{ANF}_{\text{let}}(x, \text{abort } e, r) &= \text{ANF}_{\text{let}}(y, e, (\text{let } x = \text{abort } y; r))
\end{aligned} \tag{19}$$

$$v ::= x \mid (v, v') \mid ()$$

$$o ::= v \mid f\ v \mid \iota_l\ v \mid \iota_r\ v \mid \text{abort}\ v$$

$$r, s, t ::= \text{let}\ x = o; t \mid \text{let}\ (x, y) = o; t \mid t\ \text{where}\ L \mid \text{br}\ \ell\ o \mid \text{case}\ e\ \{\iota_l\ o : s, \iota_r\ y : t\}$$

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

Fig. 25. Grammar for ANF regions

We note that, to get to ANF, we don't actually need the new label, and can instead replace each branch with $\mathsf{ANF}_{\mathsf{let}}(x, a, r)$; introducing the label is only necessary to guarantee that the transformation to ANF is *linear-space*. The specification of the functions we just defined is given by the following lemma:

Lemma 4.4 (A-normalization). *Given an arbitrary region r, we have that*

- $\mathsf{IsANF}(\mathsf{ANF}(r))$
- *If* $\Gamma \vdash r \rhd \mathsf{L}$, *then* $\Gamma \vdash r \approx \mathsf{ANF}(r) \rhd \mathsf{L}$

Similarly, given an arbitrary variable x, expression a and region r If we are also given an arbitrary expression a, then

- $\mathsf{IsANF}(r) \implies \mathsf{IsANF}(\mathsf{ANF}_{\mathsf{let}}(x, a, r))$
- *If* $\Gamma \vdash_\epsilon a : A$ *and* $\Gamma, x : A \vdash r \rhd \mathsf{L}$, *then* $\Gamma \vdash \text{let}\ x = a; r \approx \mathsf{ANF}_{\mathsf{let}}(x, a, r) \rhd \mathsf{L}$

Proof. See Appendix C.2 □

4.4.2 From ANF to (Lexical) SSA. We would now like to define a syntactic linear-space transformation from regions r to equivalent lexical SSA regions $\mathsf{SSA}(r)$. We do this by giving a pair of mutually syntactic transformations $\mathsf{SSA}(r)$ converting a region to SSA, and $\mathsf{SSA}_{\mathsf{a}}(r, L)$ from ANF regions r *targeting* labels in L to lexical SSA.

$$\mathsf{SSA}(r) = \mathsf{SSA}_{\mathsf{a}}(\mathsf{ANF}(r), \cdot)$$

$$\begin{aligned}
\mathsf{SSA}_{\mathsf{a}}(t, (\ell_i(x_i) : \{t_i\},)_i) &= t\ \text{where}\ (\ell_i(x_i) : \{t_i\},)_i \qquad \text{where } t \text{ is a terminator} \\
\mathsf{SSA}_{\mathsf{a}}((\text{let}\ x = a; r), L) &= (\text{let}\ x = a; \mathsf{SSA}_{\mathsf{a}}(r, L)) \\
\mathsf{SSA}_{\mathsf{a}}((\text{let}\ (x, y) = a; r), L) &= (\text{let}\ (x, y) = a; \mathsf{SSA}_{\mathsf{a}}(r, L)) \\
\mathsf{SSA}_{\mathsf{a}}((\text{case}\ a\ \{\iota_l\ x : s, \iota_r\ y : t\}), L) &= (\text{case}\ a\ \{\iota_l\ x : \text{br}\ \ell_l\ x, \iota_r\ y : \text{br}\ \ell_r\ y\}) \\
&\qquad \text{where}\ L, \ell_l(x) : \{\mathsf{SSA}(s)\}, \ell_r(y) : \{\mathsf{SSA}(t)\} \\
\mathsf{SSA}_{\mathsf{a}}((r\ \text{where}\ (\ell_i(x_i) : \{t_i\},)_i), L) &= \mathsf{SSA}_{\mathsf{a}}(r, (L, (\ell_i(x_i) : \{\mathsf{SSA}(t_i)\},)_i))
\end{aligned} \tag{20}$$

The specification of these functions is as follows:

Lemma 4.5 (SSA conversion). *Given an arbitrary region r,*

- $\mathsf{IsSSA}(\mathsf{SSA}(r))$
- *If* $\Gamma \vdash r \rhd \mathsf{L}$, *then* $\Gamma \vdash r \approx \mathsf{SSA}(r) \rhd \mathsf{L}$

In particular,

- *If* $\mathsf{IsANF}(r)$ *and* $\forall i, \mathsf{IsSSA}(t_i)$, *then* $\mathsf{IsSSA}(\mathsf{SSA}_{\mathsf{a}}(r, (\ell_i(x_i) : \{t_i\},)_i))$
- *If* $\Gamma \vdash r \rhd \mathsf{L}, (\ell_i(A_i),)_i$ *and* $\forall i, \Gamma \vdash t_i \rhd \mathsf{L}, (\ell_i(A_i),)_i$, *then* $\Gamma \vdash (r\ \text{where}\ (\ell_i(x_i) : \{t_i\},)_i) \approx \mathsf{SSA}_{\mathsf{a}}(r, (\ell_i(x_i) : \{t_i\},)_i) \rhd \mathsf{L}$

Proof. See Appendix C.2 □

4.4.3 From (Lexical) SSA to SSA. We now come to the final part of our argument: that lexical SSA is equivalent to standard SSA, where we take the basic-blocks-with-arguments dialect described in Figure 4 as standard. We begin by recalling how, in Figure 7, we illustrated how a lexical SSA region can alternatively be interpreted as a tuple of:

- A basic block β (the *entry block*), consisting of,
  - A sequence of instructions let $x = o$;
  - A terminator τ
- A map from labels ℓ to subtrees r (the region's *children*), L

More formally, given a lexical SSA region r, we can define the following functions to compute r's entry block and children as follows

$$\begin{aligned} \mathsf{entry}(\mathsf{let}\ x = a; r) &= (\mathsf{let}\ x = a; \mathsf{entry}(r)) \\ \mathsf{entry}(\mathsf{let}\ (x, y) = a; r) &= (\mathsf{let}\ (x, y) = a; \mathsf{entry}(r)) \\ \mathsf{entry}(\tau\ \mathsf{where}\ (\ell_i(x_i) : \{t_i\},)_i) &= \tau \end{aligned} \tag{21}$$

$$\begin{aligned} \mathsf{children}(\mathsf{let}\ x = a; r) &= \mathsf{children}(r) \\ \mathsf{children}(\mathsf{let}\ (x, y) = a; r) &= \mathsf{children}(r) \\ \mathsf{children}(\tau\ \mathsf{where}\ (\ell_i(x_i) : \{t_i\},)_i) &= [\ell_i(x_i) : \{t_i\},]_i \end{aligned} \tag{22}$$

We can similarly define a function to construct a lexical SSA program from a basic block β and a set of children as follows:

$$\begin{aligned} \mathsf{bb}(\mathsf{let}\ x = a; \beta, L) &= \mathsf{let}\ x = a; \mathsf{bb}(\beta, L) \\ \mathsf{bb}(\mathsf{let}\ (x, y) = a; \beta, L) &= \mathsf{let}\ (x, y) = a; \mathsf{bb}(\beta, L) \\ \mathsf{bb}(\tau, L) &= \tau\ \mathsf{where}\ L \end{aligned} \tag{23}$$

It is easy to see that these functions are mutually inverse: for any lexical SSA region r, we have

$$r = \mathsf{bb}(\mathsf{entry}(r), \mathsf{children}(r)) \tag{24}$$

Some other useful facts about $\mathsf{bb}(\cdot, \cdot)$ include:

- It is a congruence: if $r \approx r'$ and each $t_i \approx t_i'$, $\mathsf{bb}(r, (\ell_i(x_i) : \{t_i\},)_i) \approx \mathsf{bb}(r', (\ell_i(x_i) : \{t_i'\},)_i)$
- It is invariant up to permutations σ: $\mathsf{bb}(r, (\ell_i(x_i) : \{t_i\},)_i) \approx \mathsf{bb}(r, (\ell_{\sigma_i}(x_{\sigma_i}) : \{t_{\sigma_i}\},)_i)$
- Similarly, *if both sides of the equation are well-typed*, i.e.,
  - All t_i do not use y or any of the variables defined in s
  - All branches to ℓ_i come from κ or ℓ_j (and not from either r or G)

$$\mathsf{bb}(r, (G, \kappa(y) : \{s\}, (\ell_i(x_i) : \{t_i\},))) \approx \mathsf{bb}(r, (G, \kappa(y) : \{s\ \mathsf{where}\ (\ell_i(x_i) : \{t_i\},)\})) \tag{25}$$

and hence

$$\mathsf{bb}(r, (G, \kappa(y) : \{s\}, (\ell_i(x_i) : \{t_i\},))) \approx \mathsf{bb}(r, (G, \kappa(y) : \{\mathsf{bb}(s, (\ell_i(x_i) : \{t_i\},))\})) \tag{26}$$

In particular, we may apply this rule twice to obtain

$$\mathsf{bb}(r, (G, \kappa(y) : \{\mathsf{bb}(s, G')\}, (\ell_i(x_i) : \{t_i\},))) \approx \mathsf{bb}(r, (G, \kappa(y) : \{\mathsf{bb}(s, G', (\ell_i(x_i) : \{t_i\},))\})) \tag{27}$$

We may hence define a function $\mathsf{cfg}(\cdot)$ from lexical SSA programs r to control-flow graphs G as follows:

$$\mathsf{cfg}(r) = \mathsf{entry}(r), (\ell_i(x_i) : \{\mathsf{cfg}(t_i)\},)_{(\ell_i(x_i):\{t_i\}) \in \mathsf{children}(r)} \tag{28}$$

where we recursively flatten

$$G, \ell(x) : \{\beta, G'\} := G, \ell(x) : \{\beta\}, G' \tag{29}$$

At the level of program text, all this function is doing is "removing where-blocks;" so we'll call the result *reased*. An SSA program is *well-formed* if:

- All variable uses are well-typed
- All variable uses respect dominance-based scoping

It is easy to see that any erased program is well-formed, since our typing rules guarantee every expression is well-typed, while our lexical scoping for values ensures that variables are only visible in the children of the block β in which they are defined, which the lexical scoping of labels guarantees are dominated by β.

On the other hand, we may give an algorithm to convert any well-formed SSA program G into a well-typed lexical SSA program $r = \mathsf{reg}(G)$ as follows:

(1) Compute the dominance tree of G, rooted at its entry block β.
(2) For each child $\ell_i(x_i) : \{\beta_i\}$ of β, let G_i denote the CFG composed of the descendants of β_i (with β_i as entry block), given in the order they appear in G. Recursively compute $r_i = \mathsf{reg}(G_i)$.
(3) Return the program $\mathsf{bb}(\beta, (\ell_i(x_i) : \{r_i\},)_i)$

We will write $G \simeq G'$ to mean "G is a permutation of G'" (that is, G and G' have the same entry block, but the other labels may be reordered). It is easy to see that this algorithm yields a lexical SSA program which erases to a permutation of G, i.e., $\mathsf{cfg}(\mathsf{reg}(G)) \simeq G$, as desired. To complete our argument, it hence suffices to show that, given lexical SSA regions $\Gamma \vdash r \rhd \mathsf{L}$, $\Gamma \vdash r' \rhd \mathsf{L}$, such that $\mathsf{cfg}(r) \simeq \mathsf{cfg}(r')$, we have that $\Gamma \vdash r \approx r' \rhd \mathsf{L}$. We break this down into two lemmas:

Lemma 4.6 (Permutation Invariance). *If $G \simeq G'$ and $\Gamma \vdash \mathsf{reg}(G) \rhd \mathsf{L}$, then $\Gamma \vdash \mathsf{reg}(G') \rhd \mathsf{L}$ and $\Gamma \vdash \mathsf{reg}(G) \approx \mathsf{reg}(G') \rhd \mathsf{L}$*

Proof. See Appendix C.2 □

Lemma 4.7 (CFG Conversion). *Given a lexical SSA region r (i.e., assuming $\mathsf{IsSSA}(r)$) s.t. $\Gamma \vdash r \rhd \mathsf{L}$, we have that $\Gamma \vdash r \approx \mathsf{reg}(\mathsf{cfg}(r)) \rhd \mathsf{L}$.*

Proof. See Appendix C.2 □

It follows that, given lexical SSA regions $\Gamma \vdash r \rhd \mathsf{L}$ and $\Gamma \vdash r' \rhd \mathsf{L}$, if $\mathsf{cfg}(r) \simeq \mathsf{cfg}(r')$, we have that $\Gamma \vdash r \approx r' \rhd \mathsf{L}$, as desired, since in particular

$$r \approx \mathsf{reg}(\mathsf{cfg}(r)) \approx \mathsf{reg}(\mathsf{cfg}(r')) \approx r' \tag{30}$$

5 Denotational Semantics

Now that we have given an equational theory for SSA, we would like to build a denotational semantics for SSA: a mapping from well-typed programs $\Gamma \vdash r \rhd \mathsf{L}$ to mathematical objects $[\![\Gamma \vdash r \rhd \mathsf{L}]\!]$. This naturally gives us another way to reason about whether two programs "r, r'" are "the same:" we can ask whether $[\![\Gamma \vdash r \rhd \mathsf{L}]\!] = [\![\Gamma \vdash r' \rhd \mathsf{L}]\!]$. Our goal is to build a denotational semantics which is both *sound* and *complete*; that is, such that

$$\Gamma \vdash r \approx r' \rhd \mathsf{L} \iff [\![\Gamma \vdash r \rhd \mathsf{L}]\!] = [\![\Gamma \vdash r' \rhd \mathsf{L}]\!]$$

More specifically, we'll begin by defining a notion of a *model* $\mathcal{M}$ of SSA, and then define the denotational semantics w.r.t. a model $[\![\Gamma \vdash r \rhd \mathsf{L}]\!]_{\mathcal{M}}$. We'll then prove that, for *any* model of SSA, if $\Gamma \vdash r \approx r' \rhd \mathsf{L}$, then

$$[\![\Gamma \vdash r \rhd \mathsf{L}]\!]_{\mathcal{M}} = [\![\Gamma \vdash r' \rhd \mathsf{L}]\!]_{\mathcal{M}}$$

In other words, our equational theory is *sound*, and so never allows us to prove an equation which does not in fact hold. We'll then give an model $\mathcal{M}$ such that

$$[\![\Gamma \vdash r \rhd \mathsf{L}]\!]_{\mathcal{M}} = [\![\Gamma \vdash r' \rhd \mathsf{L}]\!]_{\mathcal{M}} \implies \Gamma \vdash r \approx r' \rhd \mathsf{L}$$

showing that our equational theory is also *complete*: if an equation holds in an *arbitrary* model of SSA, it must in fact be derivable via our theory.

We'd further like our denotational semantics to be *compositional*: the denotation of a program should be a fixed function of the denotations of its parts. To do this, we will give SSA a *categorical* semantics, interpreting (label) contexts as objects and programs as morphisms between them in some target category C. Different program structures, such as branching and sequential composition, will correspond more-or-less directly to structures in our target category.

Before we begin, we fix some notational conventions:

- We denote the *composition* of two morphisms $f : A \to B$ and $g : B \to C$ as $f ; g : A \to C$
- We denote the coproduct of a morphism $f : B \to C$ with the identity id_A as $f + A : B + A \to C + A$ or $A + f : A + B \to A + C$.

5.1 Freyd-Elgot Categories

Moggi [51] showed that the Kleisli category of a strong monad over a CCC interprets effectful higher-order functional programs. There are two mismatches in using this as a semantics for SSA. On one hand, SSA has features not necessarily supported by Moggi models such as arbitrary, unstructured cyclic control-flow. On the other hand Moggi models support features SSA (as a first-order language) does not have, such as higher-order functions and first-class computation values.

Given that we want to model SSA with some category C, we hence have to think about what structure we need C to possess so that it has exactly our desired features. Obviously, we need a way to take the product of two objects, to be able to model contexts as well as pairs. The usual way to do this is via *monoidal categories*. However, monoidal categories typically have too many equations: computations operating on independent data must always commute (this is the "sliding rule"), which is obviously not true for effects such as printing, since

$$\mathsf{print}(x); \mathsf{print}(y) \neq \mathsf{print}(y); \mathsf{print}(x)$$

Instead, we will only require that our category be *premonoidal*, as introduced by Power and Robinson [55]: "monoidal, without sliding." Indeed, the Kleisli category of a strong monad on a CCC is not always monoidal, as demonstrated by the writer monad on Set (which exposes print), but *is* always premonoidal. We define a premonoidal category as follows:

Definition 5.1 (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. 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*

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

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*

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

We say a premonoidal category is *monoidal* if every morphism is central.

One important theorem about premonoidal categories is *coherence*:

Theorem 5.2. *The subcategory $\mathcal{A}$ generated by associators, unitors, and their tensor products is an equivalence relation, i.e., for all $A, B : |\mathcal{A}|$, if $f, g : A \to B$ can be constructed using only identity, composition, associators, unitors, and their tensor products, then:*

(a) $f = g$
(b) f, g are isomorphisms in $\mathcal{A}$

In particular, as a syntactic convenience, we will often simply write "α" for the (unique) morphism between objects A and B, when it exists, satisfying the requirements of Theorem 5.2. For example, given $f : A \to B \otimes ((C \otimes I) \otimes D)$ and $g : ((B \otimes (I \otimes C)) \otimes D) \to E$, we have

$$f; \alpha; g := f; B \otimes (\lambda_C \otimes D); \alpha^{-1}_{B,C,D}; (B \otimes \rho_C^{-1}) \otimes D; g$$

Just like for higher-order functional languages, we can interpret types A as objects $[\![A]\!] : |C|$. Similarly, we can interpret variable contexts Γ by taking products of objects, as follows:

$$\boxed{[\![\Gamma]\!] : |C|} \qquad [\![\cdot]\!] = I \qquad [\![\Gamma, x : A]\!] = [\![\Gamma]\!] \otimes [\![A]\!]$$

We would like to interpret an expression-in-context $\Gamma \vdash_\epsilon a : A$ as a morphism in C from $[\![\Gamma]\!]$ to $[\![A]\!]$. However, in standard SSA, it is possible for a variable to be unused, or used multiple times. Our premonoidal structure, however, does not give us any way to *project* out of a product type, making it impossible to interpret expressions-in-context like $x : A, y : B \vdash x : A$. In order to project out individual variables, we need Cartesian structure, but a premonoidal category can only interpret *linear* expressions, that is, those which use every variable exactly once. However, it is too much to require the whole premonoidal category to be Cartesian, because that would validate the sliding rule, which is what we set out to avoid.

In the case of the Kleisli category over a CCC, the Cartesian structure of the CCC becomes the premonoidal structure of the Kleisli category. This lets us use the Cartesian structure to project out the variables, while the product in the Kleisli category still does not satisfy sliding. By analogy with this case, we can suppose that there is a subcategory $C_\perp$ of the premonoidal category C, which has the property that the premonoidal structure in C behaves like a Cartesian product in $C_\perp$.

If we generalize this structure to an arbitrary premonoidal category, we get the notion of a *Freyd category*, as introduced in Levy et al. [47]:

Definition 5.3 (Freyd category). A *Freyd category* is a premonoidal category C equipped with a wide subcategory $C_\perp \subseteq C$ of *pure* morphisms such that

- $C_\perp$ contains all associators, unitors, and symmetries
- I is a terminal object in $C_\perp$. In particular, this implies the terminal morphisms $!_A : A \to I$ are pure.
- For each A, B, $A \otimes B$ is a cartesian product of A, B in $C_\perp$
- For pure morphisms f, g, $f \otimes g = \langle \pi_l; f, \pi_r; g \rangle$

Where C is clear from context, we will write $A \to_\perp B$ to denote a pure morphism $C_\perp(A, B)$.

Alternatively, it is equivalent to require

- $C_\perp$ contains all associators, unitors, and symmetries

- I is a terminal object in $C_\bot$
- For each A, there exists a pure morphism $\Delta_A : A \to A \otimes A$ forming a comonoid with the (pure) terminal morphism $!_A : A \to I$, i.e: $\Delta_A; !_A \otimes A; \rho = \Delta_A; A \otimes !_A; \lambda = \mathrm{id}_A$
- For every pure morphism $f : A \to_\bot B$, $f; \Delta_B = \Delta_A; f \otimes f$ and $f; !_B = !_A$.

In both cases, we have that $\pi_l = A \otimes !_B; \lambda$, $\pi_r = !_A \otimes B; \rho$, $\langle f, g\rangle = \Delta_A; f \otimes g$, and $\Delta_A = \langle \mathrm{id}_A, \mathrm{id}_A\rangle$

For convenience, given $f : A \to B$ in a Freyd category, we will define the notation

$$\mathrm{let}(f) := \Delta_A; A \otimes f : A \to A \otimes B \tag{31}$$

Note that this is pure if and only if f is. This has the following useful properties:

- $\mathrm{let}(f); \pi_r = f$ and, for f pure, $\mathrm{let}(f); \pi_l = \mathrm{id}$
- $\mathrm{let}(f; g) = \mathrm{let}(f); A \otimes g$ and $\mathrm{let}(\mathrm{id}_A) = \Delta_A$
- $\mathrm{let}(\mathrm{let}(f)) = \mathrm{let}(f); \Delta_A \otimes B; \alpha_{A,A,B}$, and, therefore, given $g : A \otimes B \to C$, we have $\mathrm{let}(\mathrm{let}(f); g) = \mathrm{let}(f); \mathrm{let}(g); \pi_l \otimes C$

We now have everything we need to model effectful first-order expressions. For reasoning about substitution, we will also demand that the denotation of "*pure*" expressions $\Gamma \vdash_\bot a : A$ lies in $C_\bot(\llbracket\Gamma\rrbracket, \llbracket A\rrbracket)$. In general, we will write:

- $C_\top$ to mean just C, allowing us to write C_ϵ for an *effect* $\epsilon \in \{\bot, \top\}$.
- Morphisms in C_ϵ as $A \to_\epsilon B$, where C is clear from context.

At this point, we still have no way to interpret control-flow, i.e. case-expressions. Furthermore, if we want to model regions as morphisms, we need some way of modelling label-contexts L. At first glance, it seems sufficient for branching control-flow to require the existence of coproducts, and indeed, assuming the existence of all coproducts and an initial object, we may model label-contexts as follows:

$$\boxed{\llbracket \mathsf{L}\rrbracket : |C|} \qquad \llbracket\cdot\rrbracket = \mathbf{0} \qquad \llbracket \mathsf{L}, \ell(A)\rrbracket = \llbracket \mathsf{L}\rrbracket + \llbracket A\rrbracket$$

Regions can now be interpreted as morphisms in C from $\llbracket\Gamma\rrbracket$ to $\llbracket\mathsf{L}\rrbracket$, as desired. Just like for products, we will write "α^+" for the (unique) morphism between objects A and B, when it exists, satisfying the requirements of Theorem 5.2 where coproducts are taken as the monoidal structure; we will sometimes also write α^+_B for clarity.

It turns out that our coproducts must be *distributive* to allow us to use variables in scope before a branch. In particular, we define a *distributive* premonoidal category as follows:

Definition 5.4 (Distributive category). A premonoidal category C with all coproducts is *distributive* if, for all A, B, C, the obvious morphism

$$\delta_{A,B,C} = [(A + \iota_l), (A + \iota_r)] : (A \otimes B) + (A \otimes C) \to A \otimes (B + C)$$

has an inverse δ^{-1}. We will say a Freyd category C is distributive if it has all coproducts and the subcategory of pure morphisms $C_\bot$ is distributive (which implies, in particular, that C is distributive when taken as a premonoidal category).

We note in particular that every cartesian closed category with coproducts is a distributive Freyd category, as is the Kleisli category of a monad over a CCC with coproducts. For any finite coproduct $\Sigma_i B_i$, we will introduce the notation $\delta_\Sigma : \Sigma_i(A \otimes B_i) \to A \otimes \Sigma_i B_i$ and $\delta_\Sigma^{-1} : A \otimes \Sigma_i B_i \to \Sigma_i(A \otimes B_i)$ to denote the obvious morphisms.

This gives us enough machinery to express any *acyclic* control-flow graph, however, we still have no way to model loops. What we would like is to equip our category with an *iteration operator* taking a morphism $f : A \to B + A$, representing a "loop" which, given input A, either produces a result B or continues to another iteration with a new A, to its *fixpoint* $f^\dagger : A \to B$. Naturally, we

require that this operator satisfies various properties, and in particular is *strong*, that is, compatible with the distributor. Formally, we define a *(strong) Conway iteration operator* as follows:

Definition 5.5 (Conway iteration). A category $\mathcal{C}$ with all coproducts is said to have a *iteration operator* if we can define an operator $(-)^\dagger$ taking every morphism $f : A \to B + A$ to a morphism $f^\dagger : A \to B$, the *fixpoint* of f, with the following property: given $f : A \to B + A$, we have $f^\dagger = f; [\mathsf{id}, f^\dagger]$. We say this operator is a *Conway iteration operator* if it additionally satisfies the following properties:

- *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 $(f^\dagger)^\dagger = (f; [\mathsf{id}, \iota_r])^\dagger : A \to B$

If $\mathcal{C}$ is distributive, we say this operator is *strong* if

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

Given a wide subcategory $\mathcal{K} \subseteq \mathcal{C}$, we say $\mathcal{C}$ is $\mathcal{K}$*-uniform* if, for all $h : A \to_{\mathcal{K}} B$, $f : B \to C + B$, and $g : A \to C + A$, we have that

$$h; f = g; C + h \implies h; f^\dagger = g^\dagger$$

We will call Freyd categories equipped with an appropriate Elgot structure *strong Elgot categories*. In particular, we define

Definition 5.6 ((Strong) Elgot structure). A Freyd category $\mathcal{C}$ with all coproducts is said to have an *Elgot structure* if it has a Conway iteration operator which is $\mathcal{C}_\perp$-uniform. If $\mathcal{C}$ is distributive, we say $\mathcal{C}$ is strong Elgot if its iteration operator is strong. In particular, we say a monad is Elgot if its Kliesli category has an Elgot structure. Similarly, we say a *strong* monad is strong Elgot if its Kliesli category has a strong Elgot structure.

It turns out that, to check something is a strong Elgot category, we do not need to explicitly verify dinaturality. In particularly, we may verify the following:

Proposition 5.7. *If $(-)^\dagger$ is an iteration operator which satisfies naturality and codiagonal and is $\mathcal{K}$-uniform for $\mathcal{K}$ co-Cartesian, then it also satisfies dinaturality.*

Proof. See Lemma 31 of Goncharov and Schröder [28] □

Since in a distributive Freyd category $\mathcal{C}_\perp$ must be distributive (and hence co-cartesian), dinaturality follows from the other axioms of an Elgot category. Given a distributive Freyd category equipped with a (strong) Elgot structure, we will often want to consider the fixpoint of a morphism $f : R \otimes A \to B + A$, where our "context" R does not change between iterations. To do this, we first need to build up a morphism

$$\mathsf{rcase}(f) := \mathsf{let}(f); \pi_l \otimes (B + A); \delta^{-1} : R \otimes A \to R \otimes B + R \otimes A \tag{32}$$

which computes f and then distributes a copy of the read-only state R to each branch of the result. We may then define the fixpoint

$$\mathsf{rfix}(f) := (\mathsf{rcase}(f))^\dagger; \pi_r : R \otimes A \to B \tag{33}$$

We consider some more properties of $\mathsf{rcase}(f)$ and $\mathsf{rfix}(f)$ in Appendix B.

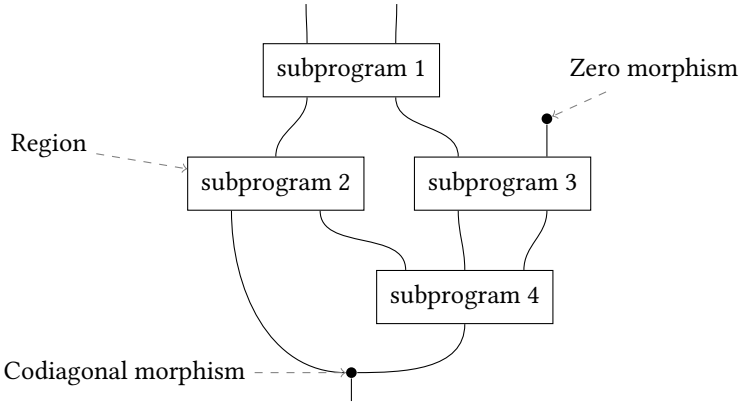


Fig. 26. A string diagram using the coproduct as symmetric monoidal structure, interpreted as a CFG

5.2 String Diagrams

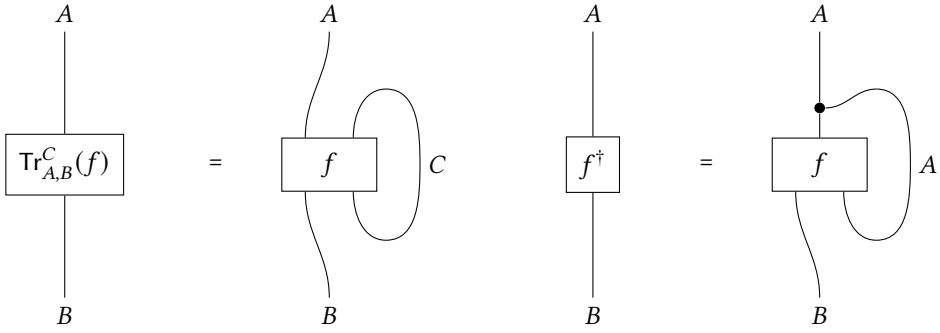
String diagrams provide a graphical calculus for reasoning about (symmetric) monoidal categories, which allows us to succinctly express complex morphisms and rewrites. Since both cartesian and co-cartesian categories are monoidal (with the product and coproduct as tensor, respectively), we can use string diagrams to reason about both. In the co-cartesian case, string diagrams behave much like control-flow diagrams, with boxes representing sub-programs, input wires entry points, and output wires exit points. In particular, a *region* is just a box with a single input wire. Continuing this analogy, we draw the codiagonal morphism $[\mathsf{id}_A, \mathsf{id}_A]$ as joining two wires, and the zero morphism as a wire coming from nowhere, as in Figure 26.

The power of string diagrams comes from the fact that many syntactically distinct ways to write equal values are obviously graphically equivalent by *isotopy*: essentially, moving boxes and wires around. String diagrams also give us an elegant way to represent and reason about Elgot structures. It turns out that Elgot structures induce a *trace* on the coproduct [30]: given $f : A + C \to B + C$, we can define

$$\mathsf{Tr}^C_{A,B}(f) = \iota_l; [f; B + \iota_r]^\dagger = \iota_l; f; [\mathsf{id}, (\iota_l; f)^\dagger] : A \to B \tag{34}$$

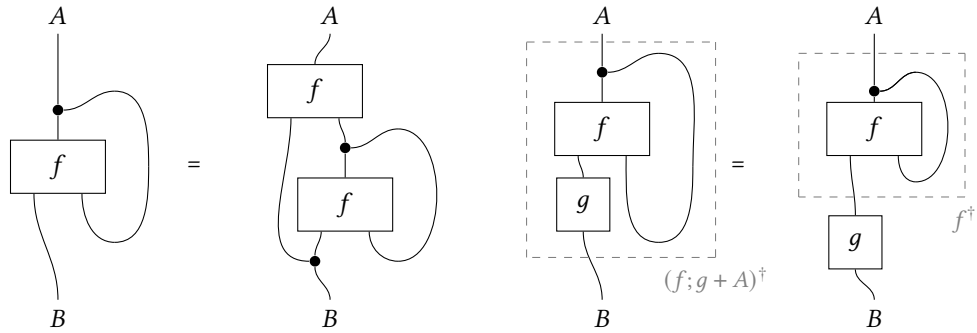
Since this satisfies the axioms of a trace over a symmetric monoidal category, we can draw it, and therefore the Elgot operator, as in Figure 27. Continuing with the control-flow diagram analogy, such traces can be interpreted as *loops*, with the Elgot axioms, now drawn as diagrams in Figure 28.

Unfortunately, unmodified string diagrams do not work for premonoidal categories, and hence for Freyd categories. The reason is because, since not all morphisms are central, premonoidal categories do not in general validate *sliding*. However, this is easy enough to fix: we can postulate a (dashed red) "state" wire which all impure morphisms require as an input and output, as in Figure 29. Since the state wire linearly threads through all impure boxes, it establishes a unique order in which they must be executed; this construction is shown to be sound in Román [57]. Pure morphisms do not have a state wire, so a diagram representing a pure morphism will simply have a dashed red "stripe" on the side. This gives us a convenient way to distinguish between string diagrams using the monoidal structure induced by the coproduct and those using the premonoidal structure induced by the tensor product in a category having both (such as a distributive premonoidal category): the latter will have a state wire, while the former will not.

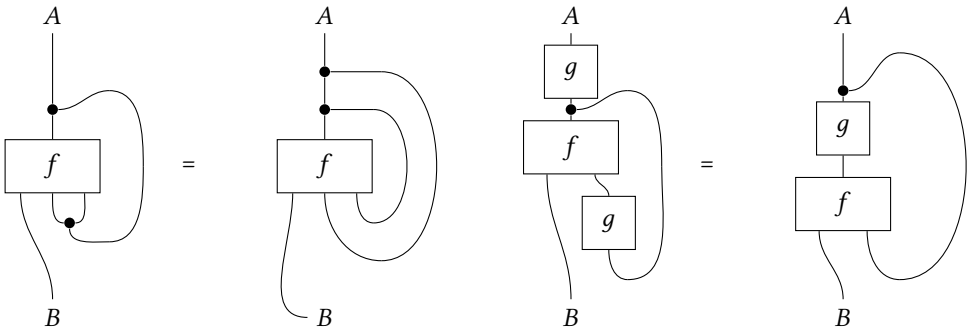


(a) The trace of $f : A + C \to B + C$ (b) The fixpoint of $f : A \to B + A$

Fig. 27. Representations of the coproduct trace and Elgot structure as string diagrams



(a) Fixpoint (b) Naturality



(c) Codiagonal (d) Dinaturality

Fig. 28. Representations of the Elgot axioms as string diagrams

5.3 Semantics

We now have all the ingredients we need to give a semantics to λ_{SSA} expressions and regions. In particular, an λ_{SSA} expression model of a signature $Sg = (\mathcal{T}, \mathcal{I})$ consists of:

- An distributive Freyd category C
- A map $[\![\cdot]\!]$ from base types X to objects $[\![X]\!] : |C|$

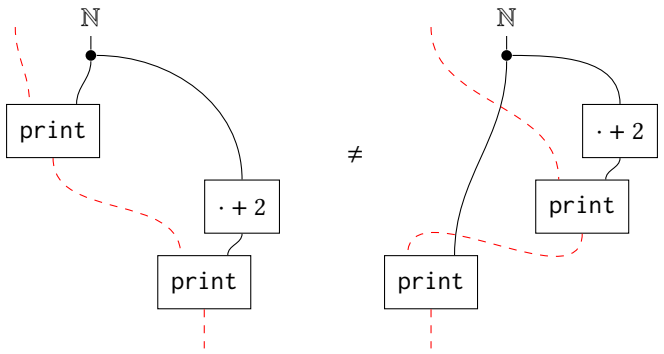


Fig. 29. A string diagram in a premonoidal category, demonstrating the necessity of using a state wire

- A map $[\![\cdot]\!]$ from primitive instructions $f \in \mathcal{I}_\epsilon(A, B)$ to morphisms $[\![f]\!] : [\![A]\!] \to_\epsilon [\![B]\!]$, where we model a type A as follows:
  - The unit type $\mathbf{1}$ is modelled as the monoidal unit I
  - The empty type $\mathbf{0}$ is modelled as the initial object $\mathbf{0}$
  - Products $A \otimes B$ are modelled as tensor products $[\![A]\!] \otimes [\![B]\!]$
  - Sum types $A + B$ are modelled as coproducts $[\![A]\!] + [\![B]\!]$

If an λ_{SSA} expression model additionally has an Elgot structure on C, we will refer to it simply as an λ_{SSA} model. We will model *contexts* and *label contexts* as tensor products and coproducts of the denotations of their parameters, respectively, as in Figure 30.

We can now interpret λ_{SSA}-expressions $\Gamma \vdash_\epsilon a : A$ over a given signature Sg as morphisms $[\![\Gamma]\!] \to_\epsilon [\![A]\!]$ using the rules in Figure 31. Up to this point, both our syntax and semantics are quite standard; in particular:

- Variables x are modelled as projections from the appropriate index in the context's denotation $\pi_{\Gamma,x} : [\![\Gamma]\!] \to_\perp [\![A]\!]$
- Applications of primitive instructions $f\ a$ are modelled as $[\![f]\!]$ precomposed with the denotation of a.
- Unary let-bindings are modelled by:
  - Duplicating the context using the diagonal morphism $\Delta_{[\![\Gamma]\!]}$
  - Passing the right copy of the context through $[\![\Gamma \vdash_\epsilon a : A]\!]$ to get an input of type $[\![\Gamma]\!] \otimes [\![A]\!] = [\![\Gamma, x : A]\!]$
  - Passing the result of this through $[\![\Gamma, x : A \vdash_\epsilon b : B]\!]$ to get the final result of type $[\![B]\!]$
- Pairs are modelled by passing the result of the diagonal morphism $\Delta_{[\![\Gamma]\!]}$ to $[\![\Gamma \vdash_\epsilon a : A]\!] \ltimes [\![\Gamma \vdash_\epsilon b : B]\!]$ i.e., first passing the left copy through $[\![\Gamma \vdash_\epsilon a : A]\!]$ and then the right copy through $[\![\Gamma \vdash_\epsilon b : B]\!]$. By the axioms of a Freyd category, for pure morphisms, this is the same as simply taking the product $\langle [\![\Gamma \vdash_\epsilon a : A]\!], [\![\Gamma \vdash_\epsilon b : B]\!] \rangle$.
- Binary let-bindings are modelled similarly to unary let-bindings, except that after passing the right copy of the context through $[\![\Gamma \vdash_\epsilon e : A \otimes B]\!]$, we re-associate $[\![\Gamma]\!] \otimes ([\![A]\!] \otimes [\![B]\!])$ to $([\![\Gamma]\!] \otimes [\![A]\!]) \otimes [\![B]\!] = [\![\Gamma, x : A, y : B]\!]$ before passing the entire result through $[\![\Gamma, x : A, y : B \vdash_\epsilon c : C]\!]$.
- The unit value $()$ is modelled as the terminal morphism $1_{[\![\Gamma]\!]}$, while abort a is modelled as the denotation of a postcomposed with the zero morphism $0_{[\![A]\!]}$. Injections, similarly, are simply modelled as the appropriate coproduct injections.

$$\boxed{[\![A]\!] : |C|}$$

$$[\![\mathbf{1}]\!] = I \qquad [\![A \otimes B]\!] = [\![A]\!] \otimes [\![B]\!] \qquad [\![\mathbf{0}]\!] = \mathbf{0} \qquad [\![A + B]\!] = [\![A]\!] + [\![B]\!]$$

$$\boxed{[\![\Gamma]\!] : |C|}$$

$$[\![\cdot]\!] = I \qquad [\![\Gamma, x : A]\!] = [\![\Gamma]\!] \otimes [\![A]\!]$$

$$\boxed{[\![\mathsf{L}]\!] : |C|}$$

$$[\![\cdot]\!] = 0 \qquad [\![\mathsf{L}, \ell(A)]\!] = [\![\mathsf{L}]\!] + [\![A]\!]$$

$$\boxed{[\![\Gamma \leq \Delta]\!] : [\![\Gamma]\!] \to_{\perp} [\![\Delta]\!]}$$

$$[\![\cdot \leq \cdot]\!] = \mathsf{id}_I \qquad [\![\Gamma, x : A \leq \Delta]\!] = \pi_l ; [\![\Gamma \leq \Delta]\!] \qquad [\![\Gamma, x : A \leq \Delta, x : A]\!] = [\![\Gamma \leq \Delta]\!] \otimes [\![A]\!]$$

$$\boxed{[\![\mathsf{L} \leq \mathsf{K}]\!] : [\![\mathsf{L}]\!] \to_{\perp} [\![\mathsf{K}]\!]}$$

$$[\![\cdot \leq \cdot]\!] = \mathsf{id}_{\mathbf{0}} \qquad [\![\mathsf{L} \leq \mathsf{K}, \ell(A)]\!] = [\![\mathsf{L} \leq \mathsf{K}]\!] ; \iota_\ell \qquad [\![\mathsf{L}, \ell(A) \leq \mathsf{K}, \ell(A)]\!] = [\![\mathsf{L} \leq \mathsf{K}]\!] + [\![A]\!]$$

Fig. 30. Denotational semantics for λ_{SSA} types, contexts, and weakenings

- A case-expression is modelled by
  - Duplicating the context using the diagonal morphism $\Delta_{[\![\Gamma]\!]}$
  - Using the right copy of the context to compute the discriminant using $[\![\Gamma \vdash_\epsilon e : A + B]\!]$
  - Applying the inverse distributor δ^{-1} to obtain a coproduct $[\![\Gamma]\!] \otimes [\![A]\!] + [\![\Gamma]\!] \otimes [\![B]\!]$
  - Computing $[\![\Gamma, x : A \vdash_\epsilon a : C]\!]$ on the right branch and $[\![\Gamma, y : B \vdash_\epsilon b : C]\!]$ on the left branch

Similarly, if we in fact have an λ_{SSA} model, we can interpret λ_{SSA} regions $\Gamma \vdash r \rhd \mathsf{L}$ as morphisms $[\![\Gamma]\!] \to [\![\mathsf{L}]\!]$; note that we don't assume anything about the effect of these morphisms. As L is a coproduct, we can view the result object of a region r as encoding both *data* and *control-flow* information. In particular, we interpret a branch $\mathsf{br}\ \ell\ a$ as simply the injection of the (pure) expression $\Gamma \vdash_\perp a : A$, our *data*, into the element of the coproduct corresponding to ℓ, which encodes the point in control-flow the rest of the program should jump to next. This is in contrast to *expressions*, which purely encode data, with no particular instructions on how to use it afterwards. Our interpretation of let-statements and case-statements, given in Figure 32, is exactly the same as that of the corresponding expressions.

Finally, we come to the interpretation of where-statements, which is where the Elgot structure comes in. The semantics of a where-block $\Gamma \vdash r$ where $(\ell_i(x_i) : \{t_i\},)_i \rhd \mathsf{L}$ can be broken down into two major components:

- The *terminator* $\mathsf{esem}_{\Gamma,\mathsf{L}}(r) : [\![\Gamma]\!] \to [\![\mathsf{L}]\!] + \Sigma_i [\![A_i]\!]$, which, given as input the context $[\![\Gamma]\!]$, executes r and then re-associates the output. The output type $[\![\mathsf{L}]\!] + \Sigma_i [\![A_i]\!]$ expresses that r may either:

$$\boxed{[\![\Gamma \vdash_\epsilon a : A]\!] : [\![\Gamma]\!] \to_\epsilon [\![A]\!]}$$

$$\begin{aligned}
[\![\Gamma \vdash_\epsilon x : A]\!] &= \pi_{\Gamma,x} \\
[\![\Gamma \vdash_\epsilon f\ a : B]\!] &= [\![f]\!] \circ [\![\Gamma \vdash_\epsilon a : A]\!] \\
[\![\Gamma \vdash_\epsilon \mathsf{let}\ x = a;\ b : B]\!] &= \mathsf{let}([\![\Gamma \vdash_\epsilon a : A]\!]); [\![\Gamma, x : A \vdash_\epsilon b : B]\!] \\
[\![\Gamma \vdash_\epsilon (a, b) : A \otimes B]\!] &= \Delta_{[\![\Gamma]\!]}; [\![\Gamma \vdash_\epsilon a : A]\!] \ltimes [\![\Gamma \vdash_\epsilon b : B]\!] \\
[\![\Gamma \vdash_\epsilon \mathsf{let}\ (x, y) = e;\ c : C]\!] &= \mathsf{let}([\![\Gamma \vdash_\epsilon e : A \otimes B]\!]); \alpha; [\![\Gamma, x : A, y : B \vdash_\epsilon c : C]\!] \\
[\![\Gamma \vdash_\epsilon () : \mathbf{1}]\!] &= 1_{[\![\Gamma]\!]} \\
[\![\Gamma \vdash_\epsilon \iota_l\ a : A + B]\!] &= [\![\Gamma \vdash_\epsilon a : A]\!]; \iota_l \\
[\![\Gamma \vdash_\epsilon \iota_r\ b : A + B]\!] &= [\![\Gamma \vdash_\epsilon b : B]\!]; \iota_r \\
[\![\Gamma \vdash_\epsilon \mathsf{case}\ e\ \{\iota_l\ x : a, \iota_r\ y : b\} : C]\!] &= \mathsf{let}([\![\Gamma \vdash_\epsilon e : A + B]\!]); \delta^{-1}_{[\![\Gamma]\!]}; \\
&\qquad [[\![\Gamma, x : A \vdash_\epsilon a : C]\!], [\![\Gamma, y : B \vdash_\epsilon b : C]\!]] \\
[\![\Gamma \vdash_\epsilon \mathsf{abort}\ a : A]\!] &= [\![\Gamma \vdash_\epsilon a : \mathbf{0}]\!]; 0_{[\![A]\!]}
\end{aligned}$$

where $\boxed{\pi_{\Gamma,x} : [\![\Gamma]\!] \to_\perp [\![A]\!]}$ $\quad \pi_{(\Gamma,x:A),x} = \pi_r \quad \pi_{(\Gamma,y:B),x} = \pi_l; \pi_{\Gamma,x}$

Fig. 31. Denotational semantics for λ_{SSA} expressions

  - Via the left injection, return immediately, jumping to an enclosing label in L
  - Via the right injection, jump to the i^{th} basic block t_i in the where-statement by returning a value in $[\![A_i]\!]$
- The "*loop*" $\mathsf{lsem}_{\Gamma,\mathsf{L}}((\ell_i(x_i) : \{t_i\},)_i) : [\![\Gamma]\!] \otimes \Sigma_i[\![A_i]\!] \to [\![\mathsf{L}]\!] + \Sigma_i[\![A_i]\!]$ which, given as input the context $[\![\Gamma]\!]$ and an input $[\![A_i]\!]$ for the i^{th} basic block t_i, executes t_i and then re-associates the output. The output type $[\![\mathsf{L}]\!] + \Sigma_i[\![A_i]\!]$ again expresses that control-flow may either exit the where-block (via $[\![\mathsf{L}]\!]$) or jump to some other basic block (via $\Sigma_i[\![A_i]\!]$).

We glue these together in the obvious manner to get the semantics for a where-block:

- Compute $\mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(r))$, which, given as input the context $[\![\Gamma]\!]$, copies the context, executes $\mathsf{esem}_{r,,}()$ and then returns the copied context and the output as $[\![\Gamma]\!] \otimes ([\![\mathsf{L}]\!] + \Sigma_i[\![A_i]\!])$
- Distribute the context into each branch of the coproduct, yielding $[\![\Gamma]\!] \otimes [\![\mathsf{L}]\!] + [\![\Gamma]\!] \otimes \Sigma_i[\![A_i]\!]$
- If we are in the left branch, project out the result to yield $[\![\mathsf{L}]\!]$, and return immediately
- Otherwise, compute $\mathsf{lsem}_{\Gamma,\mathsf{L}}((\ell_i(x_i) : \{t_i\},)_i)$ in a loop, passing in a fresh copy of the context each time. This is implemented with rfix.

This simplifies significantly in the case of a where-block defining just a single label, yielding

$$[\![\Gamma \vdash r \text{ where } \ell(x) : \{s\} \rhd \mathsf{L}]\!] = \mathsf{let}([\![\Gamma \vdash r \rhd \mathsf{L}, \ell(A)]\!]); \delta^{-1}; [\pi_r, \mathsf{rfix}([\![\Gamma, x : A \vdash s \rhd \mathsf{L}, \ell(A)]\!])] \tag{35}$$

In particular, it is a consequence of label weakening (Lemma 5.8) that, in the case where s does not call ℓ, this simplifies further to

$$[\![\Gamma \vdash r \text{ where } \ell(x) : \{s\} \rhd \mathsf{L}]\!] = \mathsf{let}([\![\Gamma \vdash r \rhd \mathsf{L}, \ell(A)]\!]); \delta^{-1}; [\pi_r, [\![\Gamma, x : A \vdash s \rhd \mathsf{L}]\!]] \tag{36}$$

5.4 Metatheory

We can now begin to state the metatheoretic properties of our denotational semantics. Before we do so, we establish the convention that whenever we have an equation involving the interpretation

$$\boxed{[\![\Gamma \vdash r \rhd \mathsf{L}]\!] : [\![\Gamma]\!] \to [\![\mathsf{L}]\!]}$$

$$\begin{aligned}
[\![\Gamma \vdash \mathsf{br}\ \ell\ a \rhd \mathsf{L}]\!] &= [\![\Gamma \vdash_\perp a : A]\!]; \iota_{\mathsf{L},\ell} \\
[\![\Gamma \vdash \mathsf{let}\ x = a; r \rhd \mathsf{L}]\!] &= \mathsf{let}([\![\Gamma \vdash_\epsilon a : A]\!]); [\![\Gamma, x : A \vdash r \rhd \mathsf{L}]\!] \\
[\![\Gamma \vdash \mathsf{let}\ (x,y) = e; r \rhd \mathsf{L}]\!] &= \mathsf{let}([\![\Gamma \vdash_\epsilon e : A \otimes B]\!]); \alpha; [\![\Gamma, x : A, y : B \vdash r \rhd \mathsf{L}]\!] \\
[\![\Gamma \vdash \mathsf{case}\ e\ \{\iota_l\ x : r, \iota_r\ y : s\} \rhd \mathsf{L}]\!] &= \mathsf{let}([\![\Gamma \vdash_\epsilon e : A + B]\!]); \delta^{-1}; \\
&\qquad [[\![\Gamma, x : A \vdash r \rhd \mathsf{L}]\!], [\![\Gamma, y : B \vdash s \rhd \mathsf{L}]\!]] \\
[\![\Gamma \vdash r\ \mathsf{where}\ (\ell_i(x_i) : \{t_i\},)_i \rhd \mathsf{L}]\!] &= \mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(r)); \delta^{-1}; [\pi_r, \mathsf{rfix}(\mathsf{lsem}_{\Gamma,\mathsf{L}}((\ell_i(x_i) : \{t_i\},)_i))]
\end{aligned}$$

where $\boxed{\iota_{\mathsf{L},\ell} : [\![A]\!] \to_\perp [\![\mathsf{L}]\!]}$ $\qquad \iota_{(\mathsf{L},\ell(A)),\ell} = \iota_r \qquad \iota_{(\mathsf{L},\kappa(B)),x} = \iota_l; \iota_{\mathsf{L},\ell}$

$$\boxed{\mathsf{esem}_{\Gamma,\mathsf{L}}(r) : [\![\Gamma]\!] \to [\![L]\!] + \Sigma_i [\![A_i]\!]}$$

$$\mathsf{esem}_{\Gamma,\mathsf{L}}(r) = [\![\Gamma \vdash r \rhd \mathsf{L}, (\ell_i(A_i),)_i]\!]; \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_i[\![A_i]\!]}$$

$$\boxed{\mathsf{lsem}_{\Gamma,\mathsf{L}}(\ell_i(x_i) : \{t_i\},)_i) : [\![\Gamma]\!] \otimes \Sigma_i [\![A_i]\!] \to [\![\mathsf{L}]\!] + \Sigma_i [\![A_i]\!]}$$

$$\mathsf{lsem}_{\Gamma,\mathsf{L}}(\ell_i(x_i) : \{t_i\},)_i) = \delta_\Sigma^{-1}; [[\![\Gamma, x_i : A_i \vdash t_i \rhd \mathsf{L}, (\ell_j(A_j),)_j]\!],]_i; \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_i[\![A_i]\!]}$$

Fig. 32. Denotational semantics for λ_{SSA} regions

of a derivation (e.g., $[\![\mathcal{D}]\!] = [\![\mathcal{D}']\!]$), we assume that all the derivations (e.g., $\mathcal{D}$ and $\mathcal{D}'$) exist and are well-formed.

We begin with weakening: as shown in Figure 30, weakenings are modelled, essentially, as projections from a larger product $[\![\Gamma]\!]$ to a smaller product $[\![\Delta]\!]$, while label-weakenings are modelled as injections from a smaller coproduct $[\![\mathsf{L}]\!]$ to a larger coproduct $[\![\mathsf{K}]\!]$; in particular, in both cases, the morphisms are pure. A simple induction can then be used to derive the following weakening lemmas:

Lemma 5.8 ((Label) Weakening). *Given $\Gamma \le \Gamma'$ and $\mathsf{L}' \le \mathsf{L}$, $\mathsf{K}' \le \mathsf{K}$, we have*

(a) For all $\Gamma \le \Delta$, $[\![\Gamma \le \Delta]\!] = [\![\Gamma \le \Gamma']\!]; [\![\Gamma' \le \Delta]\!]$
(b) For all $\mathsf{L} \le \mathsf{K}$, $[\![\mathsf{L}' \le \mathsf{K}]\!] = [\![\mathsf{L}' \le \mathsf{L}]\!]; [\![\mathsf{L} \le \mathsf{K}]\!]$
(c) $[\![\Gamma \vdash_\epsilon a : A]\!] = [\![\Gamma \le \Gamma']\!]; [\![\Gamma' \vdash_\epsilon a : A]\!]$
(d) $[\![\Gamma \vdash r \rhd \mathsf{L}]\!] = [\![\Gamma \le \Gamma']\!]; [\![\Gamma' \vdash r \rhd \mathsf{L}']\!]; [\![\mathsf{L}' \le \mathsf{L}]\!]$
(e) For all $\gamma : \Gamma \mapsto \Delta$, $[\![\gamma : \Gamma \mapsto \Delta]\!] = [\![\Gamma \le \Gamma']\!]; [\![\gamma : \Gamma' \mapsto \Delta]\!]$
(f) For all $\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}$, $[\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] = [\![\Gamma \le \Gamma']\!] \otimes [\![\mathsf{L}]\!]; [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}']\!]; [\![\mathsf{K}' \le \mathsf{K}]\!]$

Proof. See Appendix C.4. □

We can now give a denotational semantics to substitutions in which a substitution $\gamma : \Gamma \mapsto \Delta$ is interpreted as a pure morphism from $[\![\Gamma]\!]$ to $[\![\Delta]\!]$, as in Figure 33. We can now state the soundness of variable substitution as follows:

Theorem 5.9 (Soundness (Substitution)). *Given $[\![\gamma : \Gamma \mapsto \Delta]\!] : [\![\Gamma]\!] \to [\![\Delta]\!]$ pure, we have that*

(a) $[\![\Gamma \vdash_\epsilon [\gamma]a : A]\!] = [\![\gamma : \Gamma \mapsto \Delta]\!]; [\![\Delta \vdash_\epsilon a : A]\!]$
(b) $[\![\Gamma \vdash [\gamma]r \rhd \mathsf{L}]\!] = [\![\gamma : \Gamma \mapsto \Delta]\!]; [\![\Delta \vdash r \rhd \mathsf{L}]\!]$
(c) $[\![[\gamma]\rho : \Delta \mapsto \Xi]\!] = [\![\gamma : \Gamma \mapsto \Delta]\!]; [\![\rho : \Delta \mapsto \Xi]\!]$
(d) $[\![\Gamma \vdash [\gamma]\sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] = [\![\gamma : \Gamma \mapsto \Delta]\!]; [\![\Delta \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!]$

$$\boxed{[\![\gamma : \Gamma \mapsto \Delta]\!] : [\![\Gamma]\!] \to_\bot [\![\Delta]\!]}$$

$$[\![\cdot : \Gamma \mapsto \cdot]\!] = !_{[\![\Gamma]\!]} \qquad [\![\gamma, x \mapsto e : \Gamma \mapsto \Delta, x : A]\!] = \Delta_{[\![\Gamma]\!]}; [\![\gamma : \Gamma \mapsto \Delta]\!] \ltimes [\![\Gamma \vdash_\bot e : A]\!]$$

$$\boxed{[\![\Gamma \vdash \kappa : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] : [\![\Gamma]\!] \otimes [\![\mathsf{L}]\!] \to [\![\mathsf{K}]\!]}$$

$$[\![\Gamma \vdash \cdot : \cdot \rightsquigarrow \mathsf{K}]\!] = !_{[\![\Gamma]\!]} \otimes \mathbf{0}; \lambda; 0_{\mathsf{K}}$$

$$[\![\kappa, \ell(x) \mapsto r \vdash \Gamma : \mathsf{L}, \ell(A) \rightsquigarrow \mathsf{K}]\!] = \delta; [[\![\kappa \vdash \Gamma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!], [\![\Gamma, x : A \vdash r \rhd \mathsf{K}]\!]]$$

Fig. 33. Denotational semantics for λ_{SSA} (label) substitutions

Proof. See Appendix C.4. □

In particular, this implies that the semantics of substitution composition $[\gamma]\rho$ is just composition of the denotations of γ, ρ. We can derive the following important corollary:

Corollary 5.10 (Soundness (Single Substitution)). *Given $\Gamma \vdash_\bot a : A$, we have*

(1) Given $\Gamma, x : A \vdash_\bot b : B$,

$$[\![\Gamma \vdash_\epsilon [a/x]b : B]\!] = \mathsf{let}([\![\Gamma \vdash_\bot a : A]\!]); [\![\Gamma, x : A \vdash_\epsilon b : B]\!] = [\![\Gamma \vdash_\epsilon \mathsf{let}\ x = a;\ b : B]\!] \tag{37}$$

(2) Given $\Gamma, x : A \vdash r \rhd \mathsf{L}$, we have

$$[\![\Gamma \vdash [a/x]r \rhd \mathsf{L}]\!] = \mathsf{let}([\![\Gamma \vdash_\bot a : A]\!]); [\![\Gamma, x : A \vdash r \rhd \mathsf{L}]\!] = [\![\Gamma \vdash \mathsf{let}\ x = a; r \rhd \mathsf{L}]\!] \tag{38}$$

Proof. Follows immediately from the fact that

$$[\![x \mapsto a^1 : \Gamma \mapsto \Gamma, x : A]\!] = \Delta_{[\![\Gamma]\!]}; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\bot a : A]\!] = \mathsf{let}([\![\Gamma \vdash_\bot a : A]\!]) \tag{39}$$

□

We can now move on to stating the metatheoretic properties of label-substitutions in much the same manner. In particular, in Figure 33, we interpret label substitutions $\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}$ as morphisms taking a copy of the context $[\![\Gamma]\!]$ and an element of the coproduct $[\![\mathsf{L}]\!]$ to an element of the coproduct $[\![\mathsf{K}]\!]$, with an arbitrary effect. Label substitution is then sound in general, as stated in the following theorem:

Theorem 5.11 (Soundness (Label Substitution)). *Given $\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}$, we have*

(a) $[\![\Gamma \vdash [\sigma]r \rhd \mathsf{K}]\!] = \mathsf{let}([\![\Gamma \vdash r \rhd \mathsf{L}]\!]); [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!]$

(b) $[\![\Gamma \vdash [\sigma]\sigma' : \mathsf{M} \rightsquigarrow \mathsf{K}]\!] = \Delta_{[\![\Gamma]\!]} \otimes [\![\mathsf{L}]\!]; \alpha; [\![\Gamma]\!] \otimes [\![\Gamma \vdash \sigma' : \mathsf{M} \rightsquigarrow \mathsf{L}]\!]; [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!]$

Proof. See Appendix C.5 □

5.5 Equational Theory

Using the metatheory in the previous section, our goal is now to prove the equational theory given in Section 4 sound with respect to any valid λ_{SSA} model. Stated more precisely, we have the following:

Theorem 5.12 (Soundness (Equational Theory)). *We have that*

(a) $\Gamma \vdash_\epsilon a \approx a' : A \implies [\![\Gamma \vdash_\epsilon a : A]\!] = [\![\Gamma \vdash_\epsilon a' : A]\!]$

(b) $\Gamma \vdash r \approx r' \rhd \mathsf{L} \implies [\![\Gamma \vdash r \rhd \mathsf{L}]\!] = [\![\Gamma \vdash r' \rhd \mathsf{L}]\!]$

(c) $\gamma \approx \gamma' : \Gamma \mapsto \Delta \implies [\![\gamma : \Gamma \mapsto \Delta]\!] = [\![\gamma' : \Gamma \mapsto \Delta]\!]$

(d) $\sigma \approx \sigma' \vdash \Gamma : \mathsf{L} \rightsquigarrow \mathsf{K} \implies [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] = [\![\Gamma \vdash \sigma' : \mathsf{L} \rightsquigarrow \mathsf{K}]\!]$

Proof. See Appendix C.6 □

Now that we've proved the *soundness* of our equational theory, what remains is to prove that it is *complete*, i.e., that every equation which holds in all λ_{SSA} models can be derived from it, or, stated more categorically, that our syntax quotiented by the equational theory forms an initial λ_{SSA} model. Our strategy for doing this is as follows:

(1) We begin by constructing a category of expressions $\mathsf{Th}^{\otimes}(\Gamma)$ and a category of regions $\mathsf{Th}(\Gamma, \mathsf{L})$ quotiented by our equational theory, and constructing a functor from the former to the latter.
(2) We then show that the category of expressions and the category of regions have the structure of an λ_{SSA} expression model and λ_{SSA} model, respectively, and hence that expressions may be interpreted in the former and both expressions and regions in the latter.
To do so, we will first need some notation to talk about the behaviour of the equivalence classes of quotients. Suppose S and T are sets of terms. Then we will write $\Gamma \vdash_\epsilon S : A$ to mean that for every $a \in S$, we have $\Gamma \vdash_\epsilon a : A$, and similarly we will write $\Gamma \vdash_\epsilon S \approx T : A$ when for all $a \in S$ and $b \in T$, we have $\Gamma \vdash_\epsilon a \approx b : A$. We generalize in the obvious fashion to regions as well as the case when only one side of an equivalence is a set of terms.
Then, it will turn out that, for a distinguished variable $\square$ and label $\blacksquare$,
$$\begin{aligned} \Gamma, \square : \langle \Delta \rangle &\vdash_\epsilon [\![\Delta \vdash_\epsilon a : A]\!]_{\mathsf{Th}^{\otimes}(\Gamma)} : A \\ \Gamma, \square : \langle \Delta \rangle &\vdash [\![\Delta \vdash_\epsilon a : A]\!]_{\mathsf{Th}(\Gamma,\mathsf{L})} \rhd \mathsf{L}, \blacksquare(A) \\ \Gamma, \square : \langle \Delta \rangle &\vdash [\![\Delta \vdash r \rhd \mathsf{K}]\!]_{\mathsf{Th}(\Gamma,\mathsf{L})} \rhd \mathsf{L}, \blacksquare(\langle \mathsf{K} \rangle) \end{aligned} \tag{40}$$
where packing of contexts into types $\langle \Delta \rangle$ is defined as in Appendix A.
(3) Finally, we refine this result to show that
$$\begin{aligned} \Gamma, \square : [\Delta] &\vdash_\epsilon [\![\Delta \vdash_\epsilon a : A]\!]_{\mathsf{Th}^{\otimes}(\Gamma)} \approx \langle a \rangle : A \\ \Gamma, \square : [\Delta] &\vdash [\![\Delta \vdash_\epsilon a : A]\!]_{\mathsf{Th}(\Gamma,\mathsf{L})} \approx \mathsf{ret}\ \langle a \rangle \rhd \mathsf{L}, \blacksquare(A) \\ \Gamma, \square : [\Delta] &\vdash [\![\Delta \vdash r \rhd \mathsf{K}]\!]_{\mathsf{Th}(\Gamma,\mathsf{L})} \approx \langle r \rangle \rhd \mathsf{L}, \blacksquare(\langle \mathsf{K} \rangle) \end{aligned} \tag{41}$$
where $\mathsf{ret}\ a := \mathsf{br}\ \blacksquare\ a$. Since the packing operator $\langle \cdot \rangle$ on terms and regions from Appendix A is injective for pure contexts $\mathsf{eff}(\Gamma) = \bot$, and hence in particular for $\Gamma = \cdot, \mathsf{L} = \cdot$, it follows that in this case the category of expressions and the category of regions are the initial distributive λ_{SSA} expression model and λ_{SSA} model respectively.

5.6 Expressions

We'll begin by going over the entire proof of completeness for expressions, which is the simpler case. In particular, we may define the category $\mathsf{Th}^{\otimes}_{\epsilon}(\Gamma)$ of expressions with effect ϵ as follows:

- Objects $|\mathsf{Th}^{\otimes}(\Gamma)|$ are types A, B, C
- Morphisms $\mathsf{Th}^{\otimes}(\Gamma)_\epsilon(A, B) = \{e \mid \Gamma, \square : A \vdash_\epsilon e : B\}$ quotiented by $\Gamma, \square : A \vdash_\epsilon e \approx e' : B$
- Identity $(\Gamma, \square : A \vdash_\bot \square : A) \in \mathsf{Th}^{\otimes}(\Gamma)_\epsilon(A, A)$
- Composition $e; e' = (\mathsf{let}\ \square = e; e')$, which satisfies
$$\Gamma, \square : A \vdash_\epsilon e : B, \quad \Gamma, \square : B \vdash_\epsilon e' : C \qquad \Longrightarrow \qquad \Gamma, \square : A \vdash_\epsilon e; e' : C$$
We may verify that this satisfies the axioms of a category w.r.t. our equational theory

In general, the category of expressions $\mathsf{Th}^{\otimes}(\Gamma)$ is simply then given by $\mathsf{Th}^{\otimes}_{\top}(\Gamma)$, which can be viewed as the union of all $\mathsf{Th}^{\otimes}_{\epsilon}(\Gamma)$.

We now need to equip $\mathsf{Th}^{\otimes}_{\epsilon}(\Gamma)$ with the structure of a premonoidal category. Obviously, we wish to define the tensor product of types A and B to be simply $A \otimes B$; we can then begin by defining

projections

$$\Gamma, \square : A \otimes B \vdash_\epsilon \pi_l := \mathsf{let}\ (x, y) = \square; x : A \qquad \Gamma, \square : A \otimes B \vdash_\epsilon \mathsf{let}\ (x, y) = \square; y : B \tag{42}$$

By simply using the pair constructor as a cartesian product $\langle a, b\rangle = (a, b)$, this can be shown to endow $\mathsf{Th}^{\otimes}_{\perp}(\Gamma)$ with the structure of a cartesian category, allowing us to define the associators, symmetries, and unitors in the natural manner. If we then define tensor functors

$$- \otimes X : e \mapsto \mathsf{let}\ (\square, x) = \square; (e; (\square, x)) \qquad X \otimes - : e \mapsto \mathsf{let}\ (x, \square) = \square; (e; (x, \square)) \tag{43}$$

we find that $\mathsf{Th}^{\otimes}_{\epsilon}(\Gamma)$ and hence in particular $\mathsf{Th}^{\otimes}(\Gamma)$ is endowed with the structure of a Freyd category with pure subcategory $\mathsf{Th}^{\otimes}_{\perp}(\Gamma)$.

Similarly, we wish to show that $A + B$ is the coproduct of A and B in $\mathsf{Th}^{\otimes}_{\epsilon}(\Gamma)$. Since we already have obvious injection morphisms

$$\Gamma, \square : A \vdash_\epsilon \iota_l := \iota_l\ \square : A + B \qquad \Gamma, \square : B \vdash_\epsilon \iota_r := \iota_r\ \square : A + B \tag{44}$$

we can define the coproduct of morphisms $\Gamma, \square : A \vdash_\epsilon a : C$ and $\Gamma, \square : B \vdash_\epsilon b : C$ to be simply given by

$$\Gamma, \square : A + B \vdash_\epsilon [a, b] := \mathsf{case}\ \square\ \{\iota_l\ \square : a, \iota_r\ \square : b\} : C \tag{45}$$

It is straightforward to verify that this indeed induces a coproduct on $\mathsf{Th}^{\otimes}_{\epsilon}(\Gamma)$ and hence on $\mathsf{Th}^{\otimes}(\Gamma)$ All that remains is to show that $\mathsf{Th}^{\otimes}_{\epsilon}(\Gamma)$ is in fact a *distributive* Freyd category. To do so, we may define an inverse distributor morphism

$$\Gamma, \square : A \otimes (B + C) \vdash_\epsilon \delta^{-1} := \mathsf{let}\ (x, y) = \square; \mathsf{case}\ y\ \{\iota_l\ z : \iota_l(x, z), \iota_r\ z : \iota_r(x, z)\} : A \otimes B + A \otimes C \tag{46}$$

which can easily be shown to be an inverse to the obvious distributor morphism. We may now note that

$$[\![\cdot]\!]_{\mathsf{Th}^{\otimes}(\Gamma)} = \mathbf{1}, \quad [\![\Delta, x : A]\!]_{\mathsf{Th}^{\otimes}(\Gamma)} = [\![\Delta]\!]_{\mathsf{Th}^{\otimes}(\Gamma)} \otimes A \qquad \Longrightarrow \qquad [\![\Delta]\!]_{\mathsf{Th}^{\otimes}(\Gamma)} = \langle \Gamma \rangle \tag{47}$$

Therefore, it follows that, as expected, that $\Gamma, \square : \langle \Delta \rangle \vdash_\epsilon [\![\Delta \vdash_\epsilon a : A]\!]_{\mathsf{Th}^{\otimes}(\Gamma)} : A$ and it remains to show that we in fact have

$$\Gamma, \square : \langle \Delta \rangle \vdash_\epsilon [\![\Delta \vdash_\epsilon a : A]\!]_{\mathsf{Th}^{\otimes}(\Gamma)} \approx [a] : A$$

which can be done by a relatively straightforward induction, implying, since $[\cdot]$ is injective w.r.t. our equational theory for pure contexts, the following theorem:

Theorem 5.13 (Completeness (Expressions)). *We have that, for all pure* $\mathsf{eff}(\Gamma) = \perp$,

$$\Gamma \vdash_\epsilon e \approx e' : A \iff [\![\Gamma \vdash_\epsilon e : A]\!]_{\mathsf{Th}^{\otimes}(\cdot)} = [\![\Gamma \vdash_\epsilon e' : A]\!]_{\mathsf{Th}^{\otimes}(\cdot)}$$

In particular, this implies that $\mathsf{Th}(\cdot)$ *is the initial* λ_{SSA} *expression model*

Proof. See Appendix C.7 □

5.7 Regions

We define the category $\mathsf{Th}(\Gamma, \mathsf{L})$ of regions as follows:

- Objects $|\mathsf{Th}(\Gamma, \mathsf{L})|$ types A, B, C
- Morphisms $\mathsf{Th}(\Gamma, \mathsf{L})(A, B) = \{r \mid \Gamma, \square : A \vdash r \rhd \mathsf{L}, \blacksquare(B)\}$ quotiented by $\Gamma, \square : A \vdash r \approx r' \rhd \mathsf{L}, \blacksquare(B)$
- Identity $\Gamma, \square : A \vdash \mathsf{ret}\ \square \rhd \mathsf{L}, \blacksquare(A)$ where $\mathsf{ret}\ a := \mathsf{br}\ \blacksquare\ a$
- Composition $r; r' = [(\blacksquare(\square) \mapsto r')^1]r$

In particular, we may view ret as an identity-on-objects functor $\mathsf{Th}^{\otimes}_{\perp}(\Gamma) \longrightarrow \mathsf{Th}(\Gamma, \mathsf{L})$ with action on morphisms given by given by

$$\Gamma, \square : A \vdash_{\perp} e : B \qquad \mapsto \qquad \Gamma, \square(A) \vdash \mathsf{ret}\ e \rhd \mathsf{L}, \blacksquare(B) \tag{48}$$

We will use this to equip $\mathsf{Th}(\Gamma, \mathsf{L})$ with the structure of a Freyd category. In particular, taking our subcategory of pure morphisms to be the image of ret in $\mathsf{Th}(\Gamma, \mathsf{L})$, we may define the obvious tensor functors

$$- \otimes X : r \mapsto \mathsf{let}\ (\square, x) = \square; (r; \mathsf{ret}\ (\square, x)) \qquad X \otimes - : r \mapsto \mathsf{let}\ (x, \square) = \square; (r; \mathsf{ret}\ (x, \square)) \tag{49}$$

Our premonoidal structure is then completely described by requiring that ret preserves all relevant structure, i.e., that we have

$$\alpha = \mathsf{ret}\ \alpha \qquad \lambda = \mathsf{ret}\ \lambda \qquad \rho = \mathsf{ret}\ \rho \qquad \sigma = \mathsf{ret}\ \sigma \qquad \Delta = \mathsf{ret}\ \Delta \tag{50}$$

Just like for expressions, we can write the coproduct of $\Gamma, \square : A \vdash s \rhd \mathsf{L}, \blacksquare(C)$ and $\Gamma, \square : B \vdash t \rhd \mathsf{L}, \blacksquare(C)$ in $\mathsf{Th}(\Gamma, \mathsf{L})$ as

$$\Gamma, \square : A + B \vdash [s, t] := \mathsf{case}\ \square\ \{\iota_l\ \square : s, \iota_r\ \square : t\} \rhd \mathsf{L}, \blacksquare(C) \tag{51}$$

with the obvious injections $\iota_l := \mathsf{ret}\ \iota_l$ and $\iota_r = \mathsf{ret}\ \iota_r$. It turns out that in this case ret preserves coproducts as well, and we can therefore easily conclude that our category is distributive by taking inverse distributor $\delta^{-1} = \mathsf{ret}\ \delta^{-1}$.

All that remains now is to take $\mathsf{Th}(\Gamma, \mathsf{L})$ from an λ_{SSA} expression model to an λ_{SSA} model by giving it an Elgot structure. We do so by defining the fixpoint of a morphism $\Gamma, \square : A \vdash r \rhd \mathsf{L}, \blacksquare(B + A)$ as follows:

$$\Gamma, \square : A \vdash r^{\dagger} := \mathsf{br}\ \mathsf{go}\ \square\ \mathsf{where}\ \mathsf{go}(\square : A) : \{r; \mathsf{case}\ \square\ \{\iota_l\ x : \mathsf{ret}\ x, \iota_r\ y : \mathsf{br}\ \mathsf{go}\ y\}\} \rhd \mathsf{L}, \blacksquare(B) \tag{52}$$

where go is an (arbitrary) fresh label. We can verify this indeed satisfies the axioms of an Elgot structure through a somewhat tedious calculation. We may now note that

$$\begin{array}{llll} [\![\cdot]\!]_{\mathsf{Th}(\Gamma,\mathsf{L})} = \mathbf{1}, & [\![\Delta, x : A]\!]_{\mathsf{Th}(\Gamma,\mathsf{L})} = [\![\Delta]\!]_{\mathsf{Th}(\Gamma,\mathsf{L})} \otimes A & \Longrightarrow & [\![\Delta]\!]_{\mathsf{Th}(\Gamma,\mathsf{L})} = \langle \Gamma \rangle \\ [\![\cdot^{+}]\!]_{\mathsf{Th}(\Gamma,\mathsf{L})} = \mathbf{0}, & [\![\mathsf{K}, \ell(A)]\!]_{\mathsf{Th}(\Gamma,\mathsf{L})} = [\![\mathsf{K}]\!]_{\mathsf{Th}(\Gamma,\mathsf{L})} + A & \Longrightarrow & [\![\mathsf{K}]\!]_{\mathsf{Th}(\Gamma,\mathsf{L})} = \langle \mathsf{K} \rangle \end{array} \tag{53}$$

and hence that

$$\Gamma, \square : \langle \Delta \rangle \vdash [\![\Delta \vdash_{\epsilon} a : A]\!]_{\mathsf{Th}(\Gamma,\mathsf{L})} \rhd \mathsf{L}, \blacksquare(A) \qquad \Gamma, \square : \langle \Delta \rangle \vdash [\![\Delta \vdash r \rhd \mathsf{K}]\!]_{\mathsf{Th}(\Gamma,\mathsf{L})} \rhd \mathsf{L}, \blacksquare(\langle \mathsf{K} \rangle)$$

as expected. It is relatively easy to derive that

$$\Gamma, \square : \langle \Delta \rangle \vdash [\![\Delta \vdash_{\epsilon} a : A]\!]_{\mathsf{Th}(\Gamma,\mathsf{L})} \approx \mathsf{ret}\ \langle a \rangle \rhd \mathsf{L}, \blacksquare(A)$$

by a relatively straightforward induction. A much more tedious induction is required to prove that

$$\Gamma, \square : \langle \Delta \rangle \vdash [\![\Delta \vdash r \rhd \mathsf{K}]\!]_{\mathsf{Th}(\Gamma,\mathsf{L})} \approx [r] \rhd \mathsf{L}, \blacksquare(\langle \mathsf{K} \rangle)$$

since the case for where-statements is particularly complex. With a little bit more book-keeping (which can be found in the mechanization), we can state the completeness theorem as follows:

Theorem 5.14 (Completeness (Regions)). *We have that, for all pure* $\mathsf{eff}(\Gamma) = \perp$,

$$\Gamma \vdash r \approx r' \rhd \mathsf{L} \iff [\![\Gamma \vdash r \rhd \mathsf{L}]\!]_{\mathsf{Th}(\cdot,\cdot)} = [\![\Gamma \vdash r' \rhd \mathsf{L}]\!]_{\mathsf{Th}(\cdot,\cdot)}$$

In particular, this implies that $\mathsf{Th}(\cdot, \cdot)$ *is the initial* λ_{SSA} *model.*

Proof. See Appendix C.7 □

6 Concrete Models

6.1 Monads and Monad Transformers

As previously stated, every strong monad over a CCC, by virtue of providing a model of higher-order effectful programming, *also* provides a model of first-order effectful programming, i.e., (acyclic) SSA. In particular, the Kleisli category C_T of every such strong monad T is a premonoidal category and induces a Freyd category with pure morphisms given by

$$(\mathsf{C}_\mathsf{T})_\bot(A, B) = \{f; \eta_B : A \to \mathsf{T}B \mid f : A \to B\} \subseteq \mathsf{C}(A, \mathsf{T}B) = \mathsf{C}_\mathsf{T}(A, B) \tag{54}$$

It follows that every strong Elgot monad over a CCC with all coproducts is an λ_{SSA} model (since, in particular, every CCC with coproducts is distributive). This allows us to very quickly amass a large collection of λ_{SSA} models from the literature on strong Elgot monads (see e.g. [28], [26]).

Probably the simplest example of an λ_{SSA} model is given by the *option monad* $\mathsf{Option}\ A = A + \mathbf{1}$ [2], with fixpoint operation

$$f^\dagger = \mathsf{some}\ b \quad \text{if} \quad \exists n, f^n\ a = \mathsf{some}\ (\iota_l\ b) \qquad f^\dagger = \mathsf{none} \quad \text{otherwise}$$

where

$$f^0\ a = \mathsf{some}\ (\iota_r\ a) \qquad f^{n+1}\ a = \mathsf{bind}\ (f^n\ a)\ [(\mathsf{pure}; \iota_l), f]$$

This model allows us to interpret potentially divergent but otherwise purely functional programs. Another very important λ_{SSA} model is generated by the *powerset monad* $\mathcal{P}$ over Set, which has fixpoint operation

$$f^\dagger\ a = \bigcup_n \{b \mid \iota_l\ b \in f^n\ a\}$$

We can use this monad to interpret *nondeterministic* potentially divergent functional programs. We can further expand our repertoire of λ_{SSA} models by considering *monad transformers*, many of which preserve Elgot-ness. For example, if T is a strong Elgot over Set,

- For all types R, the *reader transformer* $\mathsf{ReaderT}\ R\ \mathsf{T}\ A = R \to \mathsf{T}\ A$ is strong Elgot with monad operations

$$\eta\ a = \lambda -. \eta_\mathsf{T}\ a \qquad \mathsf{bind}\ a\ f = \lambda r. \mathsf{bind}_\mathsf{T}\ (a\ r)\ (\lambda a. f\ a\ r) \tag{55}$$

  and fixpoint

$$f^\dagger = \lambda r. (\lambda a. f\ a\ r)^{\dagger_\mathsf{T}} \tag{56}$$

- For all monoids W, the *writer transformer* $\mathsf{WriterT}\ W\ \mathsf{T}\ A = \mathsf{T}\ (A \times W)$ is strong Elgot with monad operations

$$\eta\ a = \eta_\mathsf{T}\ (a, 1) \qquad \mathsf{bind}\ a\ f = \mathsf{bind}_\mathsf{T}\ a\ (\lambda(a, w). \mathsf{bind}_\mathsf{T}\ (f\ a)\ (\lambda(b, w'). (b, w \cdot w'))) \tag{57}$$

  and fixpoint

$$f^\dagger\ a = (\lambda(a, w). \mathsf{bind}_\mathsf{T}\ (f\ a)\ (\lambda(b, w'). \eta_\mathsf{T}([(\lambda c.\ (c, w \cdot w')), (\lambda c.\ (c, w \cdot w'))]\ b)))^{\dagger_\mathsf{T}}(a, 1) \tag{58}$$

- For all types S, the *state transformer* $\mathsf{StateT}\ S\ \mathsf{T}\ A = S \to \mathsf{T}(A \times S)$ is strong Elgot with monad operations

$$\eta\ a = \lambda s. \eta_\mathsf{T}\ (a, s) \qquad \mathsf{bind}\ a\ f = \lambda s. \mathsf{bind}_\mathsf{T}\ (a\ s)\ (\lambda(a, s'). f\ a\ s') \tag{59}$$

  and fixpoint

$$f^\dagger = \lambda s. (\lambda(a, s'). \mathsf{bind}_\mathsf{T}\ (f\ a\ s')\ (\lambda(b, s'). \eta_\mathsf{T}([(\lambda c.\ (c, s')), (\lambda c. (c, s'))]\ b)))^{\dagger_\mathsf{T}} \tag{60}$$

One other important source of λ_{SSA} models is via *Elgot submonads* of monads on Set, which we define as follows:

[2] In a constructive setting, we may equivalently use the partial values monad $\mathsf{Partial}\ A := \Sigma\phi : \mathsf{Prop}. \phi \to A$.

Definition 6.1 (Elgot submonad). A monad $(\mathsf{S}, \eta', \mu')$ is a submonad of (T, η, μ) if $\eta = \eta', \forall A, \mathsf{S}\ A \subseteq \mathsf{T}\ A$, and, on the set $\mathsf{S}\ (\mathsf{S}\ A) \subseteq \mathsf{T}\ (\mathsf{T}\ A)$, $\mu = \mu'$. We say S is *strong Elgot* if T is strong Elgot and, for all $f \in A \to \mathsf{S}\ (A + B)$, $f^\dagger \in A \to \mathsf{S}\ B$ (i.e. the fixpoint operation sends the Kleisli category of S to itself).

One useful property this definition has is that the intersection of arbitrarily many (Elgot) submonads forms an (Elgot) submonad, i.e., the (Elgot) submonads form a complete lattice.

6.2 Trace Models

A particularly important class of λ_{SSA} models are what we will call *trace models*, which can be viewed as programs which either diverge or produce a result, while also producing (potentially nondeterministic, potentially infinite) traces of events $\epsilon \in E$.

Definition 6.2 (Stream Action). A *stream action* $(M, I, \cdot, \Sigma)$ is a monoid actions $(M, I, \cdot)$ of *finite effects* M acting on *divergent effects* I, along with a function $\Sigma : M^\omega \to I$ mapping infinite streams of M to an element of I satisfying $\Sigma\sigma = \sigma_0 \cdot \Sigma_i \sigma_{i+1}$, where $\Sigma_i \sigma_i = \Sigma(\lambda i.\sigma_i)$.

The most important example of a stream action is the *free* stream action over a set E, called the set of *events*, where:

- $M = E^*$ is the free monoid over a set of "events" $\epsilon \in E$, i.e., the monoid of finite lists of events
- $I = E^{\leq \omega}$ is the set of *potentially* infinite streams of events $\epsilon \in E$
- $m \cdot i$ is defined in the obvious manner, i.e. by prepending the (finite) list m to the (potentially infinite) stream i
- $\Sigma\ \sigma$ is defined as the supremum $\bigsqcup_i t_i$, where $t \leq t'$ iff t is a prefix of t', $t_0 = \bot$ (the empty stream), and $t_{i+1} = t_i \cdot \sigma_i$

Definition 6.3 (Trace Monad Transformer). Given a stream action $(M, I, \cdot, \Sigma)$, we can define the *trace monad transformer* over Set as follows: $\mathsf{TraceT}\ \Sigma\ \mathsf{T}\ A = \mathsf{T}\ (A \times M + I)$ with monad operations

$$\eta = (\lambda a.\iota_l(a, 0)); \eta_{\mathsf{T}} \qquad \mathsf{bind}\ a\ f = \mathsf{bind}_{\mathsf{T}}\ a\ [\lambda(a, m).m \cdot f a, \lambda i.\eta_{\mathsf{T}}(\iota_r\ i)]$$

This yields a family of useful Elgot monads, including

- $\mathsf{Traces?}\ \Sigma = \mathsf{TraceT}\ \Sigma\ \mathcal{P}$, with the fixpoint of $f : A \to B + A$ given by

$$f^\dagger\ a = f^\dagger_\infty\ a \cup \bigcup_n f^\dagger_n\ a \qquad \text{where} \tag{61}$$

$$f^\dagger_n\ a = \{\iota_l\ (b, m) \mid \iota_l(\iota_l\ b, m) \in [\mathsf{id}_B, f]^n(\iota_r\ a)\} \cup \{\iota_r\ i \mid \iota_r\ i \in [\mathsf{id}_B, f]^n(\iota_r\ a)\} \tag{62}$$

$$f^\infty\ a = \{\iota_r \Sigma\sigma \mid \exists \alpha : A^\omega, \alpha_0 = a \wedge \forall i, \iota_l\ (\iota_r\ \alpha_{i+1}, \sigma_i) \in f\ \alpha_i\} \tag{63}$$

  We say a morphism $f : A \to \mathsf{Traces?}\ \Sigma\ B$ is *total* if all $f\ a$ are nonempty, and *deterministic* if all $f\ a$ are subsingletons.
- $\mathsf{Traces}\ \Sigma = \mathsf{TraceT}\ \Sigma\ \mathcal{P}^+$, which can be viewed as an Elgot submonad of $\mathsf{Traces?}\ \Sigma$ since the fixpoint of a total function is total
- $\mathsf{Trace?}\ \Sigma = \mathsf{TraceT}\ \Sigma\ \mathsf{Option}$, which can be viewed as an Elgot submonad of $\mathsf{Traces?}\ \Sigma$ since the fixpoint of a deterministic function is deterministic
- $\mathsf{Trace}\ \Sigma = \mathsf{TraceT}\ \Sigma\ \mathsf{Id}$, which can be viewed as an Elgot submonad of $\mathsf{Traces?}\ \Sigma$, and in particular the intersection $\mathsf{Trace?}\ \Sigma \cap \mathsf{Traces}\ \Sigma$.

As a simple, concrete example of this type of model this construction lets us define, consider the *nondeterministic printing monad*, which is simply defined as $\mathsf{Print}\ A \equiv \mathsf{Traces}\ \Sigma$, where $\Sigma : (\mathsf{byte}^*)^\omega \to \mathsf{byte}^{\leq \omega}$ denotes concatenation of bytestrings into a (potentially infinite) bytestream.

This gives us us an λ_{SSA} model supporting effectful instructions print : $\text{byte}^* \to \mathbf{1}$ and nondet : $\mathbf{1} \to A$ (for A nonempty), with semantics $[\![\text{print}]\!] = \lambda b.\{\iota_0((), b)\}$, $[\![\text{nondet}]\!] = \lambda().\{\iota_0(a, []) \mid a \in A\}$.

We can make things a little more interesting by adding a heap to the mix, defining our monad Comp = StateT Heap Print, where Heap = $\mathbb{N} \rightharpoonup \mathbb{N}$ is simply a partial function with finite support. The Kleisli category of this monad is also a Freyd category and inherits an Elgot structure from that on $\text{Set}_{\text{Print}}$; we therefore have an λ_{SSA} model supporting the instructions set : $\mathbb{N} \times \mathbb{N} \to \mathbf{1}$, get : $\mathbb{N} \to \mathbb{N}$, alloc : $\mathbb{N} \to \mathbb{N}$), and free : $\mathbb{N} \to \mathbf{1}$. We can assign semantics to set in the standard fashion, with $[\![\text{set}]\!] = \lambda(p, v)\ h.\{\iota_0((), [p \mapsto v]h, [])\}$. get is a bit more tricky, since it is unclear what to do when we try to access uninitialized memory: one option is simply to return an arbitrary value, with $[\![\text{get}]\!] = \lambda p\ h.\{\iota_0(v, h, []) \mid v = h\ p \vee p \notin h\}$. Finally, $[\![\text{alloc}]\!] = \lambda v\ h.\{\iota_0\ (p, h', []) \mid h' = h \sqcup p \mapsto v\}$ simply fills a random empty heap cell with the provided value, returning a pointer to the cell, while $[\![\text{free}]\!] = \lambda p\ h.\{\iota_0\ ((), h \setminus p, [])\}$.

As we can see, the trace monad is a very powerful tool for quickly constructing families of λ_{SSA} models, at least for relatively simple side-effects. The resulting family of models is somewhat similar to *interaction trees*, which were introduced by Xia et al. [61] to give convenient denotational semantics to imperative programs: these also form an Elgot monad. Because they form an Elgot monad, they immediately form a model of λ_{SSA}. Similar techniques can be used to tackle much more complex side-effects; to demonstrate this, we will show how the model of TSO weak given in Kavanagh and Brookes [38] can be adapted to our framework.

6.3 TSO weak memory

Rather than a sequential trace consisting of a stream of events (here, *actions*) $a_1, a_2, a_3, \ldots$, Kavanagh and Brookes [38] model concurrency by considering a *partially ordered multiset*, or *pomset*, of actions. The incomparable actions in our pomset are then precisely those which are executed in parallel. More formally, we define:

Definition 6.4 (Pomset). A *pomset* α over a set of actions $\mathcal{A}$ with a distinguished null action "*tick*" $\delta \in \mathcal{A}$ is a *nonempty* partially-ordered *carrier set* P such that every $p \in P$ has finitely many predecessors equipped with a mapping $\alpha : P \to \mathcal{A}$. A pomset is *finite* if its carrier set is. We will quotient pomsets over the removal of arbitrarily many copies of δ as long as infinite sets are only equated to other infinite pomsets. We will write unordered pomsets using multiset notation, e.g. $\{a, a, b\}$, and linearly ordered pomsets using list notation, e.g. $[a, a, b]$.

(Finite) pomsets form a monoid under sequential composition $\alpha; \beta$, which is defined by the function $[\alpha, \beta] : \text{trim}(P + Q) \to \mathcal{A}$, where $P + Q$ is given the lexicographic ordering and $\text{trim}(R)$ removes all elements of R with infinitely many predecessors. They also form a monoid under parallel composition, defining $\alpha||\beta = [\alpha, \beta] : P + Q \to \mathcal{A}$ where $P + Q$ is given the standard partial ordering (with elements of P and Q incomparable). Note in both cases the monoidal unit is $\{\delta\}$, since we only consider nonempty pomsets but allow the removal of finitely many ticks. Given any partially ordered set N and a family of pomsets α_n for $n \in N$, we can define their *sum* $\Sigma_n \alpha_n$ to be given by the function $(n, a) \mapsto \alpha_n\ a : \text{trim}(\Sigma_n P_n) \to \mathcal{A}$, where the dependent product $\Sigma_n P_n$ is given the lexicographic order. In particular, choosing $N = \mathbb{N}$ makes $\Sigma : \text{Pom}^{\omega}_{\text{fin}} \to \text{Pom}$ into a stream action monoid w.r.t the sequential composition monoid on finite pomsets.

For simplicity, we will consider an infinite set of named locations $x, y, z \in \text{Loc}$ which are subject to concurrent modification by all threads via TSO reads, writes, and fences. In particular, we define a *program order pomset* to be a pomset with $\mathcal{A}_{\text{PO}} = \mathcal{A}_w \cup \mathcal{A}_r \cup \{\delta\}$, where $\mathcal{A}_r$ consists of *reads* of the form $x = v$ and $\mathcal{A}_w$ consists of *writes* of the form $x := v$ for locations x and values $v \in \text{Word}$.

We may now define the *program order monad* PO A = Traces Σ A, where Σ is taken as a stream action on finite program order pomsets. This yields an λ_{SSA} model with support for

concurrent read and write operations $\mathsf{read}_x : \mathcal{I}_0^{\varnothing}(\mathbf{1}, \mathsf{Word})$, $\mathsf{write}_x : \mathcal{I}_0^{\varnothing}(\mathsf{Word}, \mathbf{1})$ with semantics $[\![\mathsf{read}_x]\!] = R_x^{\mathsf{PO}} = \lambda().\{(v, \{x = v\}) | v \in \mathsf{Word}\}$, $[\![\mathsf{write}_x]\!] = W_x^{\mathsf{PO}} = \lambda v.\{((), \{x := v\})\}$. The semantics of read in particular give a hint as to how this model works: rather than tracking the state of the heap, we simply *emit* a pomset of the events that would have been generated by a given execution (in the case of a read, a read event $x = v$ whenever the read returns v, and in the case of a write, the single write event $x := v$ for a write of v), and later post-filter to obtain a set of valid executions. By doing so, our semantics remains compositional, allowing us to reason about each program fragment individually, while still allowing other program fragments to perform concurrent operations affecting our potential executions. On that note, we can define the *parallel execution* of two morphisms as follows:

$$\begin{aligned} f_0||f_1 = \lambda(a_0, a_1).\{&\iota_0((b_0, b_1), \alpha_0||\alpha_1) \mid \iota_0(b_i, \alpha_i) \in f_i\ a_i\} \\ &\cup \{\iota_1(\alpha_0||\alpha_1) \mid (\iota_0(b_0, \alpha_0) \in f_0\ a \vee \iota_0\ \alpha_0 \in f_0\ a_0) \wedge \iota_1\alpha_1 \in f_1\ a_1\} \\ &\cup \{\iota_1(\alpha_0||\alpha_1) \mid \iota_1\ \alpha_0 \in f_0\ a_0 \wedge \iota_0(a_1, \alpha_1) \in f_1\ a_1\} : \mathsf{Set}_{\mathsf{PO}}(A \otimes A', B \otimes B') \end{aligned} \tag{64}$$

It's tempting to try to use this to define a tensor product of morphisms to obtain a monoidal (rather than premonoidal) category, but unfortunately, sliding still fails: $(W_x^{\mathsf{PO}}||\mathsf{id}); (\mathsf{id}||W_y^{\mathsf{PO}})$ emits the pomset $[x := a, y := b]$, while $(\mathsf{id}||W_y^{\mathsf{PO}}); (W_x^{\mathsf{PO}}||\mathsf{id})$ emits the pomset $[y := b, x := a]$.

To graduate from sequential consistency to a genuine, if maximally simple, weak memory model, we introduce *load buffering* of write actions. We will implement TSO ordering by buffering all our writes, in which case they are only visible to the local thread. On the other hand, read events will first attempt to read from the buffer, and, if there is no corresponding write in the buffer, will read an arbitrary value, in both cases pushing an event to the global pomset. At any point, we may choose to *flush* some of the buffered writes. We introduce the set $\mathcal{A}_b = \{(\overline{x} := v)\}$ of *buffer write actions*, where an action $\overline{x} := v$ by a thread denotes adding a write $x := v$ to the thread's write buffer. We may then define the set of TSO actions $\mathcal{A}_{\mathsf{TSO}} = \mathcal{A}_{\mathsf{PO}} \cup \mathcal{A}_b$; a pomset over this set is called a *TSO pomset*. A buffer will be defined to be a list of write actions $\mathsf{Buf} = \mathcal{A}_b^*$, which we will interpret as linear pomsets over $\mathcal{A}_{\mathsf{TSO}}$ ordered by index (with the empty list corresponding to $\{\delta\}$). In particular, we define the monad $\mathsf{TSO} = \mathsf{StateT}\ \mathsf{Buf}\ (\mathsf{Trace}\ \Sigma)$, where Σ is taken as a the stream action of finite TSO pomsets on TSO pomsets. We can view PO as a submonad of this monad.

Given a buffer Buf, we can define the result of reads $[\cdot]_x : \mathsf{Buf} \to \mathsf{Word} \sqcup \{\bot\}$ from the buffer by induction to be: $(L; \{\overline{x} := v\})[x] = v$,, $(L; \{\overline{y} := v\})[x] = L[x]$, $[][x] = \bot$. Note that writes at the *end* of the buffer are prioritized, since later writes overwrite earlier ones!

The semantics of reads and writes can then be given by

$$\begin{aligned} [\![\mathsf{read}_x]\!] &= R_x^{\mathsf{TSO}} = \mathsf{pflush}; (\lambda()\ L.\{(v, L, \{x = v\}) \mid L[x] = v \vee L[x] = \bot\}); \mathsf{pflush} \\ [\![\mathsf{write}_x]\!] &= W_x^{\mathsf{TSO}} = \mathsf{pflush}; (\lambda v\ L.\{(v, (L; \{\overline{x} := v\}), \{x := v\})\}); \mathsf{pflush} \end{aligned} \tag{65}$$

where buffer flushing is implemented via the morphism

$$\mathsf{pflush}_A = \lambda a\ L.\{(a, R, \alpha) | L = \alpha; R\} : \mathsf{Set}_{\mathsf{TSO}}(A, A)$$

(called split in [38]). To be able to perform synchronization, we will also need to introduce a fence : $\mathcal{I}_0^{\varnothing}(\mathbf{1}, \mathbf{1})$ instruction, which simply causes all actions before the fence to be observed before any actions after the fence. For TSO, implementing this is as simple as flushing the buffer, i.e., we define

$$[\![\mathsf{fence}]\!] = \lambda()\ L.\{((), [], L; \{\delta\})\}$$

We can now define the parallel composition of morphisms in a "fork-join" style as follows: we first flush the buffer completely (i.e., taking it and sticking it at the beginning of our pomset), then execute f and g in parallel with separate buffers, *filtering out executions which completely flush the*

buffer. Since both threads end up with an empty buffer, the resulting joined buffer is also empty, giving us the following definition:

$$\begin{aligned} f_0||f_1 = \lambda(a_0, a_1)\ L.\{&\iota_0((b_0, b_1), [], L; (\alpha_0||\alpha_1)) \mid \iota_0(b_i, [], \alpha_i) \in f_i\ a_i\ []\} \\ &\cup \{\iota_1(\alpha_0||\alpha_1) \mid (\iota_0(b_0, [], \alpha_0) \in f_0\ a \vee \iota_0\ \alpha_0 \in f_0\ a_0\ []) \wedge \iota_1\alpha_1 \in f_1\ a_1\ []\} \\ &\cup \{\iota_1(\alpha_0||\alpha_1) \mid \iota_1\ \alpha_0 \in f_0\ a_0\ [] \wedge \iota_0(a_1, [], \alpha_1) \in f_1\ a_1\ []\} \end{aligned} \tag{66}$$

One issue with this model is that our category currently has "too many" operations besides the ones we want: for example, it is a perfectly valid morphism to simply add an event into the buffer without any flushing. Of course, one thing we could do is consider the smallest Elgot subcategory generated by the atomic instructions we want to support, such as read_x, write_x, and fence, which can be a perfectly valid approach, but is somewhat unwieldy. Another approach, however, is to think about what "valid" morphisms might look like in general; this also has the benefit of letting us write down specifications that may not be trivially implementable in terms of actual instructions.

There are many potential properties we might want "valid" morphisms to have; in general, the less valid morphisms we admit, the more equations we have to work with when reasoning. One simple property we might consider is that a "valid" morphism must have at least one execution in which the buffer is completely flushed, regardless of initial state. This is a perfectly reasonable property to impose, since it ensures that we do not encounter degenerate cases where, for example, the parallel composition of two processes has an empty trace set even though both processes have a nonempty trace set (since neither empties the buffer). We might even want to go a little bit further and, representing the fact that a buffer flush can happen at any time between instructions, require that a nondeterministic buffer flush before or after an instruction does not change its semantics, i.e., that $\mathsf{pflush}_A; f; \mathsf{pflush}_B = f$. Unfortunately, this does *not* hold for the identity, since

$$\mathsf{pflush}; \mathsf{id}; \mathsf{pflush} = \mathsf{pflush}; \mathsf{pflush} = \mathsf{pflush} \neq \mathsf{id} \tag{67}$$

so this would not give us a subcategory. While one possible solution is to simply add the identity back in (and this would give us a valid subcategory), we have the issue that, for example, morphisms like the associator and unitor would still be excluded. Instead, we can take into account the fact that all pflush_A are idempotent and define a new category PTSO with objects sets and morphisms of the form

$$\mathsf{PTSO}(A, B) = \{\mathsf{pflush}_A; f; \mathsf{pflush}_B \mid f \in \mathsf{Set}_{\mathsf{TSO}}(A, B)\} \tag{68}$$

In this category, pflush_A is the identity on A, since

$$\begin{aligned} \mathsf{pflush}_A; \mathsf{pflush}_A; f; \mathsf{pflush}_B &= \mathsf{pflush}_A; f; \mathsf{pflush}_B \\ \mathsf{pflush}_A; f; \mathsf{pflush}_B; \mathsf{pflush}_B &= \mathsf{pflush}_A; f; \mathsf{pflush}_B \end{aligned} \tag{69}$$

This is especially useful since it frees us from needing to remember to sprinkle pflush everywhere when specifying things (and, of course, when that level of control is desired, we can specify in $\mathsf{Set}_{\mathsf{TSO}}$ and then lift to PTSO).

It turns out that this is a general construction we can do, which gives us another useful way to build up λ_{SSA} models that can often be quite difficult to express as simply monads. In particular, we have the following definition

Definition 6.5 (Idempotent envelope category). Let C be a category and d be a family of morphisms for each $A \in |C|$ such that each $d_A : C(A, A)$ is idempotent, i.e., $d_A; d_A = d_A$. We may then define the subsemicategory $\mathsf{Ide}(C, d)$ of C to be given by

$$\mathsf{Ide}(C, d)(A, B) = \{f \in C(A, B) \mid f = d_A; f; d_B\} = \{d_A; f; d_B \mid f \in C(A, B)\} \tag{70}$$

Note that $\mathsf{Ide}(C, d)$ is *not* a subcategory unless $d_A = \mathsf{id}_A$ for all A, in which case $\mathsf{Ide}(C, d) = C$. However, $\mathsf{Ide}(C, d)$ can be *independently* viewed as a category with identity d_A at A.

We have that, if C has coproducts and $d_{A+B} = d_A + d_B$, then $\mathsf{Ide}(C,d)$ inherits coproducts from C with injections

$$\begin{aligned} d_A; \iota_l = \iota_l; d_{A+B} &= d_A; \iota_l; d_{A+B} \in \mathsf{Ide}(C,d)(A, A+B) \\ d_B; \iota_r = \iota_r; d_{A+B} &= d_B; \iota_r; d_{A+B} \in \mathsf{Ide}(C,d)(B, A+B) \end{aligned} \tag{71}$$

since, for $f \in \mathsf{Ide}(C,d)(A,C)$, $g \in \mathsf{Ide}(C,d)(A,C)$, we have that

$$d_{A+B}; [f,g]; d_C = d_A + B; [f,g]; d_C = [d_A; f; d_C, d_B; g; d_C] = [f,g] \tag{72}$$

In particular, it follows that Ide_d preserves coproducts. If C in addition has an Elgot structure, then $\mathsf{Ide}(C,d)$ also inherits this structure (and therefore Ide_d preserves it), as for $f \in \mathsf{Ide}(C,d)(A, B+A)$, we have

$$f^\dagger = (d_A; f; d_B + d_A)^\dagger = (d_A; f; d_B + d_A)^\dagger; d_B = d_A; (f; d_B + (d_A; d_A))^\dagger; d_B = d_A; f^\dagger; d_B \tag{73}$$

On the other hand, if C is equipped with a (symmetric) premonoidal structure and we have that

$$d_{A \otimes B} = d_A \ltimes d_B = d_A \rtimes d_B \qquad d_{A \otimes I} = d_A \otimes I \qquad d_{I \otimes A} = I \otimes d_A \tag{74}$$

(note that d does *not* have to be central!), it follows that $\mathsf{Ide}(C,d)$ has (symmetric) premonoidal structure as well, with inherited tensor product functors and associators, unitors, and symmetries given by

$$\begin{gathered} d_{(A \otimes B) \otimes C}; \alpha_{A,B,C} = \alpha_{A,B,C}; d_{A \otimes (B \otimes C)} = d_{(A \otimes B) \otimes C}; \alpha_{A,B,C}; d_{A \otimes (B \otimes C)} \\ d_{A \otimes I}; \lambda_A = \lambda_A; d_A = d_{A \otimes I}; \lambda_A; d_A \qquad d_{I \otimes A}; \rho_A = \rho_A; d_A = d_{I \otimes A}; \rho_A; d_A \\ d_{A \otimes B}; \sigma_{A,B} = \sigma_{A,B}; d_{B \otimes A} = d_{A \otimes B}; \sigma_{A,B}; d_{B \otimes A} \end{gathered} \tag{75}$$

respectively. It follows that if C is distributive, so is $\mathsf{Ide}(C,d)$, since we can verify that the distributor $[d_A \otimes (d_B; \iota_l), d_A \otimes (d_C; \iota_r)]$ has the expected inverse

$$d_{A \otimes (B+C)}; \delta^{-1}; d_{(A \otimes B) + (A \otimes C)} \tag{76}$$

If we in fact have that $d_{A \otimes B} = A \otimes d_B = d_A \otimes B$ for all A, B (which implies Equation 74) and C is equipped with the structure of a Freyd category, then so is $\mathsf{Ide}(C,d)$, with projections and diagonal given by

$$\begin{gathered} d_{A \otimes B}; \pi_l = \pi_l; d_A = d_{A \otimes B}; \pi_l; d_A \qquad d_{A \otimes B}; \pi_r = \pi_r; d_B = d_{A \otimes B}; \pi_r; d_B \\ d_A; \Delta = \Delta; d_{A \otimes A} = d_A; \Delta; d_{A \otimes A} \end{gathered} \tag{77}$$

Hence, it follows that if C is an λ_{SSA} (expression) model, d_A is idempotent, $d_{A+B} = d_A + d_B$, and $d_{A \otimes B} = A \otimes d_B = d_A \otimes B$, then $\mathsf{Ide}(C,d)$ is an λ_{SSA} (expression) model as well. It turns out that all these properties hold for $d = \mathsf{pflush}$, giving us our final categorical model for TSO weak memory, $\mathsf{PTSO} = \mathsf{Ide}(\mathsf{Set}_{\mathsf{TSO}}, \mathsf{pflush})$.

6.4 Brookes-Style Models

In this section, we show how to construct a concurrency monad from a Brookes [12]-style model of concurrency, which we axiomatize. We will then explicitly show how to instantiate the model for sequentially consistent execution from Brookes [12]. We begin by defining a *countable closure operator*, given below:

Definition 6.6 (Countable Closure Operator). We define a *(countable) closure operator* $\mathsf{c} : \mathcal{P}(T) \to \mathcal{P}(T)$ on T to be a function on sets of T which:

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

We say a set X such that $\mathsf{c}(X) = X$ 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 6.7 (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 an 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.
- *Uniformity*: given arbitrary $f : A \to B+A$, $g : X \to B+X$ and $h : X \to A$ such that $h; f = g; B+h$, for all f_i, we show by induction that $h; f_i = g_i$ since $h; f_0 = 0 = g_0$ and
$$h; f_{i+1} = h; f;[\mathrm{id}, f_i] = g; B+h;[\mathrm{id}, f_i] = g;[\mathrm{id}, h; f_i] = g;[\mathrm{id}, g_i] = g_{i+1}$$
Therefore, $h; f^\dagger = \bigcup_i h; f_i = \bigcup_i g_i = g^\dagger$, as desired.

Following Brookes [12], 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

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

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. [36] gives a Brookes model of TSO, while Dvir et al. [18] gives a Brookes model of release-acquire.

6.5 Non-examples

Weak memory is very complicated, and it has been very difficult to build models which satisfy even "obvious" properties. For example, the model of Jagadeesan et al. [35] fails to even satisfy the associativity of sequencing (i.e., $[\![(S_1; S_2); S_3]\!] = [\![S_1; (S_2; S_3)]\!]$), and it took a nontrivial effort by Jeffrey et al. [37] to establish this property. However, properties like distributivity

$$[\![S; \mathsf{if}\ b\ \{T_1\}\ \mathsf{else}\ \{T_2\}]\!] = [\![\mathsf{if}\ b\ \{S; T_1\}\ \mathsf{else}\ \{S; T_2\}]\!]$$

do not hold, and there is often no semantics of loops at all. This makes them impossible to apply to SSA without extensive adaptation.

One of the easiest ways to build a model with "nice" properties is to make the model monadic; indeed, all the models presented in the above section are an example of this technique. This handles sequencing and distributivity "for free" via the monad laws and the fact that monads always preserve coproducts; however, this still requires us to verify the Conway iteration axioms to support loops, as is done in Xia et al. [62]. Our work identifies the minimal conditions required to have a compositional model of general *first-order* side effects compatible with the standard interpretation of SSA; the higher-order analogue is precisely a (complete) Elgot monad [2], as presented in Goncharov et al. [29].

7 Discussion and Related Work

7.1 Lean Formalization

We have formalized many of the lemmas and theorems presented in this paper in the Lean 4 proof assistant; as of now, our main formalization, `debruijn-ssa` (https://github.com/imbrem/debruijn-ssa), weighs about 29 kloc. While our paper uses named variables for simplicity, all our rules and syntax are formalized using de-Bruijn indices. Each formalized theorem is tagged as such in the respective proof. It was also necessary to fork Mathlib's monoidal category implementation to support Freyd categories, distributivity, and Elgot structure: the resulting library, `discretion` (https://github.com/imbrem/discretion), weighs about 27 kloc. We are currently trying to upstream support for premonoidal categories into Mathlib.

7.2 SSA, FP and IRs

Static Single Assignment (SSA) form was first introduced as a compiler intermediate representation by Alpern et al. [4] and Rosen et al. [58], with the goal of facilitating effective reasoning about program variable equivalence. To perform optimizations like common subexpression elimination (CSE) effectively, we need to determine which expressions are equal *at a given point in time*. By transforming the program into SSA form, we introduce unique variables for each assignment and use ϕ-nodes to merge variable values at points where control paths converge. Since each variable then corresponds to a unique value at a specific point in the program's execution, SSA unlocks the ability to perform *algebraic reasoning* about variable values over time. As a result, analyses like CSE become a matter of simple algebraic rewriting based on variable names.

Cytron et al. [16] provided the first efficient algorithm for converting 3-address code programs to SSA form using a minimal number of ϕ-nodes. They observed that a ϕ-node only needs to be

introduced at the earliest point where a variable may have different values based on control flow. This point is computable via the graph-theoretic notion of a *dominance frontier*, which identifies where control paths merge in the program's control flow graph. Since then, SSA has become the intermediate representation of choice for most production-grade compiler toolchains.

One of the most famous optimizations enabled by SSA's support for effective algebraic reasoning is *sparse conditional constant propagation (SCCP)*, introduced in Wegman and Zadeck [60]. This algorithm leverages the SSA property to reason about the possible set of values each variable may take using a lattice-based data-flow analysis, which models variable values in terms of abstract states like *unknown*, *constant*, or *overdefined*. Without SSA, one could naively consider all variable definitions, which would likely detect fewer constants due to imprecision. Alternatively, achieving more precise results would require a complex and computationally intensive reaching-definitions analysis.

However, while SSA is an excellent representation for implementing compilers, the semantics of ϕ-nodes can be quite unintuitive due to their lack of an obvious operational interpretation, making them challenging to reason about formally. Kelsey [39] establishes a correspondence between SSA and a subset of *continuation-passing style (CPS)*, a common intermediate representation for functional compilers. In particular, while not all CPS programs can be directly converted to SSA form—those using non-local returns like `longjmp` or `call/cc`—the typical outputs of CPS transformations avoid such features. Kelsey observed that many optimizations requiring flow analysis in CPS could be performed directly in SSA, often dramatically simplifying them.

Appel [5] builds on this work by informally showing that the functional subset of CPS programs "hidden inside" SSA is, in fact, simply nested, mutually tail-recursive functions, with each function corresponding to a basic block. He makes the key observation that the dominance-based scoping of SSA corresponds to the lexical scoping of functions. For a variable to be visible in a function, it must appear in the lexical scope of that variable's definition and therefore be dominated by it; otherwise, there would be no way to call the function.

In fact, the subset of functional programs identified by Appel [5] corresponds to *A-normal form (ANF)*, another functional intermediate representation advocated by Flanagan et al. [21]. Chakravarty et al. [14] formalizes this correspondence giving an algorithm to convert SSA programs to ANF, and then showing how SSA optimizations such as SCCP can be written to operate on ANF programs. The authors highlight that the semantic rigor of their notation, combined with the well-defined semantics of ANF, make their presentation of SCCP significantly more amenable to formal analysis.

Going in the other direction, Leißa et al. [43] introduce the Thorin intermediate representation, which consists of CPS extended with SSA-style dominance-based scoping. As SSA is a first-order language, it is often difficult to represent programs using closures effectively, and consequently, difficult to optimize them well. The authors give the example of LLVM, which often needs to generate a new struct and a large amount of boilerplate code for every closure, which, even after optimization, is often not significantly reduced. By contrast, Thorin retains the advantages of CPS for representing functional programs, while enabling SSA-like graph-based use-def analysis by prohibiting variable shadowing.

The primary focus of the works we have covered so far is establishing the correspondence between SSA, an imperative intermediate representation, and widely used functional intermediate representations. However, to advance beyond these correspondences and directly study the optimization and verification of SSA programs themselves, we require an equational theory underpinned by formal semantics. One approach found in the literature (and which we also use) is to relax SSA into forms that support richer substitution principles and well-defined operational

semantics, which can then be used to reason about SSA itself. Two papers which exemplify this approach are Benton and Kennedy [9] and Garbuzov et al. [23].

One of the difficulties of working with ANF as an intermediate representation is that it does not, in general, satisfy *substitution*: replacing a variable x with a compound expression like a function call can take a program out of the ANF-fragment. The *monadic intermediate language (MIL)* introduced by Benton and Kennedy [9] for use in the MLj compiler for Standard ML can be viewed as a relaxation of ANF (and hence, of SSA) to get a nicer substitution principle. Benton and Kennedy then use this flexibility to build up an equational theory justified by their operational semantics. One interesting feature is that, unlike Moggi's equational metalanguage [51] (and like our λ_{SSA} calculus), MIL enforces a stratification of *values* and *computations*, without supporting "computations of computations" (corresponding to nested monad types $\mathsf{T}(\mathsf{T}(A))$). This hints at Freyd categories, rather than general Kleisli categories, being a natural model of MIL-like intermediate representations.

Similarly, Garbuzov et al. [23] exhibit a correspondence between an operational semantics for SSA and an operational semantics for call-by-push-value (CBPV) [45]. They then use the normal form bisimulations of Lassen and Levy [40] to derive an equational theory for use in justifying optimizations. In particular, we can view their paper as interpreting CBPV as a relaxation of SSA more suited for developing an equational theory and for semantics work; in particular, to be able to take advantage of CBPV to give a *structural* operational semantics for (unstructured) SSA programs. The semantics of CBPV are widely studied, and hint at a large variety of potential models for SSA, but the formalization in [23] does not support any effects other than nontermination.

7.3 Formalizations of SSA

7.3.1 Other SSA type systems. Several attempts have been made to provide a type-theoretic treatment of SSA. The work most similar to ours is by Rigon et al. [56], who present a typed translation from SSA into the lambda calculus using a type-and-effect system, observing that the algorithm for converting programs to SSA form may also be viewed as a mechanism for transforming programs in a functional language with unstructured control flow into equivalent expressions.

Similarly to λ_{SSA}, they extend the lambda calculus with a mutually recursive `where`-binding, which allows them to directly translate unstructured SSA control flow. However, their calculus uses only a single syntactic category of expressions, whereas we attempt to model (generalized) SSA directly by distinguishing between expressions and regions. Their language also includes support for effect handlers, which are beyond the scope of our current study but represent an interesting direction for future work. While the authors do not provide an equational theory or a semantics for their language, we believe that our equational theory could be adapted to the fragment of their language without effect handlers.

An interesting alternative approach is demonstrated by Matsuno and Ohori [49], who give a type theory for what appear to be ordinary three-address code programs. However, every well-typed program can be placed into SSA-form by inserting ϕ-nodes in a fully type-directed way. This lets them model SSA without any ϕ-nodes, letting them use the standard semantics for three-address code.

Menon et al. [50] give a type-safe formalization of SSA, along with an operational semantics and formal definitions of dominance, definition/use points, and the SSA property for 3-address code. By augmenting SSA with first-class proof variables, they aim to give a representation which allows aggressive optimizations to preserve safety information. Their type system requires checking the SSA property separately from well-typedness, but is proven sound if the SSA property holds. Hua et al. [33] give another type system and direct operational semantics for standard SSA, and prove type safety for it.

Many operational semantics for SSA have arisen from compiler verification efforts. Barthe et al. [7] give an operational semantics as part of the CompCertSSA project, and give a semantics-preserving translation from three-address code into SSA. Herklotz et al. [31] formalise "gated SSA" and give semantics-preserving translations between it and ordinary SSA. Going beyond CompCertSSA, Zhao et al. [63] have studied the semantics of the LLVM IR itself as part of the Vellvm project; Li and Gunter [48] goes further and gives an operational semantics for LLVM in the (sequentially consistent) multithreaded setting.

There has been much less work on denotational semantics for SSA, or directly on its equational theory. Pop et al. [54] give an unusual denotational model of SSA in terms of the iteration structure of a program, which they use to better understand the loop-closing ϕ-nodes found both in the gated SSA representation as well as practical compilers such as GCC. Xia et al. [61] describe a toolkit for building denotational semantics for imperative languages, including the standard IMP language and a simple assembly language, using the *interaction tree monad* and *continuation tree monad*, which they show to satisfy the Elgot axioms. Recently, Chappe et al. [15] introduced *choice trees* (CTrees), which extend ITrees to support weak memory concurrency, and use this to model a subset of LLVM IR.

7.3.2 Mechanizations of SSA. CompcertSSA [6] is an attempt to extend the CompCert verified compiler [44] with an SSA-based middle-end. This is achieved by generating a pruned SSA IR from CompCert's RTL format. Instead of verifying the RTL-to-SSA translation within Coq, this translation is performed by unverified code, and a separate *translation validation* stage uses a verified checker to ensure correctness. After performing (verified) SSA optimizations, the IR is naively lowered back to RTL for the rest of CompCert's machinery to work on. Demange et al. [17] build on this work by providing realistic, verified implementations of Sparse Conditional Constant Propagation (SCCP) [60], Common Subexpression Elimination (CSE), and Global Value Numbering (GVN) [58] within a general framework of flow-insensitive static analysis for CompCertSSA.

While CompCert is a verified implementation of a new compiler in Coq, the Vellvm project [63] attempts to mechanize a subset of the LLVM intermediate representation (IR), covering the LLVM type system, operational semantics, and the well-formedness and structural properties of valid LLVM IR. The authors adopt a memory model for LLVM based on CompCert's [44], allowing them to leverage significant portions of CompCert's Coq infrastructure. Similarly, Bhat et al. [10] attempt to mechanize a well-defined subset of the Multi-Level Intermediate Representation (MLIR) in the Lean 4 theorem prover. Their formalization features a user-friendly front end to convert MLIR syntax into their calculus and scaffolding for defining and verifying peephole rewrites using tactics. Their framework has been tested on bitvector rewrites from LLVM, structured control flow, and fully homomorphic encryption; however, as of publication, only structured control flow was fully supported.

7.3.3 Completeness. Most other formalizations oF SSA have given *specific* semantics for it, whether operational or denotational. One of the features which makes our work distinctive is that we give our type theory semantics relative to a categorical axiomatization. Essentially, we have parameterized our interpretation over any model satisfying the Freyd-Elgot axioms. First, this lets us show that many different concrete semantics are valid models of SSA, as can be seen in section 6. This also let us show that our equational theory is complete: we have exactly the equations valid in all Freyd-Elgot models, no more and (crucially) no less. As a result, implementors and researchers proving things about compiler optimizations do not have to deal with any of the messy details of (for example) weak memory semantics, unless they are trying to study optimizations specifically involving weak memory. All the usual equations used in control- and data-flow optimizations are independent of those details, and can be validated using the equational theory.

Conversely, researchers working on models of weak memory (and other strange models of computation such as quantum computation), can use the Freyd-Elgot axioms as a target. If they ensure their models satisfy these axioms, then using an SSA-based IR in their compiler is justified.

7.4 Compositional (Relaxed) Concurrency

The most natural way to think about concurrency is often in terms of an operational semantics on an abstract machine, in which concurrent threads interleave and interact. Such semantics, naturally, are designed to reason about an entire program at a time. Batty [8] argues that a compositional semantics of concurrency is necessary to be able to reason about properties of large software systems effectively, particularly in the presence of complicating factors such as compiler optimizations and weak memory semantics. However, it is generally very challenging to reason about a small component of a concurrent system in isolation since its behaviour may be drastically affected by other components running in parallel.

One natural approach to constructing denotational models of concurrency is to consider the extension of an abstract machine's behavior. Each thread performs a sequence of atomic actions, and so from the outside, the machine can produce any interleaving of the atomic actions of each thread. Just by itself, considering sets of possible traces is not sufficiently abstract, and so Brookes [12] takes the closure of these sets under semantics-preserving transformations, such as stuttering (introducing extra identity steps) and mumbling (which fuse sequential atomic operations, thereby hiding implementation details), to obtain a model with good equational properties such as associativity ($[\![\alpha; (\beta; \gamma)]\!] = [\![(\alpha; \beta); \gamma]\!]$) and the expected identities for branches and loops.

If the machine model is *sequentially consistent* – i.e., all allowed behaviors arise from interleavings of sequences of atomic events – then this style of trace semantics is sufficient. However, real hardware often exhibits *relaxed behaviour*, in which additional behaviours which do not correspond to the interleaving of parallel threads are allowed. Even more relaxed behaviours arise from fundamental compiler optimizations, such as re-ordering of independent reads and writes. These are perfectly valid in a single-threaded context but can introduce new behaviours in multithreaded programs — for example, another thread could distinguish the order of writes. In general, we need tools to reason about interactions with a system that is not *linearizable* (i.e., in which concurrent operations cannot be reduced to interleaved atomic operations on a single thread), of which actual hardware is only one example. There are two major approaches to this problem.

One idea we might have is to augment traces with additional structure, which we can then quotient away to maintain extensionality by using an appropriate closure operator. What's nice about this method is that we can often use structures analogous to the additional state in the machine model which leads to the relaxed behaviour in the first place. For example, in Jagadeesan et al. [36], the authors extend Brookes [12] with additional state corresponding to the contents of a thread-local buffer, and then take the closure of their trace-set with respect to buffer operations such as nondeterministic flushing. This gives them a model of TSO weak memory with good equational properties and a monadic structure, which remains intuitive, since it has a connection to the original TSO model which can also be framed in terms of the abstract machine having thread-local buffers. Dvir et al. [18] use this approach to derive a trace-based semantics for release-acquire atomics inspired by their operational semantics, showing the viability of this technique even for very complex memory models.

The idea of augmenting traces leads naturally to *games*, which we can view one variant of as sequential traces of moves between a *proponent* and an *opponent*. Game semantics were first used with great success to give a semantics to *sequential computation*; for example, Abramsky and McCusker [1] give an adequate denotational semantics for sequential Algol using Hyland-Ong

games [34]. Ghica and Murawski [25] observe that the sequentiality of Hyland-Ong games corresponds to a highly constrained, deterministic form of interleaving between concurrent processes. By generalizing away many of the rules of Hyland-Ong games, and in particular *alternation* (i.e., the proponent and the opponent must take turns), the authors obtain a form of game-semantics for concurrent programs, which they use to build a fully abstract semantics for *fine-grained concurrency* (in contrast to Brookes [12], which implements *coarse-grained* concurrency using the somewhat unrealistic await primitive).

In general, generalizing traces to represent true concurrency leads us to another approach: replacing sets of *linear* traces with (sets of) *partially ordered* structures that directly represent the concurrency of their component operations. This approach directly captures the idea that executing two operations concurrently (i.e., without specifying an order between them) is fundamentally different from executing them in some particular order. The most basic such data structure is a *partially ordered multiset*, or *pomset*, which we describe in Section 6.3 of our paper. Pomsets can naturally model sequential consistency, and, like traces, can be augmented with additional structure to model relaxed memory models such as TSO weak memory, as in Kavanagh and Brookes [38].

One issue with this approach is that it can be very challenging to determine the appropriate structures to augment pomsets with in order to model more advanced memory models. For example, Jagadeesan et al. [35] introduce *pomsets with preconditions* (PwP), which augment pomsets with logical formulae, to model weak memory behaviors. However, Jeffrey et al. [37] demonstrate that sequential composition (the semicolon operator) in PwP is not associative. This lack of associativity undermines many common program optimizations and breaks the monadic structure necessary for compositional reasoning. To address this issue, Jeffrey et al. [37] propose *pomsets with predicate transformers* (PwT), which restore the associativity of sequential composition. Despite this improvement, their semantics still do not support loops, and developing a monadic structure based on PwT that fully supports recursion and iterative constructs remains future work.

Another potential generalization of pomsets, inspired by game semantics, is the use of *event structures*, which introduce a *conflict relation* to represent many potentially conflicting executions within a single mathematical object. Castellan [13] show how to use event structures to represent concurrent executions with respect to a memory model. They study a simple language supporting parallel composition of n linear programs defined as lists of loads, stores, and arithmetic operations, for which event structures provide a denotational semantics. Paviotti et al. [53] use the event structure approach to give denotational semantics for relaxed memory concurrency in a more realistic language, including semantics for branches and loops. However, they employ step indexing in their semantics, and as a result, loop unrolling is only a refinement rather than an equation. Since event structures satisfy the axioms of axiomatic domain theory [20], it should be possible to modify this semantics into one where loop unrolling is an equation, at which point it would be a model of our calculus.

7.5 Future Work

7.5.1 Substructural Types and Effects. Currently, our treatment of effects is relatively primitive, in that, while we postulate a lattice of effects, our equational theory only distinguishes between *pure* and *impure* functions, the former having effect $\bot$. Führmann [22] studies languages with semantics given in an *abstract Kleisli category*. In particular, he introduces the notion of

- *Central* operations, which commute with all other operations; we like to call such morphisms *linear*
- *Duplicable* operations f, for which let $y = f(x);\ (y, y) \approx (f(x), f(x))$
- *Discardable* operations f, for which let $y = f(x);\ e \approx e$ when e does not depend on y

While, as we make use of in our calculus, pure operations are central, duplicable, and discardable, it can be useful to study each notion in isolation, as well as to consider morphisms which are central and duplicable (which we have taken to calling *relevant*) and morphisms which are both central and discardable (which we have taken to calling *affine*). Examples of morphisms which are not pure but nonetheless relevant or affine abound; for example,

- In a programming language with nondeterminism, nondeterministic functions are *impure* (since let $y = f(x);\ (y, y)$ is a strict refinement of $(f(x), f(x))$), but are nontheless *affine*, because discarding the result of a nondeterministic function does not affect the program's behavior, assuming they have no other effect.
- In a programming language with nontermination, nonterminating functions with no other effect are impure (since let $y = f(x);\ e$ may diverge even if e does not) but are always duplicable, because duplicating a nonterminating function does not introduce new effects beyond divergence. Depending on whether the language's other effects commute with nontermination, they may in fact be relevant.

All three of these concepts still make perfect sense when interpreted as-is in a Freyd category, and are particularly useful when reasoning about models of probabilistic programming, such as Markov categories (which have many affine morphisms) [52].

Given that such relevant and affine side-effects have substitution principles which depend on how often a variable is used in a term, we call such side-effects *substructural*. We might also consider how we could support substructural, and in particular linear, *types*. Categorically, this corresponds to weakening the requirement that our base category be cartesian, instead requiring only a symmetric monoidal category. The resulting generalization of a Freyd category is called an *effectful category* by Román [57].

7.5.2 Refinement and Enrichment. Throughout this work, we have focused on the notion of program *equivalence* $\Gamma \vdash r \approx r' \rhd \mathsf{L}$. In many settings, especially when considering the nondeterminism and concurrency, we would also like to study program *refinement* $\Gamma \vdash r \sqsubseteq r' \rhd \mathsf{L}$. Semantically, this corresponds to an enrichment of our model in the category of partial orders; we conjecture that adding this to our semantics would require minimal changes. We may also consider how our syntax could be extended to support explicit parallelism and how we could reflect the corresponding algebraic laws (as given in, e.g., Hoare and van Staden [32]), such as

$$(P \parallel Q); (R \parallel S) \sqsubseteq (P; R) \parallel (Q; S) \tag{78}$$

in our equational theory. More generally, we can consider enrichment with further structures, such as distributive lattices and dcpos, and the language features these correspond to, such as nondeterministic choice between programs $P \vee C$. These features are particularly interesting from the perspective of programs as *specifications*, as they offer the potential for representing complex specifications within SSA.

7.5.3 Guarded Iteration. Elgot categories and monads were introduced in Elgot [19], and subsequently formalized in Adámek et al. [2]. Levy and Goncharov [46] generalize Elgot categories to *guarded Elgot categories* (and, correspondingly, *guarded Elgot monads*, whose Kleisli categories are guarded Elgot), to support partial non-unique iteration – in other words, categories where the fixed point does not always exist. Generalizing our language with support for guarded iteration could allow us to study applications of SSA to domains for which not all fixpoints exist, such as the representation of terminating languages (as, for example, one would find in a proof assistant) or productive processes.

Goncharov et al. [27] introduce a metalanguage for guarded iteration based on labelled iteration as presented by Geron and Levy [24]. Interestingly, they refer to the labels in their systems as *exceptions*, making the claim that this more closely matches their semantics. Similar to our work, they provide a denotational semantics for this language using the Kleisli category of a strong guarded *pre-iterative* monad T—that is, a monad with a guarded fixpoint operator—on a distributive category C. They then demonstrate that this semantics is sound and adequate with respect to a big-step operational semantics for a specific monad, namely $\mathsf{T}\ X = (X \times \mathbb{N}^*) + \mathbb{N}^\omega$ over Set. Notably, unlike our trace monad, their monad is *guarded iterative* rather than merely iterative. The infinite branch of the coproduct $\mathbb{N}^\omega$ represents an infinite sequence of natural numbers, effectively prohibiting non-productive infinite loops — that is, recursion that does not pass through a *guard*, which in this context is emitting a natural number. Their treatment of iteration is thus somewhat more general than ours, as they support guarded iteration and require only a pre-iterative monad. However, they focus on Kleisli categories rather than providing a general treatment for Freyd categories. Additionally, their language supports functional types, whereas our language is purely first-order. We are very interested in exploring the connection between guarded labelled-iteration and SSA implied by the similarities between our respective syntaxes and semantics.

Acknowledgements

This work was supported in part by a European Research Council (ERC) Consolidator Grant for the project "TypeFoundry", funded under the European Union's Horizon 2020 Framework Programme (grant agreement no. 101002277).

References

[1] Samson Abramsky and Guy McCusker. 1996. Linearity, Sharing and State: a fully abstract game semantics for Idealized Algol with active expressions. In *Linear Logic Tokyo Meeting 1996, Keio University, Mita Campus, Tokyo, Japan, March 29 - April 2, 1996 (Electronic Notes in Theoretical Computer Science, Vol. 3)*, Jean-Yves Girard, Mitsuhiro Okada, and Andre Scedrov (Eds.). Elsevier, Tokyo, Japan, 2–14. https://doi.org/10.1016/S1571-0661(05)80398-6

[2] Jirí Adámek, Stefan Milius, and Jirí Velebil. 2011. Elgot theories: a new perspective on the equational properties of iteration. *Math. Struct. Comput. Sci.* 21, 2 (2011), 417–480. https://doi.org/10.1017/S0960129510000496

[3] Frances E. Allen. 1970. Control flow analysis. *SIGPLAN Not.* 5, 7 (jul 1970), 1–19. https://doi.org/10.1145/390013.808479

[4] 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

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

[6] Gilles Barthe, Delphine Demange, and David Pichardie. 2012. A Formally Verified SSA-Based Middle-End - Static Single Assignment Meets CompCert. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 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. 7211)*, Helmut Seidl (Ed.). Springer, Tallinn, Estonia, 47–66. https://doi.org/10.1007/978-3-642-28869-2_3

[7] Gilles Barthe, Delphine Demange, and David Pichardie. 2014. Formal Verification of an SSA-Based Middle-End for CompCert. *ACM Trans. Program. Lang. Syst.* 36, 1, Article 4 (mar 2014), 35 pages. https://doi.org/10.1145/2579080

[8] 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

[9] Nick Benton and Andrew Kennedy. 1999. Monads, Effects and Transformations. *Electronic Notes in Theoretical Computer Science* 26 (1999), 3–20. https://doi.org/10.1016/S1571-0661(05)80280-4 HOOTS '99, Higher Order Operational Techniques in Semantics.

[10] Siddharth Bhat, Alex Keizer, Chris Hughes, Andrés Goens, and Tobias Grosser. 2024. Verifying Peephole Rewriting in SSA Compiler IRs. In *15th International Conference on Interactive Theorem Proving (ITP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 309)*, Yves Bertot, Temur Kutsia, and Michael Norrish (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 9:1–9:20. https://doi.org/10.4230/LIPIcs.ITP.2024.9

[11] Corrado Böhm and Giuseppe Jacopini. 1966. Flow diagrams, turing machines and languages with only two formation rules. *Commun. ACM* 9, 5 (May 1966), 366–371. https://doi.org/10.1145/355592.365646

[12] 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

[13] Simon Castellan. 2016. Weak memory models using event structures. In *Vingt-septièmes Journées Francophones des Langages Applicatifs (JFLA 2016)*, Julien Signoles (Ed.). Saint-Malo, France. https://inria.hal.science/hal-01333582

[14] 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

[15] Nicolas Chappe, Ludovic Henrio, and Yannick Zakowski. 2025. Monadic Interpreters for Concurrent Memory Models: Executable Semantics of a Concurrent Subset of LLVM IR. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Denver, CO, USA) *(CPP '25)*. Association for Computing Machinery, New York, NY, USA, 283–298. https://doi.org/10.1145/3703595.3705890

[16] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (oct 1991), 451–490. https://doi.org/10.1145/115372.115320

[17] Delphine Demange, David Pichardie, and Léo Stefanesco. 2015. Verifying Fast and Sparse SSA-Based Optimizations in Coq. In *Compiler Construction - 24th International Conference, CC 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9031)*, Björn Franke (Ed.). Springer, London, UK, 233–252. https://doi.org/10.1007/978-3-662-46663-6_12

[18] 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

[19] Calvin C. Elgot. 1975. Monadic Computation And Iterative Algebraic Theories**The material reported on here evolved from research which was intiated at the University of Bristol and supported by the Science Research Council of Great Britain. In *Logic Colloquium '73*, H.E. Rose and J.C. Shepherdson (Eds.). Studies in Logic and the Foundations of Mathematics, Vol. 80. Elsevier, Amsterdam, The Netherlands, 175–230. https://doi.org/10.1016/S0049-237X(08)71949-9

[20] Marcelo P. Fiore. 1994. *Axiomatic domain theory in categories of partial maps.* Ph. D. Dissertation. University of Edinburgh, UK. https://hdl.handle.net/1842/406

[21] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Robert Cartwright (Ed.). ACM, USA, 237–247. https://doi.org/10.1145/155090.155113

[22] 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

[23] 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

[24] Bram Geron and Paul Blain Levy. 2016. Iteration and Labelled Iteration. 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, USA, 127–146. https://doi.org/10.1016/J.ENTCS.2016.09.035

[25] Dan R. Ghica and Andrzej S. Murawski. 2008. Angelic semantics of fine-grained concurrency. *Ann. Pure Appl. Log.* 151, 2-3 (2008), 89–114. https://doi.org/10.1016/J.APAL.2007.10.005

[26] Sergey Goncharov. 2021. Uniform Elgot Iteration in Foundations. *CoRR* abs/2102.11828 (2021), 41 pages. arXiv:2102.11828 https://arxiv.org/abs/2102.11828

[27] Sergey Goncharov, Christoph Rauch, and Lutz Schröder. 2021. A metalanguage for guarded iteration. *Theor. Comput. Sci.* 880 (2021), 111–137. https://doi.org/10.1016/J.TCS.2021.04.005

[28] 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

[29] Sergey Goncharov, Lutz Schröder, Christoph Rauch, and Julian Jakob. 2018. Unguarded Recursion on Coinductive Resumptions. *Log. Methods Comput. Sci.* 14, 3 (2018). https://doi.org/10.23638/LMCS-14(3:10)2018

[30] Masahito Hasegawa. 2002. The Uniformity Principle on Traced Monoidal Categories. In *Category Theory and Computer Science, CTCS 2002, Ottawa, Canada, August 15-17, 2002 (Electronic Notes in Theoretical Computer Science, Vol. 69)*, Richard Blute and Peter Selinger (Eds.). Elsevier, Amsterdam, The Netherlands, 137–155. https://doi.org/10.1016/S1571-0661(04)80563-2

[31] Yann Herklotz, Delphine Demange, and Sandrine Blazy. 2023. Mechanised Semantics for Gated Static Single Assignment. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Boston, MA, USA) *(CPP 2023)*. Association for Computing Machinery, New York, NY, USA, 182–196. https://doi.org/10.1145/3573105.3575681

[32] Tony Hoare and Stephan van Staden. 2014. The laws of programming unify process calculi. *Sci. Comput. Program.* 85 (2014), 102–114. https://doi.org/10.1016/J.SCICO.2013.08.012

[33] Baojian Hua, Bo Xu, and Ying Gao. 2010. Explicitly typed static single-assignment form. In *2010 2nd International Conference on Education Technology and Computer*, Vol. 1. IEEE, Shanghai, China, V1–43–V1–47. https://doi.org/10.1109/ICETC.2010.5529303

[34] J. M. E. Hyland and C.-H. Luke Ong. 2000. On Full Abstraction for PCF: I, II, and III. *Inf. Comput.* 163, 2 (2000), 285–408. https://doi.org/10.1006/INCO.2000.2917

[35] Radha Jagadeesan, Alan Jeffrey, and James Riely. 2020. Pomsets with preconditions: a simple model of relaxed memory. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 194:1–194:30. https://doi.org/10.1145/3428262

[36] 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

[37] Alan Jeffrey, James Riely, Mark Batty, Simon Cooksey, Ilya Kaysin, and Anton Podkopaev. 2022. The leaky semicolon: compositional semantic dependencies for relaxed-memory concurrency. *Proc. ACM Program. Lang.* 6, POPL (2022), 1–30. https://doi.org/10.1145/3498716

[38] Ryan Kavanagh and Stephen Brookes. 2018. A Denotational Semantics for SPARC TSO. *Electronic Notes in Theoretical Computer Science* 336 (2018), 223–239. https://doi.org/10.1016/j.entcs.2018.03.025 The Thirty-third Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIII).

[39] 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

[40] 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

[41] 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.

[42] 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

[43] Roland Leißa, Marcel Köster, and Sebastian Hack. 2015. A graph-based higher-order intermediate representation. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015*, Kunle Olukotun, Aaron Smith, Robert Hundt, and Jason Mars (Eds.). IEEE Computer Society, San Francisco, CA, USA, 202–212. https://doi.org/10.1109/CGO.2015.7054200

[44] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115. https://doi.org/10.1145/1538788.1538814

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

[46] 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

[47] Paul Blain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Inf. Comput.* 185, 2 (2003), 182–210. https://doi.org/10.1016/S0890-5401(03)00088-9

[48] Liyi Li and Elsa L. Gunter. 2020. K-LLVM: A Relatively Complete Semantics of LLVM IR. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 7:1–7:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.7

[49] Yutaka Matsuno and Atsushi Ohori. 2006. A type system equivalent to static single assignment. In *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 10-12, 2006, Venice, Italy*, Annalisa Bossi and Michael J. Maher (Eds.). ACM, Venice, Italy, 249–260. https://doi.org/10.1145/1140335.1140365

[50] Vijay Menon, Neal Glew, Brian R. Murphy, Andrew McCreight, Tatiana Shpeisman, Ali-Reza Adl-Tabatabai, and Leaf Petersen. 2006. A verifiable SSA program representation for aggressive compiler optimization. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, USA, 397–408. https://doi.org/10.1145/1111037.1111072

[51] 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

[52] nLab authors. 2024. Markov category. https://ncatlab.org/nlab/show/Markov+category. Revision 62.

[53] Marco Paviotti, Simon Cooksey, Anouk Paradis, Daniel Wright, Scott Owens, and Mark Batty. 2020. Modular Relaxed Dependencies in Weak Memory Concurrency. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, Dublin, Ireland, 599–625. https://doi.org/10.1007/978-3-030-44914-8_22

[54] Sebastian Pop, Pierre Jouvelot, and George André Silber. 2009. In and Out of SSA : a Denotational Specification. In *Workshop Static Single-Assignment Form Seminar,*. Universität des Saarlandes, Autrans, France, 15 pages. https://minesparis-psl.hal.science/hal-00915979

[55] 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

[56] Leonardo Filipe Rigon, Paulo Torrens, and Cristiano D. Vasconcellos. 2020. Inferring types and effects via static single assignment. In *SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 - April 3, 2020*, Chih-Cheng Hung, Tomás Cerný, Dongwan Shin, and Alessio Bechini (Eds.). ACM, Brno, Czech Republic, 1314–1321. https://doi.org/10.1145/3341105.3373888

[57] 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

[58] 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

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

[60] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant Propagation with Conditional Branches. *ACM Trans. Program. Lang. Syst.* 13, 2 (1991), 181–210. https://doi.org/10.1145/103135.103136

[61] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119

[62] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119

[63] 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 Records and Enums

For simplicity, we defined our language to have only binary products and sums. However, we would like to support records (named n-ary products) and enums (named n-ary sums). We will implement these on top of our language using contexts; the resulting machinery will turn out to be crucial in proving the Böhm-Jacopini theorem in Appendix A.1 and completeness in Section 5.5. We begin by defining *packing* of variable contexts as follows:

$$\langle \cdot \rangle = \mathbf{1} \qquad \langle \Gamma, x : A \rangle = \langle \Gamma \rangle \otimes A \tag{79}$$

While in general we destructure pairs using binary let-bindings, it will be more convenient to start defining operations on records in terms of projections. We begin by defining projections on pairs in the obvious manner:

$$\frac{\Gamma \vdash_\epsilon e : A \otimes B}{\Gamma \vdash_\epsilon \pi_l\ e := (\mathsf{let}\ (x, y) = e;\ x) : A}\ \pi_l \qquad \frac{\Gamma \vdash_\epsilon e : A \otimes B}{\Gamma \vdash_\epsilon \pi_r\ e := (\mathsf{let}\ (x, y) = e;\ y) : A}\ \pi_r \tag{80}$$

We may then define projections on records as follows:

$$\pi_{(\Delta, x:A), y}\ e = \pi_{\Delta, y}(\pi_l\ e) \qquad \pi_{(\Delta, x:A), x}\ e = \pi_r\ e \tag{81}$$

with typing rule

$$\frac{\Gamma \vdash_\epsilon e : \langle \Delta \rangle \quad \Delta(x) = A}{\Gamma \vdash_\epsilon \pi_{\Delta, x}\ e : A}\ \pi_{\mathsf{rec}} \tag{82}$$

We will omit Δ when it is clear from context. We can define a term $\Gamma \vdash_\perp \mathsf{packed}(\Gamma) : [\Gamma]$ to "pack" up the current context into a record as follows:

$$\mathsf{packed}(\cdot) = () \qquad \mathsf{packed}(\Gamma, x : A) = (\mathsf{packed}(\Gamma), x) \tag{83}$$

We may now define packing and unpacking substitutions $\mathsf{pack}^\otimes_y(\Gamma) : \Gamma \mapsto y : \langle \Gamma \rangle$, $\mathsf{unpack}^\otimes_y(\Gamma) : y : \langle \Gamma \rangle \mapsto \Gamma$ as follows:

$$\mathsf{pack}^\otimes_y(\Gamma) = y \mapsto \mathsf{packed}(\Gamma) \qquad \mathsf{unpack}^\otimes_y(\Gamma) = (x \mapsto \pi_{\Gamma, x}\ y)_{x \in \Gamma} \tag{84}$$

Our equational theory is sufficient to show that the pack and unpack substitutions are inverses of each other:

$$\begin{aligned} [\mathsf{unpack}^\otimes_y(\Gamma)]\mathsf{pack}^\otimes_y(\Gamma) &\approx \mathsf{id}_\Gamma : \Gamma \mapsto \Gamma \\ [\mathsf{pack}^\otimes_y(\Gamma)]\mathsf{unpack}^\otimes_y(\Gamma) &\approx \mathsf{id}_{y:[\Gamma]} : y : \Gamma \mapsto y : \Gamma \end{aligned} \tag{85}$$

Similarly, we may define a "packing" operation $\langle \cdot \rangle$ on label-contexts as follows:

$$\langle \cdot \rangle = \mathbf{0} \qquad \langle \mathsf{L}, \ell(A) \rangle = \langle \mathsf{L} \rangle + A \tag{86}$$

We may then define injections on enums as follows:

$$\iota_{(\mathsf{L}, \ell(A)), \kappa}\ e = \iota_{\mathsf{L}, \kappa}(\iota_l\ e) \qquad \iota_{(\mathsf{L}, \ell(A)), \ell}\ e = \iota_r\ e \tag{87}$$

with typing rule

$$\frac{\Gamma \vdash_\epsilon e : A \quad \mathsf{L}(\ell) = A}{\Gamma \vdash_\epsilon \iota_{\mathsf{L}, \ell}\ e : \langle \mathsf{L} \rangle}\ \iota_{\mathsf{enum}} \tag{88}$$

Similarly, we can define n-ary case-statements on enums by induction as follows:

$$\begin{aligned} \mathsf{case}.\ e\ \{\} &= \infty \\ \mathsf{case}_{\mathsf{L}, \kappa(A)}\ e\ \{(\ell_i(x_i) : t_i)_i, \kappa(y) : s\} &= \mathsf{case}\ e\ \{\iota_l\ x : \mathsf{case}_{\mathsf{L}}\ x\ \{(\ell_i(A_i) : t_i)_i\}, \iota_r\ y : s\} \end{aligned} \tag{89}$$

with typing rule

$$\frac{\Gamma \vdash_\epsilon e : [\mathsf{L}] \quad \forall i. \Gamma, x_i : A_i \vdash t_i \triangleright \mathsf{K}}{\Gamma \vdash \mathsf{case}_{\mathsf{L}}\ e\ \{(\ell_i(x_i) : t_i)_i\} \triangleright \mathsf{K}}\ \mathsf{case}_{\mathsf{enum}} \tag{90}$$

where

$$\frac{}{\Gamma \vdash \infty := \mathsf{br}\ \ell\ ()\ \mathsf{where}\ \ell(x) : \{\mathsf{br}\ \ell\ x\} \rhd \mathsf{L}}\ \infty \tag{91}$$

We may now define "packing" and "unpacking" label-substitutions $\cdot \vdash \mathsf{pack}^+_\kappa(\mathsf{L}) : \mathsf{L} \rightsquigarrow \kappa([\mathsf{L}])$, $\cdot \vdash \mathsf{unpack}^+_\kappa(\mathsf{L}) : \kappa([\mathsf{L}]) \rightsquigarrow \mathsf{L}$ as follows:

$$\mathsf{pack}^+_\kappa(\cdot) = \cdot \qquad \mathsf{pack}^+_\kappa(\mathsf{L}, \ell(A)) = [\kappa(x) \mapsto \mathsf{br}\ \kappa\ \iota_l\ x]\mathsf{pack}^+_\kappa(\mathsf{L}), \ell(x) \mapsto \mathsf{br}\ \kappa\ \iota_r\ x \tag{92}$$

$$\mathsf{unpack}^+_\kappa(\mathsf{L}) = \kappa(x) \mapsto \mathsf{unpack}^+(\mathsf{L})\ x \tag{93}$$

where we have

$$\frac{\Gamma \vdash_\epsilon e : \mathsf{L}}{\Gamma \vdash \mathsf{unpack}^+(\mathsf{L})\ e \rhd \mathsf{L}}\ \mathsf{unpack}^+ \tag{94}$$

defined as follows:

$$\mathsf{unpack}^+(\cdot)\ e = \infty \qquad \mathsf{unpack}^+(\mathsf{L}, \ell(A))\ e = \mathsf{case}\ e\ \{\iota_l\ x : \mathsf{unpack}^+(\mathsf{L})\ x, \iota_r\ y : \mathsf{br}\ \ell\ y\} \tag{95}$$

We can further show that, in this case for all label contexts L, these substitutions are mutually inverse, i.e., that

$$\begin{aligned} &\Gamma \vdash [\mathsf{unpack}^+_\kappa(\mathsf{L})]\mathsf{pack}^+_\kappa(\mathsf{L}) \approx \mathsf{id}_\mathsf{L} : \mathsf{L} \rightsquigarrow \mathsf{L} \\ &\Gamma \vdash [\mathsf{pack}^+_\kappa(\mathsf{L})]\mathsf{unpack}^+_\kappa(\mathsf{L}) \approx \mathsf{id}_{\kappa([\mathsf{L}])} : \kappa([\mathsf{L}]) \rightsquigarrow \kappa([\mathsf{L}]) \end{aligned} \tag{96}$$

Finally, fixing a distinguished variable $\square$, it will be very useful to define a "*variable packing*" operation on expressions and regions as follows:

$$\Gamma \vdash r \rhd \mathsf{L} \implies \square : [\Gamma] \vdash (\langle r \rangle^\otimes := [\mathsf{unpack}^\otimes_\square(\Gamma)]r) \rhd \mathsf{L} \tag{97}$$

In particular, for Γ pure, the packing operation $[\cdot]$ is an injection on expressions and regions w.r.t. the equational theory, since $\mathsf{unpack}^\otimes_\square(\Gamma)$ has an inverse. Similarly, fixing a distinguished label $\blacksquare$, we may define a "label packing" operation on regions as follows:

$$\Gamma \vdash r \rhd \mathsf{L} \implies \Gamma \vdash \langle r \rangle^+ := [\mathsf{pack}^+_\blacksquare(\mathsf{L})]r \rhd \blacksquare([\mathsf{L}]) \tag{98}$$

Since the packing substitution has an inverse, it similarly follows that label packing is an injection w.r.t. the equational theory, i.e.

$$\Gamma \vdash r \approx r' \rhd \mathsf{L} \iff \Gamma \vdash \langle r \rangle^+ \approx \langle r' \rangle^+ \rhd \mathsf{L} \tag{99}$$

Finally, we can define a "packing" operation on regions to be given by label-packing followed by variable-packing, or vice versa (since it turns out the operations commute), as follows:

$$\Gamma \vdash r \rhd \mathsf{L} \implies \square : [\Gamma] \vdash \langle r \rangle := \langle \langle r \rangle^+ \rangle^\otimes = \langle \langle r \rangle^\otimes \rangle^+ \rhd \blacksquare(\mathsf{L}) \tag{100}$$

We similarly have that this is an injection for Γ pure, i.e.

$$\Gamma \vdash r \approx r' \rhd \mathsf{L} \iff \Gamma \vdash \langle r \rangle \approx \langle r' \rangle \rhd \mathsf{L} \tag{101}$$

A.1 Böhm-Jacopini for SSA

Now that we have given our equational theory, we want to show it is "good enough" to reason about the properties inherent to all SSA-based languages. In particular, we wish to show that we have enough power to reason about interconversion between data-flow and control-flow. For example, a state machine can be implemented either as a switch on a state value, or as a set of mutually-tail-recursive functions (i.e., the state can be encoded in the program counter). We demonstrate this by using some of the machinery from the previous section to state and prove a form of the Böhm-Jacopini theorem [11] for SSA.

The Böhm-Jacopini theorem states that every general control-flow graph program can be rewritten to an equivalent program which uses only structured control-flow: i.e., it can be rewritten to a

program using only conditional branching, sequencing, and loops. To adapt this result to SSA, we need to express branching, sequencing and loops as SSA regions $\Gamma \vdash r \rhd \mathsf{L}$, so that we can build up an inductive set of structured regions. We will be maximally strict, and allow branching to an exit label in L only as the terminal statement in a structured program (and, in particular, not from within a loop!). It is obvious that we can represent conditional branching using case-statements, but we need convenient primitives for sequencing and looping. Sequencing can be expressed using a where-block as follows (with ℓ fresh):

$$\frac{\Gamma \vdash r \rhd \blacksquare(A) \quad \Gamma, \square : A \vdash s \rhd \mathsf{L}}{\Gamma \vdash \mathsf{seq}(r,s) := ([\blacksquare(x) \mapsto \mathsf{br}\ \ell\ x]r \text{ where } \ell(\square) : \{s\}) \rhd \mathsf{L}}\ \mathsf{seq} \tag{102}$$

where "$\square$" is a distinguished input variable, and "$\blacksquare$" is a distinguished output label. Another, equivalent, way of writing sequencing is:

$$\Gamma \vdash \mathsf{seq}(r,s) \approx [\blacksquare(\square) \mapsto s]r \rhd \mathsf{L} \tag{103}$$

That is, when we exit r, we jump to (a copy of) s.

Expressing structured looping is a bit more complicated, since we do not have mutable variables and hence cannot directly express while loops. If we recall that loops can be expressed as tail-recursive procedures which carry the loop state in the argument, then we can define a "functional do-while loop" as follows:

$$\frac{\Gamma \vdash_\epsilon e : A \quad \Gamma, \square : A \vdash r \rhd \blacksquare(B + A)}{\Gamma \vdash \mathsf{loop}(e,r) := (\mathsf{br}\ \ell\ e \text{ where } \ell(\square) : \{\mathsf{seq}(r, \mathsf{case}\ \square\ \{\iota_l\ x : \mathsf{br}\ \blacksquare\ x, \iota_r\ y : \mathsf{br}\ \ell\ y\})\}) \rhd \blacksquare(B)}\ \mathsf{loop} \tag{104}$$

We can now define the inductive predicate $\Gamma \vdash^s r \rhd \mathsf{L}$, which says that r is a structured region with input variables Γ and output labels L, as in Figure 34. Note that this defines a subset of the well-typed regions: every region r such that $\Gamma \vdash^s r \rhd \mathsf{L}$ is also well-typed as $\Gamma \vdash r \rhd \mathsf{L}$.

It now remains to give an algorithm to convert a region r targeting label-context L into an equivalent structured region $\mathsf{WH}_\mathsf{L}(r)$. In particular, we define

$$\mathsf{WH}_\mathsf{L}(r) = [\mathsf{unpack}^+_\blacksquare(\mathsf{L})]\mathsf{PW}_L(r) \tag{105}$$

where we define

$$\begin{aligned}
\mathsf{PW}_\mathsf{L}(\mathsf{br}\ \ell\ a) &= \mathsf{br}\ \blacksquare\ \iota_{\mathsf{L},\ell}\ a \\
\mathsf{PW}_\mathsf{L}(\mathsf{let}\ x = a; r) &= \mathsf{let}\ x = a; \mathsf{PW}_\mathsf{L}(r) \\
\mathsf{PW}_\mathsf{L}(\mathsf{let}\ (x,y) = a; r) &= \mathsf{let}\ (x,y) = a; \mathsf{PW}_\mathsf{L}(r) \\
\mathsf{PW}_\mathsf{L}(\mathsf{case}\ e\ \{\iota_l\ x : r, \iota_r\ y : s\}) &= \mathsf{case}\ e\ \{\iota_l\ x : \mathsf{PW}_\mathsf{L}(r), \iota_r\ y : \mathsf{PW}_\mathsf{L}(s)\} \\
\mathsf{PW}_\mathsf{L}(r \text{ where } (\ell_i(x_i) : \{t_i\},)_i) &= \mathsf{seq}(\mathsf{PW}_\mathsf{L}(r), \mathsf{case}\ \mathsf{ua}\ \square\ \{\iota_l\ x : \mathsf{br}\ \blacksquare\ x \\
&\qquad , \iota_r\ y : \mathsf{loop}(y, \mathsf{case}_\mathsf{R}\ \square\ \{\ell_i(x_i) : \mathsf{seq}(\mathsf{PW}_\mathsf{L}(t_i), \mathsf{br}\ \blacksquare\ (\mathsf{ua}\ \square)\})\})\} \\
\text{where} \quad \mathsf{R} &= (\ell_i(A_i),)
\end{aligned} \tag{106}$$

and

$$\mathsf{ua}_{\mathsf{L},\cdot}\ e = e \qquad \mathsf{ua}_{\mathsf{L},(\mathsf{R},\ell(A))}\ e = \mathsf{case}\ e\ \{\iota_l\ x : \mathsf{case}\ \mathsf{ua}_{\mathsf{L},\mathsf{R}}\ x\ \{\iota_l\ z : \iota_l\ z, \iota_r\ w : \iota_r\ \iota_l\ w\}, \iota_r\ y : \iota_r\ (\iota_r\ y)\} \tag{107}$$

Note that the base case $\mathsf{PW}_\mathsf{L}(\mathsf{br}\ \ell\ a)$ is The $\mathsf{PW}_\mathsf{L}(r)$ function does the actual transformation, and we can see that most of the cases are trivial except for:

- The base case, in which we simply replace a branch $\mathsf{br}\ \ell\ a$ with a branch to the output label $\blacksquare$ with the appropriate injection $\iota_{\mathsf{L},\ell}\ a$ as parameter.

$$\frac{\Gamma \vdash_\perp a : A \quad \mathsf{L}\ \ell = A}{\Gamma \vdash^{\mathrm{s}} \mathsf{br}\ \ell\ a \rhd \mathsf{L}}\ \text{s-br} \qquad \frac{\Gamma \vdash_\epsilon a : A \quad \Gamma, x : A \vdash^{\mathrm{s}} r \rhd \mathsf{L}}{\Gamma \vdash^{\mathrm{s}} \mathsf{let}\ x = a; r \rhd \mathsf{L}}\ \text{s-let}_1\text{-r}$$

$$\frac{\Gamma \vdash_\epsilon e : A \otimes B \quad \Gamma, x : A, y : B \vdash^{\mathrm{s}} r \rhd \mathsf{L}}{\Gamma \vdash^{\mathrm{s}} \mathsf{let}\ (x, y) = e; r \rhd \mathsf{L}}\ \text{s-let}_2\text{-r}$$

$$\frac{\Gamma \vdash_\epsilon e : A + B \quad \Gamma, x : A \vdash^{\mathrm{s}} r \rhd \mathsf{L} \quad \Gamma, y : B \vdash^{\mathrm{s}} s \rhd \mathsf{L}}{\Gamma \vdash^{\mathrm{s}} \mathsf{case}\ e\ \{\iota_l\ x : r, \iota_r\ y : s\} \rhd \mathsf{L}}\ \text{s-case-r}$$

$$\frac{\Gamma \vdash^{\mathrm{s}} r \rhd \blacksquare(A) \quad \Gamma, \square : A \vdash^{\mathrm{s}} s \rhd \mathsf{L}}{\Gamma \vdash^{\mathrm{s}} \mathsf{seq}(r, s) \rhd \mathsf{L}}\ \text{s-seq} \qquad \frac{\Gamma \vdash_\epsilon e : \blacksquare(A) \quad \Gamma, \square : A \vdash^{\mathrm{s}} r \rhd \blacksquare(B + A)}{\Gamma \vdash^{\mathrm{s}} \mathsf{loop}(e, r) \rhd \blacksquare(B)}\ \text{s-loop}$$

Fig. 34. Typing rules for structured regions

- The r where $(\ell_i(x_i) : \{t_i\},)_i$ case, in which we replace the set of labels bound by the where clause with a tag and a while loop containing a case statement branching on the tag. (This uses the ua function, which implements associativity of n-ary coproducts. That is, given $\Gamma \vdash_\perp e : [L, R]$, we have that $\Gamma \vdash_\perp \mathsf{ua}_{\mathsf{L}}\ e : [L] + [R]$.)

The Böhm-Jacopini theorem for SSA can then be written:

Theorem A.1 (Böhm-Jacopini for SSA). *For all* $\Gamma \vdash r \rhd \mathsf{L}$, *we have that* $\Gamma \vdash^{\mathrm{s}} \mathsf{WH}_{\mathsf{L}}(r) \rhd \mathsf{L}$ *is structured, and* $\Gamma \vdash r \approx \mathsf{WH}_{\mathsf{L}}(r) \rhd \mathsf{L}$. *In particular, we have that* $\Gamma \vdash^{\mathrm{s}} \mathsf{PW}_{\mathsf{L}}(r) \rhd \blacksquare(\langle \mathsf{L} \rangle)$ *is structured, and* $\Gamma \vdash \langle r \rangle^+ \approx \mathsf{PW}_r(\rhd)\blacksquare(\langle \mathsf{L} \rangle)$.

Proof. See Appendix C.3 □

B The Environment Comonad

If C is a Freyd category, then given an object $R \in |C|$, the functor $R \otimes \cdot$ is a comonad with counit $\pi_r : R \otimes A \to A$ and comultiplication $\Delta_R \otimes A; \alpha : R \otimes A \to R \otimes (R \otimes A)$, often called the *environment comonad* or *coreader comonad*. In Set, the co-Kleisli category of $R \otimes \cdot$ is isomorphic to the Kleisli category of the reader monad $R \to \cdot$, and thus can be equipped with the structure of a distributive Elgot category. We might therefore intuit that, in general, if C is a distributive Elgot category, the co-Kleisli category of $R \otimes \cdot$ has this structure, even if e.g. C lacks exponentials. The rest of this section is dedicated to proving this, which, beyond providing yet another class of λ_{SSA} models, will also give us a some useful equations and notation to use in our proofs later in the appendix. We begin by giving an explicit definition:

Definition B.1 (Environment Comonad). Given a Freyd category C and an object (the *environment*) $R \in |C|$, the *environment comonad* $C_{R\otimes\cdot}$ is given by the functor $A \mapsto R \otimes A$ and has counit π_r and comultiplication $\Delta_R \otimes A$. It follows that composition of co-Kleisli morphisms $f : R \otimes A \to B$, $g : R \otimes B \to C$ is given by

$$f \gg_R g := \Delta_R; \alpha; R \otimes f; g : R \otimes A \to C \tag{108}$$

In particular, we can define an identity on objects functor $\mathsf{env}_R : C \to C_{R\otimes\cdot}$ as follows:

$$\mathsf{env}_R(f) := \pi_r; f \tag{109}$$

Where it is clear from context, we will leave out the environment R.

Given $f : R \otimes A \to B$, we will introduce the syntax sugar

$$\mathsf{rlet}(f) := \Delta_R; \alpha; R \otimes f : R \otimes A \to R \otimes B \tag{110}$$

In particular, we have that $f \gg g = \mathsf{rlet}(f); g$ and $\mathsf{rlet}(\mathsf{env}(f)) := R \otimes f$. It is trivial to verify (for example, by drawing string diagrams) that

$$\begin{aligned} &\mathsf{rlet}(\mathsf{rlet}(f); g) = \mathsf{rlet}(f); \mathsf{rlet}(g) \implies f \gg (g \gg h) = (f \gg g) \gg h \\ &\mathsf{rlet}(f); \pi_r = f \implies f \gg \pi_r = f \qquad \mathsf{rlet}(\pi_r) = \mathsf{id} \implies \pi_r \gg f = f \end{aligned} \tag{111}$$

and hence that $C_{R\otimes\cdot}$ is indeed a category. To show it is in fact a *Freyd category*, we can define tensor functors

$$\begin{aligned} f \otimes_R X &:= \alpha^{-1}; f \otimes X : R \otimes (A \otimes X) \to B \otimes X \\ X \otimes_R f &:= R \otimes \sigma; f \otimes_R X; \sigma : R \otimes (X \otimes A) \to X \otimes B \end{aligned} \tag{112}$$

We can define

- Associators $\alpha^R_{A,B,C} := \mathsf{env}_R(\alpha_{A,B,C})$
- Unitors $\lambda^R_A := \mathsf{env}_R(\lambda_A)$ and $\rho^R_A := \mathsf{env}_R(\rho_A)$
- Symmetries $\sigma^R_{A,B} := \mathsf{env}_R(\sigma_{A,B})$
- Projections $\pi^R_l := \mathsf{env}_R(\pi_l)$ and $\pi^R_r := \mathsf{env}_R(\pi_r)$
- Terminal morphisms $!^R_A := \mathsf{env}_R(!_A) = !_{R\otimes A}$
- Diagonals $\Delta^R_A := \mathsf{env}_R(\Delta_A)$
- Pure morphisms $C_{R\otimes\cdot\perp}(A, B) = C_\perp(R \otimes A, B)$

Lemma B.2. *We can always equip $C_{R\otimes}$ with the structure of a Freyd category as described above ; furthermore, $\mathsf{env}_R(\cdot)$ strictly preserves premonoidal structure.*

Proof. We have that

$$\begin{aligned} \mathsf{env}_R(f \otimes A) &= \pi_r; f \otimes A = \alpha^{-1}; (\pi_r; f) \otimes A = \mathsf{env}_R(f) \otimes_R A \\ \mathsf{env}_R(A \otimes f) &= \pi_r; A \otimes f = R \otimes \sigma; \alpha^{-1}; (\pi_r; f) \otimes A; \sigma = A \otimes_R \mathsf{env}_R(f) \end{aligned} \tag{113}$$

and hence that env_R preserves products. Since env_R is a functor, to check we indeed have a Freyd category, it suffices to show that:

- $\alpha^R_{A,B,C}$ is natural: we have that, given $f : R \otimes A \to A'$, $g : R \otimes B \to B'$, $h : R \otimes C \to C'$

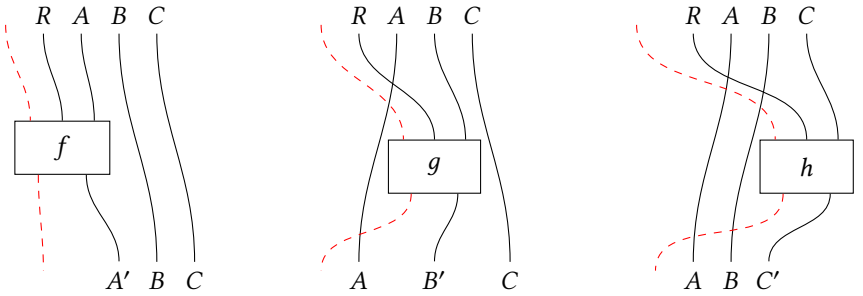
$$(f \otimes_R B) \otimes_R C \gg \alpha_{A',B,C} = \alpha_{A,B,C} \gg f \otimes_R (B \otimes C) \tag{114}$$
$$(A \otimes_R g) \otimes C \gg \alpha_{A,B',C} = \alpha_{A,B,C} \gg A \otimes_R (g \otimes_R C) \tag{115}$$
$$(A \otimes B) \otimes_R h \gg \alpha_{A,B,C'} = \alpha_{A,B,C} \gg A \otimes_R (B \otimes_R h) \tag{116}$$
$$\tag{117}$$

  since, for each equation, both sides are equal to the same string diagram in Figure 35a, 35b, and 35c respectively.
- λ^R_A is natural: we have that, for $f : R \otimes A \to A'$,

$$\begin{aligned} f \otimes_R \mathbf{1} \gg \lambda^R_{A'} &= \Delta_R; \alpha; R \otimes (\alpha^{-1}; f \otimes \mathbf{1}); \pi_r; \pi_l \\ &= R \otimes \pi_l; f \\ &= \Delta_R; \alpha; R \otimes (\pi_r; \pi_l); f \qquad = \lambda^R_A \gg f \end{aligned} \tag{118}$$

  In general, we note that, for all $g : R \otimes A \to A' \otimes \mathbf{1}$, $g \gg \lambda^R = g; \pi_l$.
- ρ^R_A is natural:

$$\begin{aligned} \mathbf{1} \otimes_R f \gg \rho^R_{A'} &= \Delta_R \otimes (\mathbf{1} \otimes A); \alpha; R \otimes (R \otimes \sigma; \alpha^{-1}; f \otimes \mathbf{1}; \sigma); \pi_r; \pi_r \\ &= R \otimes \pi_r; f \\ &= \Delta_R; \alpha; R \otimes (\pi_r; \pi_r); f \qquad = \rho^R_A \gg f \end{aligned} \tag{119}$$



(a) Equation 114 (b) Equation 115 (c) Equation 116

Fig. 35. Naturality of the associator in the co-Kleisli category of the environment comonad

In general, we note that, for all $g : R \otimes A \to \mathbf{1} \otimes A'$, $g \gg \rho^R = g; \pi_r$.

- $\sigma^R_{A,B}$ is natural: given $f : R \otimes A \to A'$ and $g : R \otimes B \to B'$, we have that

$$\begin{aligned} f \otimes_R B \gg \sigma^R_{A',B} &= \Delta_R \otimes (A \otimes B); \alpha; R \otimes (\alpha^{-1}; f \otimes B); \pi_r; \sigma \\ &= \alpha^{-1}; f \otimes B; \sigma \\ &= R \otimes \sigma; R \otimes \sigma; f \otimes B; \sigma \\ &= \Delta_R \otimes (B \otimes A); R \otimes (\pi_r; \sigma); R \otimes \sigma; \alpha^{-1}; f \otimes B; \sigma \quad = \sigma^R_{A,B} \gg B \otimes_R f \end{aligned} \tag{120}$$

and

$$\begin{aligned} A \otimes g \gg \sigma^R_{A,B'} &= \Delta_R \otimes (A \otimes B); \alpha; R \otimes (R \otimes \sigma; \alpha^{-1}; g \otimes A; \sigma); \pi_r; \sigma \\ &= R \otimes \sigma; \alpha^{-1}; g \otimes A; \cancel{\sigma; \sigma} \\ &= \Delta_R \otimes (A \otimes B); \alpha; R \otimes (\pi_r; \sigma); \alpha^{-1}; g \otimes A \qquad = \sigma^R_{A,B} \gg g \otimes_R A \end{aligned} \tag{121}$$

- Pure morphisms are central: given $f : R \otimes A \to A'$ and $g : R \otimes B \to B'$ pure, we have that

$$\begin{aligned} f \otimes_R B \gg A' \otimes_R g &= \Delta_R; \alpha; R \otimes (\alpha^{-1}; f \otimes B; \sigma); \alpha^{-1}; g \otimes A'; \sigma \\ &= \Delta_R; \alpha; R \otimes (R \otimes \sigma; \alpha^{-1}; g \otimes A; \sigma); \alpha^{-1}; f \otimes B' \\ &= A \otimes_R g \gg f \otimes_R B' \end{aligned} \tag{122}$$

since both sides correspond to the diagram in Figure 36. We will write this as $f \otimes_R g$.

- $!^R_A$ is terminal for pure morphisms: yes, since $!^R_A$ is just $!_{R \otimes A}$ which is terminal in C
- Δ^R_A duplicates pure morphisms: we have

$$\begin{aligned} f \gg \Delta^R_B &= \Delta_R; \alpha; R \otimes f; \pi_r; \Delta_{A'} \\ &= f; \Delta_{A'} = \Delta_{R \otimes A}; f \otimes f \\ &= \Delta_R; \alpha; R \otimes (\alpha^{-1}; f \otimes A'); R \otimes \sigma; \alpha^{-1}; f \otimes A'; \sigma \quad = \Delta^R_A \gg f \otimes_R f \end{aligned} \tag{123}$$

□

Now, assume C is distributive. We wish to show that $A + B$ is a coproduct in $C_{R\otimes\cdot}$, and furthermore that $C_{R\otimes\cdot}$ is distributive; note that even the former may not be the case without distributivity! We proceed as follows:

Lemma B.3. *If C is a distributive Freyd category, then $C_{R\otimes\cdot}$ is also distributive Freyd*

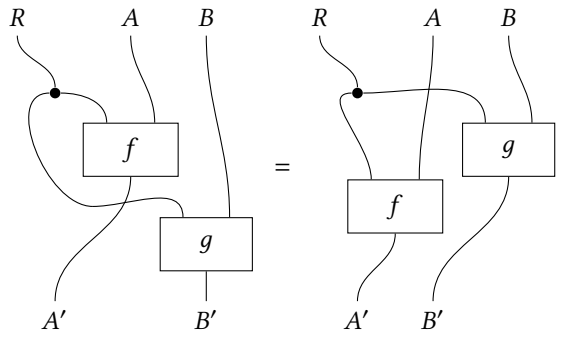


Fig. 36. Centrality of pure morphisms in the environment comonad's co-Kleisli category

Proof. Given $f : R \otimes A \to B$ and $g : R \otimes A \to C$, we may define the coproduct and injections

$$[f,g]_R := \delta^{-1}; [f,g] : R \otimes A \to B + C \qquad \iota_l^R := \pi_r; \iota_r : R \otimes A \to A + B \qquad \iota_r^R := \pi_r; \iota_r : R \otimes B \to A + B \tag{124}$$

To verify this is indeed a coproduct, we can check that

$$\begin{aligned} \iota_l^R \gg [f,g]_R &= \Delta_R; \alpha; R \otimes (\pi_r; \iota_r); \delta^{-1}; [f,g] = R \otimes \iota_r; \delta^{-1}; [f,g] = f \\ \iota_r^R \gg [f,g]_R &= \Delta_R; \alpha; R \otimes (\pi_r; \iota_l); \delta^{-1}; [f,g] = R \otimes \iota_l; \delta^{-1}; [f,g] = g \end{aligned} \tag{125}$$

This morphism is obviously unique, since we have for all $h : R \otimes (A + B) \to C$

$$[\iota_l^R \gg h, \iota_r^R \gg h]_R = \delta^{-1}; [\Delta_R; \alpha; R \otimes \iota_l; \pi_r; h, \Delta_R; \alpha; R \otimes \iota_r; \pi_r; h] = \cancel{\delta^{-1}; [R \otimes \iota_r, R \otimes \iota_l]}; h \tag{126}$$

To show that $R_{C\otimes\cdot}$ is indeed distributive, we need $\delta^{R-1} := \mathsf{env}_R(\delta^{-1}) = \pi_r; \delta^R$ to be the inverse to $\delta^R := [A \otimes \iota_l^R, A \otimes \iota_r^R]_R$, which can easily be derived from the functoriality of $\mathsf{env}_R(\cdot)$, since

$$\mathsf{env}_R(\delta) = \pi_r; [\iota_l; \iota_r] = \delta^{-1}; [\pi_r; \iota_l, \pi_r \iota_r] = \delta^R \tag{127}$$

□

Note in particular that this allows us to define sums

$$\begin{aligned} f +_R g &= [f \gg \iota_l^R, g \gg \iota_r^R]_R \\ &= \delta^{-1}; [\Delta_R \otimes (A + B); \alpha; R \otimes f; \pi_r; \iota_l, \Delta_R \otimes (A + B); \alpha; R \otimes g; \pi_r; \iota_r] \quad = \delta^{-1}; f + g \end{aligned} \tag{128}$$

Our final task is to show that, if C is a strong Elgot category, then so is $R_{C\otimes\cdot}$, and, in particular, with fixpoint operator $\mathsf{rfix}(f)$. Note that, just like we needed distributivity to have coproducts at all, we will need strength to have an Elgot structure. We begin by stating some generally useful properties of $\mathsf{rcase}(f)$ and $\mathsf{rfix}(f)$. In particular, we have that: For $f : R \otimes A \to B + C$:

- Given $h : R \otimes X \to A$, we have

$$\begin{aligned} \mathsf{rlet}(h); \mathsf{rcase}(f) &= \Delta_R \otimes X; \alpha; R \otimes h; \Delta_R \otimes A; \alpha; R \otimes f; \delta^{-1} \\ &= \Delta_R \otimes X; \alpha; R \otimes (\Delta_R \otimes X; \alpha; R \otimes h; f); \delta^{-1} \\ &= \mathsf{rcase}(\mathsf{rlet}(h); f) \qquad\qquad = \mathsf{rcase}(h \gg f) \end{aligned} \tag{129}$$

- Given $g : R \otimes B \to X$, $h : R \otimes C \to Y$, we have

$$\begin{aligned}
\mathsf{rcase}(f \gg g +_R h) &= \mathsf{rlet}(f \gg g +_R h); \delta^{-1} \\
&= \mathsf{rlet}(f); \mathsf{rlet}(g +_R h); \delta^{-1} \\
&= \mathsf{rlet}(f); \Delta_R \otimes (B + C); \alpha; R \otimes (\delta^{-1}; g + h); \delta^{-1} \\
&= \mathsf{rlet}(f); \Delta_R \otimes (B + C); \alpha; R \otimes \delta^{-1}; \delta^{-1}; (R \otimes g) + (R \otimes h) \\
&= \mathsf{rlet}(f); \delta^{-1}; \mathsf{rlet}(g) + \mathsf{rlet}(h) \\
&= \mathsf{rcase}(f); \mathsf{rlet}(g) + \mathsf{rlet}(h) \\
&= \mathsf{rlet}(f); \mathsf{rlet}(g) +_R \mathsf{rlet}(h)
\end{aligned} \tag{130}$$

Similarly, for $f : R \otimes A \to B + A$:

- Given $h : R' \to_\perp R$, $h \otimes A; \mathsf{rfix}(f) = \mathsf{rfix}(h \otimes A; f)$: we have that

$$\begin{aligned}
h \otimes A; \mathsf{rcase}(f) &= h \otimes A; \Delta_R \otimes A; \alpha; R \otimes f; \delta^{-1} \\
&= \Delta_{R'} \otimes A; \alpha; h \otimes (h \otimes A; f); \delta^{-1} \\
&= \Delta_{R'} \otimes A; \alpha; R \otimes (h \otimes A; f); \delta^{-1}; h \otimes B + h \otimes A \\
&= \mathsf{rcase}(h \otimes A; f); h \otimes B + h \otimes A
\end{aligned} \tag{131}$$

  and hence by uniformity that

$$\begin{aligned}
h \otimes A; \mathsf{rfix}(f) &= h \otimes A; (\mathsf{rcase}(f))^\dagger; \pi_r \\
&= (\mathsf{rcase}(h \otimes A; f); h \otimes B + R' \otimes A)^\dagger; \pi_r \\
&= (\mathsf{rcase}(h \otimes A; f))^\dagger; h \otimes B; \pi_r \qquad = \mathsf{rfix}(h \otimes A; f)
\end{aligned} \tag{132}$$

- $\mathsf{rlet}(\mathsf{rfix}(f)) = (\mathsf{rcase}(f))^\dagger$: we have that

$$\begin{aligned}
&(\Delta_R \otimes A; \alpha); (R \otimes \mathsf{rcase}(f); \delta^{-1}; R \otimes \pi_r + R \otimes (R \otimes A)) \\
&= \Delta_R \otimes A; \alpha; R \otimes (\mathsf{let}(f); \pi_l \otimes (B + A); \delta^{-1}); \delta^{-1}; R \otimes \pi_r + R \otimes (R \otimes A) \\
&= \Delta_R \otimes A; \alpha; R \otimes f; \Delta_R \otimes (B + A); \alpha; R \otimes \delta^{-1}; \delta^{-1}; R \otimes \pi_r + R \otimes (R \otimes A) \\
&= \Delta_R \otimes A; \alpha; R \otimes f; \delta^{-1}; (\Delta_R \otimes A; \alpha; R \otimes \pi_r) + (\Delta_R \otimes A; \alpha) \\
&= \Delta_{R \otimes A}; (R \otimes A) \otimes f; \pi_l \otimes (B + A); \delta^{-1}; R \otimes A + (\Delta_R \otimes A; \alpha) \\
&= \mathsf{rcase}(f); R \otimes A + (\Delta_R \otimes A; \alpha)
\end{aligned} \tag{133}$$

  It follows by uniformity and strength that

$$\begin{aligned}
\mathsf{rlet}(\mathsf{rfix}(f)) = \mathsf{let}(\mathsf{rfix}(f)); \pi_l \otimes B &= \Delta_{R \otimes A}; (R \otimes A) \otimes \mathsf{rfix}(f); \pi_l \otimes B \\
&= \Delta_R \otimes A; \alpha; R \otimes \mathsf{rfix}(f) \\
&= \Delta_R \otimes A; \alpha; R \otimes ((\mathsf{rcase}(f))^\dagger; \pi_r) \\
&= \Delta_R \otimes A; \alpha; (R \otimes \mathsf{rcase}(f); \delta^{-1}; R \otimes \pi_r + R \otimes (R \otimes A))^\dagger \\
&= (\mathsf{rcase}(f))^\dagger
\end{aligned} \tag{134}$$

  as desired.

We also state some generally useful properties of $\mathsf{rlet}(f)$:

- For all $f : R \otimes A \to C$, $g : R \otimes B \to C$,

$$\begin{aligned} \mathsf{rlet}(\delta^{-1}; [f, g]) &= \Delta_R \otimes (A + B); \alpha; R \otimes (\delta^{-1}; [f, g]) \\ &= \delta^{-1}; [\Delta_R \otimes A; \alpha; f, \Delta_R \otimes B; \alpha; g] \\ &= \delta^{-1}; [\mathsf{rlet}(f), \mathsf{rlet}(g)] \end{aligned} \tag{135}$$

We may now state our desired result as follows:

Lemma B.4. *If C is a premonoidal strong Elgot category, then so is $C_{R\otimes}$. with fixpoint operator* $\mathsf{rfix}(f)$.

Proof. We check each of the Elgot axioms as follows:

- *Fixpoint:* given $f : R \otimes A \to B + A$, we have that

$$\begin{aligned} \mathsf{rfix}(f) &= (\mathsf{rcase}(f))^\dagger; \pi_r = (\mathsf{rlet}(f); \delta^{-1})^\dagger; \pi_r \\ &= \mathsf{rlet}(f); \delta^{-1}; [\mathsf{id}_{R\otimes B}, (\mathsf{rlet}(f); \delta^{-1})^\dagger]; \pi_r \\ &= \mathsf{rlet}(f); \delta^{-1}; [\pi_r, (\mathsf{rcase}(f))^\dagger; \pi_r] \qquad = f \gg [\pi_r, \mathsf{rfix}(f)]_R \end{aligned} \tag{136}$$

  as desired.
- *Naturality:* given $f : R \otimes A \to B + A$, $g : R \otimes B \to C$, we have that

$$\begin{aligned} \mathsf{rfix}(f \gg g +_R A) &= (\mathsf{rlet}(\Delta_R \otimes A; \alpha; R \otimes f; \delta^{-1}; g + \pi_r); \delta^{-1})^\dagger; \pi_r \\ &= (\mathsf{rlet}(\Delta_R \otimes A; \alpha; R \otimes f; \delta^{-1}); \delta^{-1}; R \otimes g + R \otimes \pi_r)^\dagger; \pi_r \\ &= (\Delta_R \otimes A; \alpha; R \otimes (\Delta_R \otimes A; \alpha; R \otimes f; \delta^{-1}); \delta^{-1}; (\pi_r; g) + R \otimes \pi_r)^\dagger \\ &= (\Delta_R \otimes A; \alpha; R \otimes f; \delta^{-1}; (\Delta_R \otimes B; \alpha; \pi_r; g) + (\Delta_R \otimes A; \alpha; R \otimes \pi_r))^\dagger \\ &= (\Delta_R \otimes A; \alpha; R \otimes f; \delta^{-1}; g + R \otimes A)^\dagger \\ &= (\Delta_R \otimes A; \alpha; R \otimes f; \delta^{-1})^\dagger; g = (\mathsf{rcase}(f))^\dagger; g \end{aligned} \tag{137}$$

  On the other hand, we have that

$$\begin{aligned} \mathsf{rfix}(f) \gg g &= \Delta_R \otimes A; \alpha; R \otimes (\mathsf{rcase}(f); \pi_r + R \otimes A)^\dagger; g \\ &= \Delta_R \otimes A; \alpha; (R \otimes (\mathsf{rcase}(f); \pi_r + R \otimes A); \delta^{-1})^\dagger; g \\ &= \Delta_R \otimes A; \alpha; (R \otimes (\Delta_R \otimes A; \alpha; R \otimes f; \delta^{-1}; \pi_r + R \otimes A); \delta^{-1})^\dagger; g \\ &= \Delta_R \otimes A; \alpha; (R \otimes (\Delta_R \otimes A; \alpha; R \otimes f; \delta^{-1}); \delta^{-1}; R \otimes \pi_r + R \otimes (R \otimes A))^\dagger; g \end{aligned} \tag{138}$$

  By uniformity, it hence suffices to show that

$$\begin{aligned} &(\Delta_R \otimes A; \alpha); (R \otimes (\Delta_R \otimes A; \alpha; R \otimes f; \delta^{-1}); \delta^{-1}; R \otimes \pi_r + R \otimes (R \otimes A)) \\ &= \Delta_R \otimes A; \alpha; R \otimes f; \delta^{-1}; (\Delta_R \otimes B; \alpha; R \otimes \pi_r) + (\Delta_R \otimes A; \alpha) \\ &= \mathsf{rcase}(f); (R \otimes B) + (\Delta_R \otimes A; \alpha) \end{aligned} \tag{139}$$

  to yield the desired result.

- *Codiagonal:* given $f : R \otimes A \to (B + A) + A$, we have

$$\begin{aligned}
&\mathsf{rfix}(\mathsf{rfix}(f)) \\
&= (\Delta_R \otimes A; \alpha\ ; R \otimes ((\mathsf{rcase}(f))^\dagger; \pi_r); \delta^{-1})^\dagger; \pi_r \\
&= (\Delta_R \otimes A; \alpha; R \otimes (\mathsf{rcase}(f); \pi_r + R \otimes A)^\dagger; \delta^{-1})^\dagger; \pi_r \\
&= (\Delta_R \otimes A; \alpha; (R \otimes (\mathsf{rcase}(f); \pi_r + R \otimes A); \delta^{-1})^\dagger; \delta^{-1})^\dagger; \pi_r \\
&= (\Delta_R \otimes A; \alpha; (R \otimes (\mathsf{rcase}(f); \pi_r + R \otimes A); \delta^{-1}; \delta^{-1} + R \otimes (R \otimes A))^\dagger)^\dagger; \pi_r \\
&= (\Delta_R \otimes A; \alpha; (R \otimes (\mathsf{rcase}(f); \pi_r + R \otimes A); \delta^{-1}; \delta^{-1} + R \otimes (R \otimes A))^\dagger; \pi_r + R \otimes A)^\dagger \\
&= (\Delta_R \otimes A; \alpha; (R \otimes (\mathsf{rcase}(f); \pi_r + R \otimes A); \delta^{-1}; (\delta^{-1}; (\pi_r + R \otimes A)) + R \otimes (R \otimes A))^\dagger)^\dagger
\end{aligned} \tag{140}$$

On the other hand, we have

$$\begin{aligned}
\mathsf{rfix}(f \gg [\pi_r, \iota_l^R]_R) &= \mathsf{rfix}(\Delta_R \otimes A; \alpha; R \otimes f; \delta^{-1}; [\pi_r, \pi_r; \iota_l]) \\
&= \mathsf{rfix}(f; [\mathsf{id}, \iota_l]) \\
&= (\Delta_R \otimes A; \alpha; R \otimes (f; [\mathsf{id}, \iota_l]); \delta^{-1})^\dagger; \pi_r \\
&= (\Delta_R \otimes A; \alpha; R \otimes f; \delta^{-1}; \delta^{-1} + R \otimes A; [\mathsf{id}, \iota_l])^\dagger; \pi_r \\
&= ((\Delta_R \otimes A; \alpha; R \otimes f; \delta^{-1}; \delta^{-1} + R \otimes A)^\dagger)^\dagger; \pi_r \\
&= ((\Delta_R \otimes A; \alpha; R \otimes f; \delta^{-1}; \delta^{-1} + R \otimes A)^\dagger; \pi_r + R \otimes A)^\dagger \\
&= ((\Delta_R \otimes A; \alpha; R \otimes f; \delta^{-1}; \delta^{-1} + R \otimes A; (\pi_r + R \otimes A) + R \otimes A)^\dagger)^\dagger
\end{aligned} \tag{141}$$

By uniformity, it therefore suffices to show that

$$\begin{aligned}
&(\Delta_R \otimes A; \alpha); (R \otimes (\mathsf{rcase}(f); \pi_r + R \otimes A); \delta^{-1}; (\delta^{-1}; (\pi_r + R \otimes A)) + R \otimes (R \otimes A)) \\
&= \Delta_R \otimes A; \alpha; R \otimes (\Delta_R \otimes A; \alpha; R \otimes f; \delta^{-1}; \pi_r + R \otimes A); \delta^{-1}; (\delta^{-1}; (\pi_r + R \otimes A)) + R \otimes (R \otimes A) \\
&= \Delta_R \otimes A; \alpha; R \otimes (\Delta_R \otimes A; \alpha; R \otimes f; \delta^{-1}); \\
&\quad \delta^{-1}; R \otimes \pi_r + R \otimes (R \otimes A); (\delta^{-1}; (\pi_r + R \otimes A)) + R \otimes (R \otimes A) \\
&= \Delta_R \otimes A; \alpha; R \otimes f; \delta^{-1}; \delta^{-1} + (R \otimes A); (\pi_r + R \otimes A) + (\Delta_R \otimes A; \alpha)
\end{aligned} \tag{142}$$

to obtain the desired result.

- *Uniformity:* Given $f : R \otimes A \to B + A$, $g : R \otimes X \to B + X$ and $h : R \otimes X \to_\perp A$ such that $h \gg f = g \gg B +_R h$, we have that

$$\begin{aligned}
\mathsf{rlet}(h); \mathsf{rcase}(f) &= \mathsf{rcase}(h \gg f) \\
&= \mathsf{rcase}(g \gg B +_R h) \\
&= \mathsf{rcase}(g); (R \otimes B) + \mathsf{rlet}(h)
\end{aligned} \tag{143}$$

and hence by uniformity that

$$h \gg \mathsf{rfix}(f) = \mathsf{rlet}(h); (\mathsf{rcase}(f))^\dagger; \pi_r = (\mathsf{rcase}(g))^\dagger; \pi_r = \mathsf{rfix}(g) \tag{144}$$

□

C Definitions and Proofs

C.1 Syntactic Metatheory

$$(\gamma, x \mapsto e)(x) = e \qquad (\gamma, y \mapsto e)(x) = \gamma(x) \qquad (\cdot)(x) = x$$

$$[\gamma]x = \gamma(x) \qquad [\gamma](\text{let } x = a;\ e) = \text{let } x = [\gamma]a;\ [\gamma]e \qquad [\gamma](a, b) = ([\gamma]a, [\gamma]b) \qquad [\gamma]() = ()$$
$$[\gamma](\text{let } (x, y) = a;\ e) = \text{let } (x, y) = [\gamma]a;\ [\gamma]e \qquad [\gamma](\iota_l\ a) = \iota_l\ [\gamma]a \qquad [\gamma](\iota_r\ b) = \iota_r\ [\gamma]b$$
$$[\gamma](\text{case } e\ \{\iota_l\ x : a, \iota_r\ y : b\}) = \text{case } [\gamma]e\ \{\iota_l\ x : [\gamma]a, \iota_r\ y : [\gamma]b\}$$
$$[\gamma](\text{abort } a) = \text{abort } [\gamma]a$$

$$[\gamma](\text{br } \ell\ a) = \text{br } \ell\ [\gamma]a \qquad [\gamma](\text{let } x = a; r) = \text{let } x = [\gamma]a; [\gamma]r$$
$$[\gamma](\text{let } (x, y) = e; r) = \text{let } (x, y) = [\gamma]e; [\gamma]r$$
$$[\gamma](\text{case } e\ \{\iota_l\ x : r, \iota_r\ y : s\}) = \text{case } [\gamma]e\ \{\iota_l\ x : [\gamma]r, \iota_r\ y : [\gamma]s\}$$
$$[\gamma](r \text{ where } (\ell_i(x_i) : \{t_i\},)_i) = [\gamma]r \text{ where } (\ell_i(x_i) : \{[\gamma]t_i\},)_i$$

$$[\gamma](\cdot) = \cdot \qquad [\gamma](\gamma', x \mapsto e) = ([\gamma]\gamma', x \mapsto [\gamma]e)$$

$$[\gamma](\cdot) = \cdot \qquad [\gamma](\sigma, \ell(x) \mapsto r) = ([\gamma]\sigma, \ell(x) \mapsto [\gamma]r)$$

Fig. 37. Capture-avoiding substititon for λ_{SSA} terms, regions, and (label) substitutions; in particular, we assume bound variables and labels are α-converted so as not to appear in γ/σ.

C.2 Lexical SSA and ANF

Lemma 4.4 (A-normalization). *Given an arbitrary region r, we have that*

- $\mathsf{IsANF}(\mathsf{ANF}(r))$
- *If* $\Gamma \vdash r \rhd \mathsf{L}$, *then* $\Gamma \vdash r \approx \mathsf{ANF}(r) \rhd \mathsf{L}$

Similarly, given an arbitrary variable x, expression a and region r If we are also given an arbitrary expression a, then

- $\mathsf{IsANF}(r) \implies \mathsf{IsANF}(\mathsf{ANF}_{\mathsf{let}}(x, a, r))$
- *If* $\Gamma \vdash_\epsilon a : A$ *and* $\Gamma, x : A \vdash r \rhd \mathsf{L}$, *then* $\Gamma \vdash \text{let } x = a; r \approx \mathsf{ANF}_{\mathsf{let}}(x, a, r) \rhd \mathsf{L}$

Proof. We may show that $\mathsf{ANF}(r)$ is always in ANF and that $\mathsf{ANF}_{\mathsf{let}}(x, a, r)$ is in ANF if r is by a straightforward induction. To prove the rest of the lemma, we begin by proving the correctness of $\mathsf{ANF}_{\mathsf{let}}(x, a, r)$ by induction on expressions a:

- If $\Gamma \vdash^{\mathsf{anf}}_{\epsilon} a : A$ is atomic, we are done; otherwise
- If $a = f\ e$, then, since e is a subterm of a, by induction, we have

$$\text{let } x = a; r \approx \text{let } y = e; \text{let } x = f\ y; r \approx \mathsf{ANF}_{\mathsf{let}}(y, e, \text{let } x = f\ y; r) = \mathsf{ANF}_{\mathsf{let}}(x, a, r) \tag{145}$$

  as desired.
- The cases for pairs, injections, and aborts containing expressions are analogous

- If $a = \mathsf{case}\ e\ \{\iota_l\ y : b, \iota_r\ z : c\}$, then we may rewrite

$$\mathsf{let}\ x = a; r \approx \mathsf{let}\ w = e; \mathsf{case}\ w\ \{\iota_l\ y : \mathsf{let}\ b = x;\ r, \iota_r\ z : \mathsf{let}\ c = x;\ r\} \tag{146}$$

  Since r is in ANF, by induction (since b, c are subterms of a), this is equivalent to

$$\mathsf{let}\ w = e; \mathsf{case}\ w\ \{\iota_l\ y : \mathsf{ANF}_{\mathsf{let}}(b, x, r), \iota_r\ z : \mathsf{ANF}_{\mathsf{let}}(c, x, r)\} \tag{147}$$

  which is equal to $\mathsf{ANF}_{\mathsf{let}}(x, a, r)$, as desired.
- The cases for unary and binary let are analogous to the above

We may now prove the correctness of $\mathsf{ANF}(r)$ by a straightforward induction on r:

- If $r = \mathsf{br}\ \ell\ a$, by β-reduction, we have that

$$r \approx \mathsf{let}\ x = a;\ \mathsf{br}\ \ell\ x \approx \mathsf{ANF}_{\mathsf{let}}(x, a, \mathsf{br}\ \ell\ x) = \mathsf{ANF}(r) \tag{148}$$

- If $r = \mathsf{let}\ x = a; r'$, then by induction we have that

$$r \approx \mathsf{let}\ x = a; \mathsf{ANF}(r') \approx \mathsf{ANF}_{\mathsf{let}}(x, a, \mathsf{ANF}(r')) = \mathsf{ANF}(r) \tag{149}$$

  as desired.
- If $r = \mathsf{let}\ (x, y) = a; r'$, then by induction we have that

$$\begin{aligned} r &\approx \mathsf{let}\ z = a; \mathsf{let}\ (x, y) = z; r' \\ &\approx \mathsf{let}\ z = a; \mathsf{let}\ (x, y) = z; \mathsf{ANF}(r') \\ &\approx \mathsf{ANF}_{\mathsf{let}}(z, a, \mathsf{let}\ (x, y) = z; \mathsf{ANF}(r')) \approx \mathsf{ANF}(r) \end{aligned} \tag{150}$$

  The proof for case-statements is analogous
- The case for control-flow graphs follows trivially by induction

□

Lemma 4.5 (SSA conversion). *Given an arbitrary region r,*

- $\mathsf{IsSSA}(\mathsf{SSA}(r))$
- *If* $\Gamma \vdash r \rhd \mathsf{L}$, *then* $\Gamma \vdash r \approx \mathsf{SSA}(r) \rhd \mathsf{L}$

In particular,

- *If* $\mathsf{IsANF}(r)$ *and* $\forall i, \mathsf{IsSSA}(t_i)$, *then* $\mathsf{IsSSA}(\mathsf{SSA}_{\mathsf{a}}(r, (\ell_i(x_i) : \{t_i\},)_i))$
- *If* $\Gamma \vdash r \rhd \mathsf{L}, (\ell_i(A_i),)_i$ *and* $\forall i, \Gamma \vdash t_i \rhd \mathsf{L}, (\ell_i(A_i),)_i$, *then* $\Gamma \vdash (r\ \mathsf{where}\ (\ell_i(x_i) : \{t_i\},)_i) \approx \mathsf{SSA}_{\mathsf{a}}(r, (\ell_i(x_i) : \{t_i\},)_i) \rhd \mathsf{L}$

Proof. Given that $\mathsf{SSA}_{\mathsf{a}}(r, G) \approx r$ where G for r in ANF, it is trivial to see that

$$\mathsf{SSA}(r) := \mathsf{SSA}_{\mathsf{a}}(\mathsf{ANF}(r), \cdot) \approx (\mathsf{ANF}(r)\ \mathsf{where}\ \cdot) \approx \mathsf{ANF}(r) \approx r \tag{151}$$

We hence only need to prove the second part of the lemma. We proceed by induction on ANF regions r as follows:

- If r is a terminator, this holds trivially by reflexivity
- If $r = \mathsf{let}\ x = a; r'$, then by induction, we have that

$$\mathsf{SSA}_{\mathsf{a}}(r, G) := \mathsf{let}\ x = a; \mathsf{SSA}_{\mathsf{a}}(r', G) \approx \mathsf{let}\ x = a; r'\ \mathsf{where}\ G \approx \mathsf{let}\ x = a; r'\ \mathsf{where}\ G \approx r\ \mathsf{where}\ G \tag{152}$$

  as desired. The case for binary let-statements is analogous.

- If $r = \text{case } a\ \{\iota_l\ x : s, \iota_r\ y : t\}$, then by induction, we have that

$$\begin{aligned}\text{SSA}_\text{a}(r, G) &:= (\text{case } a\ \{\iota_l\ x : \text{br } \ell_l\ x, \iota_r\ y : \text{br } \ell_r\ y\}) \text{ where } G, \ell_l(x) : \{\text{SSA}(s)\}, \ell_r(y) : \{\text{SSA}(t)\} \\ &:= (\text{case } a\ \{\iota_l\ x : \text{br } \ell_l\ x, \iota_r\ y : \text{br } \ell_r\ y\}) \text{ where } G, \ell_l(x) : \{s\}, \ell_r(y) : \{t\} \\ &\approx ((\text{case } a\ \{\iota_l\ x : \text{br } \ell_l\ x, \iota_r\ y : \text{br } \ell_r\ y\}) \text{ where } \ell_l(x) : \{s\}, \ell_r(y) : \{t\}) \text{ where } G \\ &\approx \text{case } a\ \{\iota_l\ x : s, \iota_r\ y : t\} \text{ where } G \approx r \text{ where } G\end{aligned} \tag{153}$$

- If $r = r'$ where $\ell_i(x_i) : \{t_i\},)_i$, then by induction, we have that

$$\begin{aligned}\text{SSA}_\text{a}(r, G) &:= \text{SSA}_\text{a}(r', G, (\ell_i(x_i) : \{\text{SSA}(t_i)\},)_i) \\ &\approx r' \text{ where } G, (\ell_i(x_i) : \{t_i\},)_i \\ &\approx (r' \text{ where } (\ell_i(x_i) : \{t_i\},)_i \text{ where } G \approx r \text{ where } G\end{aligned} \tag{154}$$

□

Lemma 4.6 (Permutation Invariance). *If $G \simeq G'$ and $\Gamma \vdash \text{reg}(G) \rhd \mathsf{L}$, then $\Gamma \vdash \text{reg}(G') \rhd \mathsf{L}$ and $\Gamma \vdash \text{reg}(G) \approx \text{reg}(G') \rhd \mathsf{L}$*

Proof. We proceed by induction on the size of G. If G consists only of an entry block, there is only one permutation, so we are done. Otherwise, assume $G = \beta, (\ell_i(x_i) : \{t_i\},)_{i \in I}$. Furthermore, let:

- $\ell_i(x_i) : \{\beta_i\}$ be the children of β in G in order
- $G_i = (\kappa_{i,j}(y_{i,j}) : \{t_{i,j}\},)_{j \in I}$ be the CFG composed of the descendants of β_i in G, with β_i as entry block, in order
- $\ell'_i(x'_i) : \{\beta'_i\}$ be the children of β in G' in order
- G'_i be the CFG composed of the descendants of β'_i in G', with β'_i as entry block, in order

By assumption, there exists some permutation $\sigma : I \longrightarrow I$ such that $G' = \beta, (\ell_{\sigma_i}(x_{\sigma_i}) : \{t_{\sigma_i}\},)_{i \in I}$. It follows that, since the dominance relation on labels is permutation-invariant, there exists some permutation ρ such that $\forall i \in I, \ell'_i(x'_i) : \{\beta'_i\} = \ell_{\rho_i}(x_{\rho_i}) : \{\beta_{\rho_i}\}$, as well as permutations τ_i such that $G'_i = (\ell_{\rho_i,\tau_{ij}}(x_{\rho_i,\tau_{ij}}) : \{t_{\rho_i,\tau_{ij}}\},)_{j \in I}$, implying in particular that $G'_i \simeq G_{\rho_i}$. By induction, we hence have that, for all i, $\text{reg}(G'_i) \approx \text{reg}(G_{\rho_i})$, and hence that

$$\begin{aligned}\text{reg}(G') &:= \text{bb}(\beta, (\ell_{\rho_i}(x_{\rho_i}) : \{\text{reg}(G'_i)\},)_{i \in I}) \\ &\approx \text{bb}(\beta, (\ell_{\rho_i}(x_{\rho_i}) : \{\text{reg}(G_{\rho_i})\},)_{i \in I}) \\ &\approx \text{bb}(\beta, (\ell_i(x_i) : \{\text{reg}(G_i)\},)_{i \in I}) \approx \text{reg}(G)\end{aligned} \tag{155}$$

□

Lemma 4.7 (CFG Conversion). *Given a lexical SSA region r (i.e., assuming $\text{IsSSA}(r)$) s.t. $\Gamma \vdash r \rhd \mathsf{L}$, we have that $\Gamma \vdash r \approx \text{reg}(\text{cfg}(r)) \rhd \mathsf{L}$.*

Proof. We will proceed by induction on the length of $G = \text{cfg}(r)$. If G consists of only an entry block β, then we trivially have that $r = \text{reg}(\text{cfg}(r))$, and so we are done. Otherwise, assume $r = \text{bb}(\beta, (\ell_i(x_i) : \{t_i\},)_{i \in I})$. Clearly, $\text{cfg}(r)$ will have entry block β; moreover, since every block in $\text{cfg}(r)$ other than those of the form $\text{entry}(t_i)$ can only be reached from within the region t_i (due to lexical scoping of labels), we have $\beta_i = \text{entry}(t_{\rho_i})$ for some injection ρ, where β_i are the children of β in the dominance tree of G. In particular, we can write $\{\ell_i(x_i) : \{\text{entry}(t_i)\},\}_i$ as the disjoint union of:

- The immediate children of β, $\kappa_i(y_i) : \{\beta_i\}$, where $\kappa_i = \ell_{\rho_i}$, $y_i = x_{\rho_i}$, and $\beta_i = \text{entry}(t_{\rho_i})$

- The nodes dominated by each β_i but not immediately dominated by β; we will write the collection of such nodes for each i as $\kappa_{i,j}(y_{i,j}) : \{\beta_{i,j}\}$, where $\kappa_{i,j} = \ell_{\rho_{i,j}}$, $y_{i,j} = x_{\rho_{i,j}}$, and $\beta_{i,j} = \mathsf{entry}(t_{\rho_{i,j}})$.

As in the algorithm to compute $\mathrm{reg}(\cdot)$, let G_i denote the control-flow graph with entry block β_i consisting of all blocks dominated by β_i. We may write

$$G \simeq \beta, (\kappa_i(y_i) : \{G_i\},)_{i\in I} \qquad \forall i \in I, G_i = \beta_i, (\kappa_{i,j}(y_{i,j}) : \{\beta_{i,j}\},)_{j\in I}, R_i \tag{156}$$

where R_i is the "remainder" of the control-flow graph G_i. Note the equations for G_i are actually equalities, rather than being equivalence up to permutation $\simeq$, since the $\beta_{i,j}$, appear in G before any other elements, being immediate children of β. It follows that

$$\mathrm{reg}(\mathrm{cfg}(r)) = \mathrm{reg}(\mathrm{cfg}(\mathsf{bb}(\beta, (\ell_i(x_i) : \{t_i\},)_i))) = \mathsf{bb}(\beta, (\kappa_i(y_i) : \{\mathrm{reg}(G_i)\},)_i) \tag{157}$$

Now, define the lexical SSA regions

$$r_i = \mathsf{bb}(\beta_i, ((\kappa_{i,j}(y_{i,j}) : \{t_{\rho_{i,j}}\},)_j, \mathsf{children}(t_i))) \tag{158}$$

It is easy to show that $\mathrm{cfg}(r_i) \simeq G_i$, since every basic block dominated by β_i must be dominated via either some $t_{\rho_{i,j}}$ or some child of t_i. Since by induction $\mathrm{reg}(\mathrm{cfg}(r_i)) \approx r_i$, and by the previous lemma $\mathrm{reg}(\mathrm{cfg}(_)r_i) \approx \mathrm{reg}(G_i)$, it follows that

$$\mathrm{reg}(\mathrm{cfg}(r)) \approx \mathsf{bb}(\beta, (\kappa_i(y_i) : \{r_i\},)_i) \tag{159}$$

Define

$$T_i = (\kappa_j(y_j) : \{r_j\},)_{j<i}, (\kappa_j(y_j) : \{t_j\}, (\kappa_{j,k}(y_{j,k}) : \{t_{\rho_{j,k}}\},)_k,)_{j\geq i} \tag{160}$$

Using Equation 27, we may show that, for all i, $\mathsf{bb}(\beta, T_{i+1}) \approx \mathsf{bb}(\beta, T_i)$, since

- For all j, $t_{\rho_{i,j}}$ cannot use variables defined in β_i, or r would not typecheck
- For all j, if r_k or $t_{\rho_{k,j'}}$ calls $\kappa_{i,j}$, then $k = i$, or $\beta_{i,j}$ would not be dominated by β_i
- Similarly, β cannot call $\kappa_{i,j}$ or $\beta_{i,j}$ would be a direct descendant of β

We hence have by induction that

$$\mathrm{reg}(\mathrm{cfg}(r)) \approx \mathsf{bb}(\beta, T_N) \approx \mathsf{bb}(\beta, T_0) \approx r \tag{161}$$

as desired, since T_0 is a permutation of $(\ell_i(x_i) : \{t_i\},)_i$. □

C.3 Böhm-Jacopini

Lemma C.1. *The following facts hold:*

- *Given* $\Gamma \vdash r \rhd \blacksquare(A)$ *and* $\Gamma, \square : A \vdash s \rhd \mathsf{L}$, *we have that*

$$\begin{aligned}&[\![\Gamma \vdash \mathrm{seq}(r, s) := ([\blacksquare(x) \mapsto \mathsf{br}\ \ell\ x]r \text{ where } \ell(\square) : \{s\}) \rhd \mathsf{L}]\!]\\ &= \mathsf{let}([\![\Gamma \vdash [\blacksquare(x) \mapsto \mathsf{br}\ \ell\ x]r \rhd \ell(A)]\!]); [\![\Gamma]\!] \otimes \alpha^+_A; [\![\Gamma, \square : A \vdash s \rhd \mathsf{L}]\!]\\ &= \mathsf{let}([\![\Gamma \vdash r \rhd \ell(A)]\!]); [\![\Gamma]\!] \otimes \alpha^+_A; [\![\Gamma, \square : A \vdash s \rhd \mathsf{L}]\!]\end{aligned} \tag{162}$$

- *Given* $\Gamma \vdash_\epsilon e : A, \Gamma, \square : A \vdash r \rhd \blacksquare(B + A)$, *we have that*

$$\begin{aligned}&[\![\Gamma \vdash \mathrm{loop}(e, r) := (\mathsf{br}\ \ell\ e \text{ where } \ell(\square) : \{\mathrm{seq}(r, \mathsf{case}\ \square\ \{\iota_l\ x : \mathsf{br}\ \blacksquare\ x, \iota_r\ y : \mathsf{br}\ \ell\ y\})\}) \rhd \blacksquare(B)]\!]\\ &= \mathsf{let}([\![\Gamma \vdash_\epsilon e : A]\!]); \mathrm{rfix}([\![\Gamma, \square : A \vdash r \rhd \blacksquare(B + A)]\!]; \alpha^+_{B+A})\end{aligned} \tag{163}$$

- *Given* $\mathsf{R} = (\ell_i(A_i),)_i$ *and* $\Gamma, x_i : A_i \vdash t_i \rhd \mathsf{L}$, *we have*

$$\begin{aligned}&[\![\Gamma, c : \langle \mathsf{R} \rangle \vdash \mathsf{case}_\mathsf{R}\ c\ \{\ell_i(x_i) : t_i\} \rhd \mathsf{L}]\!]\\ &= [\![\Gamma]\!] \otimes \alpha^+_{\Sigma_i [\![A_i]\!]}; \delta^{-1}_\Sigma; [[\![\Gamma, x_i : A_i \vdash t_i \rhd \mathsf{L}]\!],]_i\end{aligned} \tag{164}$$

We have that

$$[\![\Gamma, \square : \bot \vdash_{[\mathsf{L}]} \mathsf{ua}\ \square : \langle \mathsf{L}\rangle + \langle \mathsf{R}\rangle]\!] = \pi_r; \alpha^+_{[\![\mathsf{L}]\!]+[\![\mathsf{R}]\!]} \tag{165}$$

We have that

$$[\![\Gamma \vdash \mathsf{pack}^+_\kappa(\mathsf{L}) : \mathsf{L} \rightsquigarrow \kappa(\langle \mathsf{L}\rangle)]\!] = \pi_r; \alpha^+_{\mathbf{0}+[\![\mathsf{L}]\!]} \tag{166}$$

and

$$[\![\Gamma \vdash \mathsf{unpack}^+_\kappa(\mathsf{L}) : \kappa(\langle \mathsf{L}\rangle) \rightsquigarrow \mathsf{L}]\!] = \pi_r; \alpha^+_{[\![\mathsf{L}]\!]} \tag{167}$$

In particular, given $\Gamma \vdash r \rhd \mathsf{L}$*, we have that*

$$[\![\Gamma \vdash \langle r\rangle^+ \rhd \blacksquare(\langle \mathsf{L}\rangle)]\!] = [\![\Gamma \vdash r \rhd \mathsf{L}]\!]; \alpha^+_{\mathbf{0}+[\![\mathsf{L}]\!]} \tag{168}$$

Theorem A.1 (Böhm-Jacopini for SSA). *For all* $\Gamma \vdash r \rhd \mathsf{L}$*, we have that* $\Gamma \vdash^{\mathsf{s}} \mathsf{WH}_{\mathsf{L}}(r) \rhd \mathsf{L}$ *is structured, and* $\Gamma \vdash r \approx \mathsf{WH}_{\mathsf{L}}(r) \rhd \mathsf{L}$*. In particular, we have that* $\Gamma \vdash^{\mathsf{s}} \mathsf{PW}_{\mathsf{L}}(r) \rhd \blacksquare(\langle \mathsf{L}\rangle)$ *is structured, and* $\Gamma \vdash \langle r\rangle^+ \approx \mathsf{PW}_r(\rhd)\blacksquare(\langle \mathsf{L}\rangle)$.

Proof. We begin by showing the correctness of $\mathsf{PW}_{\mathsf{L}}(r)$ by induction on r:

- If $r = \mathsf{br}\ \ell\ a$, we have that $\mathsf{PW}_{\mathsf{L}}(a) = \mathsf{pack}^+(\mathsf{L})_\ell(a) = [\mathsf{br}\ \ell\ a]^+$ by definition
- If $r = \mathsf{let}\ x = a; s$, we have by induction that

$$\mathsf{PW}_{\mathsf{L}}(\mathsf{let}\ x = a; s) = \mathsf{let}\ x = a; \mathsf{PW}_{\mathsf{L}}(s) \approx \mathsf{let}\ x = a; [s]^+ = [\mathsf{let}\ x = a; s]^+$$

- If $r = \mathsf{let}\ (x, y) = a; s$, we have by induction that

$$\mathsf{PW}_{\mathsf{L}}(\mathsf{let}\ (x, y) = a; s) = \mathsf{let}\ (x, y) = a; \mathsf{PW}_{\mathsf{L}}(s) \approx \mathsf{let}\ (x, y) = a; \langle s\rangle^+ = \langle \mathsf{let}\ (x, y) = a; s\rangle^+$$

- If $r = \mathsf{case}\ a\ \{\iota_l\ x : s, \iota_r\ y : t\}$, we have by induction that

$$\begin{aligned}\mathsf{PW}_{\mathsf{L}}(\mathsf{case}\ a\ \{\iota_l\ x : s, \iota_r\ y : t\}) &= \mathsf{case}\ a\ \{\iota_l\ x : \mathsf{PW}_{\mathsf{L}}(s), \iota_r\ y : \mathsf{PW}_{\mathsf{L}}(t)\}\\ &\approx \mathsf{case}\ a\ \{\iota_l\ x : \langle s\rangle^+, \iota_r\ y : \langle t\rangle^+\}\\ &= \langle \mathsf{case}\ a\ \{\iota_l\ x : s, \iota_r\ y : t\}\rangle^+\end{aligned}$$

- Assume $r = s$ where $(\ell_i(t_i) : \{x_i\},)_i$. Define $\mathsf{R} = (\ell_i(A_i),)_i$. By induction, we have that $\forall i, \mathsf{PW}_{\mathsf{L},\mathsf{R}}(t_i) \approx \langle t_i\rangle^+$, and hence by soundness

$$[\![\Gamma, x_i : A_i \vdash \mathsf{PW}_{\mathsf{L}}(t_i) \rhd \blacksquare(\langle \mathsf{L}, \mathsf{R}\rangle)]\!] = [\![\Gamma, x_i : A_i \vdash \langle t_i\rangle^+ \rhd \blacksquare(\langle \mathsf{L}, \mathsf{R}\rangle)]\!] \tag{169}$$

Now, define

$$D = \mathsf{case}_{\mathsf{L}}\ \square\ \{\ell_i(x_i) : \mathsf{seq}(\mathsf{PW}_{\mathsf{L}}(t_i), \mathsf{br}\ \blacksquare\ (\mathsf{ua}\ \square))\}$$

and $L = \mathsf{loop}(y, D)$. It follows that

$$\begin{aligned}&[\![\Gamma, \square : [R] \vdash D \rhd \blacksquare(\langle L\rangle + \langle R\rangle)]\!]\\ &= [\![\Gamma]\!] \otimes \alpha^+_{\Sigma_i[\![A_i]\!]}; \delta^{-1}_\Sigma; [[\![\Gamma, x_i : A_i \vdash \mathsf{seq}(\mathsf{PW}_{\mathsf{L}}(t_i), \mathsf{br}\ \blacksquare\ (\mathsf{ua}\ \square)) \rhd \blacksquare(\langle \mathsf{L}\rangle + \langle \mathsf{R}\rangle)]\!],]_i\\ &= [\![\Gamma]\!] \otimes \alpha^+_{\Sigma_i[\![A_i]\!]}; \delta^{-1}_\Sigma; [[\![\Gamma, x_i : A_i \vdash \mathsf{PW}_{\mathsf{L}}(t_i) \rhd \blacksquare(\langle \mathsf{L}, \mathsf{R}\rangle)]\!]; \alpha^+_{\mathbf{0}+([\![\mathsf{L}]\!]+[\![\mathsf{R}]\!])},]_i\\ &= [\![\Gamma]\!] \otimes \alpha^+_{\Sigma_i[\![A_i]\!]}; \delta^{-1}_\Sigma; [[\![\Gamma, x_i : A_i \vdash \langle t_i\rangle^+ \rhd \blacksquare(\langle \mathsf{L}, \mathsf{R}\rangle)]\!]; \alpha^+_{\mathbf{0}+([\![\mathsf{L}]\!]+[\![\mathsf{R}]\!])},]_i\\ &= [\![\Gamma]\!] \otimes \alpha^+_{\Sigma_i[\![A_i]\!]}; \delta^{-1}_\Sigma; [[\![\Gamma, x_i : A_i \vdash t_i \rhd \mathsf{L}, \mathsf{R}]\!]; \alpha^+_{\mathbf{0}+([\![\mathsf{L}]\!]+[\![\mathsf{R}]\!])},]_i\end{aligned} \tag{170}$$

and therefore that

$$\begin{aligned}&[\![\Gamma, y : \langle R\rangle \vdash L \rhd \blacksquare(\langle \mathsf{L}\rangle)]\!]\\ &= \mathsf{let}([\![\Gamma, y : \langle R\rangle \vdash_\bot y : A]\!]); \mathsf{rfix}([\![\Gamma, \square : \langle R\rangle \vdash D \rhd \blacksquare(\langle L\rangle + \langle R\rangle)]\!]; \alpha^+_{(\mathbf{0}+[\![\mathsf{L}]\!])+[\![\mathsf{R}]\!]})\\ &= \mathsf{rfix}([\![\Gamma]\!] \otimes \alpha^+_{\Sigma_i[\![A_i]\!]}; \delta^{-1}_\Sigma; [[\![\Gamma, x_i : A_i \vdash t_i \rhd \langle \mathsf{L}, \mathsf{R}\rangle]\!]; \alpha^+_{(\mathbf{0}+[\![\mathsf{L}]\!])+[\![\mathsf{R}]\!]},]_i)\\ &= [\![\Gamma]\!] \otimes \alpha^+_{\Sigma_i[\![A_i]\!]}; \mathsf{rfix}(\mathsf{lsem}_{\Gamma,\mathsf{L}}((\ell_i(x_i) : \{t_i\},)_i)); \alpha^+_{\mathbf{0}+[\![\mathsf{L}]\!]}\end{aligned} \tag{171}$$

Hence, we have that

$$\begin{aligned}
&[\![\Gamma, \square : \langle \mathsf{L}, \mathsf{R}\rangle \vdash \mathsf{case\ ua}\ \square\ \{\iota_l\ x : \mathsf{br}\ \blacksquare\ x, \iota_r\ y : L\} \rhd \blacksquare([\mathsf{L}])]\!] \\
&= [\![\Gamma]\!] \otimes \alpha^+_{[\![\mathsf{L}]\!]+[\![\mathsf{R}]\!]}; \delta^{-1}; [[\![\Gamma, x : \langle \mathsf{L}\rangle \vdash \mathsf{br}\ \blacksquare\ x \rhd \blacksquare(\langle \mathsf{L}\rangle)]\!], [\![\Gamma, y : \langle \mathsf{R}\rangle \vdash L \rhd \blacksquare(\langle \mathsf{L}\rangle)]\!]] \\
&= [\![\Gamma]\!] \otimes \alpha^+_{[\![\mathsf{L}]\!]+[\![\mathsf{R}]\!]}; \delta^{-1}; [\pi_r; \alpha^+_{\mathbf{0}+[\![\mathsf{L}]\!]}, [\![\Gamma]\!] \otimes \alpha^+_{\Sigma_i [\![A_i]\!]}; \mathsf{rfix}(\mathsf{lsem}_{\Gamma,\mathsf{L}}((\ell_i(x_i) : \{t_i\},)_i)); \alpha^+_{\mathbf{0}+[\![\mathsf{L}]\!]}] \\
&= [\![\Gamma]\!] \otimes \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_i [\![A_i]\!]}; \delta^{-1}; [\pi_r, \mathsf{rfix}(\mathsf{lsem}_{\Gamma,\mathsf{L}}((\ell_i(x_i) : \{t_i\},)_i))]; \alpha^+_{\mathbf{0}+[\![\mathsf{L}]\!]}
\end{aligned} \tag{172}$$

It hence suffices by completeness (Theorem 5.14) to show that

$$\begin{aligned}
&[\![\Gamma \vdash \mathsf{seq}(\mathsf{PW}_\mathsf{L}(r), \mathsf{case\ ua}\ \square\ \{\iota_l\ x : \mathsf{br}\ \blacksquare\ x, \iota_r\ y : L\}) \rhd \blacksquare(\langle \mathsf{L}\rangle)]\!] \\
&= \mathsf{let}([\![\Gamma \vdash \mathsf{PW}_\mathsf{L}(r) \rhd \blacksquare(\langle \mathsf{L}, \mathsf{R}\rangle)]\!]); [\![\Gamma]\!] \otimes \alpha^+_{[\![\mathsf{L},\mathsf{R}]\!]}; \\
&\qquad [\![\Gamma, \square : \langle \mathsf{L}, \mathsf{R}\rangle \vdash \mathsf{case\ ua}\ \square\ \{\iota_l\ x : \mathsf{br}\ \blacksquare\ x, \iota_r\ y : L\} \rhd \blacksquare(\langle \mathsf{L}\rangle)]\!] \\
&= \mathsf{let}([\![\Gamma \vdash \mathsf{PW}_\mathsf{L}(r) \rhd \blacksquare(\langle \mathsf{L}, \mathsf{R}\rangle)]\!]); [\![\Gamma]\!] \otimes \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_i [\![A_i]\!]}; \delta^{-1}; \\
&\qquad [\pi_r, \mathsf{rfix}(\mathsf{lsem}_{\Gamma,\mathsf{L}}((\ell_i(x_i) : \{t_i\},)_i))]; \alpha^+_{\mathbf{0}+[\![\mathsf{L}]\!]} \\
&= [\![\Gamma \vdash r\ \mathsf{where}\ (\ell_i(x_i) : \{t_i\},)_i \rhd \mathsf{L}]\!]; \alpha^+_{\mathbf{0}+[\![\mathsf{L}]\!]} \\
&= [\![\Gamma \vdash [r\ \mathsf{where}\ (\ell_i(x_i) : \{t_i\},)_i]^+ \rhd \mathsf{L}]\!]
\end{aligned} \tag{173}$$

□

C.4 Substitution

Lemma C.2. *The following facts hold:*

(a) For all $f : A \to B \otimes C$, $g : (A \otimes B) \otimes C \to D$, we have

$$\mathsf{let}(f); \alpha; \mathsf{let}(g); (\pi_l; \pi_l) \otimes D = \mathsf{let}(\mathsf{let}(f); \alpha; g) \tag{174}$$

(b) For all $f : A \to B + C$, $g : A \otimes B \to D$, $h : A \otimes C \to D$, we have

$$\mathsf{let}(f); \delta^{-1}; [\mathsf{let}(g); \pi_l \otimes D, \mathsf{let}(h); \pi_l \otimes D] = \mathsf{let}(\mathsf{let}(f); \delta^{-1}; [g, h]) \tag{175}$$

(c) For all $f_i : R \otimes A_i \to B$, we have

$$\delta_\Sigma^{-1}; [\mathsf{let}(f_i); \pi_l \otimes B,]_i = \mathsf{let}(\delta_\Sigma^{-1}; [f_i]_i); \pi_l \otimes B \tag{176}$$

i.e.,

$$\delta_\Sigma^{-1}; [\mathsf{rlet}(f_i),]_i = \mathsf{rlet}(\delta_\Sigma^{-1}; [f_i]_i) \tag{177}$$

(d) For all $f : A \to B + C$, $g : A \otimes B \to B'$, $h : A \otimes C \to C'$, we have

$$\begin{aligned}
\mathsf{let}(\mathsf{let}(f); \delta^{-1}; g + h) &= \mathsf{let}(f); \delta^{-1}; [\mathsf{let}(g); \pi_l \otimes \iota_l, \mathsf{let}(h); \pi_l \otimes \iota_r] \\
&= \mathsf{let}(f); \delta^{-1}; (\mathsf{let}(g); \pi_l \otimes B') + (\mathsf{let}(h); \pi_l \otimes C'); \delta \\
&= \mathsf{let}(f); \delta^{-1}; \mathsf{rlet}(g) + \mathsf{rlet}(h); \delta
\end{aligned} \tag{178}$$

In particular, we have

$$\begin{aligned}
\mathsf{let}(\mathsf{let}(f); \delta^{-1}; \pi_r + h) &= \mathsf{let}(f); \delta^{-1}; [A \otimes \iota_l, \mathsf{let}(h); \pi_l \otimes \iota_r] \\
&= \mathsf{let}(f); \delta^{-1}; (A \otimes B) + (\mathsf{let}(h); \pi_l \otimes C'); \delta \\
&= \mathsf{let}(f); \delta^{-1}; (A \otimes B) + \mathsf{rlet}(h); \delta
\end{aligned} \tag{179}$$

and

$$\begin{aligned}
\mathsf{let}(\mathsf{let}(f); \delta^{-1}; g + \pi_r) &= \mathsf{let}(f); \delta^{-1}; [\mathsf{let}(g); \pi_l \otimes \iota_l, A \otimes \iota_r] \\
&= \mathsf{let}(f); \delta^{-1}; (\mathsf{let}(g); \pi_l \otimes B') + (A \otimes C); \delta \\
&= \mathsf{let}(f); \delta^{-1}; \mathsf{rlet}(g) + (A \otimes C); \delta
\end{aligned} \tag{180}$$

Lemma 5.8 ((Label) Weakening). *Given* $\Gamma \leq \Gamma'$ *and* $\mathsf{L}' \leq \mathsf{L}$, $\mathsf{K}' \leq \mathsf{K}$, *we have*

(a) For all $\Gamma \leq \Delta$, $[\![\Gamma \leq \Delta]\!] = [\![\Gamma \leq \Gamma']\!]; [\![\Gamma' \leq \Delta]\!]$
(b) For all $\mathsf{L} \leq \mathsf{K}$, $[\![\mathsf{L}' \leq \mathsf{K}]\!] = [\![\mathsf{L}' \leq \mathsf{L}]\!]; [\![\mathsf{L} \leq \mathsf{K}]\!]$
(c) $[\![\Gamma \vdash_\epsilon a : A]\!] = [\![\Gamma \leq \Gamma']\!]; [\![\Gamma' \vdash_\epsilon a : A]\!]$
(d) $[\![\Gamma \vdash r \rhd \mathsf{L}]\!] = [\![\Gamma \leq \Gamma']\!]; [\![\Gamma' \vdash r \rhd \mathsf{L}']\!]; [\![\mathsf{L}' \leq \mathsf{L}]\!]$
(e) For all $\gamma : \Gamma \mapsto \Delta$, $[\![\gamma : \Gamma \mapsto \Delta]\!] = [\![\Gamma \leq \Gamma']\!]; [\![\gamma : \Gamma' \mapsto \Delta]\!]$
(f) For all $\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}$, $[\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] = [\![\Gamma \leq \Gamma']\!] \otimes [\![\mathsf{L}]\!]; [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}']\!]; [\![\mathsf{K}' \leq \mathsf{K}]\!]$

Proof. To show that variable weakenings compose (a), we proceed by induction on the derivation of $\Gamma \leq \Gamma'$ as follows:

- wk-nil: if $\Gamma, \Gamma' = \cdot$, then $\Delta = \cdot$, and so we trivially have $[\![\cdot \leq \cdot]\!]; [\![\cdot \leq \cdot]\!] = [\![\cdot \leq \cdot]\!] = \mathsf{id}$
- wk-skip: we have $\Gamma = \Xi, x : A$, and so by induction

$$[\![\Xi, x : A \leq \Gamma']\!]; [\![\Gamma' \leq \Delta]\!] = \pi_l; [\![\Xi \leq \Gamma']\!]; [\![\Gamma' \leq \Delta]\!] = \pi_l; [\![\Xi \leq \Delta]\!] = [\![\Xi, x : A \leq \Delta]\!] \tag{181}$$

  as desired
- wk-cons: we have $\Gamma = \Xi, x : A$ and $\Gamma' = \Xi', x : A$. We proceed by case analysis on $\Gamma' \leq \Delta$:
  - wk-skip: we have

$$\begin{aligned} [\![\Xi, x : A \leq \Xi', x : A]\!]; [\![\Xi' \leq \Delta]\!] &= [\![\Xi \leq \Xi']\!] \otimes [\![A]\!]; [\![\Xi', x : A \leq \Delta]\!] \\ &= [\![\Xi \leq \Xi']\!] \otimes [\![A]\!]; \pi_l; [\![\Xi' \leq \Delta]\!] \\ &= \pi_l; [\![\Xi \leq \Xi']\!]; [\![\Xi' \leq \Delta]\!] \\ &= \pi_l; [\![\Xi \leq \Delta]\!] \\ &= [\![\Xi, x : A \leq \Delta]\!] \end{aligned} \tag{182}$$

    as desired
  - wk-cons: we have $\Delta = \Delta', x : A$, and so by induction

$$\begin{aligned} [\![\Xi, x : A \leq \Xi', x : A]\!]; [\![\Xi', x : A \leq \Delta', x : A]\!] &= [\![\Xi \leq \Xi']\!] \otimes [\![A]\!]; [\![\Xi' \leq \Delta']\!] \otimes [\![A]\!] \\ &= ([\![\Xi \leq \Xi']\!]; [\![\Xi' \leq \Delta']\!]) \otimes [\![A]\!] \\ &= [\![\Xi \leq \Delta']\!] \otimes [\![A]\!] \\ &= [\![\Xi, x : A \leq \Delta]\!] \end{aligned} \tag{183}$$

    as desired

We can analogously show (b) (i.e., that label weakenings compose) by induction on the derivation of $\mathsf{L} \leq \mathsf{K}$ as follows:

- lwk-nil: if $\mathsf{L} = \mathsf{K} = \cdot$, then $\mathsf{L}' = \cdot$, so the result follows trivially from the fact that $[\![\cdot]\!] = \mathsf{L}'$ is the initial object
- lwk-skip: we have $\mathsf{K} = \mathsf{K}', \ell(A)$, and therefore

$$[\![\mathsf{L}' \leq \mathsf{L}]\!]; [\![\mathsf{L} \leq \mathsf{K}', \ell(A)]\!] = [\![\mathsf{L}' \leq \mathsf{L}]\!]; [\![\mathsf{L} \leq \mathsf{K}']\!]; \iota_r = [\![\mathsf{L}' \leq \mathsf{K}']\!]; \iota_r = [\![\mathsf{L}' \leq \mathsf{K}]\!] \tag{184}$$

- lwk-cons: we have $\mathsf{L} = \mathsf{R}, \ell(A)$ and $\mathsf{K} = \mathsf{K}', \ell(A)$. We proceed by case splitting on $\mathsf{L} \leq \mathsf{K}'$:
  - lwk-skip: we have

$$\begin{aligned} [\![\mathsf{L}' \leq \mathsf{R}, \ell(A)]\!]; [\![\mathsf{R}, \ell(A) \leq \mathsf{K}', \ell(A)]\!] &= [\![\mathsf{L}' \leq \mathsf{R}]\!]; \iota_r; [\![\mathsf{R} \leq \mathsf{K}']\!] + [\![A]\!] \\ &= [\![\mathsf{L}' \leq \mathsf{R}]\!]; [\![\mathsf{R} \leq \mathsf{K}']\!]; \iota_r \\ &= [\![\mathsf{L}' \leq \mathsf{K}']\!]; \iota_r \\ &= [\![\mathsf{L}' \leq \mathsf{K}', \ell(A)]\!] \end{aligned} \tag{185}$$

– lwk-cons: we have that $\mathsf{L}' = \mathsf{R}', \ell(A)$, and therefore

$$\begin{aligned}\llbracket \mathsf{R}', \ell(A) \leq \mathsf{R}, \ell(A) \rrbracket ; \llbracket \mathsf{R}, \ell(A) \leq \mathsf{K}', \ell(A) \rrbracket &= \llbracket \mathsf{R}' \leq \mathsf{R} \rrbracket + \llbracket A \rrbracket ; \llbracket \mathsf{R} \leq \mathsf{K}' \rrbracket + \llbracket A \rrbracket \\ &= (\llbracket \mathsf{R}' \leq \mathsf{R} \rrbracket ; \llbracket \mathsf{R} \leq \mathsf{K}' \rrbracket) + \llbracket A \rrbracket \\ &= \llbracket \mathsf{R}' \leq \mathsf{K}' \rrbracket + \llbracket A \rrbracket \\ &= \llbracket \mathsf{R}', \ell(A) \leq \mathsf{K}', \ell(A) \rrbracket \end{aligned} \tag{186}$$

We can now show weakening for expressions $\Delta \vdash_\epsilon a : A$ (c) by induction on the typing derivation as follows:

- var: we need to show that

$$\llbracket \Gamma \leq \Delta \rrbracket ; \llbracket \Delta \vdash_\epsilon x : A \rrbracket = \llbracket \Gamma \leq \Delta \rrbracket ; \pi_{\Delta,x} = \llbracket \Gamma \vdash_\epsilon x : A \rrbracket = \pi_{\Gamma,x} \tag{187}$$

  we proceed by induction on $\Gamma \leq \Delta$:
  – wk-nil: this case yields a contradiction, since if $\Delta = \cdot$ it cannot define x.
  – wk-cons: given $\Gamma = \Gamma', y : B$, $\Delta = \Delta', y : B$,
    * If $x = y$, then $B = A$, and

$$\llbracket \Gamma', x : A \leq \Delta', x : A \rrbracket ; \pi_{(\Delta, x:A),x} = \llbracket \Gamma' \leq \Delta' \rrbracket \otimes \llbracket A \rrbracket ; \pi_r = \pi_r = \pi_{(\Gamma, x:A),x} \tag{188}$$

      as desired.
    * Otherwise, we have by induction that

$$\begin{aligned}\llbracket \Gamma', y : B \leq \Delta', y : B \rrbracket ; \pi_{(\Delta', y:B),x} &= \llbracket \Gamma' \leq \Delta' \rrbracket \otimes \llbracket B \rrbracket ; \pi_l ; \pi_{\Delta',x} \\ &= \pi_l ; \llbracket \Gamma' \leq \Delta' \rrbracket ; \pi_{\Delta',x} = \pi_l ; \pi_{\Gamma',x} = \pi_{\Gamma,x} \end{aligned} \tag{189}$$

  – wk-skip: we have $\Gamma = \Gamma', y : B$, and hence

$$\llbracket \Gamma', y : B \leq \Delta \rrbracket ; \pi_{\Delta,x} = \pi_l ; \llbracket \Gamma' \leq \Delta \rrbracket ; \pi_{\Delta,x} = \pi_l ; \pi_{\Gamma',x} = \pi_{\Gamma,x} \tag{190}$$

- let_1: we have

$$\begin{aligned}&\llbracket \Gamma \leq \Delta \rrbracket ; \llbracket \Delta \vdash_\epsilon \mathsf{let}\ x = a;\ b : B \rrbracket \\ &= \llbracket \Gamma \leq \Delta \rrbracket ; \mathsf{let}(\llbracket \Delta \vdash_\epsilon a : A \rrbracket) ; \llbracket \Delta, x : A \vdash_\epsilon b : B \rrbracket \\ &= \mathsf{let}(\llbracket \Gamma \leq \Delta \rrbracket ; \llbracket \Delta \vdash_\epsilon a : A \rrbracket) ; \llbracket \Gamma \leq \Delta \rrbracket \otimes \llbracket A \rrbracket ; \llbracket \Delta, x : A \vdash_\epsilon b : B \rrbracket \\ &= \mathsf{let}(\llbracket \Gamma \vdash_\epsilon a : A \rrbracket) ; \llbracket \Gamma, x : A \leq \Delta, x : A \rrbracket ; \llbracket \Delta, x : A \vdash_\epsilon b : B \rrbracket \\ &= \mathsf{let}(\llbracket \Gamma \vdash_\epsilon a : A \rrbracket) ; \llbracket \Gamma, x : A \vdash_\epsilon b : B \rrbracket \\ &= \llbracket \Gamma \vdash_\epsilon \mathsf{let}\ x = a;\ b : B \rrbracket \end{aligned} \tag{191}$$

- unit: follows immediately since weakenings are pure and $\mathbf{1}$ is the terminal object.
- pair: we have

$$\begin{aligned}&\llbracket \Gamma \leq \Delta \rrbracket ; \llbracket \Delta \vdash_\epsilon (a, b) : A \otimes B \rrbracket \\ &= \llbracket \Gamma \leq \Delta \rrbracket ; \Delta_\Delta ; \llbracket \Delta \vdash_\epsilon a : A \rrbracket \otimes \llbracket \Delta \vdash_\epsilon b : B \rrbracket \\ &= \Delta_\Gamma ; (\llbracket \Gamma \leq \Delta \rrbracket ; \llbracket \Delta \vdash_\epsilon a : A \rrbracket) \otimes (\llbracket \Gamma \leq \Delta \rrbracket ; \llbracket \Delta \vdash_\epsilon b : B \rrbracket) \\ &= \Delta_\Gamma ; \llbracket \Gamma \vdash_\epsilon a : A \rrbracket \otimes \llbracket \Gamma \vdash_\epsilon b : B \rrbracket \\ &= \llbracket \Gamma \vdash_\epsilon (a, b) : A \otimes B \rrbracket \end{aligned} \tag{192}$$

- let_2: we have

$$\begin{aligned}
&[\![\Gamma \leq \Delta]\!]; [\![\Delta \vdash_\epsilon \mathsf{let}\ (x,y) = a;\ b : B]\!] \\
&= [\![\Gamma \leq \Delta]\!]; \mathsf{let}([\![\Delta \vdash_\epsilon a : A \otimes B]\!]); \alpha; [\![\Delta, x : A, y : B \vdash_\epsilon b : B]\!] \\
&= \mathsf{let}([\![\Gamma \leq \Delta]\!]; [\![\Delta \vdash_\epsilon a : A \otimes B]\!]); \alpha; [\![\Gamma \leq \Delta]\!] \otimes [\![A]\!] \otimes [\![B]\!]; [\![\Delta, x : A, y : B \vdash_\epsilon b : B]\!] \\
&= \mathsf{let}([\![\Gamma \vdash_\epsilon a : A \otimes B]\!]); \alpha; [\![\Gamma, x : A, y : B \leq \Delta, x : A, y : B]\!]; [\![\Delta, x : A, y : B \vdash_\epsilon b : B]\!] \\
&= \mathsf{let}([\![\Gamma \vdash_\epsilon a : A \otimes B]\!]); \alpha; [\![\Gamma, x : A, y : B \vdash_\epsilon b : B]\!] \\
&= [\![\Gamma \vdash_\epsilon \mathsf{let}\ (x,y) = a;\ b : B]\!]
\end{aligned} \tag{193}$$

- case: we have

$$\begin{aligned}
&[\![\Gamma \leq \Delta]\!]; [\![\Delta \vdash_\epsilon \mathsf{case}\ a\ \{\iota_l\ x : s, \iota_r\ y : t\} : B]\!] \\
&= [\![\Gamma \leq \Delta]\!]; \mathsf{let}([\![\Delta \vdash_\epsilon a : A]\!]); \delta^{-1}; [[\![\Delta, x : A \vdash_\epsilon s : B]\!], [\![\Delta, y : A \vdash_\epsilon t : B]\!]] \\
&= \mathsf{let}([\![\Gamma \leq \Delta]\!]; [\![\Gamma \vdash_\epsilon a : A]\!]); \delta^{-1}; \\
&\qquad [[\![\Gamma \leq \Delta]\!] \otimes [\![A]\!]; [\![\Delta, x : A \vdash_\epsilon s : C]\!], [\![\Gamma \leq \Delta]\!] \otimes [\![B]\!]; [\![\Delta, y : B \vdash_\epsilon t : C]\!]] \\
&= \mathsf{let}([\![\Delta \vdash_\epsilon a : A]\!]); \\
&\qquad [[\![\Gamma, x : A \leq \Delta, x : A]\!]; [\![\Delta, x : A \vdash_\epsilon s : C]\!], [\![\Gamma, y : B \leq \Delta, y : B]\!]; [\![\Delta, y : B \vdash_\epsilon t : C]\!]] \\
&= \mathsf{let}([\![\Delta \vdash_\epsilon a : A]\!]); [[\![\Gamma, x : A \vdash_\epsilon s : C]\!], [\![\Gamma, y : B \vdash_\epsilon t : C]\!]]
\end{aligned} \tag{194}$$

- op: we have

$$\begin{aligned}
[\![\Gamma \leq \Delta]\!]; [\![\Gamma \vdash_\epsilon f\ a : B]\!] &= [\![\Gamma \leq \Delta]\!]; [\![\Delta \vdash_\epsilon a : A]\!]; [\![f \in \mathcal{I}_\epsilon(A,B)]\!] \\
&= [\![\Gamma \vdash_\epsilon a : A]\!]; [\![f \in \mathcal{I}_\epsilon(A,B)]\!] \\
&= [\![\Gamma \vdash_\epsilon f\ a : B]\!]
\end{aligned} \tag{195}$$

- inl, inr: analogous to the op case

Similarly, we can show weakening for regions $\Delta \vdash r \rhd \mathsf{L}$ (d) by induction on the typing derivation as follows:

- br: we have that

$$\begin{aligned}
[\![\Gamma \leq \Delta]\!]; [\![\Delta \vdash \mathsf{br}\ \ell\ a \rhd \mathsf{L}]\!]; [\![\mathsf{L} \leq \mathsf{K}]\!] &= [\![\Gamma \leq \Delta]\!]; [\![\Delta \vdash_\bot a : A]\!]; \iota_{\mathsf{L},\ell}; [\![\mathsf{L} \leq \mathsf{K}]\!] \\
&= [\![\Gamma \vdash_\bot a : A]\!]; \iota_{\mathsf{L},\ell}; [\![\mathsf{L} \leq \mathsf{K}]\!]
\end{aligned} \tag{196}$$

  It hence suffices to show that $\iota_{\mathsf{L},\ell}; [\![\mathsf{L} \leq \mathsf{K}]\!] = \iota_{\mathsf{K},\ell}$, which we can do by induction on $\mathsf{L} \leq \mathsf{K}$:
  - lwk-nil: this case yields a contradiction, since if $\mathsf{K} = \cdot$ it cannot define the label ℓ.
  - lwk-skip: we have $\mathsf{K} = \mathsf{K}', \kappa(B)$, and hence

$$\iota_{\mathsf{L}',\ell}; [\![\mathsf{L} \leq \mathsf{K}', \kappa(B)]\!] = \iota_{\mathsf{L}',\ell}; [\![\mathsf{L} \leq \mathsf{K}', \kappa(B)]\!]; \iota_l = \iota_{\mathsf{K}',\ell}; \iota_l = \iota_{\mathsf{K},\ell} \tag{197}$$

  - lwk-cons: we have $\mathsf{L} = \mathsf{L}', \kappa(B)$ and $\mathsf{K} = \mathsf{K}', \kappa(B)$.
    * If $\kappa = \ell$, then $B = A$ and

$$\iota_{(\mathsf{L}', \ell(A)),\ell}; [\![\mathsf{L}, \ell(A) \leq \mathsf{K}', \ell(A)]\!] = \iota_r; [\![\mathsf{L}' \leq \mathsf{K}']\!] + [\![A]\!] = \iota_r = \iota_{\mathsf{K},\ell} \tag{198}$$

    * Otherwise, we have by induction that

$$\begin{aligned}
\iota_{(\mathsf{L}', \kappa(B)),\ell}; [\![\mathsf{L}, \kappa(B) \leq \mathsf{K}', \kappa(B)]\!] &= \iota_{\mathsf{L}',\ell}; \iota_l; [\![\mathsf{L}' \leq \mathsf{K}']\!] + [\![B]\!] \\
&= \iota_{\mathsf{L}',\ell}; [\![\mathsf{L}' \leq \mathsf{K}']\!]; \iota_l \\
&= \iota_{\mathsf{K}',\ell}; \iota_l \qquad\qquad = \iota_{\mathsf{K},\ell}
\end{aligned} \tag{199}$$

- let_1-r: we have by induction that

$$\begin{aligned}
&[\![\Gamma \le \Delta]\!]; [\![\Delta \vdash \mathsf{let}\ x = a; r \rhd \mathsf{L}]\!]; [\![\mathsf{L} \le \mathsf{K}]\!] \\
&= [\![\Gamma \le \Delta]\!]; \mathsf{let}([\![\Delta \vdash_\epsilon a : A]\!]); [\![\Delta, x : A \vdash r \rhd \mathsf{L}]\!]; [\![\mathsf{L} \le \mathsf{K}]\!] \\
&= \mathsf{let}([\![\Gamma \le \Delta]\!]; [\![\Delta \vdash_\epsilon a : A]\!]); [\![\Gamma \le \Delta]\!] \otimes [\![A]\!]; [\![\Delta, x : A \vdash r \rhd \mathsf{L}]\!]; [\![\mathsf{L} \le \mathsf{K}]\!] \\
&= \mathsf{let}([\![\Gamma \vdash_\epsilon a : A]\!]); [\![\Gamma, x : A \le \Delta, x : A]\!]; [\![\Delta, x : A \vdash r \rhd \mathsf{L}]\!]; [\![\mathsf{L} \le \mathsf{K}]\!] \\
&= \mathsf{let}([\![\Gamma \vdash_\epsilon a : A]\!]); [\![\Gamma, x : A \vdash r \rhd \mathsf{K}]\!] \\
&= [\![\Gamma \vdash \mathsf{let}\ x = a; r \rhd \mathsf{K}]\!]
\end{aligned} \tag{200}$$

- let_2-r: we have by induction that

$$\begin{aligned}
&[\![\Gamma \le \Delta]\!]; [\![\Delta \vdash \mathsf{let}\ (x, y) = a; r \rhd \mathsf{L}]\!]; [\![\mathsf{L} \le \mathsf{K}]\!] \\
&= [\![\Gamma \le \Delta]\!]; \mathsf{let}([\![\Delta \vdash_\epsilon a : A \otimes B]\!]); \alpha; [\![\Delta, x : A, y : B \vdash r \rhd \mathsf{L}]\!]; [\![\mathsf{L} \le \mathsf{K}]\!] \\
&= \mathsf{let}([\![\Gamma \le \Delta]\!]; [\![\Delta \vdash_\epsilon a : A \otimes B]\!]); \alpha; [\![\Gamma \le \Delta]\!] \otimes [\![A]\!] \otimes [\![B]\!]; [\![\Delta, x : A, y : B \vdash r \rhd \mathsf{L}]\!]; [\![\mathsf{L} \le \mathsf{K}]\!] \\
&= \mathsf{let}([\![\Gamma \vdash_\epsilon a : A \otimes B]\!]); \alpha; [\![\Gamma, x : A, y : B \le \Delta, x : A, y : B]\!]; [\![\Delta, x : A, y : B \vdash r \rhd \mathsf{L}]\!]; [\![\mathsf{L} \le \mathsf{K}]\!] \\
&= \mathsf{let}([\![\Gamma \vdash_\epsilon a : A \otimes B]\!]); \alpha; [\![\Gamma, x : A, y : B \vdash r \rhd \mathsf{K}]\!]
\end{aligned} \tag{201}$$

- case-r: we have by induction that

$$\begin{aligned}
&[\![\Gamma \le \Delta]\!]; [\![\Delta \vdash \mathsf{case}\ a\ \{\iota_l\ x : r, \iota_r\ y : s\} \rhd \mathsf{L}]\!]; [\![\mathsf{L} \le \mathsf{K}]\!] \\
&= [\![\Gamma \le \Delta]\!]; \mathsf{let}([\![\Delta \vdash_\epsilon a : A]\!]); \delta^{-1}; \\
&\qquad [[\![\Delta, x : A \vdash s \rhd \mathsf{L}]\!], [\![\Delta, y : B \vdash t \rhd \mathsf{L}]\!]]; [\![\mathsf{L} \le \mathsf{K}]\!] \\
&= \mathsf{let}([\![\Gamma \le \Delta]\!]; [\![\Delta \vdash_\epsilon a : A]\!]); \delta^{-1}; [\\
&\qquad [\![\Gamma \le \Delta]\!] \otimes [\![A]\!]; [\![\Delta, x : A \vdash s \rhd \mathsf{L}]\!]; [\![\mathsf{L} \le \mathsf{K}]\!], \\
&\qquad [\![\Gamma \le \Delta]\!] \otimes [\![B]\!]; [\![\Delta, y : B \vdash t \rhd \mathsf{L}]\!]; [\![\mathsf{L} \le \mathsf{K}]\!]] \\
&= \mathsf{let}([\![\Gamma \le \Delta]\!]; [\![\Delta \vdash_\epsilon a : A]\!]); \delta^{-1}; [\\
&\qquad [\![\Gamma, x : A \le \Delta, x : A]\!]; [\![\Delta, x : A \vdash s \rhd \mathsf{L}]\!]; [\![\mathsf{L} \le \mathsf{K}]\!], \\
&\qquad [\![\Gamma, y : B \le \Delta, y : B]\!]; [\![\Delta, y : B \vdash t \rhd \mathsf{L}]\!]; [\![\mathsf{L} \le \mathsf{K}]\!]] \\
&= \mathsf{let}([\![\Gamma \vdash_\epsilon a : A]\!]); \delta^{-1}; [[\![\Gamma, x : A \vdash s \rhd \mathsf{K}]\!], [\![\Gamma, y : B \vdash t \rhd \mathsf{K}]\!]] \\
&= [\![\Gamma \vdash \mathsf{case}\ a\ \{\iota_l\ x : s, \iota_r\ y : t\} \rhd \mathsf{K}]\!]
\end{aligned} \tag{202}$$

- cfg: Let $L = \mathsf{lsem}_{\Delta,\mathsf{L}}((\ell_i(x_i) : \{t_i\},)_i)$ and $\mathsf{R} = (\ell_i(A_i),)_i$. We have by induction that

$$\begin{aligned}
&[\![\Gamma \le \Delta]\!]; L; [\![\mathsf{L} \le \mathsf{K}]\!] + \Sigma_i [\![A_i]\!] \\
&= [\![\Gamma \le \Delta]\!]; \delta_\Sigma^{-1}; [(\![\Delta, x_i : A_i \vdash t_i \rhd \mathsf{L}, \mathsf{R}]\!],)_i]; \alpha^+_{[\![\mathsf{L}]\!] + \Sigma_i [\![A_i]\!]}; [\![\mathsf{L} \le \mathsf{K}]\!] + \Sigma_i [\![A_i]\!] \\
&= \delta_\Sigma^{-1}; [[\![\Gamma \le \Delta]\!] \otimes [\![A_i]\!]; ([\![\Delta, x_i : A_i \vdash t_i \rhd \mathsf{L}, \mathsf{R}]\!]; \alpha_{[\![\mathsf{L}]\!] + [\![\mathsf{R}]\!]}; [\![\mathsf{L} \le \mathsf{K}]\!] + [\![\mathsf{R}]\!],)_i]; \alpha^+_{[\![\mathsf{L}]\!] + \Sigma_i [\![A_i]\!]} \\
&= \delta_\Sigma^{-1}; [([\![\Gamma, x_i : A_i \le \Delta, x_i : A_i]\!]; [\![\Delta, x_i : A_i \vdash t_i \rhd \mathsf{L}, \mathsf{R}]\!]; [\![\mathsf{L}, \mathsf{R} \le \mathsf{K}, \mathsf{R}]\!],)_i]; \alpha^+_{[\![\mathsf{L}]\!] + \Sigma_i [\![A_i]\!]} \\
&= \delta_\Sigma^{-1}; [([\![\Gamma, x_i : A_i \vdash t_i \rhd \mathsf{K}, \mathsf{R}]\!],)_i]; \alpha^+_{[\![\mathsf{L}]\!] + \Sigma_i [\![A_i]\!]} \\
&= \mathsf{lsem}_{\Gamma,\mathsf{K}}((\ell_i(x_i) : \{t_i\},)_i)
\end{aligned} \tag{203}$$

It follows that

$$\begin{aligned}
&[\![\Gamma \le \Delta]\!]; [\![\Delta \vdash r \text{ where } (\ell_i(x_i) : \{t_i\},)_i \rhd \mathsf{L}]\!]; [\![\mathsf{L} \le \mathsf{K}]\!] \\
&= [\![\Gamma \le \Delta]\!]; \mathsf{let}([\![\Delta \vdash r \rhd \mathsf{L}, \mathsf{R}]\!]); [\![\Delta]\!] \otimes \alpha^{+}_{[\![L]\!]+\Sigma_i[\![A_i]\!]}); \delta^{-1}; [\pi_r, \mathsf{rfix}(L)]; [\![\mathsf{L} \le \mathsf{K}]\!] \\
&= \mathsf{let}([\![\Gamma \le \Delta]\!]; [\![\Delta \vdash r \rhd \mathsf{L}, \mathsf{R}]\!]; \alpha^{+}_{[\![L]\!]+\Sigma_i[\![A_i]\!]}); \delta^{-1}; [\pi_r, [\![\Gamma \le \Delta]\!]; \mathsf{rfix}(L)]; [\![\mathsf{L} \le \mathsf{K}]\!] \\
&= \mathsf{let}([\![\Gamma \vdash r \rhd \mathsf{L}, \mathsf{R}]\!]; \alpha^{+}_{[\![L]\!]+\Sigma_i[\![A_i]\!]}); [\pi_r; [\![\mathsf{L} \le \mathsf{K}]\!], \mathsf{rfix}(\mathsf{lsem}_{\Gamma,\mathsf{K}}((\ell_i(x_i) : \{t_i\},)_i))] \\
&= \mathsf{let}([\![\Gamma \vdash r \rhd \mathsf{L}, \mathsf{R}]\!]; \alpha_{[\![\mathsf{L}]\!]+\Sigma_i[\![A_i]\!]}; [\![\mathsf{L} \le \mathsf{K}]\!] + \Sigma_i[\![A_i]\!]); [\pi_r, \mathsf{rfix}(\mathsf{lsem}_{\Gamma,\mathsf{K}}((\ell_i(x_i) : \{t_i\},)_i))] \\
&= \mathsf{let}([\![\Gamma \vdash r \rhd \mathsf{L}, \mathsf{R}]\!]; \alpha_{[\![\mathsf{L}]\!]+\Sigma_i[\![A_i]\!]}; [\![\mathsf{L} \le \mathsf{K}]\!] + \Sigma_i[\![A_i]\!]); [\pi_r, \mathsf{rfix}(\mathsf{lsem}_{\Gamma,\mathsf{K}}((\ell_i(x_i) : \{t_i\},)_i))] \\
&= \mathsf{let}([\![\Gamma \vdash r \rhd \mathsf{L}, \mathsf{R}]\!]; [\![\mathsf{L}, \mathsf{R} \le \mathsf{K}, \mathsf{R}]\!]; \alpha_{[\![\mathsf{K}]\!]+\Sigma_i[\![A_i]\!]}); [\pi_r, \mathsf{rfix}(\mathsf{lsem}_{\Gamma,\mathsf{K}}((\ell_i(x_i) : \{t_i\},)_i))] \\
&= \mathsf{let}([\![\Gamma \vdash r \rhd \mathsf{K}, \mathsf{R}]\!]; \alpha_{[\![\mathsf{K}]\!]+\Sigma_i[\![A_i]\!]}); [\pi_r, \mathsf{rfix}(\mathsf{lsem}_{\Gamma,\mathsf{K}}((\ell_i(x_i) : \{t_i\},)_i))] \\
&= [\![\Gamma \vdash r \text{ where } (\ell_i(x_i) : \{t_i\},)_i \rhd \mathsf{K}]\!]
\end{aligned} \tag{204}$$

Weakening for substitutions (e) and label substitututions (f) then follow by a trivial induction. □

Lemma C.3 (Substitution Projection). *For $\gamma : \Gamma \mapsto \Delta$, $\mathsf{eff}(\Delta) = \bot$ and $\Delta(x) = A$, we have*

$$[\![\gamma : \Gamma \mapsto \Delta]\!]; \pi_{\Delta,x} = [\![\Gamma \vdash_\epsilon [\gamma]x : A]\!] \tag{205}$$

Proof. We proceed by induction on Δ:

- If $\Delta = \cdot$, then $\Delta(x) = A$ is a contradiction.
- If $\Delta = \Delta', x : A$, then $\gamma = \gamma', x \mapsto [\gamma]x$, so we have

$$[\![\gamma : \Gamma \mapsto \Delta]\!]; \pi_{\Delta,x} = \Delta_{[\![\Gamma]\!]}; [\![\gamma' : \Gamma \mapsto \Delta']\!] \otimes [\![\Gamma \vdash_\bot [\gamma]x : A]\!]; \pi_r = [\![\Gamma \vdash_\epsilon [\gamma]x : A]\!] \tag{206}$$

  as desired.
- If $\Delta = \Delta', y : B$ (with $y \neq x$), then $\gamma = \gamma', y \mapsto [\gamma]y$, so by induction we have

$$\begin{aligned}
[\![\gamma : \Gamma \mapsto \Delta]\!]; \pi_{\Delta,x} &= \Delta_{[\![\Gamma]\!]}; [\![\gamma' : \Gamma \mapsto \Delta']\!] \otimes [\![\Gamma \vdash_\bot [\gamma]y : B]\!]; \pi_l; \pi_{\Delta',x} \\
&= [\![\gamma' : \Gamma \mapsto \Delta']\!]; \pi_{\Delta',x} = [\![\Gamma \vdash_\epsilon [\gamma']x : A]\!] = [\![\Gamma \vdash_\epsilon [\gamma]x : A]\!]
\end{aligned} \tag{207}$$

□

Theorem 5.9 (Soundness (Substitution)). *Given $[\![\gamma : \Gamma \mapsto \Delta]\!] : [\![\Gamma]\!] \to [\![\Delta]\!]$ pure, we have that*

(a) $[\![\Gamma \vdash_\epsilon [\gamma]a : A]\!] = [\![\gamma : \Gamma \mapsto \Delta]\!]; [\![\Delta \vdash_\epsilon a : A]\!]$
(b) $[\![\Gamma \vdash [\gamma]r \rhd \mathsf{L}]\!] = [\![\gamma : \Gamma \mapsto \Delta]\!]; [\![\Delta \vdash r \rhd \mathsf{L}]\!]$
(c) $[\![[\gamma]\rho : \Delta \mapsto \Xi]\!] = [\![\gamma : \Gamma \mapsto \Delta]\!]; [\![\rho : \Delta \mapsto \Xi]\!]$
(d) $[\![\Gamma \vdash [\gamma]\sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] = [\![\gamma : \Gamma \mapsto \Delta]\!]; [\![\Delta \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!]$

Proof. Fix $\gamma : \Gamma \mapsto \Delta$ with $\mathsf{eff}(\Delta) = \bot$. We will begin by showing the soundness of substitution for expressions (a): we proceed by induction on the derivation $\Delta \vdash_\epsilon e : E$:

- If $e = x$ is a variable, then by Lemma C.3, we have

$$[\![\gamma : \Gamma \mapsto \Delta]\!]; [\![\Delta \vdash_\epsilon x : A]\!] = [\![\gamma : \Gamma \mapsto \Delta]\!]; \pi_{\Delta,x} = [\![\Gamma \vdash_\epsilon [\gamma]x : A]\!] \tag{208}$$

  as desired.
- If $e = f\ a$ is an operation, then by induction we have that

$$\begin{aligned}
[\![\gamma : \Gamma \mapsto \Delta]\!]; [\![\Gamma \vdash_\epsilon f\ a : B]\!] &= [\![\gamma : \Gamma \mapsto \Delta]\!]; [\![\Delta \vdash_\epsilon a : A]\!]; [\![f \in \mathcal{I}_B(\epsilon, A)]\!] \\
&= [\![\Gamma \vdash_\epsilon [\gamma]a : A]\!]; [\![f \in \mathcal{I}_B(\epsilon, A)]\!] = [\![\Gamma \vdash_\epsilon [\gamma](f\ a) : B]\!]
\end{aligned} \tag{209}$$

  as desired. The cases for left injections, right injections, and abort are analogous

- If $e = (\text{let } x = a;\ b)$ is a unary let-binding, then by induction we have that

$$\begin{aligned}
&[\![\gamma : \Gamma \mapsto \Delta]\!] \otimes [\![A]\!]; [\![\Delta, x : A \vdash_\epsilon b : B]\!] \\
&= [\![(\gamma, x \mapsto x) : (\Gamma, x : A) \mapsto (\Delta, x : A)]\!]; [\![\Delta, x : A \vdash_\epsilon b : B]\!] \\
&= [\![\Gamma, x : A \vdash_\epsilon [\gamma, x \mapsto x]b : B]\!] = [\![\Gamma, x : A \vdash_\epsilon [\gamma]b : B]\!]
\end{aligned} \tag{210}$$

  as we can assume that x is a fresh variable. Hence, it follows that

$$\begin{aligned}
&[\![\gamma : \Gamma \mapsto \Delta]\!]; [\![\Delta \vdash_\epsilon \text{let } x = a;\ b : B]\!] \\
&= [\![\gamma : \Gamma \mapsto \Delta]\!]; \Delta_{[\![\Delta]\!]}; [\![\Delta]\!] \otimes [\![\Delta \vdash_\epsilon a : A]\!]; [\![\Delta, x : A \vdash_\epsilon b : B]\!] \\
&= \Delta_{[\![\Gamma]\!]}; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon [\gamma]a : A]\!]; [\![\gamma : \Gamma \mapsto \Delta]\!] \otimes [\![A]\!]; [\![\Delta, x : A \vdash_\epsilon b : B]\!] \\
&= \Delta_{[\![\Gamma]\!]}; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon [\gamma]a : A]\!]; [\![\Gamma, x : A \vdash_\epsilon [\gamma]b : B]\!] \\
&= [\![\Gamma \vdash_\epsilon [\gamma](\text{let } x = a;\ b) : B]\!]
\end{aligned} \tag{211}$$

  as desired.
- If $e = (a, b)$ is a pair, then by induction we have that

$$\begin{aligned}
&[\![\gamma : \Gamma \mapsto \Delta]\!]; [\![\Delta \vdash_\epsilon (a, b) : A \times B]\!] \\
&= [\![\gamma : \Gamma \mapsto \Delta]\!]; \Delta_{[\![\Delta]\!]}; [\![\Delta \vdash_\epsilon a : A]\!] \ltimes [\![\Delta \vdash_\epsilon b : B]\!] \\
&= \Delta_{[\![\Gamma]\!]}; ([\![\gamma : \Gamma \mapsto \Delta]\!] \otimes [\![\Delta \vdash_\epsilon a : A]\!]) \ltimes ([\![\gamma : \Gamma \mapsto \Delta]\!] \otimes [\![\Delta \vdash_\epsilon b : B]\!]) \\
&= \Delta_{[\![\Gamma]\!]}; [\![\Gamma \vdash_\epsilon [\gamma]a : A]\!] \ltimes [\![\Gamma \vdash_\epsilon [\gamma]b : B]\!] = [\![\Gamma \vdash_\epsilon [\gamma](a, b) : A \times B]\!]
\end{aligned} \tag{212}$$

- If $e = (\text{let } (x, y) = a;\ b)$ is a binary let-binding, then by induction we have that

$$\begin{aligned}
&[\![\gamma : \Gamma \mapsto \Delta]\!] \otimes ([\![A]\!] \otimes [\![B]\!]); \alpha; [\![\Delta, x : A, y : B \vdash_\epsilon b : C]\!] \\
&= \alpha; [\![(\gamma, x \mapsto x, y \mapsto y) : (\Gamma, x : A, y : B) \mapsto (\Delta, x : A, y : B)]\!]; [\![\Delta, x : A, y : B \vdash_\epsilon b : C]\!] \\
&= [\![\Gamma, x : A, y : B \vdash_\epsilon [\gamma, x \mapsto x, y \mapsto y]b : C]\!] = [\![\Gamma, x : A, y : B \vdash_\epsilon [\gamma]b : C]\!]
\end{aligned} \tag{213}$$

  as we can assume that x, y are fresh variables. Hence, it follows that

$$\begin{aligned}
&[\![\gamma : \Gamma \mapsto \Delta]\!]; [\![\Delta \vdash_\epsilon \text{let } (x, y) = a;\ b : C]\!] \\
&= [\![\gamma : \Gamma \mapsto \Delta]\!]; \Delta_{[\![\Delta]\!]}; [\![\Delta]\!] \otimes [\![\Delta \vdash_\epsilon a : A \otimes B]\!]; \alpha; [\![\Delta, x : A, y : B \vdash_\epsilon b : C]\!] \\
&= \Delta_{[\![\Gamma]\!]}; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon [\gamma]a : A \otimes B]\!]; [\![\gamma : \Gamma \mapsto \Delta]\!] \otimes ([\![A]\!] \otimes [\![B]\!]); \alpha; [\![\Delta, x : A, y : B \vdash_\epsilon b : C]\!] \\
&= \Delta_{[\![\Gamma]\!]}; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon [\gamma]a : A \otimes B]\!]; [\![\Gamma, x : A, y : B \vdash_\epsilon [\gamma]b : C]\!] \\
&= [\![\Gamma \vdash_\epsilon [\gamma](\text{let } (x, y) = a;\ b) : C]\!]
\end{aligned} \tag{214}$$

  as desired.
- If $e = \text{case } a\ \{\iota_l\ x : b, \iota_r\ y : c\}$ is a case-expression, then by induction we have that

$$\begin{aligned}
&[\![\gamma : \Gamma \mapsto \Delta]\!] \otimes [\![A]\!]; [\![\Delta, x : A \vdash_\epsilon b : C]\!] \\
&= [\![(\gamma, x \mapsto x) : (\Gamma, x : A) \mapsto (\Delta, x : A)]\!]; [\![\Delta, x : A \vdash_\epsilon b : C]\!] \\
&= [\![\Gamma, x : A \vdash_\epsilon [\gamma, x \mapsto x]b : C]\!] = [\![\Gamma, x : A \vdash_\epsilon [\gamma]b : C]\!]
\end{aligned} \tag{215}$$

  and

$$\begin{aligned}
&[\![\gamma : \Gamma \mapsto \Delta]\!] \otimes [\![B]\!]; [\![\Delta, y : B \vdash_\epsilon c : C]\!] \\
&= [\![(\gamma, y \mapsto y) : (\Gamma, y : B) \mapsto (\Delta, y : B)]\!]; [\![\Delta, y : B \vdash_\epsilon c : C]\!] \\
&= [\![\Gamma, y : B \vdash_\epsilon [\gamma, x \mapsto x]c : C]\!] = [\![\Gamma, y : B \vdash_\epsilon [\gamma]c : C]\!]
\end{aligned} \tag{216}$$

as we can assume that x, y are fresh variables. Hence, it follows that

$$\begin{aligned}
&[\![\gamma : \Gamma \mapsto \Delta]\!]; \Delta_{[\![\Delta]\!]}; [\![\Delta]\!] \otimes [\![\Delta \vdash_\epsilon a : A + B]\!]; \delta^{-1}_{[\![\Delta]\!]}; [[\![\Delta, x : A \vdash_\epsilon b : C]\!], [\![\Delta, y : B \vdash_\epsilon c : C]\!]] \\
&= \Delta_{[\![\Gamma]\!]}; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon [\gamma]a : A + B]\!]; \delta^{-1}_{[\![\Gamma]\!]}; \\
&\quad [[\![\gamma : \Gamma \mapsto \Delta]\!] \otimes [\![A]\!]; [\![\Delta, x : A \vdash_\epsilon b : C]\!], [\![\gamma : \Gamma \mapsto \Delta]\!] \otimes [\![B]\!]; [\![\Delta, y : B \vdash_\epsilon c : C]\!]] \\
&= \Delta_{[\![\Gamma]\!]}; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon [\gamma]a : A + B]\!]; \delta^{-1}_{[\![\Gamma]\!]}; [[\![\Gamma, x : A \vdash_\epsilon [\gamma]b : C]\!], [\![\Gamma, y : B \vdash_\epsilon [\gamma]c : C]\!]] \\
&= [\![\Gamma \vdash_\epsilon [\gamma](\text{case } a \ \{\iota_l \ x : b, \iota_r \ y : c\}) : C]\!]
\end{aligned} \tag{217}$$

as desired.

- If $e = ()$ is the null expression, since $[\![\gamma : \Gamma \mapsto \Delta]\!]$ is pure, the desired result holds trivially since $\mathbf{1}$ is the terminal object in the category of pure morphisms.

We may now prove the soundness of substitution for regions as follows: assuming $\Gamma \vdash r \rhd \mathsf{L}$, we proceed by induction on r as follows:

- If $r = \text{br } \ell \ a$, then we have that

$$\begin{aligned}
[\![\gamma : \Gamma \mapsto \Delta]\!]; [\![\Delta \vdash \text{br } \ell \ A \rhd \mathsf{L}]\!] &= [\![\gamma : \Gamma \mapsto \Delta]\!]; [\![\Delta \vdash_\bot a : A]\!]; \iota_{\mathsf{L},\ell} \\
&= [\![\Gamma \vdash_\bot [\gamma]a : A]\!]; \iota_{\mathsf{L},\ell} = [\![\Gamma \vdash [\gamma](\text{br } \ell \ a) \rhd \mathsf{L}]\!]
\end{aligned} \tag{218}$$

- If $r = (\text{let } x = a; t)$, then we have that, by induction,

$$\begin{aligned}
&[\![\gamma : \Gamma \mapsto \Delta]\!] \otimes [\![A]\!]; [\![\Delta, x : A \vdash t \rhd \mathsf{L}]\!] \\
&= [\![(\gamma, x \mapsto x) : (\Gamma, x : A) \mapsto (\Delta, x : A)]\!]; [\![\Delta, x : A \vdash t \rhd \mathsf{L}]\!] \\
&= [\![\Gamma, x : A \vdash [\gamma, x \mapsto x]t \rhd \mathsf{L}]\!] = [\![\Gamma, x : A \vdash [\gamma]t \rhd \mathsf{L}]\!]
\end{aligned} \tag{219}$$

since x can be taken to be a free variable. Hence,

$$\begin{aligned}
&[\![\gamma : \Gamma \mapsto \Delta]\!]; [\![\Delta \vdash \text{let } x = a; \ t \rhd \mathsf{L}]\!] \\
&= [\![\gamma : \Gamma \mapsto \Delta]\!]; \Delta_{[\![\Delta]\!]}; [\![\Delta]\!] \otimes [\![\Delta \vdash_\epsilon a : A]\!]; [\![\Delta, x : A \vdash t \rhd \mathsf{L}]\!] \\
&= \Delta_{[\![\Gamma]\!]}; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon [\gamma]a : A]\!]; [\![\gamma : \Gamma \mapsto \Delta]\!] \otimes [\![A]\!]; [\![\Delta, x : A \vdash t \rhd \mathsf{L}]\!] \\
&= \Delta_{[\![\Gamma]\!]}; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon [\gamma]a : A]\!]; [\![\Gamma, x : A \vdash [\gamma]t \rhd \mathsf{L}]\!] \\
&= [\![\Gamma \vdash [\gamma](\text{let } x = a; \ t) \rhd \mathsf{L}]\!]
\end{aligned} \tag{220}$$

- If $r = (\text{let } (x, y) = a; t)$, then we have that, by induction,

$$\begin{aligned}
&[\![\gamma : \Gamma \mapsto \Delta]\!] \otimes ([\![A]\!] \otimes [\![B]\!]); \alpha; [\![\Delta, x : A, y : B \vdash t \rhd \mathsf{L}]\!] \\
&= \alpha; [\![(\gamma, x \mapsto x, y \mapsto y) : (\Gamma, x : A, y : B) \mapsto (\Delta, x : A, y : B)]\!]; [\![\Delta, x : A, y : B \vdash t \rhd \mathsf{L}]\!] \\
&= [\![\Gamma, x : A, y : B \vdash [\gamma, x \mapsto x, y \mapsto y]t \rhd \mathsf{L}]\!] = [\![\Gamma, x : A, y : B \vdash [\gamma]t \rhd \mathsf{L}]\!]
\end{aligned} \tag{221}$$

since x, y can be taken to be free variables. Hence,

$$\begin{aligned}
&[\![\gamma : \Gamma \mapsto \Delta]\!]; [\![\Delta \vdash \text{let } (x, y) = a; \ t \rhd \mathsf{L}]\!] \\
&= [\![\gamma : \Gamma \mapsto \Delta]\!]; \Delta_{[\![\Delta]\!]}; [\![\Delta]\!] \otimes [\![\Delta \vdash_\epsilon a : A \otimes B]\!]; \alpha; [\![\Delta, x : A, y : B \vdash t \rhd \mathsf{L}]\!] \\
&= \Delta_{[\![\Gamma]\!]}; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon [\gamma]a : A \otimes B]\!]; [\![\gamma : \Gamma \mapsto \Delta]\!] \otimes ([\![A]\!] \otimes [\![B]\!]); \alpha; [\![\Delta, x : A, y : B \vdash t \rhd \mathsf{L}]\!] \\
&= \Delta_{[\![\Gamma]\!]}; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon [\gamma]a : A \otimes B]\!]; [\![\Gamma, x : A, y : B \vdash [\gamma]t \rhd \mathsf{L}]\!] \\
&= [\![\Gamma \vdash [\gamma](\text{let } (x, y) = a; \ t) \rhd \mathsf{L}]\!]
\end{aligned} \tag{222}$$

- If $r = \text{case } a\ \{\iota_l\ x : s, \iota_r\ y : t\}$, then we have that, by induction

$$\begin{aligned}
&[\![\gamma : \Gamma \mapsto \Delta]\!] \otimes [\![A]\!]; [\![\Delta, x : A \vdash s \rhd \mathsf{L}]\!] \\
&= [\![(\gamma, x \mapsto x) : (\Gamma, x : A) \mapsto (\Delta, x : A)]\!]; [\![\Delta, x : A \vdash s \rhd \mathsf{L}]\!] \\
&= [\![\Gamma, x : A \vdash [\gamma, x \mapsto x]s \rhd \mathsf{L}]\!] = [\![\Gamma, x : A \vdash [\gamma]s \rhd \mathsf{L}]\!]
\end{aligned} \tag{223}$$

  and

$$\begin{aligned}
&[\![\gamma : \Gamma \mapsto \Delta]\!] \otimes [\![B]\!]; [\![\Delta, y : B \vdash t \rhd \mathsf{L}]\!] \\
&= [\![(\gamma, y \mapsto y) : (\Gamma, y : B) \mapsto (\Delta, y : B)]\!]; [\![\Delta, y : B \vdash t \rhd \mathsf{L}]\!] \\
&= [\![\Gamma, y : B \vdash [\gamma, y \mapsto y]t \rhd \mathsf{L}]\!] = [\![\Gamma, y : B \vdash [\gamma]t \rhd \mathsf{L}]\!]
\end{aligned} \tag{224}$$

  since x, y can be taken to be free variables. Hence,

$$\begin{aligned}
&[\![\gamma : \Gamma \mapsto \Delta]\!]; \Delta_{[\![\Delta]\!]}; [\![\Delta]\!] \otimes [\![\Delta \vdash_\epsilon a : A + B]\!]; \delta^{-1}_{[\![\Delta]\!]}; [[\![\Delta, x : A \vdash s \rhd \mathsf{L}]\!], [\![\Delta, y : B \vdash t \rhd \mathsf{L}]\!]] \\
&= \Delta_{[\![\Gamma]\!]}; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon [\gamma]a : A + B]\!]; \delta^{-1}_{[\![\Gamma]\!]}; \\
&\quad [[\![\gamma : \Gamma \mapsto \Delta]\!] \otimes [\![A]\!]; [\![\Delta, x : A \vdash s \rhd \mathsf{L}]\!], [\![\gamma : \Gamma \mapsto \Delta]\!] \otimes [\![B]\!]; [\![\Delta, y : B \vdash t \rhd \mathsf{L}]\!]] \\
&= \Delta_{[\![\Gamma]\!]}; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon [\gamma]a : A + B]\!]; \delta^{-1}_{[\![\Gamma]\!]}; [[\![\Gamma, x : A \vdash [\gamma]s \rhd \mathsf{L}]\!], [\![\Gamma, y : B \vdash [\gamma]t \rhd \mathsf{L}]\!]] \\
&= [\![\Gamma \vdash [\gamma](\text{case } a\ \{\iota_l\ x : s, \iota_r\ y : t\}) \rhd \mathsf{L}]\!]
\end{aligned} \tag{225}$$

  as desired.
- Assume $r = s$ where $(\ell_i(x_i) : \{t_i\},)_i$. Define $\mathsf{R} = (\ell_i(A_i),)_i$ and $S = [\![\gamma : \Gamma \mapsto \Delta]\!]$. We have by induction that, for all i,

$$\begin{aligned}
&[\![\gamma : \Gamma \mapsto \Delta]\!] \otimes [\![A_i]\!]; [\![\Delta, x_i : A_i \vdash t_i \rhd \mathsf{L}, \mathsf{R}]\!] \\
&= [\![(\gamma, x_i \mapsto x_i) : (\Gamma, x_i : A_i) \mapsto (\Delta, x_i : A_i)]\!]; [\![\Delta, x_i : A_i \vdash t_i \rhd \mathsf{L}, \mathsf{R}]\!] \\
&= [\![\Gamma, x_i : A_i \vdash [\gamma, x_i \mapsto x_i]t_i \rhd \mathsf{L}, \mathsf{R}]\!] = [\![\Gamma, x_i : A_i \vdash [\gamma]t_i \rhd \mathsf{L}, \mathsf{R}]\!]
\end{aligned} \tag{226}$$

  and therefore that

$$\begin{aligned}
&S \otimes \Sigma_i [\![A_i]\!]; \mathsf{lsem}_{\Delta,\mathsf{L}}((\ell_i(x_i) : \{t_i\},)_i) \\
&= S \otimes \Sigma_i [\![A_i]\!]; \delta^{-1}_\Sigma; [[\![\Delta, x_i : A_i \vdash t_i \rhd \mathsf{L}, \mathsf{R}]\!],]_i; \alpha^+_{[\![\mathsf{L}]\!] + \Sigma_i [\![A_i]\!]} \\
&= \delta^{-1}_\Sigma; [S \otimes [\![A_i]\!]; [\![\Delta, x_i : A_i \vdash t_i \rhd \mathsf{L}, \mathsf{R}]\!],]_i; \alpha^+_{[\![\mathsf{L}]\!] + \Sigma_i [\![A_i]\!]} \\
&= \delta^{-1}_\Sigma; [[\![\Gamma, x_i : A_i \vdash [\gamma]t_i \rhd \mathsf{L}, \mathsf{R}]\!],]_i; \alpha^+_{[\![\mathsf{L}]\!] + \Sigma_i [\![A_i]\!]} \\
&= \mathsf{lsem}_{\Gamma,\mathsf{L}}((\ell_i(x_i) : \{[\gamma]t_i\},)_i)
\end{aligned} \tag{227}$$

  It follows that

$$\begin{aligned}
&S; [\![\Delta \vdash s \text{ where } (\ell_i(x_i) : \{t_i\},)_i \rhd \mathsf{L}]\!] \\
&= S; \mathsf{let}([\![\Delta \vdash s \rhd \mathsf{L}, \mathsf{R}]\!]; \alpha^+_{[\![\mathsf{L}]\!] + \Sigma_i [\![A_i]\!]}); \delta^{-1}; [\pi_r, \mathsf{lsem}_{\Delta,\mathsf{L}}((\ell_i(x_i) : \{t_i\},)_i)] \\
&= \mathsf{let}(S; [\![\Delta \vdash s \rhd \mathsf{L}, \mathsf{R}]\!]; \alpha^+_{[\![\mathsf{L}]\!] + \Sigma_i [\![A_i]\!]}); S \otimes ([\![\mathsf{L}]\!] + \Sigma_i [\![A_i]\!]); \delta^{-1}; [\pi_r, \mathsf{lsem}_{\Delta,\mathsf{L}}((\ell_i(x_i) : \{t_i\},)_i)] \\
&= \mathsf{let}([\![\Gamma \vdash [\gamma]s \rhd \mathsf{L}, \mathsf{R}]\!]; \alpha^+_{[\![\mathsf{L}]\!] + \Sigma_i [\![A_i]\!]}); \delta^{-1}; [\pi_r, S \otimes ([\![\mathsf{L}]\!] + \Sigma_i [\![A_i]\!]); \mathsf{lsem}_{\Delta,\mathsf{L}}((\ell_i(x_i) : \{t_i\},)_i)] \\
&= \mathsf{let}([\![\Gamma \vdash [\gamma]s \rhd \mathsf{L}, \mathsf{R}]\!]; \alpha^+_{[\![\mathsf{L}]\!] + \Sigma_i [\![A_i]\!]}); \delta^{-1}; [\pi_r, \mathsf{lsem}_{\Gamma,\mathsf{L}}((\ell_i(x_i) : \{[\gamma]t_i\},)_i)] \\
&= [\![\Gamma \vdash [\gamma]s \text{ where } (\ell_i(x_i) : \{[\gamma]t_i\},)_i \rhd \mathsf{L}]\!]
\end{aligned} \tag{228}$$

  as desired.

Composition of substitutions then follows by a trivial induction, as does substitution for label substitutions. □

C.5 Label Substitution

Lemma C.4 (Label Substitution Injection). *For* $\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}$ *and* $\mathsf{L}(\ell) = A$, *we have*

$$[\![\Gamma]\!] \otimes \iota_{\mathsf{L},\ell}; [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] = [\![\Gamma, x : A \vdash [\gamma](\mathsf{br}\ \ell\ x) \rhd \mathsf{K}]\!] \tag{229}$$

Proof. We proceed by induction on L:

- If $\mathsf{L} = \cdot$, then $\mathsf{L}(\ell) = A$ is a contradiction
- If $\mathsf{L} = \mathsf{L}', \ell(A)$, then $\sigma = \sigma', \ell(x) \mapsto [\sigma](\mathsf{br}\ \ell\ x)$, so we have

$$\begin{aligned} &[\![\Gamma]\!] \otimes \iota_{\mathsf{L},\ell}; [\![\Gamma \vdash \sigma : \mathsf{L}, \ell(A) \rightsquigarrow \mathsf{K}]\!] \\ &= [\![\Gamma]\!] \otimes \iota_r; \delta^{-1}; [[\![\Gamma \vdash \sigma' : \mathsf{L} \rightsquigarrow \mathsf{K}]\!], [\![\Gamma, x : A \vdash [\sigma](\mathsf{br}\ \ell\ x) \rhd \mathsf{K}]\!]] \\ &= [\![\Gamma, x : A \vdash [\sigma](\mathsf{br}\ \ell\ x) \rhd \mathsf{K}]\!] \end{aligned} \tag{230}$$

- If $\mathsf{L} = \mathsf{L}', \kappa(A)$, then $\sigma = \sigma', \kappa(x) \mapsto [\sigma](\mathsf{br}\ \kappa\ x)$, so by induction we have

$$\begin{aligned} &[\![\Gamma]\!] \otimes \iota_{\mathsf{L},\ell}; [\![\Gamma \vdash \sigma : \mathsf{L}, \ell(A) \rightsquigarrow \mathsf{K}]\!] \\ &= [\![\Gamma]\!] \otimes (\iota_{\mathsf{L}',\ell}; \iota_l); \delta^{-1}; [[\![\Gamma \vdash \sigma' : \mathsf{L} \rightsquigarrow \mathsf{K}]\!], [\![\Gamma, x : B \vdash [\sigma](\mathsf{br}\ \kappa\ x) \rhd \mathsf{K}]\!]] \\ &= [\![\Gamma]\!] \otimes \iota_{\mathsf{L}',\ell}; [\![\Gamma \vdash \sigma' : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] \\ &= [\![\Gamma, x : A \vdash [\sigma'](\mathsf{br}\ \ell\ x) \rhd \mathsf{K}]\!] \\ &= [\![\Gamma, x : A \vdash [\sigma](\mathsf{br}\ \ell\ x) \rhd \mathsf{K}]\!] \end{aligned} \tag{231}$$

□

Lemma C.5 (Label Substitution Splitting). *For* $\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}$, *where* $\mathsf{L} = (\ell_i(A_i),)_i$, *we have*

$$[\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] = [\![\Gamma]\!] \otimes \alpha_{\Sigma_i [\![A_i]\!]}; \delta_\Sigma^{-1}; [[\![\Gamma, x_i : A_i \vdash \sigma\ \ell_i\ x_i \rhd \mathsf{K}]\!],]_i \tag{232}$$

and therefore

$$\delta_\Sigma^{-1}; [[\![\Gamma, x_i : A_i \vdash \sigma\ \ell_i\ x_i \rhd \mathsf{K}]\!],]_i = \alpha_{[\![\mathsf{L}]\!]}; [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] \tag{233}$$

In particular, we have that, given $\forall i, \Gamma, x_i : A_i \vdash t_i \rhd \mathsf{K}$,

$$[\![\Gamma \vdash (\ell_i(x_i) : \{A_i\},)_i) : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] = [\![\Gamma]\!] \otimes \alpha_{\Sigma_i [\![A_i]\!]}; \delta_\Sigma^{-1}; [[\![\Gamma, x_i : A_i \vdash t_i \rhd \mathsf{K}]\!],]_i \tag{234}$$

Lemma C.6 (Label Substitution Extension). *Given* $\sigma = \sigma_l, \sigma_r, \Gamma \vdash \sigma : \mathsf{L}, \mathsf{R} \rightsquigarrow \mathsf{L}', \mathsf{R}', \Gamma \vdash \sigma_l : \mathsf{L} \rightsquigarrow \mathsf{L}', \Gamma \vdash \sigma_r : \mathsf{R} \rightsquigarrow \mathsf{R}'$, *we have*

$$[\![\Gamma \vdash \sigma : \mathsf{L}, \mathsf{R} \rightsquigarrow \mathsf{L}', \mathsf{R}']\!] = [\![\Gamma]\!] \otimes \alpha_{[\![\mathsf{L}]\!]+[\![\mathsf{R}]\!]}; [\![\Gamma \vdash \sigma_l : \mathsf{L} \rightsquigarrow \mathsf{L}']\!] + [\![\Gamma \vdash \sigma_r : \mathsf{R} \rightsquigarrow \mathsf{R}']\!]; \alpha_{[\![\mathsf{L}',\mathsf{R}']\!]} \tag{235}$$

In particular, for $\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}$, *we have*

$$[\![\Gamma \vdash \sigma^{\downarrow} : \mathsf{R}, \mathsf{L} \rightsquigarrow \mathsf{R}, \mathsf{K}]\!] = [\![\Gamma]\!] \otimes \alpha^+_{[\![\mathsf{R}]\!]+[\![\mathsf{L}]\!]}; \delta^{-1}; \pi_r + [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!]; \alpha^+_{[\![\mathsf{R},\mathsf{K}]\!]} \tag{236}$$

$$[\![\Gamma \vdash \sigma^{\downarrow} : \mathsf{L}, \mathsf{R} \rightsquigarrow \mathsf{K}, \mathsf{R}]\!] = [\![\Gamma]\!] \otimes \alpha^+_{[\![\mathsf{L}]\!]+[\![\mathsf{R}]\!]}; \delta^{-1}; [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] + \pi_r; \alpha^+_{[\![\mathsf{K},\mathsf{R}]\!]} \tag{237}$$

since

$$\Gamma \vdash \mathsf{id} : \mathsf{R} \rightsquigarrow \mathsf{R} = \pi_r \tag{238}$$

Theorem 5.11 (Soundness (Label Substitution)). *Given* $\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}$, *we have*

(a) $[\![\Gamma \vdash [\sigma] r \rhd \mathsf{K}]\!] = \mathsf{let}([\![\Gamma \vdash r \rhd \mathsf{L}]\!]); [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!]$

(b) $[\![\Gamma \vdash [\sigma]\sigma' : \mathsf{M} \rightsquigarrow \mathsf{K}]\!] = \Delta_{[\![\Gamma]\!]} \otimes [\![\mathsf{L}]\!]; \alpha; [\![\Gamma]\!] \otimes [\![\Gamma \vdash \sigma' : \mathsf{M} \rightsquigarrow \mathsf{L}]\!]; [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!]$

Proof. Fix $\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}$. We begin by proving the soundness of label substitution for regions as follows: assuming $\Gamma \vdash r \rhd \mathsf{L}$, we proceed by induction on r as follows:

- If $r = \text{br } \ell\ a$, then by Lemma C.4 we have that

$$\begin{aligned}
[\![\Gamma \vdash [\sigma](\text{br } \ell\ a) \rhd \mathsf{K}]\!] &= [\![\Gamma \vdash [a/x][\sigma](\text{br } \ell\ x) \rhd \mathsf{K}]\!] \\
&= \text{let}([\![\Gamma \vdash_\perp a : A]\!]); [\![\Gamma, x : A \vdash [\sigma](\text{br } \ell\ x) \rhd \mathsf{K}]\!] \\
&= \text{let}([\![\Gamma \vdash_\perp a : A]\!]); [\![\Gamma]\!] \otimes \iota_{\mathsf{L},\ell}; [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] \\
&= \text{let}([\![\Gamma \vdash_\perp a : A]\!]; \iota_{\mathsf{L},\ell}); [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] \\
&= \text{let}([\![\Gamma \vdash \text{br } \ell\ a \rhd \mathsf{L}]\!]); [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!]
\end{aligned} \tag{239}$$

  as desired.
- If $r = (\text{let } x = a; t)$, then we have by induction that

$$\begin{aligned}
&[\![\Gamma \vdash [\sigma](\text{let } x = a; t) \rhd \mathsf{K}]\!] = [\![\Gamma \vdash \text{let } x = a; [\sigma]t \rhd \mathsf{K}]\!] \\
&= \text{let}([\![\Gamma \vdash_\epsilon a : A]\!]); [\![\Gamma, x : A \vdash [\sigma]t \rhd \mathsf{K}]\!] \\
&= \text{let}([\![\Gamma \vdash_\epsilon a : A]\!]); \text{let}([\![\Gamma, x : A \vdash t \rhd \mathsf{L}]\!]); [\![\Gamma, x : A \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] \\
&= \text{let}([\![\Gamma \vdash_\epsilon a : A]\!]); \text{let}([\![\Gamma, x : A \vdash t \rhd \mathsf{L}]\!]); \pi_l \otimes [\![\mathsf{L}]\!]; [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] \\
&= \text{let}(\text{let}([\![\Gamma \vdash_\epsilon a : A]\!]); [\![\Gamma, x : A \vdash t \rhd \mathsf{L}]\!]); [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] \\
&= \text{let}([\![\Gamma \vdash \text{let } x = a; t \rhd \mathsf{L}]\!]); [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!]
\end{aligned} \tag{240}$$

  as desired.
- If $r = (\text{let } (x, y) = a; t)$, then we have by induction that

$$\begin{aligned}
&[\![\Gamma \vdash [\sigma](\text{let } (x, y) = a; t) \rhd \mathsf{K}]\!] = [\![\Gamma \vdash \text{let } (x, y) = a; [\sigma]t \rhd \mathsf{K}]\!] \\
&= \text{let}([\![\Gamma \vdash_\epsilon a : A \otimes B]\!]); \alpha; [\![\Gamma, x : A, y : B \vdash [\sigma]t \rhd \mathsf{K}]\!] \\
&= \text{let}([\![\Gamma \vdash_\epsilon a : A \otimes B]\!]); \alpha; \text{let}([\![\Gamma, x : A, y : B \vdash t \rhd \mathsf{L}]\!]); [\![\Gamma, x : A, y : B \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] \\
&= \text{let}([\![\Gamma \vdash_\epsilon a : A \otimes B]\!]); \alpha; \text{let}([\![\Gamma, x : A, y : B \vdash t \rhd \mathsf{L}]\!]); (\pi_l; \pi_l) \otimes [\![\mathsf{L}]\!]; [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] \\
&= \text{let}(\text{let}([\![\Gamma \vdash_\epsilon a : A \otimes B]\!]); \alpha; [\![\Gamma, x : A, y : B \vdash t \rhd \mathsf{L}]\!]); [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] \\
&= \text{let}([\![\Gamma \vdash \text{let } (x, y) = a; t \rhd \mathsf{L}]\!]); [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!]
\end{aligned} \tag{241}$$

  as desired, since
- If $r = \text{case } a\ \{\iota_l\ x : s, \iota_r\ y : t\}$, then we have by induction that

$$\begin{aligned}
&[\![\Gamma \vdash [\sigma](\text{case } a\ \{\iota_l\ x : s, \iota_r\ y : t\}) \rhd \mathsf{K}]\!] = [\![\Gamma \vdash \text{case } a\ \{\iota_l\ x : [\sigma]s, \iota_r\ y : [\sigma]t\} \rhd \mathsf{K}]\!] \\
&= \text{let}([\![\Gamma \vdash_\epsilon a : A + B]\!]); \delta^{-1}; [[\![\Gamma, x : A \vdash [\sigma]s \rhd \mathsf{K}]\!], [\![\Gamma, y : B \vdash [\sigma]t \rhd \mathsf{K}]\!]] \\
&= \text{let}([\![\Gamma \vdash_\epsilon a : A + B]\!]); \delta^{-1}; [\\
&\qquad \text{let}([\![\Gamma, x : A \vdash s \rhd \mathsf{L}]\!]); [\![\Gamma, x : A \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!], \\
&\qquad \text{let}([\![\Gamma, y : B \vdash t \rhd \mathsf{L}]\!]); [\![\Gamma, y : B \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!]] \\
&= \text{let}([\![\Gamma \vdash_\epsilon a : A + B]\!]); \delta^{-1}; \\
&\qquad [\text{let}([\![\Gamma, x : A \vdash s \rhd \mathsf{L}]\!]); \pi_l \otimes [\![\mathsf{L}]\!], \text{let}([\![\Gamma, y : B \vdash t \rhd \mathsf{L}]\!]); \pi_l \otimes [\![\mathsf{L}]\!]]; [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] \\
&= \text{let}(\text{let}([\![\Gamma \vdash_\epsilon a : A + B]\!]); \delta^{-1}; [[\![\Gamma, x : A \vdash s \rhd \mathsf{L}]\!], [\![\Gamma, y : B \vdash t \rhd \mathsf{L}]\!]]); [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] \\
&= \text{let}([\![\Gamma \vdash \text{case } a\ \{\iota_l\ x : s, \iota_r\ y : t\} \rhd \mathsf{L}]\!]); [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!]
\end{aligned} \tag{242}$$

  as desired.

- If $r = s$ where $(\ell_i(x_i) : \{t_i\},)_i$, then we have by induction, taking $\mathsf{R} = (\ell_i(A_i),)_i$,

$$\begin{aligned}
&\mathsf{esem}_{\Gamma,\mathsf{K}}([\sigma^\uparrow]s) = [\![\Gamma \vdash [\sigma^\uparrow]s \rhd \mathsf{K}, \mathsf{R}]\!]; \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_i[\![A_i]\!]} \\
&= \mathsf{let}([\![\Gamma \vdash s \rhd \mathsf{L}, \mathsf{R}]\!]); [\![\Gamma \vdash \sigma^\uparrow : \mathsf{L}, \mathsf{R} \rightsquigarrow \mathsf{K}, \mathsf{R}]\!]; \alpha^+_{[\![\mathsf{K}]\!]+\Sigma_i[\![A_i]\!]} \\
&= \mathsf{let}([\![\Gamma \vdash s \rhd \mathsf{L}, \mathsf{R}]\!]); [\![\Gamma]\!] \otimes \alpha^+_{[\![\mathsf{L}]\!]+[\![\mathsf{R}]\!]}; \delta^{-1}; [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] + \pi_r; \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_i[\![A_i]\!]} \\
&= \mathsf{let}([\![\Gamma \vdash s \rhd \mathsf{L}, \mathsf{R}]\!]); [\![\Gamma]\!] \otimes \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_i[\![A_i]\!]}; \delta^{-1}; [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] + \pi_r \\
&= \mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(s)); \delta^{-1}; [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] + \pi_r
\end{aligned} \tag{243}$$

and

$$\begin{aligned}
&\mathsf{lsem}_{\Gamma,\mathsf{K}}((\ell_i(x_i) : \{[\sigma^\uparrow]t_i\},)_i) = \delta_\Sigma^{-1}; [[\![\Gamma, x_i : A_i \vdash [\sigma^\uparrow]t_i \rhd \mathsf{K}, \mathsf{R}]\!]]_i; \alpha^+_{[\![\mathsf{K}]\!]+\Sigma_i[\![A_i]\!]} \\
&= \delta_\Sigma^{-1}; [\mathsf{let}([\![\Gamma, x_i : A_i \vdash t_i \rhd \mathsf{L}, \mathsf{R}]\!]); [\![\Gamma, x_i : A_i \vdash \sigma^\uparrow : \mathsf{L}, \mathsf{R} \rightsquigarrow \mathsf{K}, \mathsf{R}]\!]]_i; \alpha^+_{[\![\mathsf{K}]\!]+\Sigma_i[\![A_i]\!]} \\
&= \delta_\Sigma^{-1}; [\mathsf{let}([\![\Gamma, x_i : A_i \vdash t_i \rhd \mathsf{L}, \mathsf{R}]\!]); \pi_l \otimes [\![\mathsf{L}, \mathsf{R}]\!]]_i; [\![\Gamma \vdash \sigma^\uparrow : \mathsf{L}, \mathsf{R} \rightsquigarrow \mathsf{K}, \mathsf{R}]\!]; \alpha^+_{[\![\mathsf{K}]\!]+\Sigma_i[\![A_i]\!]} \\
&= \delta_\Sigma^{-1}; [\mathsf{let}([\![\Gamma, x_i : A_i \vdash t_i \rhd \mathsf{L}, \mathsf{R}]\!]); \pi_l \otimes \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_j[\![A_j]\!]}]_i; \delta^{-1}; [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] + \pi_r \\
&= \mathsf{let}(\delta_\Sigma^{-1}; [[\![\Gamma, x_i : A_i \vdash t_i \rhd \mathsf{L}, \mathsf{R}]\!]; \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_j[\![A_j]\!]},]_i); \pi_l \otimes -; \delta^{-1}; [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] + \pi_r \\
&= \mathsf{let}(\delta_\Sigma^{-1}; [[\![\Gamma, x_i : A_i \vdash t_i \rhd \mathsf{L}, \mathsf{R}]\!],]_i; \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_i[\![A_i]\!]}); \pi_l \otimes -; \delta^{-1}; [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] + \pi_r \\
&= \mathsf{let}(\mathsf{lsem}_{\Gamma,\mathsf{L}}((\ell_i(x_i) : \{t_i\},)_i)); \pi_l \otimes -; \delta^{-1}; [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] + \pi_r
\end{aligned} \tag{244}$$

Letting $L = \mathsf{lsem}_{\Gamma,\mathsf{L}}((\ell_i(x_i) : \{t_i\},)_i)$ and $S = [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!]$, we have that

$$\begin{aligned}
&\mathsf{rcase}(\mathsf{lsem}_{\Gamma,\mathsf{K}}((\ell_i(x_i) : \{[\sigma^\uparrow]t_i\},)_i)) \\
&= \mathsf{let}(\mathsf{let}(L); \pi_l \otimes ([\![\mathsf{L}]\!] + \Sigma_i[\![A_i]\!]); \delta^{-1}; S + \pi_r); \pi_l \otimes ([\![\mathsf{K}]\!] + \Sigma_i[\![A_i]\!]); \delta^{-1} \\
&= \mathsf{let}(\mathsf{let}(L)); ([\![\Gamma]\!] \otimes \Sigma_i[\![A_i]\!]) \otimes (\pi_l \otimes ([\![\mathsf{L}]\!] + \Sigma_i[\![A_i]\!]); \delta^{-1}; S + \pi_r); \pi_l \otimes ([\![\mathsf{K}]\!] + \Sigma_i[\![A_i]\!]); \delta^{-1} \\
&= \mathsf{let}(L); \Delta_{[\![\Gamma]\!] \otimes \Sigma_i[\![A_i]\!]} \otimes -; \alpha; ([\![\Gamma]\!] \otimes \Sigma_i[\![A_i]\!]) \otimes (\pi_l \otimes ([\![\mathsf{L}]\!] + \Sigma_i[\![A_i]\!]); \delta^{-1}; S + \pi_r); \pi_l \otimes -; \delta^{-1} \\
&= \mathsf{let}(L); (\pi_l; \Delta_{[\![\Gamma]\!]}) \otimes ([\![\mathsf{L}]\!] + \Sigma_i[\![A_i]\!]); \alpha; [\![\Gamma]\!] \otimes (\delta^{-1}; S + \pi_r); \delta^{-1} \\
&= \mathsf{let}(L); (\pi_l; \Delta_{[\![\Gamma]\!]}) \otimes ([\![\mathsf{L}]\!] + \Sigma_i[\![A_i]\!]); \alpha; [\![\Gamma]\!] \otimes \delta^{-1}; \delta^{-1}; ([\![\Gamma]\!] \otimes [\![\Gamma]\!]) \otimes S + ([\![\Gamma]\!] \otimes [\![\Gamma]\!]) \otimes \pi_r \\
&= \mathsf{let}(L); \pi_l \otimes ([\![\mathsf{L}]\!] + \Sigma_i[\![A_i]\!]); \delta^{-1}; (\Delta_{[\![\Gamma]\!]} \otimes [\![\mathsf{L}]\!]; \alpha; [\![\Gamma]\!] \otimes S) + (\Delta_{[\![\Gamma]\!]} \otimes \Sigma_i[\![A_i]\!]; \alpha; [\![\Gamma]\!] \otimes \pi_r) \\
&= \mathsf{rcase}(L); \Delta_{[\![\Gamma]\!]} \otimes [\![\mathsf{L}]\!]; \alpha; [\![\Gamma]\!] \otimes S + [\![\Gamma]\!] \otimes \Sigma_i[\![A_i]\!]
\end{aligned} \tag{245}$$

implying by naturality that

$$\begin{aligned}
\mathsf{rfix}(\mathsf{lsem}_{\Gamma,\mathsf{K}}((\ell_i(x_i) : \{[\sigma^\uparrow]t_i\},)_i)) &= (\mathsf{rcase}(L); \Delta_{[\![\Gamma]\!]} \otimes [\![\mathsf{L}]\!]; \alpha; [\![\Gamma]\!] \otimes S + [\![\Gamma]\!] \otimes \Sigma_i[\![A_i]\!])^\dagger; \pi_r \\
&= (\mathsf{rcase}(L))^\dagger; \Delta_{[\![\Gamma]\!]} \otimes [\![\mathsf{L}]\!]; \alpha; [\![\Gamma]\!] \otimes S; \pi_r \\
&= (\mathsf{rcase}(L))^\dagger; S
\end{aligned} \tag{246}$$

Hence, we have that

$$\begin{aligned}
&[\![\Gamma \vdash [\sigma](s \text{ where } (\ell_i(x_i) : \{t_i\},)_i) \rhd \mathsf{K}]\!] = [\![\Gamma \vdash [\sigma^\dagger]s \text{ where } (\ell_i(x_i) : \{[\sigma^\dagger]t_i\},)_i \rhd \mathsf{K}]\!] \\
&= \mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{K}}([\sigma^\dagger]s)); \delta^{-1}; [\pi_r, \mathsf{rfix}(\mathsf{lsem}_{\Gamma,\mathsf{K}}((\ell_i(x_i) : \{[\sigma^\dagger]t_i\},)_i))] \\
&= \mathsf{let}(\mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(s)); \delta^{-1}; S + \pi_r); \delta^{-1}; [\pi_r, (\mathsf{rcase}(L))^\dagger; S] \\
&= \mathsf{let}(\mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(s))); [\![\Gamma]\!] \otimes (\delta^{-1}; S + \pi_r); \delta^{-1}; [\pi_r, (\mathsf{rcase}(L))^\dagger; S] \\
&= \mathsf{let}(\mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(s))); [\![\Gamma]\!] \otimes \delta^{-1}; \delta^{-1}; [[\![\Gamma]\!] \otimes S; \pi_r, [\![\Gamma]\!] \otimes \pi_r; (\mathsf{rcase}(L))^\dagger; S] \\
&= \mathsf{let}(\mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(s))); [\![\Gamma]\!] \otimes \delta^{-1}; \delta^{-1}; [\pi_r, [\![\Gamma]\!] \otimes \pi_r; (\mathsf{rcase}(L))^\dagger]; S \\
&= \mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(s)); \Delta_{[\![\Gamma]\!]} \otimes ([\![\mathsf{L}]\!] + \Sigma_i [\![A_i]\!]); \alpha; [\![\Gamma]\!] \otimes \delta^{-1}; \delta^{-1}; [\pi_r, [\![\Gamma]\!] \otimes \pi_r; (\mathsf{rcase}(L))^\dagger]; S \\
&= \mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(s)); \delta^{-1}; [\Delta_{[\![\Gamma]\!]} \otimes [\![\mathsf{L}]\!]; \alpha; \pi_r, \Delta_{[\![\Gamma]\!]} \otimes \Sigma_i [\![A_i]\!]; \alpha; [\![\Gamma]\!] \otimes \pi_r; (\mathsf{rcase}(L))^\dagger]; S \\
&= \mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(s)); \delta^{-1}; [\Delta_{[\![\Gamma]\!]} \otimes [\![\mathsf{L}]\!]; \alpha; [\![\Gamma]\!] \otimes \pi_r, (\mathsf{rcase}(L))^\dagger]; S \\
&= \mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(s)); \delta^{-1}; [\Delta_{[\![\Gamma]\!]} \otimes [\![\mathsf{L}]\!]; \alpha; [\![\Gamma]\!] \otimes \pi_r, \mathsf{let}(\mathsf{rfix}(L)); \pi_l \otimes ([\![\mathsf{L}]\!] + \Sigma_i [\![A_i]\!])]; S \\
&= \mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(s)); \delta^{-1}; [\Delta_{[\![\Gamma]\!]} \otimes [\![\mathsf{L}]\!]; \alpha; [\![\Gamma]\!] \otimes \pi_r, \Delta_{[\![\Gamma]\!]} \otimes \Sigma_i [\![A_i]\!]; \alpha; [\![\Gamma]\!] \otimes \mathsf{rfix}(L)]; S \\
&= \mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(s)); \Delta_{[\![\Gamma]\!]} \otimes ([\![\mathsf{L}]\!] + \Sigma_i [\![A_i]\!]); \alpha; [\![\Gamma]\!] \otimes \delta^{-1}; \delta^{-1}; [[\![\Gamma]\!] \otimes \pi_r, [\![\Gamma]\!] \otimes \mathsf{rfix}(L)]; S \\
&= \mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(s)); \Delta_{[\![\Gamma]\!]} \otimes ([\![\mathsf{L}]\!] + \Sigma_i [\![A_i]\!]); \alpha; [\![\Gamma]\!] \otimes (\delta^{-1}; [\pi_r, \mathsf{rfix}(L)]); S \\
&= \mathsf{let}(\mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(s))); [\![\Gamma]\!] \otimes (\delta^{-1}; [\pi_r, \mathsf{rfix}(L)]); S \\
&= \mathsf{let}(\mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(s)); \delta^{-1}; [\pi_r, \mathsf{rfix}(L)]); S \\
&= \mathsf{let}([\![\Gamma \vdash s \text{ where } (\ell_i(x_i) : \{t_i\},)_i \rhd \mathsf{L}]\!]); [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!]
\end{aligned} \tag{247}$$

as desired.

Composition of label substitutions then follows by a trivial induction. □

C.6 Equational Theory

Theorem 5.12 (Soundness (Equational Theory)). *We have that*

(a) $\Gamma \vdash_\epsilon a \approx a' : A \implies [\![\Gamma \vdash_\epsilon a : A]\!] = [\![\Gamma \vdash_\epsilon a' : A]\!]$
(b) $\Gamma \vdash r \approx r' \rhd \mathsf{L} \implies [\![\Gamma \vdash r \rhd \mathsf{L}]\!] = [\![\Gamma \vdash r' \rhd \mathsf{L}]\!]$
(c) $\gamma \approx \gamma' : \Gamma \mapsto \Delta \implies [\![\gamma : \Gamma \mapsto \Delta]\!] = [\![\gamma' : \Gamma \mapsto \Delta]\!]$
(d) $\sigma \approx \sigma' \vdash \Gamma : \mathsf{L} \rightsquigarrow \mathsf{K} \implies [\![\Gamma \vdash \sigma : \mathsf{L} \rightsquigarrow \mathsf{K}]\!] = [\![\Gamma \vdash \sigma' : \mathsf{L} \rightsquigarrow \mathsf{K}]\!]$

Proof. We begin our proof by showing soundness of the equational theory for expressions, i.e. (a), by rule induction.

- *Congruence*: these follow trivially by induction
- initial, terminal: both of these follow trivially from the universal property of the initial/terminal object, respectively.
- let_1-β: this follows directly from Corollary 5.10
- let_1-η: we have

$$\begin{aligned}
&[\![\Gamma \vdash_\epsilon \mathsf{let}\ x = a;\ x : A]\!] \\
&= \Delta_{[\![\Gamma]\!]}; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon a : A]\!]; [\![\Gamma, x : A \vdash_\epsilon x : A]\!] \\
&= \Delta_{[\![\Gamma]\!]}; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon a : A]\!]; \pi_r \\
&= [\![\Gamma \vdash_\epsilon a : A]\!]
\end{aligned}$$

as desired.

- let_1-op: we have

$$\begin{aligned}
&[\![\Gamma \vdash_\epsilon \mathsf{let}\ x = a;\ \mathsf{let}\ y = f\ x;\ c : C]\!] \\
&= \mathsf{let}([\![\Gamma \vdash_\epsilon a : A]\!]); \mathsf{let}([\![\Gamma, x : A \vdash_\epsilon f\ x : B]\!]); [\![\Gamma, x : A, y : B \vdash_\epsilon c : C]\!] \\
&= \mathsf{let}([\![\Gamma \vdash_\epsilon a : A]\!]); \mathsf{let}(\pi_r; [\![f]\!]); \pi_l \otimes [\![B]\!]; [\![\Gamma, y : B \vdash_\epsilon c : C]\!] \\
&= \mathsf{let}(\mathsf{let}([\![\Gamma \vdash_\epsilon a : A]\!]); \pi_r; [\![f]\!]); [\![\Gamma, y : B \vdash_\epsilon c : C]\!] \\
&= \mathsf{let}([\![\Gamma \vdash_\epsilon a : A]\!]; [\![f]\!]); [\![\Gamma, y : B \vdash_\epsilon c : C]\!] \\
&= [\![\Gamma \vdash_\epsilon \mathsf{let}\ y = f\ a;\ c : C]\!]
\end{aligned}$$

as desired. The let_1-abort case is analogous.

- let_1-let_1: we have that

$$\begin{aligned}
&[\![\Gamma \vdash_\epsilon \mathsf{let}\ y = (\mathsf{let}\ x = a;\ b;\ c) : C]\!] \\
&= \mathsf{let}(\mathsf{let}([\![\Gamma \vdash_\epsilon a : A]\!]); [\![\Gamma, x : A \vdash_\epsilon b : B]\!]); [\![\Gamma, y : B \vdash_\epsilon c : C]\!] \\
&= \mathsf{let}([\![\Gamma \vdash_\epsilon a : A]\!]); \mathsf{let}([\![\Gamma, x : A \vdash_\epsilon b : B]\!]); \pi_l \otimes [\![B]\!]; [\![\Gamma, y : B \vdash_\epsilon c : C]\!] \\
&= \mathsf{let}([\![\Gamma \vdash_\epsilon a : A]\!]); \mathsf{let}([\![\Gamma, x : A \vdash_\epsilon b : B]\!]); [\![\Gamma, x : A, y : B \vdash_\epsilon c : C]\!] \\
&= [\![\Gamma \vdash_\epsilon \mathsf{let}\ x = a;\ \mathsf{let}\ y = b;\ c : C]\!]
\end{aligned}$$

as desired.

- let_1-let_2: we have that

$$\begin{aligned}
&[\![\Gamma \vdash_\epsilon \mathsf{let}\ z = (\mathsf{let}\ (x, y) = e;\ c);\ d : D]\!] \\
&= \mathsf{let}(\mathsf{let}([\![\Gamma \vdash_\epsilon e : A \otimes B]\!]); \alpha; [\![\Gamma, x : A, y : B \vdash_\epsilon c : C]\!]); [\![\Gamma, z : C \vdash_\epsilon d : D]\!] \\
&= \mathsf{let}(\Gamma \vdash_\epsilon e : A \otimes B); \alpha; \mathsf{let}([\![\Gamma, x : A, y : B \vdash_\epsilon c : C]\!]); (\pi_l; \pi_l) \otimes [\![\mathbf{C}]\!]; [\![\Gamma, z : C \vdash_\epsilon d : D]\!] \\
&= \mathsf{let}(\Gamma \vdash_\epsilon e : A \otimes B); \alpha; \mathsf{let}([\![\Gamma, x : A, y : B \vdash_\epsilon c : C]\!]); [\![\Gamma, x : A, y : B, z : C \vdash_\epsilon d : D]\!] \\
&= [\![\Gamma \vdash_\epsilon \mathsf{let}\ (x, y) = e;\ \mathsf{let}\ z = c;\ d : D]\!]
\end{aligned}$$

as desired.

- let_1-case: follows from the properties of the coproduct; in particular, we have that

$$
\begin{aligned}
&[\![\Gamma \vdash_\epsilon \text{case } e\ \{\iota_l\ x : \text{let } z = a;\ d, \iota_r\ y : \text{let } z = b;\ d\} : D]\!] \\
&= \Delta; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon e : A + B]\!]; \delta^{-1}; [\\
&\quad \Delta; [\![\Gamma, x : A]\!] \otimes [\![\Gamma, x : A \vdash_\epsilon a : C]\!]; [\![\Gamma, x : A, z : C \vdash_\epsilon d : D]\!], \\
&\quad \Delta; [\![\Gamma, y : B]\!] \otimes [\![\Gamma, y : B \vdash_\epsilon b : C]\!]; [\![\Gamma, y : B, z : C \vdash_\epsilon d : D]\!]] \\
&= \Delta; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon e : A + B]\!]; \delta^{-1}; [\\
&\quad \Delta \otimes [\![A]\!]; \alpha; [\![\Gamma]\!] \otimes [\![\Gamma, x : A \vdash_\epsilon a : C]\!]; [\![\Gamma, z : C \vdash_\epsilon d : D]\!], \\
&\quad \Delta \otimes [\![B]\!]; \alpha; [\![\Gamma]\!] \otimes [\![\Gamma, y : B \vdash_\epsilon b : C]\!]; [\![\Gamma, z : C \vdash_\epsilon d : D]\!]] \\
&= \Delta; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon e : A + B]\!]; \delta^{-1}; [\\
&\quad \Delta \otimes [\![A]\!]; \alpha; [\![\Gamma]\!] \otimes [\![\Gamma, x : A \vdash_\epsilon a : C]\!], \\
&\quad \Delta \otimes [\![B]\!]; \alpha; [\![\Gamma]\!] \otimes [\![\Gamma, y : B \vdash_\epsilon b : C]\!]]; [\![\Gamma, z : C \vdash_\epsilon d : D]\!] \\
&= \Delta; \Delta \otimes [\![\Gamma \vdash_\epsilon e : A + B]\!]; \delta^{-1}; \\
&\quad [\alpha; [\![\Gamma]\!] \otimes [\![\Gamma, x : A \vdash_\epsilon a : C]\!], \alpha; [\![\Gamma]\!] \otimes [\![\Gamma, y : B \vdash_\epsilon b : C]\!]]; [\![\Gamma, z : C \vdash_\epsilon d : D]\!] \\
&= \Delta; \Delta \otimes [\![\Gamma \vdash_\epsilon e : A + B]\!]; \alpha; [\![\Gamma]\!] \otimes (\delta^{-1}; [[\![\Gamma, x : A \vdash_\epsilon a : C]\!], [\![\Gamma, y : B \vdash_\epsilon b : C]\!]]); \\
&\quad [\![\Gamma, z : C \vdash_\epsilon d : D]\!] \\
&= \Delta; [\![\Gamma]\!] \otimes (\Delta; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon e : A + B]\!]; \delta^{-1}; [[\![\Gamma, x : A \vdash_\epsilon a : C]\!], [\![\Gamma, y : B \vdash_\epsilon b : C]\!]]); \\
&\quad [\![\Gamma, z : C \vdash_\epsilon d : D]\!] \\
&= [\![\Gamma \vdash_\epsilon \text{let } z = (\text{case } e\ \{\iota_l\ x : a, \iota_r\ y : b\});\ d : D]\!]
\end{aligned}
$$

- let_2-bind: we have

$$
\begin{aligned}
&[\![\Gamma \vdash_\epsilon \text{let } z = e;\ \text{let } (x, y) = z;\ c : C]\!] \\
&= \text{let}([\![\Gamma \vdash_\epsilon e : A \otimes B]\!]); \text{let}([\![\Gamma, z : A \otimes B \vdash_\epsilon z : A \otimes B]\!]); \alpha; [\![\Gamma, z : A \otimes B, x : A, y : B \vdash_\epsilon c : C]\!] \\
&= \text{let}([\![\Gamma \vdash_\epsilon e : A \otimes B]\!]); \text{let}(\pi_r); \pi_l \otimes ([\![A]\!] \otimes [\![B]\!]); \alpha; [\![\Gamma, x : A, y : B \vdash_\epsilon c : C]\!] \\
&= \text{let}(\text{let}([\![\Gamma \vdash_\epsilon e : A \otimes B]\!]); \pi_r); \alpha; [\![\Gamma, x : A, y : B \vdash_\epsilon c : C]\!] \\
&= \text{let}([\![\Gamma \vdash_\epsilon e : A \otimes B]\!]); \alpha; [\![\Gamma, x : A, y : B \vdash_\epsilon c : C]\!] \\
&= [\![\Gamma \vdash_\epsilon \text{let } (x, y) = e;\ c : C]\!]
\end{aligned}
$$

- let_2-η: follows from the properties of the product; in particular, we have that

$$
\begin{aligned}
&[\![\Gamma \vdash_\epsilon \text{let } (x, y) = e;\ (x, y) : A \otimes B]\!] \\
&= \Delta; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon e : A \otimes B]\!]; \Delta; [\![\Gamma, x : A, y : B \vdash_\perp x : A]\!] \otimes [\![\Gamma, x : A, y : B \vdash_\perp y : B]\!] \\
&= \Delta; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon e : A \otimes B]\!]; \Delta; (\pi_l; \pi_r) \otimes \pi_r = [\![\Gamma \vdash_\epsilon e : A \otimes B]\!]
\end{aligned}
$$

- case-inl: follows from the properties of the coproduct and inverse distributor; in particular, we have that

$$
\begin{aligned}
&[\![\Gamma \vdash_\epsilon \text{case } \iota_l\ a\ \{\iota_l\ x : c, \iota_r\ y : d\} : C]\!] \\
&= \Delta; [\![\Gamma]\!] \otimes ([\![\Gamma \vdash_\epsilon a : A]\!]; \iota_l); \delta^{-1}; [[\![\Gamma, x : A \vdash_\epsilon c : C]\!], [\![\Gamma, y : B \vdash_\epsilon d : C]\!]] \\
&= \Delta; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon a : A]\!]; \iota_l; [[\![\Gamma, x : A \vdash_\epsilon c : C]\!], [\![\Gamma, y : B \vdash_\epsilon d : C]\!]] \\
&= \Delta; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon a : A]\!]; \iota_l; [\![\Gamma, x : A \vdash_\epsilon c : C]\!] \\
&= [\![\Gamma \vdash_\epsilon \text{let } x = a;\ c : C]\!]
\end{aligned}
$$

We can validate case-inr analogously

- case-η: follows from the properties of the coproduct and distributor; in particular, we have

$$\begin{aligned}
&[\![\Gamma \vdash_\epsilon \mathsf{case}\ e\ \{\iota_l\ x : \iota_l\ x, \iota_r\ y : \iota_r\ y\} : A + B]\!] \\
&= \Delta; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon e : A + B]\!]; \delta^{-1}; [[\![\Gamma, x : A \vdash_\epsilon x : A]\!]; \iota_l, [\![\Gamma, y : B \vdash_\epsilon y : B; \iota_r]\!]] \\
&= \Delta; [\![\Gamma]\!] \otimes [\![\Gamma \vdash_\epsilon e : A + B]\!]; \delta^{-1}; (\pi_r + \pi_r) = [\![\Gamma \vdash_\epsilon e : A + B]\!]
\end{aligned}$$

We now proceed to tackle the equational theory for regions r in the same manner. In particular, we proceed by rule induction as follows:

- cfg-β_1: Define $P = [\![\Gamma \vdash_\perp a : A_k]\!]$, $L = \mathsf{lsem}_{\Gamma,\mathsf{L}}((\ell_i(x_i) : \{t_i\},)_i)$ and $\mathsf{R} = (\ell_i(A_i),)_i$. We have that

$$\begin{aligned}
&[\![\Gamma \vdash \mathsf{br}\ \ell_k\ a\ \mathsf{where}\ (\ell_i(x_i) : \{t_i\},)_i \rhd \mathsf{L}]\!] \\
&= \mathsf{let}(P; \iota_{(\mathsf{L},\mathsf{R}),\ell_k}; \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_i[\![A_i]\!]}); \delta^{-1}; [\pi_r, \mathsf{rfix}(L)] \\
&= \mathsf{let}(P; \iota_k); [\![\Gamma]\!] \otimes \iota_r; \delta^{-1}; [\pi_r, \mathsf{rfix}(L)] \\
&= \mathsf{let}(P; \iota_k); \mathsf{rfix}(L) \\
&= \mathsf{let}(P; \iota_k); \mathsf{rcase}(L); [\pi_r, \mathsf{rfix}(L)] \\
&= \mathsf{let}(P); \mathsf{rcase}([\![\Gamma]\!] \otimes \iota_k; L); [\pi_r, \mathsf{rfix}(L)] \\
&= \mathsf{let}(P); \mathsf{rcase}([\![\Gamma, x_k : A_k \vdash t_k \rhd \mathsf{L}, \mathsf{R}]\!]; \alpha^+_{[\![L]\!]+\Sigma_i[\![A_i]\!]}); [\pi_r, \mathsf{rfix}(L)] \\
&= \mathsf{let}(P); \mathsf{rlet}([\![\Gamma, x_k : A_k \vdash t_k \rhd \mathsf{L}, \mathsf{R}]\!]; \alpha^+_{[\![L]\!]+\Sigma_i[\![A_i]\!]}); \delta^{-1}; [\pi_r, \mathsf{rfix}(L)] \\
&= \mathsf{let}(\mathsf{let}(P); [\![\Gamma, x_k : A_k \vdash t_k \rhd \mathsf{L}, \mathsf{R}]\!]; \alpha^+_{[\![L]\!]+\Sigma_i[\![A_i]\!]}); \delta^{-1}; [\pi_r, \mathsf{rfix}(L)] \\
&= \mathsf{let}([\![\Gamma \vdash \mathsf{let}\ x_k = a; t_k \rhd \mathsf{L}, \mathsf{R}]\!]; \alpha^+_{[\![L]\!]+\Sigma_i[\![A_i]\!]}); \delta^{-1}; [\pi_r, \mathsf{rfix}(L)] \\
&= [\![\Gamma \vdash \mathsf{let}\ x_k = a; t_k\ \mathsf{where}\ (\ell_i(x_i) : \{A_i\},)_i \rhd \mathsf{L}]\!]
\end{aligned} \tag{248}$$

as desired.

- cfg-β_2: Define $P = [\![\Gamma \vdash_\perp b : B]\!]$, $L = \mathsf{lsem}_{\Gamma,\mathsf{L}}((\ell_i(x_i) : \{t_i\},)_i)$ and $\mathsf{R} = (\ell_i(A_i),)_i$. We have that

$$\begin{aligned}
&[\![\Gamma \vdash \mathsf{br}\ \kappa\ b\ \mathsf{where}\ (\ell_i(x_i) : \{t_i\},)_i \rhd \mathsf{L}]\!] \\
&= \mathsf{let}(P; \iota_{(\mathsf{L},\mathsf{R}),\kappa}; \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_i[\![A_i]\!]}); \delta^{-1}; [\pi_r, \mathsf{rfix}(L)] \\
&= \mathsf{let}(P; \iota_{\mathsf{L},\kappa}); [\![\Gamma]\!] \otimes \iota_l; \delta^{-1}; [\pi_r, \mathsf{rfix}(L)] \\
&= \mathsf{let}(P; \iota_{\mathsf{L},\kappa}); \pi_r = P; \iota_{\mathsf{L},\kappa} = [\![\Gamma \vdash \mathsf{br}\ \kappa\ b \rhd \mathsf{L}]\!]
\end{aligned} \tag{249}$$

as desired.

- cfg-η: Let $\mathsf{L} = (\kappa_j(B_j),)_j$, and define $\mathsf{R} = (\ell_i(A_i),)_i$ and $L = \mathsf{lsem}_{\Gamma,\mathsf{L}}((\ell_i(x_i) : \{t_i\},)_i)$. Using the cfg-$\beta_1$ and cfg-β_2 cases proved above, we have that

$$
\begin{aligned}
&[\![\Gamma \vdash \mathsf{cfgs}\ \{(\ell_i(x_i) : \{t_i\},)_i\} : \mathsf{L}, \mathsf{R} \rightsquigarrow \mathsf{L}]\!] \\
&= [\![\Gamma]\!] \otimes \alpha_{[\![\mathsf{L}]\!]+\Sigma_i[\![A_i]\!]}; \delta^{-1}; [\\
&\quad [\![\Gamma \vdash \mathsf{cfgs}\ \{(\ell_i(x_i) : \{t_i\},)_i\} : \mathsf{L} \rightsquigarrow \mathsf{L}]\!], \\
&\quad [\![\Gamma \vdash \mathsf{cfgs}\ \{(\ell_i(x_i) : \{t_i\},)_i\} : \mathsf{R} \rightsquigarrow \mathsf{L}]\!]] \\
&= [\![\Gamma]\!] \otimes \alpha_{\Sigma_i[\![B_i]\!]+\Sigma_i[\![A_i]\!]}; \delta^{-1}; [\\
&\quad \delta_\Sigma^{-1}; [[\![\Gamma, y_j : B_j \vdash \mathsf{br}\ \kappa_j\ y_j\ \mathsf{where}\ (\ell_i(x_i) : \{t_i\},)_i \rhd \mathsf{L}]\!],]_j, \\
&\quad \delta_\Sigma^{-1}; [[\![\Gamma, x_j : A_j \vdash \mathsf{br}\ \ell_j\ x_j\ \mathsf{where}\ (\ell_i(x_i) : \{t_i\},)_i \rhd \mathsf{L}]\!],]_j] \\
&= [\![\Gamma]\!] \otimes \alpha_{\Sigma_i[\![B_i]\!]+\Sigma_i[\![A_i]\!]}; \delta^{-1}; [\\
&\quad \delta_\Sigma^{-1}; [[\![\Gamma, y_j : B_j \vdash_\perp y_j : B_j]\!]; \iota_{\mathsf{L},\kappa_j},]_j, \\
&\quad \delta_\Sigma^{-1}; [\mathsf{let}([\![\Gamma, x_j : A_j \vdash_\perp x_j : A_j]\!]; \iota_j); \mathsf{rfix}(\pi_l \otimes \Sigma_k[\![A_k]\!]; L)]_j] \\
&= [\![\Gamma]\!] \otimes \alpha_{\Sigma_i[\![B_i]\!]+\Sigma_i[\![A_i]\!]}; \delta^{-1}; [\delta_\Sigma^{-1}; [\pi_r; \iota_{\mathsf{L},\kappa_j},]_j, \delta_\Sigma^{-1}; [\mathsf{let}(\pi_r; \iota_j); \pi_l \otimes \Sigma_k[\![A_k]\!]]_j; \mathsf{rfix}(L)] \\
&= [\![\Gamma]\!] \otimes \alpha_{\Sigma_i[\![B_i]\!]+\Sigma_i[\![A_i]\!]}; \delta^{-1}; [\pi_r; \alpha_{\mathsf{L}}, \mathsf{rlet}(\delta_\Sigma^{-1}; [\pi_r; \iota_j]_j); \mathsf{rfix}(L)] \\
&= [\![\Gamma]\!] \otimes \alpha_{\mathsf{L}+\Sigma_i[\![A_i]\!]}; \delta^{-1}; [\pi_r, \mathsf{rlet}(\pi_r); \mathsf{rfix}(L)] \\
&= [\![\Gamma]\!] \otimes \alpha_{\mathsf{L}+\Sigma_i[\![A_i]\!]}; \delta^{-1}; [\pi_r, \mathsf{rfix}(L)]
\end{aligned}
\tag{250}
$$

  It follows by label-substitution that that

$$
\begin{aligned}
&[\![\Gamma \vdash [\mathsf{cfgs}\ \{(\ell_i(x_i) : \{t_i\},)_i\}]r \rhd \mathsf{L}]\!] \\
&= \mathsf{let}([\![\Gamma \vdash r \rhd \mathsf{L}, \mathsf{R}]\!]); [\![\Gamma \vdash \mathsf{cfgs}\ \{(\ell_i(x_i) : \{t_i\},)_i\} : \mathsf{L}, \mathsf{R} \rightsquigarrow \mathsf{L}]\!] \\
&= \mathsf{let}([\![\Gamma \vdash r \rhd \mathsf{L}, \mathsf{R}]\!]); [\![\Gamma]\!] \otimes \alpha_{\mathsf{L}+\Sigma_i[\![A_i]\!]}; \delta^{-1}; [\pi_r, \mathsf{rfix}(L)] \\
&= \mathsf{esem}_{\Gamma,\mathsf{L}}(r); \delta^{-1}; [\pi_r, \mathsf{rfix}(L)] \\
&[\![\Gamma \vdash r\ \mathsf{where}\ (\ell_i(x_i) : \{t_i\},)_i \rhd \mathsf{L}]\!]
\end{aligned}
\tag{251}
$$

  as desired.
- codiag: Define $R = [\![\Gamma \vdash r \rhd \mathsf{L}, \ell(A)]\!]$, and $S = [\![\Gamma, y : A \vdash s \rhd \mathsf{L}, \ell(A), \kappa(A)]\!]$ We have that

$$
\begin{aligned}
&[\![\Gamma \vdash r\ \mathsf{where}\ \ell(x) : \{\mathsf{br}\ \kappa\ x\ \mathsf{where}\ \kappa(y) : \{s\}\} \rhd \mathsf{L}]\!] \\
&= \mathsf{let}(R); \delta^{-1}; [\mathsf{id}, \mathsf{rfix}([\![\Gamma, x : A \vdash \mathsf{br}\ \kappa\ x\ \mathsf{where}\ \kappa(y) : \{s\} \rhd \mathsf{L}, \ell(A)]\!])] \\
&= \mathsf{let}(R); \delta^{-1}; [\mathsf{id}, \mathsf{rfix}(\mathsf{let}(\pi_r; \iota_r); \delta^{-1}; [\pi_r, \mathsf{rfix}([\![\Gamma, x : A, y : A \vdash s \rhd \mathsf{L}, \ell(A), \kappa(A)]\!])])] \\
&= \mathsf{let}(R); \delta^{-1}; [\mathsf{id}, \mathsf{rfix}(\mathsf{let}(\pi_r; \iota_r); \delta^{-1}; [\pi_l \otimes A; \pi_r, \mathsf{rfix}(\pi_l \otimes [\![A]\!]; S)])] \\
&= \mathsf{let}(R); \delta^{-1}; [\mathsf{id}, \mathsf{rfix}(\mathsf{let}(\pi_r; \iota_r); \pi_l \otimes [\![A]\!]; \delta^{-1}; [\pi_r, \mathsf{rfix}(S)])] \\
&= \mathsf{let}(R); \delta^{-1}; [\mathsf{id}, \mathsf{rfix}(\iota_r^{[\![\Gamma]\!]} \gg [\pi_r, \mathsf{rfix}(S)]_{[\![\Gamma]\!]})] \\
&= \mathsf{let}(R); \delta^{-1}; [\mathsf{id}, \mathsf{rfix}(\mathsf{rfix}(S))] = \mathsf{let}(R); \delta^{-1}; [\mathsf{id}, \mathsf{rfix}(S \gg [\pi_r, \iota_r^{[\![\Gamma]\!]}]_{[\![\Gamma]\!]})] \\
&= \mathsf{let}(R); \delta^{-1}; [\mathsf{id}, \mathsf{rfix}(\mathsf{let}(S); \delta^{-1}; [\pi_r, \pi_r; \iota_r])] = \mathsf{let}(R); \delta^{-1}; [\mathsf{id}, \mathsf{rfix}(S; [\mathsf{id}, \iota_r])] \\
&= \mathsf{let}(R); \delta^{-1}; [\mathsf{id}, \mathsf{rfix}([\![\Gamma, y : A \vdash [\ell/\kappa]s \rhd \mathsf{L}, \ell(A)]\!])] = [\![\Gamma \vdash r\ \mathsf{where}\ \ell(y) : \{[\ell/\kappa]s\} \rhd \mathsf{L}]\!]
\end{aligned}
\tag{252}
$$

  as desired.

- uni: Define $R = [\![\Gamma \vdash r \rhd \mathsf{L}, \ell(A)]\!]$, $E = [\![\Gamma, x : A \vdash_\perp e : B]\!]$, $S = [\![\Gamma, y : B \vdash s \rhd \mathsf{L}, \kappa(B)]\!]$, and $T = [\![\Gamma, x : A \vdash s \rhd \mathsf{L}, \ell(A)]\!]$. We have by induction that

$$\begin{aligned}[\![\Gamma, x : A \vdash \mathsf{let}\ y = e; s \rhd \mathsf{L}, \kappa(B)]\!] &= \mathsf{rlet}(E); S = \\ [\![\Gamma, x : A \vdash t\ \mathsf{where}\ \ell(x) : \{\mathsf{br}\ \kappa\ e\} \rhd \mathsf{L}, \kappa(B)]\!] &= \mathsf{rcase}(T); [\![\mathsf{L}]\!] + E\end{aligned} \tag{253}$$

  It follows in particular that

$$\mathsf{rlet}(E); \mathsf{rfix}(S) = \mathsf{rfix}(T) \tag{254}$$

  and hence that

$$\begin{aligned}&[\![\Gamma \vdash (r\ \mathsf{where}\ \ell(x) : \{\mathsf{br}\ \kappa\ e\})\ \mathsf{where}\ \kappa(y) : \{s\} \rhd \mathsf{L}]\!] \\ &= \mathsf{let}(\mathsf{let}(R); \delta^{-1}; \pi_r + E); \delta^{-1}; [\pi_r, \mathsf{rfix}(S)] \\ &= \mathsf{let}(R); \delta^{-1}; ([\![\Gamma]\!] \otimes [\![\mathsf{L}]\!]) + \mathsf{rlet}(E); [\pi_r, \mathsf{rfix}(S)] = \mathsf{let}(R); \delta^{-1}; [\pi_r, \mathsf{rlet}(E); \mathsf{rfix}(S)] \\ &= \mathsf{let}(R); \delta^{-1}; [\pi_r, \mathsf{rfix}(T)] = [\![\Gamma \vdash r\ \mathsf{where}\ \ell(x) : \{t\} \rhd \mathsf{L}]\!]\end{aligned} \tag{255}$$

  as desired.
- dinat: We define $\mathsf{R} = (\ell_i(A_i),)_i$, $\mathsf{R}' = (\kappa_j(B_j),)_j$, $S = [\![\Gamma \vdash \sigma : \mathsf{R} \rightsquigarrow \mathsf{R}']\!]$, and $S' = [\![\Gamma]\!] \otimes \alpha^+_{[\![R]\!]}; S; \alpha^+_{\Sigma_j [\![B_j]\!]}$. We have that

$$\begin{aligned}\mathsf{esem}_{\Gamma,\mathsf{L}}([\sigma^1]r) &= [\![\Gamma \vdash [\sigma^1]r \rhd \mathsf{L}, \mathsf{R}']\!]; \alpha^+_{[\![\mathsf{L}]\!] + \Sigma_j [\![B_j]\!]} \\ &= \mathsf{let}([\![\Gamma \vdash r \rhd \mathsf{L}, \mathsf{R}]\!]); [\![\Gamma]\!] \otimes \alpha^+_{[\![\mathsf{L}]\!] + [\![\mathsf{R}]\!]}; \delta^{-1}; \pi_r + (S; \alpha^+_{\Sigma_j [\![B_j]\!]}) \\ &= \mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(r)); \delta^{-1}; \pi_r + S'\end{aligned} \tag{256}$$

  and, writing $L = \mathsf{lsem}_{\Gamma,\mathsf{L}}((\ell_i(x_i) : \{t_i\},)_i)$,

$$\begin{aligned}&\mathsf{let}((\kappa_i(x_i) : \{[\sigma^1]t_i\},)_i) = \delta_\Sigma^{-1}; [[\![\Gamma, x_i : B_i \vdash [\sigma^1]t_i \rhd \mathsf{L}, \mathsf{R}']\!],]_i; \alpha^+_{[\![\mathsf{L}]\!] + \Sigma_j [\![B_j]\!]} \\ &= \delta_\Sigma^{-1}; [\mathsf{let}([\![\Gamma, x_i : B_i \vdash t_i \rhd \mathsf{L}, \mathsf{R}]\!]); [\![\Gamma, x_i : B_i \vdash \sigma^1 : \mathsf{L}, \mathsf{R} \rightsquigarrow \mathsf{L}, \mathsf{R}']\!],]_i; \alpha^+_{[\![\mathsf{L}]\!] + \Sigma_j [\![B_j]\!]} \\ &= \delta_\Sigma^{-1}; [\mathsf{rlet}([\![\Gamma, x_i : B_i \vdash t_i \rhd \mathsf{L}, \mathsf{R}]\!]),]_i; [\![\Gamma \vdash \sigma^1 : \mathsf{L}, \mathsf{R} \rightsquigarrow \mathsf{L}, \mathsf{R}']\!]; \alpha^+_{[\![\mathsf{L}]\!] + \Sigma_j [\![B_j]\!]} \\ &= \mathsf{rlet}(\delta_\Sigma^{-1}; [[\![\Gamma, x_i : B_i \vdash t_i \rhd \mathsf{L}, \mathsf{R}]\!]]_i); [\![\Gamma]\!] \otimes \alpha^+_{[\![\mathsf{L}]\!] + [\![\mathsf{R}]\!]}; \delta^{-1}; \pi_r + (S; \alpha^+_{\Sigma_j [\![B_j]\!]}) \\ &= \mathsf{rlet}(\delta_\Sigma^{-1}; [[\![\Gamma, x_i : B_i \vdash t_i \rhd \mathsf{L}, \mathsf{R}]\!],]_i; \alpha^+_{[\![\mathsf{L}]\!] + \Sigma_j [\![B_j]\!]}); \delta^{-1}; \pi_r + S' \\ &= \mathsf{rcase}(L); \pi_r + S'\end{aligned} \tag{257}$$

  Furthermore, we have that, letting $G = [\![\Gamma \vdash (\kappa_j(x_j) \mapsto t_j,)_j : \mathsf{R}' \rightsquigarrow \mathsf{L}, \mathsf{R}]\!]$,

$$\begin{aligned}&[\![\Gamma, x_i : A_i \vdash [(\kappa_j(x_j) \mapsto t_j,)_j{}^1](\sigma\ \ell_i\ x_i) \rhd \mathsf{L}, \mathsf{R}]\!] \\ &= \mathsf{let}([\![\Gamma, x_i : A_i \vdash \sigma\ \ell_i\ x_i \rhd \mathsf{L}, \mathsf{R}']\!]); [\![\Gamma, x_i : A_i \vdash (\kappa_j(x_j) \mapsto t_j,)_j{}^1 : \mathsf{L}, \mathsf{R}' \rightsquigarrow \mathsf{L}, \mathsf{R}]\!] \\ &= \mathsf{let}([\![\Gamma, x_i : A_i \vdash \sigma\ \ell_i\ x_i \rhd \mathsf{L}, \mathsf{R}']\!]); \pi_l \otimes [\![\mathsf{L}, \mathsf{R}']\!]; [\![\Gamma \vdash (\kappa_j(x_j) \mapsto t_j,)_j{}^1 : \mathsf{L}, \mathsf{R}' \rightsquigarrow \mathsf{L}, \mathsf{R}]\!] \\ &= \mathsf{rlet}([\![\Gamma, x_i : A_i \vdash \sigma\ \ell_i\ x_i \rhd \mathsf{L}, \mathsf{R}']\!]); [\![\Gamma]\!] \otimes \alpha^+_{[\![\mathsf{L}]\!] + [\![\mathsf{R}']\!]}; \delta^{-1}; [\pi_r; \iota_l; \alpha^+_{[\![\mathsf{L},\mathsf{R}]\!]}, G] \\ &= \mathsf{rlet}([\![\Gamma, x_i : A_i \vdash \sigma\ \ell_i\ x_i \rhd \mathsf{L}, \mathsf{R}']\!]; \alpha^+_{[\![\mathsf{L}]\!] + [\![\mathsf{R}']\!]}); \delta^{-1}; [\pi_r; \iota_l; \alpha^+_{[\![\mathsf{L},\mathsf{R}]\!]}, G] \\ &= \mathsf{rcase}([\![\Gamma, x_i : A_i \vdash \sigma\ \ell_i\ x_i \rhd \mathsf{L}, \mathsf{R}']\!]; \alpha^+_{[\![\mathsf{L}]\!] + [\![\mathsf{R}']\!]}); [\pi_r; \iota_l; \alpha^+_{[\![\mathsf{L},\mathsf{R}]\!]}, G] \\ &= \mathsf{rcase}([\![\Gamma, x_i : A_i \vdash \sigma\ \ell_i\ x_i \rhd \mathsf{R}']\!]; \iota_r); [\pi_r; \iota_l; \alpha^+_{[\![\mathsf{L},\mathsf{R}]\!]}, G] \\ &= \mathsf{rlet}([\![\Gamma, x_i : A_i \vdash \sigma\ \ell_i\ x_i \rhd \mathsf{R}']\!]); G\end{aligned} \tag{258}$$

It follows that

$$\begin{aligned}
&\mathsf{lsem}_{\Gamma,\mathsf{L}}((\ell_i(x_i) : \{[(\kappa_j(x_j) \mapsto t_j,)^1](\sigma\ \ell_i\ x_i)\},)_i) \\
&= \delta_\Sigma^{-1}; [[\![\Gamma, x_i : A_i \vdash [(\kappa_j(x_j) \mapsto t_j,)_j{}^1](\sigma\ \ell_i\ x_i) \rhd \mathsf{L}, \mathsf{R}]\!],]_i; \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_i[\![A_i]\!]} \\
&= \delta_\Sigma^{-1}; [\mathsf{rlet}([[\![\Gamma, x_i : A_i \vdash \sigma\ \ell_i\ x_i \rhd \mathsf{R}']\!]); G]_i; \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_i[\![A_i]\!]} \\
&= \delta_\Sigma^{-1}; [\mathsf{rlet}([[\![\Gamma, x_i : A_i \vdash \sigma\ \ell_i\ x_i \rhd \mathsf{R}']\!]); G]_i; \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_i[\![A_i]\!]} \\
&= \delta_\Sigma^{-1}; [\mathsf{rlet}([[\![\Gamma, x_i : A_i \vdash \sigma\ \ell_i\ x_i \rhd \mathsf{R}']\!])]_i; G; \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_i[\![A_i]\!]} \\
&= \mathsf{rlet}(\delta_\Sigma^{-1}; [[[\![\Gamma, x_i : A_i \vdash \sigma\ \ell_i\ x_i \rhd \mathsf{R}']\!]]_i); G; \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_i[\![A_i]\!]} \\
&= \mathsf{rlet}(\delta_\Sigma^{-1}; [[[\![\Gamma, x_i : A_i \vdash \sigma\ \ell_i\ x_i \rhd \mathsf{R}']\!]]_i); G; \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_i[\![A_i]\!]} \\
&= \mathsf{rlet}([[\![\Gamma]\!]] \otimes \alpha_{[\![\mathsf{R}]\!]}; S); G; \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_i[\![A_i]\!]} \\
&= \mathsf{rlet}(S'); [[\![\Gamma]\!]] \otimes \alpha_{[\![\mathsf{R}']\!]}; [[\![\Gamma \vdash (\kappa_j(x_j) \mapsto t_j,)_j : \mathsf{R}' \rightsquigarrow \mathsf{L}, \mathsf{R}]\!]; \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_i[\![A_i]\!]} \\
&= \mathsf{rlet}(S'); \delta_\Sigma^{-1}; [[\![\Gamma, x_i : A_i \vdash t_i \rhd \mathsf{R}'\mathsf{L}, \mathsf{R}]\!],]_i; \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_i[\![A_i]\!]} \\
&= \mathsf{rlet}(S'); L
\end{aligned} \tag{259}$$

We therefore have

$$\begin{aligned}
&[[\![\Gamma \vdash [\sigma^1]r \text{ where } (\kappa_i(x_i) : \{[\sigma^1]t_i\},)_i \rhd \mathsf{L}]\!] \\
&= \mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}([\sigma^1]r)); \delta^{-1}; [\pi_r, \mathsf{rfix}(\mathsf{lsem}_{\Gamma,\mathsf{L}}((\kappa_i(x_i) : \{[\sigma^1]t_i\},)_i))] \\
&= \mathsf{let}(\mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(r)); \delta^{-1}; \pi_r + S'); \delta^{-1}; [\pi_r, \mathsf{rfix}(\mathsf{rcase}(L); \pi_r + S')] \\
&= \mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(r)); \delta^{-1}; [[\![\Gamma]\!]] \otimes [[\![\mathsf{L}]\!]] + \mathsf{rlet}(S); [\pi_r, \mathsf{rfix}(\mathsf{rcase}(L); \pi_r + S')] \\
&= \mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(r)); \delta^{-1}; [\pi_r, \mathsf{rlet}(S'); \mathsf{rfix}(\mathsf{rcase}(L); \pi_r + S')] \\
&= \mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(r)); \delta^{-1}; [\pi_r, \mathsf{rfix}(\mathsf{rlet}(S'); L)] \\
&= \mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(r)); \delta^{-1}; [\pi_r, \mathsf{rfix}(\mathsf{lsem}_{\Gamma,\mathsf{L}}((\ell_i(x_i) : \{[(\kappa_j(x_j) \mapsto t_j,)^1](\sigma\ \ell_i\ x_i)\},)_i))] \\
&= [[\![\Gamma \vdash r \text{ where } [(\kappa_j(x_j) \mapsto t_j,)^1](\sigma\ \ell_i\ x_i),)_i \rhd \mathsf{L}]\!]
\end{aligned} \tag{260}$$

as desired.

All other cases are analogous to those for expressions, and so omitted. □

Lemma C.7 (where-fusion). *The rewrite rules cfg-fuse$_1$ (Eqn. 16) and cfg-fuse$_2$ are sound, where we define*

$$\frac{\begin{array}{c}\forall i \in I.\Gamma, x_i : A_i \vdash t_i \rhd \mathsf{L}, (\ell_j(A_j),)_{j\in I}, \kappa(B) \\ \Gamma, y : B \vdash s \rhd \mathsf{L}, (\ell_j(A_j),)_{j\in I}, \kappa(B), (\ell'_{j'}(A'_{j'}),)_{j'\in I'}, \\ \Gamma \vdash r \rhd \mathsf{L}, (\ell_i(A_i),)_i, \kappa(B) \quad \forall i' \in I'.\Gamma, x'_{i'} : A'_{i'} \vdash t'_{i'} \rhd \mathsf{L}, (\ell_j(A_j),)_{j\in I}, \kappa(B), (\ell'_{j'}(A'_{j'}),)_{j'\in I'}\end{array}}{\begin{array}{c}\Gamma \vdash r \text{ where } (\ell_i(x_i) : \{t_i\},)_{i\in I}, \kappa(y) : \{s \text{ where } (\ell'_{i'}(x'_{i'}) : \{t'_{i'}\},)_{i'\in I'}\} \\ \approx r \text{ where } (\ell_i(x_i) : \{t_i\},)_{i\in I}, \kappa(y) : \{s\}, (\ell'_{i'}(x'_{i'}) : \{t'_{i'}\},)_{i'\in I'} \rhd \mathsf{L}\end{array}}\ \textit{cfg-fuse}_2 \tag{261}$$

Proof. We begin with cfg-fuse$_1$. Let $\mathsf{R} = (\ell_j(A_j),)_j$, $\mathsf{K} = (\kappa_k(B_k),)_k$, $S = (\kappa_i(y_i) : \{s_i\},)_i$, $T = (\ell_i(x_i) : \{t_i\},)_j$, and $D_S = \mathsf{lsem}_{\Gamma,\mathsf{L},\mathsf{R}}(S)$, $D_T = \mathsf{lsem}_{\Gamma,\mathsf{L}}(T)$, $D_G = \mathsf{lsem}_{\Gamma,\mathsf{L}}(S,T)$ We have that

$$
\begin{aligned}
&[\![\Gamma \vdash (r \text{ where } S) \text{ where } T \rhd \mathsf{L}]\!] \\
&= \mathsf{let}(\mathsf{let}(\mathsf{esem}_{\Gamma,(\mathsf{L},\mathsf{R})}(r)); \delta^{-1}; [\pi_r, \mathsf{rfix}(D_S)]; \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_j[\![A_j]\!]}); \delta^{-1}; [\pi_r, \mathsf{rfix}(D_T)] \\
&= \mathsf{let}(\mathsf{esem}_{\Gamma,(\mathsf{L},\mathsf{R})}(r)); \mathsf{rlet}(\delta^{-1}; [\pi_r, \mathsf{rfix}(D_S)]; \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_j[\![A_j]\!]}); \delta^{-1}; [\pi_r, \mathsf{rfix}(D_T)] \\
&= \mathsf{let}(\mathsf{esem}_{\Gamma,(\mathsf{L},\mathsf{R})}(r)); \delta^{-1}; [\mathsf{rlet}(\pi_r; \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_j[\![A_j]\!]}), \mathsf{rlet}(\mathsf{rfix}(D_S); \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_j[\![A_j]\!]})]; \delta^{-1}; [\pi_r, \mathsf{rfix}(D_T)] \\
&= \mathsf{let}(\mathsf{esem}_{\Gamma,(\mathsf{L},\mathsf{R})}(r)); [\\
&\quad \pi_r; \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_j[\![A_j]\!]} \gg [\pi_r, \mathsf{rfix}(D_T)]_{[\![\Gamma]\!]}, \\
&\quad \mathsf{rfix}(D_S); \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_j[\![A_j]\!]} \gg [\pi_r, \mathsf{rfix}(D_T)]_{[\![\Gamma]\!]}]_{[\![\Gamma]\!]} \\
&= \mathsf{let}(\mathsf{esem}_{\Gamma,(\mathsf{L},\mathsf{R})}(r)); [\\
&\quad \alpha^{R+}_{[\![\mathsf{L}]\!]+\Sigma_j[\![A_j]\!]} \gg [\pi_r, \mathsf{rfix}(D_T)]_{[\![\Gamma]\!]}, \\
&\quad \mathsf{rfix}(D_S); \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_j[\![A_j]\!]} \gg [\pi_r, \mathsf{rfix}(D_T)]_{[\![\Gamma]\!]}]_{[\![\Gamma]\!]} \\
&= \mathsf{let}(\Gamma \vdash r \rhd \mathsf{L}, \mathsf{R}, \mathsf{K}); \alpha^{\mathsf{R}+}_{[\![\mathsf{L}]\!]+([\![R]\!]+\Sigma_j[\![A_j]\!])} \gg [\pi_r, \\
&\quad [\mathsf{rfix}(D_T), \mathsf{rfix}(D_S); \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_j[\![A_j]\!]} \gg [\pi_r, \mathsf{rfix}(D_T)]_{[\![\Gamma]\!]}]_{[\![\Gamma]\!]}]_{[\![\Gamma]\!]} \\
&= \mathsf{let}(\Gamma \vdash r \rhd \mathsf{L}, \mathsf{R}, \mathsf{K}; \alpha^{\mathsf{R}+}_{[\![\mathsf{L}]\!]+([\![R]\!]+\Sigma_j[\![A_j]\!])}); \delta^{-1}; \\
&\quad [\pi_r, [\mathsf{rfix}(D_T), \mathsf{rfix}(D_S); \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_j[\![A_j]\!]} \gg [\pi_r, \mathsf{rfix}(D_T)]_{[\![\Gamma]\!]}]_{[\![\Gamma]\!]}] \\
&= \mathsf{let}(\mathsf{esem}_{\Gamma,\mathsf{L}}(r)); \delta^{-1}; [\pi_r, [\mathsf{rfix}(D_T), \mathsf{rfix}(D_S); \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_j[\![A_j]\!]} \gg [\pi_r, \mathsf{rfix}(D_T)]_{[\![\Gamma]\!]}]_{[\![\Gamma]\!]}]
\end{aligned}
\tag{262}
$$

For this to be equal to $[\![\Gamma \vdash r \text{ where } S, T \rhd \mathsf{L}]\!]$, it therefore suffices to show that

$$
\mathsf{rfix}(D_G) = [\mathsf{rfix}(D_T), \mathsf{rfix}(D_S); \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_j[\![A_j]\!]} \gg [\pi_r, \mathsf{rfix}(D_T)]_{[\![\Gamma]\!]}]_{[\![\Gamma]\!]} \tag{263}
$$

We note that, by re-association and weakening, we have that

$$
\begin{aligned}
D_G &= [\![\Gamma]\!] \otimes \alpha^+_{[\![\mathsf{R}]\!]+[\![\mathsf{K}]\!]}; \delta^{-1}; [\delta^{-1}_\Sigma; [[\![\Gamma, x_i : A_i \vdash t_i \rhd \mathsf{L}, \mathsf{R}, \mathsf{K}]\!]; \alpha^+_{[\![\mathsf{L}]\!]+\Sigma_j[\![C_j]\!]},]_i, D_S] \\
&= [\![\Gamma]\!] \otimes \alpha^+_{[\![\mathsf{R}]\!]+[\![\mathsf{K}]\!]}; \delta^{-1}; [D_T; [\![\mathsf{L}]\!] + (\alpha^+_{\mathsf{R}}; [\![\mathsf{R} \le \mathsf{R}, \mathsf{K}]\!]; \alpha^+_{\Sigma_i[\![C_i]\!]}), D_S] \\
&= [\![\Gamma]\!] \otimes \alpha^+_{[\![\mathsf{R}]\!]+[\![\mathsf{K}]\!]}; \delta^{-1}; [D_T; [\![\mathsf{L}]\!] + (\iota_{l,\Sigma_i[\![A_i]\!]+\Sigma_i[\![B_i]\!]}; \alpha^+_{\Sigma_i[\![C_i]\!]}), D_S] \\
&= \alpha^{+[\![\Gamma]\!]}_{\mathsf{R}+\mathsf{K}} \gg [D_T \gg [\![\mathsf{L}]\!] +_R (\iota^{[\![\Gamma]\!]}_l; \alpha^+_{\Sigma_i[\![C_i]\!]}),]_{[\![\Gamma]\!]}
\end{aligned}
\tag{264}
$$

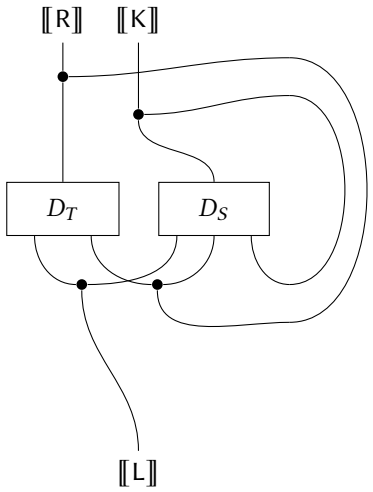
where $C_{1..k} = A_{1..k}$ and $C_{k+1..n} = B_{1..n-k}$. We note in particular that $[\![\mathsf{R} \le \mathsf{R}, \mathsf{K}]\!]$ is up to isomorphism the left injection $\iota_l : \Sigma_i[\![A_i]\!] \to \Sigma_i[\![A_i]\!] + \Sigma_i[\![B_i]\!]$. We can now derive Equation 263, and hence the soundness of cfg-fuse$_1$, via the string-diagrams in Figure 38, which are drawn in the co-Kleisli category inducted by $[\![\Gamma]\!]$. cfg-fuse$_2$ then follows by repeated application of cfg-fuse$_1$, as desired. □

C.7 Completeness

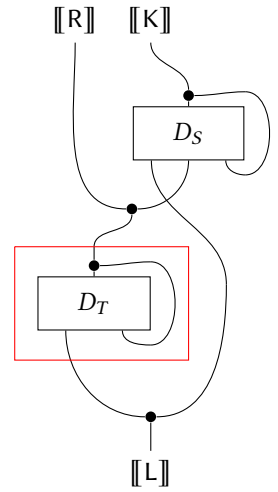
Theorem 5.13 (Completeness (Expressions)). *We have that, for all pure* $\mathsf{eff}(\Gamma) = \bot$,

$$
\Gamma \vdash_\epsilon e \approx e' : A \iff [\![\Gamma \vdash_\epsilon e : A]\!]_{\mathsf{Th}^\otimes(\cdot)} = [\![\Gamma \vdash_\epsilon e' : A]\!]_{\mathsf{Th}^\otimes(\cdot)}
$$

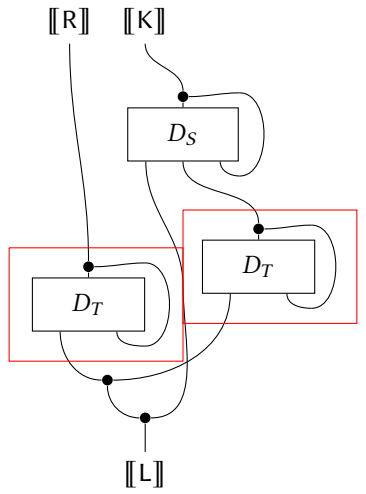
In particular, this implies that $\mathsf{Th}(\cdot)$ *is the initial* λ_{SSA} *expression model*



(a) $\mathsf{rfix}(D_G)$, with D_G drawn as per the right of Eqn. 264



(b) Equivalent to 38a by isotopy and associativity of the codiagonal. We highlight $\mathsf{rfix}(D_T)$, which is used in the next step.



(c) $[\mathsf{rfix}(D_T), \mathsf{rfix}(D_S); \alpha^+_{⟦\mathsf{L}⟧+\Sigma_j ⟦A_j⟧} \gg [\pi_r, \mathsf{rfix}(D_T)]_{⟦\Gamma⟧}]_{⟦\Gamma⟧}$.
Equivalent to 38b by duplication of $\mathsf{rfix}(D_T)$.

Fig. 38. String diagrams validating the soundness of cfg-fuse$_1$ (Eqn. 16), drawn in the co-Kleisli category induced by $⟦\Gamma⟧$

Proof. We begin by showing $\mathsf{Th}^{\otimes}(\Gamma)$ is an λ_{SSA} expression model by validating all the equations for a distributive Freyd category. These are formalized as follows:

- $\mathsf{Th}^{\otimes}(\Gamma)$ is a category: formalized in `Rewrite/Term/Compose/Seq.lean`
- $\mathsf{Th}^{\otimes}(\Gamma)$ is a Freyd category: formalized in `Rewrite/Term/Compose/Product.lean`
- $\mathsf{Th}^{\otimes}(\Gamma)$ has coproducts: formalized in `Rewrite/Term/Compose/Sum.lean`
- $\mathsf{Th}^{\otimes}(\Gamma)$ is distributive: formalized in `Rewrite/Term/Compose/Distrib.lean`

We then prove that the packing of each type constructor is equivalent to its denotational semantics in $\mathsf{Th}^{\otimes}(\Gamma)$ in `Rewrite/Term/Compose/Completeness.lean`. Finally, we show that packing and unpacking are mutual inverses for Γ pure in `Term.Eqv.packed_unpacked` and `Term.Eqv.unpacked_packed` in `Rewrite/Term/Structural/Product.lean`, completing the proof of initiality. □

Theorem 5.14 (Completeness (Regions)). *We have that, for all pure* $\mathsf{eff}(\Gamma) = \bot$,

$$\Gamma \vdash r \approx r' \rhd \mathsf{L} \iff [\![\Gamma \vdash r \rhd \mathsf{L}]\!]_{\mathsf{Th}(\cdot,\cdot)} = [\![\Gamma \vdash r' \rhd \mathsf{L}]\!]_{\mathsf{Th}(\cdot,\cdot)}$$

In particular, this implies that $\mathsf{Th}(\cdot,\cdot)$ *is the initial* λ_{SSA} *model.*

Proof. We begin by showing $\mathsf{Th}(\Gamma, \mathsf{L})$ is an λ_{SSA} region model by validating all the equations for a distributive Elgot category. These are formalized as follows:

- $\mathsf{Th}(\Gamma, \mathsf{L})$ is a category: formalized in `Rewrite/Region/Compose/Seq.lean`
- $\mathsf{Th}(\Gamma, \mathsf{L})$ is a Freyd category: formalized in `Rewrite/Region/Compose/Product.lean`
- $\mathsf{Th}(\Gamma, \mathsf{L})$ has coproducts: formalized in `Rewrite/Region/Compose/Sum.lean`
- $\mathsf{Th}(\Gamma, \mathsf{L})$ is distributive: formalized in `Rewrite/Region/Compose/Distrib.lean`
- $\mathsf{Th}(\Gamma, \mathsf{L})$ is a strong Elgot category: formalized in `Rewrite/Region/Compose/Elgot.lean`

We then prove that the packing of each type constructor is equivalent to its denotational semantics in $\mathsf{Th}(\Gamma, \mathsf{L})$ in `Rewrite/Region/Compose/Completeness.lean`. Finally, we show that packing and unpacking are mutual inverses for Γ pure in `Region.Eqv.packed_unpacked` and `Region.Eqv.unpacked_packed` in `Rewrite/Region/Structural.lean`, completing the proof of initiality. □