Data Layout from a Type-Theoretic Perspective (extended version)

Henry DeYoung[1] Frank Pfenning[2,3]

Computer Science Department
Carnegie Mellon University
Pittsburgh, United States

Abstract

The specifics of data layout can be important for the efficiency of functional programs and interaction with external libraries. In this paper, we develop a type-theoretic approach to data layout that could be used as a typed intermediate language in a compiler or to give a programmer more control. Our starting point is a computational interpretation of the semi-axiomatic sequent calculus for intuitionistic logic that defines abstract notions of cells and addresses. We refine this semantics so addresses have more structure to reflect possible alternative layouts without fundamentally departing from intuitionistic logic. We then add recursive types and explore example programs and properties of the resulting language. This document is an extended version of an invited paper that appeared at MFPS 2022; additional proofs are included here.

Keywords: sequent calculus, Curry–Howard correspondence, futures-based concurrency, data layout

1 Introduction

The Curry-Howard isomorphism establishes a firm relationship between the propositions of intuitionistic logic and types for functional programs. Even if both sides of this correspondence are intuitionistic, the precise relation between proofs and functional programs varies significantly with the proof system. Curry [4], for example, related Hilbert-style axiomatic proofs to combinators and combinatory reduction. Howard [12], on the other hand, related natural deductions to terms in the typed λ-calculus. As a further example, Herbelin [11] related an intuitionistic sequent calculus with a "stoup" (LJT) to a λ-calculus with a form of explicit substitution. The computational mechanism in each of these examples is quite different and in each case derives from a proof-theoretic notion of reduction. The logical motivation for these reductions stems from the desire to prove, constructively at the meta-level, the *consistency* of specific systems of inference rules [8]. Moreover, proofs that are fully reduced (called *normal* or *cut-free*, depending on the system) exhibit the *subformula property*, which means they can serve as *verifications* [7,14].

The basis for this paper is the recently discovered *semi-axiomatic sequent calculus* [6] and its correspondence to futures [2,10]. Briefly, the semi-axiomatic sequent calculus starts from the intuitionistic sequent

---

[1] Email: hdeyoung@cs.cmu.edu

[2] Email: fp@cs.cmu.edu

calculus and replaces the right rules for positive connectives (positive conjunctions $A_1 \wedge A_2$, positive unit $\top$, and disjunctions $A_1 \vee A_2$) and the left rules for negative connectives (implications $A_1 \supset A_2$ and negative conjunctions $A_1 \mathbin{\&} A_2$) by axioms. For example, the six axioms $A_1, A_2 \vdash A_1 \wedge A_2$; $A_k \vdash A_1 \vee A_2$ for $k \in \{1, 2\}$; $A_1, A_1 \supset A_2 \vdash A_2$; and $A_1 \mathbin{\&} A_2 \vdash A_k$ for $k \in \{1, 2\}$ replace the usual $\wedge$R, $\vee$R$_k$, $\supset$L, and $\mathbin{\&}$L$_k$ rules, respectively, while the $\wedge$L, $\vee$L, $\supset$R, and $\mathbin{\&}$R rules remain unchanged. This restructuring leads to a failure of Gentzen's cut elimination theorem [8]: some cuts (called *snips*) are essential and cannot be eliminated. Fortunately, all snips arise from the new axioms and can be shown to obey a subformula property [6, Theorem 7] derived from the evident subformula property of the new axioms (A_1 and A_2 are subformulas of $A_1 \wedge A_2$, etc.).

Computationally, a process is assigned to each proof in the semi-axiomatic sequent calculus, arriving at a type theory dubbed SAX [6, section 5]. Both cuts and snips are treated as the allocation of a future, with the two premises of the cut or snip computing in parallel. The first premise computes and writes the value of the future, while the second may read its value, potentially blocking until it has been written. The *type* of the future (that is, the *cut formula*) determines the form of value that it ultimately holds.

The key observation underlying this paper is that we have the freedom to give different computational interpretations to cuts and snips. The subformula property of the new axioms, reflected in the snips, is manifest in *projections* from the values of larger types to those of smaller types. For example, ordinarily we might think of assigning terms to the axiom $A_1, A_2 \vdash A_1 \wedge A_2$ as $x_1{:}A_1, x_2{:}A_2 \vdash \langle x_1, x_2 \rangle : A_1 \wedge A_2$. Instead, if we ensure that $x : A_1 \wedge A_2$ has already been allocated, the axiom expresses *address projections* $x \cdot \pi_1 : A_1, x \cdot \pi_2 : A_2 \vdash x : A_1 \wedge A_2$. Taking it further, this axiom can be seen as *merely computing the addresses of A_1 and A_2, given the address for the pair $A_1 \wedge A_2$*. Under this interpretation, only cuts allocate memory for a future. A snip is then the parallel composition of two processes that respectively write to and read from a shared portion of a future that has already been allocated.

When we go beyond the usual propositions-as-types correspondence and add recursive types and recursive processes to our language, we have to slightly modulate our approach. For example, a type of lists of booleans satisfying the equation $boollist = \oplus\{\texttt{nil}: \mathbf{1}, \texttt{cons}: bool \times boollist\}$ would require an unbounded (or unpredictable) amount of space for a value of type $boollist$. In order to avoid this problem, we introduce a type constructor $\downarrow A$, originating in adjoint logic [3,19,17], that is inhabited by addresses for cells of type A, *i.e.*, pointers. Logically, this has no force in the sense that $A \dashv\vdash \downarrow A$, but it affects the fine structure of proofs. Every recursive type must be *guarded* by a $\downarrow$ shift, as in $boollist = \oplus\{\texttt{nil}: \mathbf{1}, \texttt{cons}: \downarrow bool \times \downarrow boollist\}$. This expresses a layout where a binary number is either just the tag $\texttt{nil}$ or a tag $\texttt{cons}$ together with a pair of addresses, $\texttt{cons}\ a_1\ a_2$. Through different ways to place $\downarrow$ shifts we can control the layout of the data. For example, here we would likely prefer $boollist = \oplus\{\texttt{nil}: \mathbf{1}, \texttt{cons}: bool \times \downarrow boollist\}$, where the address of a boolean is replaced by the boolean itself.

In summary, the main contributions of this paper are:

- a reformulated semi-axiomatic sequent calculus that syntactically distinguishes cuts and snips so they can be assigned a different dynamics (section 3); and simultaneously

- a new type theory, derived by Curry–Howard interpretation of the reformulated semi-axiomatic sequent calculus, for specifying data layout for futures-based concurrency (also in section 3); and

- proofs of type preservation and progress for the type theory (including equirecursive types and recursive processes, section 3.6); as well as

- a procedure for compiling SAX processes into SNAX processes (section 4).

Related work is discussed in section 5. This document is an extended version of an invited paper that appeared at MFPS 2022 [5]; the additional material is a discussion of compiling from SAX into SNAX found in section 4 and proofs of its correctness in appendix C.

2   SAX: A semi-axiomatic type theory for shared memory concurrency

The semi-axiomatic sequent calculus is a presentation of intuitionistic logic that blends inference rules of the sequent calculus with axioms of the Hilbert calculus [6]. Perhaps surprisingly, there is a Curry–Howard correspondence between the semi-axiomatic sequent calculus and a type theory for futures [2,10],

a form of write-once shared memory concurrency. Here we give a detailed review of the semi-axiomatic sequent calculus and its SAX type theory (though slightly altered to use explicit rules of weakening and contraction) because they serve as the cornerstones for our SNAX type theory.

Judgmental aspects.

Following the usual Curry–Howard pattern, the propositions of intuitionistic logic become SAX types, sequents become SAX static typing judgments, and their proofs become SAX processes. Sequents and SAX typing judgments correspond as follows:

$$\underbrace{A_1, A_2, \ldots, A_n}_{antecedents} \vdash \underbrace{A}_{succedent} \qquad \underbrace{a_1{:}A_1, a_2{:}A_2, \ldots, a_n{:}A_n}_{may\ read\ from} \vdash P :: \underbrace{(a : A)}_{must\ write\ to}$$

where the typing judgment is read as "Process P may read data of types $A_1, A_2, \ldots, A_n$ from addresses $a_1, a_2, \ldots, a_n$, respectively, and must write data of type A to address a." This discipline also serves to enforce the invariant that each address is written by exactly one process. For this reason, we will sometimes refer to the address that a process must write to as the process's *destination*.

In both the semi-axiomatic proof theory and the SAX type theory, we use the metavariable Γ to stand for contexts – of either antecedents or readable addresses, respectively. SAX addresses have no structure, being only variables x that will be mapped to concrete addresses α at runtime.

$$Contexts \quad \Gamma ::= (\cdot) \mid \Gamma, A \qquad\qquad Contexts \qquad \Gamma ::= (\cdot) \mid \Gamma, a{:}A$$
$$Addresses \quad a, b, c, d ::= x \mid \alpha$$

Weakening and contraction.

For reasons that will become clear in section 3, we diverge slightly from the presentation of the semi-axiomatic sequent calculus and its SAX type theory seen in [6] and use explicit rules for weakening and contraction. With respect to process syntax, both weakening and contraction are silent. (The explicit rules for weakening and contraction mean that our SAX typing rules will not immediately yield a syntax-directed type checking algorithm. But conversion to the implicit weakening and contraction of [6] is standard, and those rules are directly suitable for type checking.)

$$\frac{\Gamma \vdash C}{\Gamma, A \vdash C}\ \text{W} \qquad\qquad \frac{\Gamma \vdash P :: (c : C)}{\Gamma, a{:}A \vdash P :: (c : C)}\ \text{W}$$

$$\frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}\ \text{C} \qquad\qquad \frac{\Gamma, a{:}A, a{:}A \vdash P :: (c : C)}{\Gamma, a{:}A \vdash P :: (c : C)}\ \text{C}$$

Cut and identity.

The semi-axiomatic sequent calculus's cut rule (inherited from the sequent calculus) corresponds to SAX's static typing rule for a notion of *futures* for concurrent computation, $x \leftarrow P; Q$.

$$\frac{\Gamma_1 \vdash A \quad \Gamma_2, A \vdash C}{\Gamma_1, \Gamma_2 \vdash C}\ \text{CUT} \qquad\qquad \frac{\Gamma_1 \vdash P :: (x : A) \quad \Gamma_2, x{:}A \vdash Q :: (c : C) \quad (x\ \text{fresh})}{\Gamma_1, \Gamma_2 \vdash x \leftarrow P; Q :: (c : C)}\ \text{CUT}$$

Operationally, $x \leftarrow P; Q$ runs by first allocating memory for data of type A and then running, in parallel, processes P and Q to write data to addresses x and c, respectively. If Q tries to read from address x before P has written to x, then Q will block until P does write to x.[4]

---

[4] It is also possible to assign a sequential semantics (akin to call-by-value) or a call-by-need semantics to cuts [18]. In this paper, we will use the simplest and most general semantics, which is concurrent.

The semi-axiomatic sequent calculus's identity rule (also inherited from the sequent calculus) corresponds to the SAX typing rule for a primitive operation, copy a b, for copying data from address b to address a. This copy operation is shallow: it does not follow pointers to copy recursively.

$$\frac{}{A \vdash A} \ \text{ID} \qquad \frac{}{b{:}A \vdash \text{copy } a \ b :: (a : A)} \ \text{ID} \qquad a \ \boxed{} \quad b \ \boxed{S} \quad \rightsquigarrow \quad a \ \boxed{S} \quad b \ \boxed{S}$$

As we have done here, we will use pictures throughout this section to provide intuition about the way that data is laid out in SAX; in section 3, these will serve as a point of comparison with the layouts offered by our SNAX type theory.

Pairs, type $A_1 \times A_2$.

In the Curry–Howard isomorphism between natural deduction and the simply typed λ-calculus, conjunctions $A_1 \wedge A_2$ are interpreted as product types $A_1 \times A_2$ that describe pairs of values of types A_1 and A_2, respectively, (as well as describing the expressions that evaluate to such values). With the shift in perspective to the semi-axiomatic sequent calculus and shared memory concurrency, conjunction has a slightly different – but very closely related – interpretation: types $A_1 \times A_2$ describe addresses a to which a pair $\langle a_1, a_2 \rangle$ of addresses of types A_1 and A_2, respectively, must be written (as well as describing the processes that must write to such addresses a).

The $\wedge$A axiom of the semi-axiomatic sequent calculus thus corresponds to a static typing rule for the process write a $\langle a_1, a_2 \rangle$ that writes a pair of addresses $\langle a_1, a_2 \rangle$ to address a.

$$\frac{}{A_1, A_2 \vdash A_1 \wedge A_2} \ \wedge\text{A} \qquad \frac{}{a_1{:}A_1, a_2{:}A_2 \vdash \text{write } a \ \langle a_1, a_2 \rangle :: (a : A_1 \times A_2)} \ \times\text{A} \qquad a \ \boxed{} \quad \rightsquigarrow \quad a \ \boxed{}$$

As a proof-theoretic aside, in the semi-axiomatic sequent calculus, the above $\wedge$A axiom is used in place of the sequent calculus's full-fledged right rule, $\wedge$R, that has premises $\Gamma \vdash A_1$ and $\Gamma \vdash A_2$. Using the cut and identity rules, the $\wedge$A axiom and the usual right rule are interderivable and therefore characterize the same notion of intuitionistic conjunction. The same pattern will hold for the other logical connectives in the semi-axiomatic sequent calculus: depending on whether the connective's polarity [1,9] is positive or negative, either the sequent calculus right or left rule will be replaced with an equivalently expressive axiom. Moreover, right axioms and right rules will write, whereas left axioms and left rules will read.

Under our shared memory interpretation, the left rule for conjunction becomes a static typing rule for the process read a $(\langle x_1, x_2 \rangle \Rightarrow P)$ that reads the pair of addresses that is stored at address a:

$$\frac{\Gamma, A_1, A_2 \vdash C}{\Gamma, A_1 \wedge A_2 \vdash C} \ \wedge\text{L} \qquad \frac{\Gamma, x_1{:}A_1, x_2{:}A_2 \vdash P :: (c : C)}{\Gamma, a{:}A_1 \times A_2 \vdash \text{read } a \ (\langle x_1, x_2 \rangle \Rightarrow P) :: (c : C)} \ \times\text{L}$$

Operationally, the pair of addresses $\langle a_1, a_2 \rangle$ stored at address a is read from memory; then variables x_1 and x_2 are bound to addresses a_1 and a_2, respectively, and execution continues according to process P.

Example. At this point, we can consider our first, very simple example process. The commutativity of conjunction can be captured by a semi-axiomatic proof of $A_1 \wedge A_2 \vdash A_2 \wedge A_1$, and its computational content is a shared memory process of type $p : A_1 \times A_2 \vdash q : A_2 \times A_1$ that creates a new pair at address q by swapping the components of the existing pair at address p:

$$p : A_1 \times A_2 \vdash \text{read } p \ (\langle x_1, x_2 \rangle \Rightarrow \text{write } q \ \langle x_2, x_1 \rangle) :: (q : A_2 \times A_1)$$

$$\frac{\dfrac{}{A_1, A_2 \vdash A_2 \wedge A_1} \ \wedge\text{A}}{A_1 \wedge A_2 \vdash A_2 \wedge A_1} \ \wedge\text{L}$$

The process first reads from address p and binds variables x_1 and x_2 to the pair of addresses that are stored there. Then it writes those same addresses in reverse order as a pair at address q.

4

Unit, type 1.

The unit type $\mathbf{1}$ is the nullary form of the product type $A_1 \times A_2$ and arises in Curry–Howard correspondence with $\top$. The $\top_A$ axiom becomes a typing rule for the construct $\mathsf{write}\, a\, \langle\rangle$, and the $\top_L$ rule (which, also being an instance of weakening, is uninteresting in terms of provability, but is computationally relevant) becomes a typing rule for the construct $\mathsf{read}\, a\, (\langle\rangle \Rightarrow P)$.

$$\frac{}{\cdot \vdash \top}\, \top_A \qquad\qquad \frac{}{\cdot \vdash \mathsf{write}\, a\, \langle\rangle :: (a:\mathbf{1})}\, \mathbf{1}_A \qquad a\ \boxed{} \ \rightsquigarrow\ a\ \boxed{\langle\rangle}$$

$$\frac{\Gamma \vdash C}{\Gamma, \top \vdash C}\, \top_L \qquad\qquad \frac{\Gamma \vdash P :: (c:C)}{\Gamma, a{:}\mathbf{1} \vdash \mathsf{read}\, a\, (\langle\rangle \Rightarrow P) :: (c:C)}\, \mathbf{1}_L$$

Tagged unions, type $\oplus\{\ell\colon A_\ell\}_{\ell\in L}$.

Disjunction corresponds to a labeled sum type, $\oplus\{\ell\colon A_\ell\}_{\ell\in L}$, for tagged unions. Being a positive connective, like conjunction and truth, disjunction's $\vee_{A_k}$ axiom in the semi-axiomatic sequent calculus becomes a typing rule for writing a tagged address. Specifically, the process $\mathsf{write}\, a\, k\langle a_k\rangle$ writes a tag k and an address a_k into memory at address a.
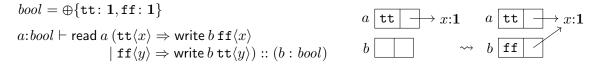
$$\frac{(k \in \{1,2\})}{A_k \vdash A_1 \vee A_2}\, \vee_{A_k} \qquad \frac{(k \in L)}{a_k{:}A_k \vdash \mathsf{write}\, a\, k\langle a_k\rangle :: (a : \oplus\{\ell : A_\ell\}_{\ell\in L})}\, \oplus_{A_k} \qquad a\ \boxed{}\ \rightsquigarrow\ a\ \boxed{k\ } {\scriptstyle\begin{matrix}a_k\\\uparrow\end{matrix}}$$

Symmetrically – and adhering to the pattern for positive types – the semi-axiomatic sequent calculus's $\vee_L$ rule becomes a typing rule for the reading construct $\mathsf{read}\, a\, (\ell\langle x_\ell\rangle \Rightarrow P_\ell)_{\ell\in L}$ that branches on the tag that it reads from address a.

$$\frac{\forall \ell \in \{1,2\}\colon \Gamma, A_\ell \vdash C}{\Gamma, A_1 \vee A_2 \vdash C}\, \vee_L \qquad \frac{\forall \ell \in L\colon \Gamma, x_\ell{:}A_\ell \vdash P_\ell :: (c:C)}{\Gamma, a{:}\oplus\{\ell : A_\ell\}_{\ell\in L} \vdash \mathsf{read}\, a\, (\ell\langle x_\ell\rangle \Rightarrow P_\ell)_{\ell\in L} :: (c:C)}\, \oplus_L$$

The process $\mathsf{read}\, a\, (\ell\langle x_\ell\rangle \Rightarrow P_\ell)_{\ell\in L}$ reads the tag, say $k \in L$, stored at address a and selects the corresponding branch. The variable x_k is bound to the address that was tagged by k, and execution continues according to P_k.

Example. At this point, we can consider another simple example. Booleans can be described with the type $\oplus\{\mathtt{tt}\colon \mathbf{1}, \mathtt{ff}\colon \mathbf{1}\}$, which we abbreviate as *bool*. The following process reads the boolean stored at address a and then writes its negation at address b. The diagram shows the process's execution when the tag stored at address a is $\mathtt{tt}$; the other case is symmetric.

$$bool = \oplus\{\mathtt{tt}\colon \mathbf{1}, \mathtt{ff}\colon \mathbf{1}\}$$

$$a{:}bool \vdash \mathsf{read}\, a\, (\mathtt{tt}\langle x\rangle \Rightarrow \mathsf{write}\, b\, \mathtt{ff}\langle x\rangle$$
$$| \ \mathtt{ff}\langle y\rangle \Rightarrow \mathsf{write}\, b\, \mathtt{tt}\langle y\rangle) :: (b : bool)$$

Functions, type $A_1 \to A_2$.

Being a negative proposition, the implication $A_1 \supset A_2$ follows a story dual to that of the positive conjunction $A_1 \wedge A_2$. Unlike the positive types, which write with axioms and read with left rules, the function type $A_1 \to A_2$ that corresponds to implication writes with a right rule and reads with an axiom.

The semi-axiomatic sequent calculus's $\supset_R$ rule therefore becomes a typing rule for the process $\mathsf{write}\, a\, (\langle x, z\rangle \Rightarrow P)$. In practice, this process might write (a pointer to) a closure to address a, but closures exist at a lower level of abstraction than the Curry–Howard correspondence between the semi-axiomatic sequent calculus and SAX supports. For this reason, we think of the process $\mathsf{write}\, a\, (\langle x, z\rangle \Rightarrow P)$ as writing the *continuation* $(\langle x, z\rangle \Rightarrow P)$.

$$\frac{\Gamma, A_1 \vdash A_2}{\Gamma \vdash A_1 \supset A_2}\, \supset_R \qquad \frac{\Gamma, x{:}A_1 \vdash P :: (z : A_2)}{\Gamma \vdash \mathsf{write}\, a\, (\langle x, z\rangle \Rightarrow P) :: (a : A_1 \to A_2)}\, \to_R \qquad a\ \boxed{}\ \rightsquigarrow\ a\ \boxed{\langle x, z\rangle \Rightarrow P}$$

5

The semi-axiomatic sequent calculus's $\supset$A axiom becomes a typing rule for the construct $\mathsf{read}\, a\, \langle a_1, a_2 \rangle$.

$$\frac{}{A_1 \supset A_2, A_1 \vdash A_2}\ {\supset}\text{A} \qquad\qquad \frac{}{a{:}A_1 \to A_2, a_1{:}A_1 \vdash \mathsf{read}\, a\, \langle a_1, a_2 \rangle :: (a_2 : A_2)}\ {\to}\text{A}$$

This construct reads the continuation ($\langle x, z \rangle \Rightarrow P$) stored at address a and passes it two addresses: a_1, the address of the function argument of type A_1 to which the continuation should be applied; and a_2, the address to which the called function should write its result of type A_2. The variables x and z are bound to these addresses, respectively, and execution continues according to P.

Other types.

Other SAX types and process constructs also emerge from this Curry–Howard reading of the semi-axiomatic sequent calculus. For example, it is possible to adapt the negative polarity conjunction from intuitionistic logic to a type of lazy records (e.g., like call-by-push-value [13]). Its SAX typing rules are dual to those for tagged unions [6]; moreover, because function types already exemplify the key aspects of negative types in SAX, we do not present the details of negative conjunction and lazy records in this paper.

Another possible SAX type is $\downarrow A$, which arises from the downshift of adjoint logic [3,19,17]. From a purely logical standpoint, $\downarrow A$ is not especially interesting because $\downarrow A$ is logically equivalent to A. Neither is $\downarrow A$ particularly useful in SAX processes: it merely serves to introduce indirections beyond those already present in abundance in SAX. However, the type $\downarrow A$ is also present in SNAX and becomes much more useful there, so we postpone further discussion of type $\downarrow A$, processes $\mathsf{write}\, a\, \langle b \rangle$ and $\mathsf{read}\, a\, (\langle x \rangle \Rightarrow P)$, and their typing rules and operational semantics to section 3.

2.1 Adding recursion to SAX

For most practical examples, recursively defined types and processes are needed. Recursion in SAX goes beyond a strict Curry–Howard correspondence with the semi-axiomatic sequent calculus, but only in the same way that recursive functional programming goes beyond a strict Curry–Howard correspondence with natural deduction.

Instead of adding an explicit μ, fold, and unfold operators, we use recursive type definitions and recursive process definitions. Recursive type definitions have the form $t = A$; we choose to treat them equirecursively so that t and its unfolding, A, are indistinguishable. Under this interpretation, type definitions like $t = t$ would not be sensible, so SAX requires type definitions to be *contractive*: each type name t must (eventually) unfold to a logical type constructor like $\times$ or $\to$.

Recursive process definitions have the form $\mathsf{proc}\, p\, (z{:}C)\, (x_1{:}A_1) \cdots (x_n{:}A_n) = P$, where the argument $x_1{:}A_1, \ldots, x_n{:}A_n$ may be read by process P, and $z{:}C$ is the destination to which P will write. (If the process P diverges, it escapes the obligation to write by postponing that obligation indefinitely.) Calls to these recursively defined processes are made by the process construct $\mathsf{call}\, p\, c\, a_1 \cdots a_n$. Its typing rule is the following where Σ is a signature that holds recursive type and process definitions. (See section 2.3 for more details on signatures.)

$$\frac{(\mathsf{proc}\, p\, (z{:}C)\, (x_1{:}A_1) \cdots (x_n{:}A_n) = P) \in \Sigma}{a_1{:}A_1, \ldots, a_n{:}A_n \vdash_\Sigma \mathsf{call}\, p\, c\, a_1 \cdots a_n :: (c : C)}\ \text{CALL}$$

Operationally, $\mathsf{call}\, p\, c\, a_1 \cdots a_n$ will lookup the definition for p and execution will continue according to the definition's body, substituting addresses for the argument and destination variables. Accordingly, recursive process definitions are required to be contractive.

2.2 Extended example: mapping a function across a linked list of booleans

As an extended example of the SAX type theory, we can consider the recursive type *boollist* that describes linked lists of booleans. (A polymorphic type of linked lists is not currently possible in SAX, but adding

parametric polymorphism to SAX is a primary goal of future work.) The type *boollist* is defined as follows.

$$boollist = \oplus\{\texttt{nil}: \mathbf{1}, \texttt{cons}: bool \times boollist\}$$

(a) xs [`nil` | →] → $u{:}\mathbf{1}$ (b) xs [`cons` | →] → p [| | →] → $xs'{:}boollist$, with $x{:}bool$ above.

Specifically, an address xs of type *boollist* stores either: *(a)* the tag `nil` and an address u of type $\mathbf{1}$; or *(b)* the tag `cons` and an address p of type $bool \times boollist$, which itself stores a pair of addresses of types *bool* and *boollist*, respectively. Aside from the high degree of indirection and the use of tags to replace null pointers, this is a fairly recognizable representation of a linked list of booleans.

A *map* function for mapping a unary boolean function f across a linked list xs of booleans and writing the resulting list to destination ys is given by the following recursive definition.

$$
\begin{array}{ll}
\textsf{proc } map \ (ys : boollist) \ (f : bool \to bool) \ (xs : boollist) = & \\
\quad \textsf{read } xs \ (& \text{\% read and branch on tag at } xs \\
\quad\quad \texttt{nil}\langle u\rangle \Rightarrow \textsf{copy } ys \ xs & \text{\% copy (empty) input list } xs \text{ to destination } ys \\
\quad\quad |\ \texttt{cons}\langle p\rangle \Rightarrow \textsf{read } p \ (\langle x, xs'\rangle \Rightarrow & \text{\% read pair at } p \\
\quad\quad\quad\quad y \leftarrow \textsf{read } f \ \langle x, y\rangle; & \text{\% allocate } y \text{ and call } f \text{ with destination } y \\
\quad\quad\quad\quad ys' \leftarrow \textsf{call } map \ ys' \ f \ xs'; & \text{\% allocate } ys' \text{ and call map recursively with dest. } ys' \\
\quad\quad\quad\quad q \leftarrow \textsf{write } q \ \langle y, ys'\rangle; & \text{\% allocate } q \text{ and write pair of } y \text{ and } ys' \\
\quad\quad\quad\quad \textsf{write } ys \ \texttt{cons}\langle q\rangle)) & \text{\% write the tagged pair to the original destination } ys
\end{array}
$$

2.3 Details of the SAX type theory

The types in SAX are as described above; recursive type definitions $t = A$ and recursive process definitions $\textsf{proc } p \ z \ x_1 \cdots x_n = P$ are collected in signatures Σ. A signature indexes the SAX typing judgment: $\Gamma \vdash_\Sigma P :: (a : A)$. However, because none of the typing rules affect the signature, it is frequently elided.

$$
\begin{array}{lrl}
Types & A, B, C & ::= A \times B \mid \mathbf{1} \mid \oplus\{\ell: A_\ell\}_{\ell \in L} \mid {\downarrow}A \mid A \to B \mid t \\
Signatures & \Sigma & ::= (\cdot) \mid \Sigma, t = A \mid \textsf{proc } p \ z \ x_1 \cdots x_n = P
\end{array}
$$

SAX processes P and Q have one of five forms: allocations, copies, writes, reads, and calls.

$$
\begin{array}{lrl}
Processes & P, Q & ::= x \leftarrow P; Q \mid \textsf{copy } a \ b \mid \textsf{write } a \ S \mid \textsf{read } a \ T \mid \textsf{call } p \ d \ a_1 \cdots a_n \\
Storables & S & ::= \langle a_1, a_2\rangle \mid \langle\rangle \mid k\langle a\rangle \mid \langle a\rangle \mid (\langle x, z\rangle \Rightarrow P) \\
Co\text{-}storables & T & ::= (\langle x_1, x_2\rangle \Rightarrow P) \mid (\langle\rangle \Rightarrow P) \mid (\ell\langle x_\ell\rangle \Rightarrow P_\ell)_{\ell \in L} \mid (\langle x\rangle \Rightarrow P) \mid \langle a_1, a_2\rangle
\end{array}
$$

Writes rely on a syntactic category of storables S, which are the data that may be written into a memory cell. There is one storable for each type constructor: pairs of addresses, $\langle a_1, a_2\rangle$; unit value, $\langle\rangle$; tagged address, $k\langle a\rangle$; pointers, $\langle a\rangle$; and function continuations, $(\langle x, z\rangle \Rightarrow P)$. Reads rely on a syntactic category of co-storables T that are dual to storables. Once again, there is one form of co-storable for each type constructor: continuations for pairs, unit values, tagged unions, and pointers; and pairs of addresses to be passed to function continuations. We will not repeat the process typing rules here.

The operational semantics of the SAX type theory is based on multiset rewriting, using three semantic objects: $\textsf{thread}(a, P)$ denotes a running process P that must write to address a; $\textsf{cell}(a, \square)$ denotes an empty memory cell at address a, *i.e.*, one that has been allocated but not yet written; and $!\textsf{cell}(a, S)$ denotes a filled memory cell at address a, *i.e.*, one that has been allocated and now stores S. The '!' is notation borrowed from linear logic and denotes that filled cells implicitly persist across rewriting steps.

A configuration $\mathcal{C}$ is a collection of filled cells and threads with corresponding empty cells:

$$
\begin{array}{lrl}
Configurations & \mathcal{C} & ::= (\cdot) \mid \mathcal{C}_1 \, \mathcal{C}_2 \mid \textsf{thread}(a, P) \, \textsf{cell}(a, \square) \mid !\textsf{cell}(a, S) \\
Configuration \ contexts & \Phi & ::= (\cdot) \mid \Phi, a{:}A
\end{array}
$$

We say that a configuration $\mathcal{C}$ is *final* if it consists only of filled cells $!\mathsf{cell}(a, S)$. Configurations are typed with a judgment $\Phi \vDash \mathcal{C} :: \Phi'$. Configuration contexts Φ have the same syntactic structure as contexts Γ, but configuration contexts Φ are not subject to contraction; in the judgment $\Phi \vDash \mathcal{C} :: \Phi'$, the addresses in Φ are therefore presumed to be distinct. Also, in $\mathcal{C}$ and Φ, we write a, b, and c for readability, but they are all fresh runtime addresses α (not variables x). The configuration typing judgment has the rules:

$$\frac{}{\Phi \vDash (\cdot) :: \Phi} \;\text{EMP} \qquad\qquad \frac{\Phi \vDash \mathcal{C}_1 :: \Phi' \quad \Phi' \vDash \mathcal{C}_2 :: \Phi''}{\Phi \vDash \mathcal{C}_1\, \mathcal{C}_2 :: \Phi''} \;\text{JOIN}$$

$$\frac{(a \notin \operatorname{dom} \Phi) \quad (\Phi \supseteq \Gamma) \quad \Gamma \vdash P :: (a : A)}{\Phi \vDash \mathsf{thread}(a, P)\, \mathsf{cell}(a, \square) :: (\Phi, a{:}A)} \;\text{THREAD} \qquad \frac{(a \notin \operatorname{dom} \Phi) \quad (\Phi \supseteq \Gamma) \quad \Phi \vdash \mathsf{write}\, a\, S :: (a : A)}{\Phi \vDash !\mathsf{cell}(a, S) :: (\Phi, a{:}A)} \;\text{CELL}$$

Notice that both the THREAD and CELL rules check that the address a is not already present in the domain of Γ, to ensure that each address has a unique type. Both rules also rely on the static typing judgment for SAX processes.

The concurrent operational semantics is then described by the following multiset rewriting clauses.

$\mathsf{thread}(c, (x \leftarrow P; Q))$ $\qquad\qquad\qquad$ (α fresh) $\qquad$ where $S \triangleright T$ is given by
$\qquad \longmapsto \mathsf{thread}(\alpha, [\alpha/x]P)\, \mathsf{cell}(\alpha, \square)\, \mathsf{thread}(c, [\alpha/x]Q)$ $\qquad$ $\langle a_1, a_2 \rangle \triangleright (\langle x_1, x_2 \rangle \Rightarrow P) = [a_1/x_1, a_2/x_2]P$

$\mathsf{thread}(a, \mathsf{copy}\, a\, b)\, \mathsf{cell}(a, \square)\, !\mathsf{cell}(b, S) \longmapsto !\mathsf{cell}(a, S)$ $\qquad\qquad\quad$ $\langle\rangle \triangleright (\langle\rangle \Rightarrow P) = P$

$\mathsf{thread}(a, \mathsf{write}\, a\, S)\, \mathsf{cell}(a, \square) \longmapsto !\mathsf{cell}(a, S)$ $\qquad\quad$ $k\langle a \rangle \triangleright (\ell\langle x_\ell \rangle \Rightarrow P_\ell)_{\ell \in L} = [a/x_k]P_k \quad (k \in L)$

$\mathsf{thread}(c, \mathsf{read}\, a\, T)\, !\mathsf{cell}(a, S) \longmapsto \mathsf{thread}(c, S \triangleright T)$ $\qquad\qquad\quad$ $\langle a \rangle \triangleright (\langle x \rangle \Rightarrow P) = [a/x]P$

$\mathsf{thread}(c, \mathsf{call}\, p\, c\, a_1 \cdots a_n)$ $\qquad\qquad\qquad\qquad\qquad\qquad$ $(\langle x, z \rangle \Rightarrow P) \triangleright \langle a_1, a_2 \rangle = [a_1/x, a_2/z]P$
$\qquad \longmapsto \mathsf{thread}(c, [c/z, a_1/x_1, \ldots, a_n/x_n]P)$
$\qquad$ (where $\mathsf{proc}\, p\, z\, x_1 \cdots x_n = P$)

Preservation and progress hold for the SAX type theory [6].

Theorem 2.1 (Preservation) *If* $\Phi_0 \vDash \mathcal{C} :: \Phi$ *and* $\mathcal{C} \longmapsto \mathcal{C}'$, *then* $\Phi_0 \vDash \mathcal{C}' :: \Phi'$ *for some* $\Phi' \supseteq \Phi$.

Theorem 2.2 (Progress) *If* $\vDash \mathcal{C} :: \Phi$, *then either* $\mathcal{C}$ *is final or* $\mathcal{C} \longmapsto \mathcal{C}'$ *for some* $\mathcal{C}'$.

3   SNAX type theory for data layout

The SAX type theory does not take layout considerations into account in the sense that addresses remain entirely abstract. As a concrete example, a single `cons` node in a linked list of booleans can be thought of as laid out in SAX with three indirections, while a more compact flat layout would be more memory-efficient:

Instead of `cons` $\to \cdots$, *the flat layout* `cons | tt` $\to \cdots$ *is likely preferable.*

SAX's rather extreme level of indirection arises from its heavy reliance on the CUT rule. This suggests that if we want to account for data layout in a SAX-like type theory, an understanding of cut elimination and the structure of cut-free proofs in the semi-axiomatic sequent calculus may provide some insight.

3.1   Cut elimination in the semi-axiomatic sequent calculus

Unfortunately, the semi-axiomatic sequent calculus does not immediately satisfy Gentzen-style cut elimination. As a counterexample, there is a semi-axiomatic proof of $B, B \supset A_1, A_2 \vdash A_1 \wedge A_2$, namely

$$\frac{\dfrac{}{B, B \supset A_1 \vdash A_1} \;\supset\text{A} \quad \dfrac{}{A_1, A_2 \vdash A_1 \wedge A_2} \;\wedge\text{A}}{B, B \supset A_1, A_2 \vdash A_1 \wedge A_2} \;\text{CUT},$$

8

but there is no *cut-free* proof: the cut that appears here is essential and cannot be eliminated.

However, notice that the above cut has a *subformula property*: the cut formula (here A_1) is a proper subformula of one of the conclusion sequent's formulas (here $A_1 \wedge A_2$). Moreover, this subformula property derives from the use of the cut formula within the occurrence of the $\wedge$A axiom. Such formulas that are used by axioms are said to be *eligible* to act as the cut formula in one of these well-behaved cuts, which are called *snips*. Proof-theoretically, both eligibility and snips are thought of as *properties* of proofs: we can implicitly ignore that a formula is eligible and that a cut is a snip because eligibility is not part of the structure of the proof rules themselves.

Although semi-axiomatic proofs cannot be transformed to be fully cut-free, they can be transformed so that the only remaining cuts are these well-behaved snips. For example, the above proof can be reformulated using a snip as follows; the eligible formulas are indicated by underlining.

$$\dfrac{\dfrac{}{\underline{B}, B \supset A_1 \vdash \underline{A_1}} \supset \text{A} \quad \dfrac{}{\underline{A_1}, \underline{A_2} \vdash A_1 \wedge A_2} \wedge \text{A}}{\underline{B}, B \supset A_1, \underline{A_2} \vdash A_1 \wedge A_2} \text{SNIP}$$

For more on the proof-theoretic view of eligibility as a property of proofs, we refer the reader to [6].

3.2 Making eligibility first-class

As we saw in section 2, the SAX type theory is a Curry–Howard interpretation of the semi-axiomatic sequent calculus that ignores eligibility and snips, viewing them as mere technical refinements used in recovering (a modified form of) cut elimination. However, in this section, we elevate eligibility and snips to be first-class concepts in the proof theory and thereby obtain, by Curry–Howard correspondence, a type theory that gives a clean, logically grounded account of data layout in shared memory concurrency. We will call the resulting type theory SNAX, short for "SAX with snips".

Eligibility, addresses, contexts, and judgments.

To make eligibility a structural component of the semi-axiomatic sequent calculus, contexts Γ now contain ordinary antecedents A and eligible antecedents $\underline{A}$ – eligibility is no longer a refinement property, but instead intrinsic to an antecedent. Owing to the subformula structure that underlies eligibility, an eligible antecedent $\underline{A}$ in a proof will correspond to an eligible address $a{\cdot}p : A$, where p is a *projection* and $a \cdot p$ is a new form of address. For example, we will see shortly that just as A_1 is an eligible antecedent within the axiom for $A_1 \wedge A_2$, so will $a \cdot \pi_1$ be the address of the first component of a pair that itself begins at address a. (We will sometimes elide the $\cdot$ operator in $a \cdot p$, especially when π_1 and π_2 are involved.)

$$
\begin{array}{llll}
\textit{Contexts} & \Gamma ::= (\cdot) \mid \Gamma, A \mid \Gamma, \underline{A} & \textit{Contexts} & \Gamma ::= (\cdot) \mid \Gamma, a{:}A \mid \Gamma, \underline{a{\cdot}p{:}A} \\
& & \textit{Addresses} & a, b, c, d ::= x \mid \alpha \mid a \cdot p \\
& & \textit{Projections} & p ::= \pi_1 \mid \pi_2 \mid \overline{\ell} \mid p_1 \cdot p_2
\end{array}
$$

We say that a *strictly extends* c and write $a \succ c$ whenever $a = c \cdot p$ for some projection p; we also say that a *(weakly) extends* c and write $a \succcurlyeq c$ whenever either $a = c$ or $a \succ c$. Note that every address has a variable (either a static x or a runtime α) at its head. This reflects the idea that memory will be allocated in blocks and, at the level of abstraction that SNAX provides, each address exists relative to the base address of the block to which it belongs.

Projections provide a clean, logical description of the *abstract* layout of data. On the other hand, machine code relies on *concrete* address arithmetic. To bridge this gap, we envision that, at a lower level of abstraction, projections would be concretized. One simple concretization $(a)^\star$ underlying our diagrams would be the following:

$$
\begin{array}{lll}
(a \cdot \overline{k})^\star = a^\star + 1 & \text{where} \quad |A_1 \times A_2| = |A_1| + |A_2| & |{\downarrow}A| = 1 \\
(a \cdot \pi_1)^\star = a^\star & |\mathbf{1}| = 0 & |A_1 \to A_2| = 1 \\
(a \cdot \pi_2)^\star = a^\star + |A_1| \ \text{(when } a : A_1 \times A_2\text{)} & |\oplus\{\ell{:}\, A_\ell\}_{\ell \in L}| = 1 + \max_{\ell \in L} |A_\ell| &
\end{array}
$$

9

However, this is not the only possible concretization. Practical compilers employ layout optimizations, such as bit-packing to reduce the space needed for the tags of consecutive tagged unions. By using projections, SNAX is agnostic about the particular concretization chosen and remains flexible.

Aside from the changes to addresses and contexts, sequents and typing judgments look the same as in SAX: $\Gamma \vdash A$ and $\Gamma \vdash P :: (a : A)$, respectively. As a general convention, for each typing judgment $\Gamma \vdash P :: (a : A)$, we presuppose that $a \not\geq b$ for all $b{:}B \in \Gamma$; the typing rules will maintain this invariant. (This property is also provable for all $\underline{b{:}B} \in \Gamma$ [see Lemma B.2].) If we did not have this well-formedness condition, there would incorrectly be two writers to address a: one implicitly as part of the writer to b, and the other being P.

To achieve (its modified form of) cut elimination, the semi-axiomatic sequent calculus also uses eligible succedents. However, because the SNAX type theory does not model the internal layout of function closures, eligible succedents do not appear in SNAX. As a consequence, cut elimination (even in modified form) will not hold for SNAX. Since we are primarily interested in operational aspects, we avoid here the technical complications required to restore it, since type preservation and progress *do* hold.

Weakening and contraction.

To correctly maintain the connection between an eligible antecedent and the axiom from which it derives its eligibility, some care must be taken with weakening and contraction. Ordinary antecedents and their corresponding addresses are still subject to weakening, but eligible ones may *not* be weakened away (or otherwise, their eligibility would fail to be derived from an axiom).

$$\frac{\Gamma \vdash C}{\Gamma, A \vdash C}\ \text{W} \qquad\qquad \frac{\Gamma \vdash P :: (c : C)}{\Gamma, a{:}A \vdash P :: (c : C)}\ \text{W}$$

(no weakening for $\underline{A}$ and $\underline{a{:}A}$)

Contraction for two ordinary antecedents and their corresponding addresses occurs as in SAX. Furthermore, contracting an eligible antecedent $\underline{A}$ and an ordinary antecedent A together into $\underline{A}$ is permitted: the eligibility of the resulting $\underline{A}$ will still be traceable to an axiom. But contracting two copies of $\underline{a{:}A}$ into one is not permitted in SNAX. Such a contraction rule would not be harmful, but neither would it be useful since no two eligible antecedents can be assigned the same address in a derivable judgment (see Lemma B.3). In this way, eligible antecedents $\underline{A}$ and their corresponding addresses $\underline{a{:}A}$ follow an almost linear discipline.

$$\frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}\ \text{C} \qquad\qquad \frac{\Gamma, a{:}A, a{:}A \vdash P :: (c : C)}{\Gamma, a{:}A \vdash P :: (c : C)}\ \text{C}$$

$$\frac{\Gamma, \underline{A}, A \vdash C}{\Gamma, \underline{A} \vdash C}\ \text{C}_\text{E} \qquad\qquad \frac{\Gamma, \underline{a{:}A}, a{:}A \vdash P :: (c : C)}{\Gamma, \underline{a{:}A} \vdash P :: (c : C)}\ \text{C}_\text{E}$$

(no contraction for $\underline{A}, \underline{A}$ and $\underline{a{:}A}, \underline{a{:}A}$)

Cut, snip, and identity.

The essential idea of the SNAX type theory is that, once eligibility and snips are first-class, we have the freedom to give different computational interpretations to cuts and snips. SNAX retains the CUT rule from SAX but also has a distinct SNIP$^+$ rule. The CUT rule and $x \leftarrow P; Q$ construct continue to act as in SAX, allocating memory for data of type A and then running P and Q in parallel. (The type A should be inferred or given, if SNAX is used as a source language, so that allocation can depend on the type.)

$$\frac{\Gamma_1 \vdash A \quad \Gamma_2, A \vdash C}{\Gamma_1, \Gamma_2 \vdash C}\ \text{CUT} \qquad\qquad \frac{\Gamma_1 \vdash P :: (x : A) \quad \Gamma_2, x{:}A \vdash Q :: (c : C) \quad (x\ \text{fresh})}{\Gamma_1, \Gamma_2 \vdash x \leftarrow P; Q :: (c : C)}\ \text{CUT}$$

But the SNIP$^+$ rule is different. Its $P;Q$ construct does not (re-)allocate memory at address a because it is an eligible address and, as such, refers to a location *within* a block already allocated by an earlier CUT rule. Instead, it simply runs processes P and Q in parallel, with reads at address a that occur in Q blocking until that address has been written to by P.[5]

$$\frac{\Gamma_1 \vdash A \quad \Gamma_2, \underline{A} \vdash C}{\Gamma_1, \Gamma_2 \vdash C} \ \text{SNIP}^+ \qquad\qquad \frac{\Gamma_1 \vdash P :: (a : A) \quad \Gamma_2, \underline{a{:}A} \vdash Q :: (c : C)}{\Gamma_1, \Gamma_2 \vdash P;Q :: (c : C)} \ \text{SNIP}^+$$

In the pure proof theory, there would also be a symmetric SNIP$^-$ rule for eligible succedents. However, as previously mentioned, SNAX ignores eligibility in succedents and therefore does not include SNIP$^-$.

Eligibility does not enter into the identity rule; it remains as in SAX. (Note that the ID typing rule is one place where it is essential to have the presupposition on typing judgments $\Gamma \vdash P :: (a : A)$ that $a \not\ge b$ for all $b{:}B \in \Gamma$. Without it, the ID rule would not be operationally sensible.)

$$\frac{}{A \vdash A} \ \text{ID} \qquad\qquad \frac{}{b{:}A \vdash \mathsf{copy}\ a\ b :: (a : A)} \ \text{ID}$$

Pairs, type $A_1 \times A_2$.

Recall from section 2 that SAX uses the construct $\mathsf{write}\ a\ \langle a_1, a_2 \rangle$ to write a pair of addresses $\langle a_1, a_2 \rangle$ to address a. Because these addresses point to the data that are conceptually the components of a pair of values, the layout is very indirect.

In contrast, for SNAX, we first observe that the purely logical axiom $\wedge$A has both antecedents A_1 and A_2 eligible (as denoted by the underlining), because both are proper subformulas that are used in an axiom. This subformula structure is then reflected in the $\times$A typing rule by assigning addresses $a\pi_1$ and $a\pi_2$ to A_1 and A_2, respectively. Because these addresses are locally calculable from a, they are not needed in the process syntax $\mathsf{write}\ a\ \langle \_, \_ \rangle$.

$$\frac{}{\underline{A_1}, \underline{A_2} \vdash A_1 \wedge A_2} \ \wedge\text{A} \qquad\qquad \frac{}{\underline{a\pi_1{:}A_1}, \underline{a\pi_2{:}A_2} \vdash \mathsf{write}\ a\ \langle \_, \_ \rangle :: (a : A_1 \times A_2)} \ \times\text{A}$$

The above picture depicts a particular flat layout for SNAX pairs in which $a\pi_1$ precedes $a\pi_2$. At a formal level, SNAX abstracts away from these particulars: SNAX requires only that $a\pi_1$ and $a\pi_2$ are calculable from $a : A_1 \times A_2$ and that $a\pi_1 \cdot p_1 \neq a\pi_2 \cdot p_2$ for all projections p_1 and p_2. For example, as far as SNAX is concerned, an equally correct picture could have $a\pi_2$ preceding $a\pi_1$.

SNAX also tweaks the construct for reading an address a of type $A_1 \times A_2$. Instead of binding variables with $\mathsf{read}\ a\ (\langle x_1, x_2 \rangle \Rightarrow P)$, we now use $\mathsf{read}\ a\ (\langle \_, \_ \rangle \Rightarrow P)$. Bound variables are no longer needed because we know that a SNAX pair at address a will always have its components located at the relative addresses $a\pi_1$ and $a\pi_2$. Logically, though, the static typing rule for reading at type $A_1 \times A_2$ is still derived from the semi-axiomatic sequent calculus's $\wedge$L rule.

$$\frac{\Gamma, A_1, A_2 \vdash C}{\Gamma, A_1 \wedge A_2 \vdash C} \ \wedge\text{L} \qquad\qquad \frac{\Gamma, a\pi_1{:}A_1, a\pi_2{:}A_2 \vdash P :: (c : C)}{\Gamma, a{:}A_1 \times A_2 \vdash \mathsf{read}\ a\ (\langle \_, \_ \rangle \Rightarrow P) :: (c : C)} \ \times\text{L}$$

The $\times$L rule demonstrates that, while eligible antecedents always correspond to addresses that are projections $a \cdot p$, the converse is not true: not all address projections correspond to eligible antecedents. Here, although $a\pi_1{:}A_1$ and $a\pi_2{:}A_2$ appear in the premise of the $\times$L rule, the antecedents A_1 and A_2 are *not* eligible in the premise of the $\wedge$L rule.

(In common layouts we considered, such as the one depicted in the above diagram, the processes $\mathsf{write}\ a\ \langle \_, \_ \rangle$ and $\mathsf{read}\ a\ (\langle \_, \_ \rangle \Rightarrow P)$ do not actually write nor read runtime information, their names notwithstanding. It would be possible to consider erasing $\mathsf{write}\ a\ \langle \_, \_ \rangle$ from the syntax and reducing

---

[5] Similar to the situation for SAX, it is possible to give sequential and call-by-need semantics to the SNAX CUT and SNIP$^+$ constructs.

read $a\,(\langle\_,\_\rangle \Rightarrow P)$ to P. We do not do so because it diverges from the logical foundations and complicates type checking.)

Example. Because SNAX tracks eligibility explicitly, the structure of the proof of commutativity of conjunction changes: $\textsc{snip}^+$ and $\textsc{id}$ rules are needed to mediate the ordinary A_1 and A_2 of the $\wedge\textsc{l}$ rule's premise and the eligible $\underline{A_1}$ and $\underline{A_2}$ of the $\wedge\textsc{a}$ axiom. The corresponding SNAX process involves address projections and explicit copying of data so that the flat layout of pairs is respected.

$$
\cfrac{
\cfrac{
\overline{A_2 \vdash A_2}\ \text{ID} \quad
\cfrac{
\cfrac{\overline{A_1 \vdash A_1}\ \text{ID} \quad \overline{\underline{A_2},\underline{A_1} \vdash A_2 \wedge A_1}\ \wedge\text{A}}
{\underline{A_2},A_1 \vdash A_2 \wedge A_1}\ \textsc{snip}^+
}
{A_1, A_2 \vdash A_2 \wedge A_1}\ \textsc{snip}^+
}
{A_1 \wedge A_2 \vdash A_2 \wedge A_1}\ \wedge\text{L}
}
$$

$$
p : A_1 \times A_2 \vdash \mathsf{read}\ p\,(\langle\_,\_\rangle \Rightarrow \\
\mathsf{copy}\,(q\pi_1)\,(p\pi_2)\,; \\
\mathsf{copy}\,(q\pi_2)\,(p\pi_1)\,; \\
\mathsf{write}\ q\,\langle\_,\_\rangle) \qquad :: (q : A_2 \times A_1)
$$

Unit, type 1.

The rules for $\top$ in the semi-axiomatic sequent calculus do not involve eligibility – after all, $\top$ has no proper subformula – and so the SNAX process constructs and typing rules involving $\mathbf{1}$ are as in SAX. They are repeated here for convenience.

$$
\cfrac{}{\cdot \vdash \mathsf{write}\ a\,\langle\rangle :: (a : \mathbf{1})}\ \mathbf{1}\text{A}
\qquad\qquad
\cfrac{\Gamma \vdash P :: (c : C)}{\Gamma, a{:}\mathbf{1} \vdash \mathsf{read}\ a\,(\langle\rangle \Rightarrow P) :: (c : C)}\ \mathbf{1}\text{L}
$$

However, there is one operational difference: Now that SNAX provides an abstraction that supports conceptually flat layouts of pairs, we can think of data of type $\mathbf{1}$ as taking no space at runtime. This proves useful in some of the examples that will follow.

Tagged unions, type $\oplus\{\ell: A_\ell\}_{\ell\in L}$.

Disjunction still corresponds to a labeled sum type, $\oplus\{\ell: A_\ell\}_{\ell\in L}$, for tagged unions, as in SAX. However, instead of the indirect layout of SAX, the presence of an eligible antecedent in the $\vee\text{A}_k$ rule suggests a flat layout for tagged unions in which the tag and the underlying data are laid out side-by-side. In SNAX, this is abstracted using a new form of address projection, $a \cdot \overline{k}$, where k is the tag.

Instead of SAX's $\mathsf{write}\ a\,k\langle a_k\rangle$ for writing a tagged address, SNAX uses the construct $\mathsf{write}\ a\,k\langle\_\rangle$. This can be seen as fixing address a_k to be $a{\cdot}\overline{k}$ and then eliding it because it is calculable from the address a and tag k. This process is typed by the $\oplus\text{A}_k$ rule, which corresponds to the $\vee\text{A}_k$ axiom for disjunction. Following the pattern seen for pairs, the eligible antecedent $\underline{A_k}$ in the $\vee\text{A}_k$ axiom corresponds to the eligible $a{\cdot}\overline{k}{:}A_k$ in the $\oplus\text{A}_k$ typing rule.

$$
\cfrac{(k \in \{1,2\})}{\underline{A_k} \vdash A_1 \vee A_2}\ \vee\text{A}_k
\qquad\qquad
\cfrac{(k \in L)}{\underline{a{\cdot}\overline{k}{:}A_k} \vdash \mathsf{write}\ a\,k\langle\_\rangle :: (a : \oplus\{\ell: A_\ell\}_{\ell\in L})}\ \oplus\text{A}_k
$$

The SNAX construct for reading an address a of type $\oplus\{\ell: A_\ell\}_{\ell\in L}$ also differs slightly from its SAX counterpart. Once again, instead of binding variables x_ℓ in the branches of $\mathsf{read}\ a\,(\ell\langle x_\ell\rangle \Rightarrow P_\ell)_{\ell\in L}$, we take advantage of knowing that a tag ℓ at address a will always have its underlying data located at $a{\cdot}\overline{\ell}$ and use the construct $\mathsf{read}\ a\,(\ell\langle\_\rangle \Rightarrow P_\ell)_{\ell\in L}$. Logically, the typing rule is still derived from the semi-axiomatic sequent calculus's $\vee\text{L}$ rule. All of this follows the pattern seen for the type $A_1 \times A_2$.

$$
\cfrac{\forall \ell \in \{1,2\}\colon \Gamma, A_\ell \vdash C}{\Gamma, A_1 \vee A_2 \vdash C}\ \vee\text{L}
\qquad\qquad
\cfrac{\forall \ell \in L\colon \Gamma, a{\cdot}\overline{\ell}{:}A_\ell \vdash P_\ell :: (c : C)}{\Gamma, a{:}\oplus\{\ell: A_\ell\}_{\ell\in L} \vdash \mathsf{read}\ a\,(\ell\langle\_\rangle \Rightarrow P_\ell)_{\ell\in L} :: (c : C)}\ \oplus\text{L}
$$

Example. We can now revisit booleans of type $bool = \oplus\{\mathtt{tt}: \mathbf{1}, \mathtt{ff}: \mathbf{1}\}$ in the context of SNAX. Each address $a{:}bool$ stores only a tag, $\mathtt{tt}$ or $\mathtt{ff}$, and no space needs to be reserved for $a{\cdot}\overline{\mathtt{tt}} : \mathbf{1}$ or $a{\cdot}\overline{\mathtt{ff}} : \mathbf{1}$. A

process for reading a boolean at address a and writing its negation to address b is as follows; its execution in the case that a holds tag tt is shown.

$$bool = \oplus\{\texttt{tt}\colon \mathbf{1}, \texttt{ff}\colon \mathbf{1}\}$$

$a{:}bool \vdash \textsf{read } a \; (\texttt{tt}\langle\_\rangle \Rightarrow \textsf{write } (b \cdot \overline{\texttt{ff}}) \; \langle\rangle; \textsf{write } b \, \texttt{ff}\langle\_\rangle$
$\qquad\qquad | \; \texttt{ff}\langle\_\rangle \Rightarrow \textsf{write } (b \cdot \overline{\texttt{tt}}) \; \langle\rangle; \textsf{write } b \, \texttt{tt}\langle\_\rangle) :: (b : bool)$

$a \;\boxed{\texttt{tt}}\;\; b\;\boxed{} \;\;\rightsquigarrow\;\; a\;\boxed{\texttt{tt}}\;\; b\;\boxed{\texttt{ff}}$

Pointers, type $\downarrow\! A$.

The semi-axiomatic sequent calculus can include the shift proposition $\downarrow\! A$ from adjoint logic [3,19,17]. Because the semi-axiomatic sequent calculus consists of a single adjoint layer, A and $\downarrow\! A$ are logically equivalent. From a provability perspective, this makes $\downarrow\! A$ uninteresting, but the corresponding type, which we also write as $\downarrow\! A$, nevertheless has computational significance.[6]

As a proposition of positive polarity, $\downarrow\! A$ follows the pattern of having its axiom correspond to a typing rule for writes at that type.

$$\frac{}{\underline{A} \vdash \downarrow\! A}\; {\downarrow}\textsc{a} \qquad\qquad \frac{}{b{:}A \vdash \textsf{write } a\; \langle b\rangle :: (a : \downarrow\! A)}\; {\downarrow}\textsc{a} \qquad a\;\boxed{} \;\;\rightsquigarrow\;\; a\;\boxed{\uparrow}$$

From a computational standpoint, we can interpret the structure of the ${\downarrow}\textsc{a}$ axiom as writing an address of type A into an address of type $\downarrow\! A$. In other words, $\downarrow\! A$ is the type of pointers to data of type A.

In the proof theory, the ${\downarrow}\textsc{a}$ axiom must use an eligible antecedent $\underline{A}$ in order for cut elimination to hold. However, requiring the ${\downarrow}\textsc{a}$ *typing rule* to use an eligible address would be far too restrictive computationally – it would force a pointer at address a to point to only a specific projection of a, say $a{\cdot}{\downarrow}$. We certainly want pointers to arbitrary addresses, so we allow the ${\downarrow}\textsc{a}$ typing rule to use an ordinary $b{:}A$. (The ${\downarrow}\textsc{a}$ typing rule is another place the well-formedness condition on typing judgments is essential.)

Again following the pattern for positive propositions, the ${\downarrow}\textsc{l}$ rule corresponds to the typing rule for a construct for reading at type $\downarrow\! A$, namely $\textsf{read } a\; (\langle x\rangle \Rightarrow P)$.

$$\frac{\Gamma, A \vdash C}{\Gamma, \downarrow\! A \vdash C}\; {\downarrow}\textsc{l} \qquad\qquad \frac{\Gamma, x{:}A \vdash P :: (c : C)}{\Gamma, a{:}{\downarrow}\! A \vdash \textsf{read } a\; (\langle x\rangle \Rightarrow P) :: (c : C)}\; {\downarrow}\textsc{l}$$

Unlike the constructs for reading pairs and tagged values, $\textsf{read } a\; (\langle x\rangle \Rightarrow P)$ has a bound variable and no projection. At runtime, the address stored at address a is read; then the variable x is bound to that address, and execution continues according to process P. Using a variable, not a projection, in this rule is necessary because the pointer may refer to an arbitrary location, not just to one calculable from a.

Example. By judiciously inserting or removing $\downarrow$ shifts within a type, different layouts can be effected. As an example, we can revisit the types *bool* and *boollist*. Having only a $\downarrow$ shift in front of the recursive call to *boollist* yields a flat layout, with pointer indirection only between list elements:

$$bool = \oplus\{\texttt{tt}\colon \mathbf{1}, \texttt{ff}\colon \mathbf{1}\}$$
$$boollist = \oplus\{\texttt{nil}\colon \mathbf{1}, \texttt{cons}\colon bool \times \downarrow\! boollist\}$$

$\boxed{\texttt{cons} \mid \texttt{tt} \mid \;\;} \rightarrow \cdots$

At the other extreme, having a $\downarrow$ shift in front of each type constructor effects an indirection-heavy, SAX-like layout within SNAX. Each $\downarrow$ shift introduces a pointer into the layout.

$$bool = \oplus\{\texttt{tt}\colon \downarrow\!\mathbf{1}, \texttt{ff}\colon \downarrow\!\mathbf{1}\}$$
$$boollist = \oplus\{\texttt{nil}\colon \downarrow\!\mathbf{1}, \texttt{cons}\colon \downarrow(\downarrow\! bool \times \downarrow\! boollist)\}$$

Layouts with intermediate degrees of indirection can be achieved by using fewer $\downarrow$ shifts.

---

[6] In future work, we wish to extend the semi-axiomatic sequent calculus and its correspondence with SNAX to have several adjoint layers, to support both linear and persistent data, for example. In such a system, $\downarrow\! A$ would have logical force, as A and $\downarrow\! A$ would not be logically equivalent in general.

Functions, type $A_1 \to A_2$.

Unlike the data of positive types such as $A_1 \times A_2$, we will not model the internal layout of function closures. As previously mentioned, we therefore ignore the eligibility that appears in the semi-axiomatic sequent calculus's $\supset$A axiom. For this reason, SNAX directly inherits the process constructs and static typing rules for $A_1 \to A_2$ from SAX. Operationally, they behave as before. (Although we do not present the details in this paper, it is also possible to adapt negative conjunction from intuitionistic logic to the SNAX type theory as a lazy record type.)

$$\frac{\Gamma, A_1 \vdash A_2}{\Gamma \vdash A_1 \supset A_2} \; \supset\text{R} \qquad\qquad \frac{\Gamma, x{:}A_1 \vdash P :: (z : A_2)}{\Gamma \vdash \mathsf{write}\, a\, (\langle x, z\rangle \Rightarrow P) :: (a : A_1 \to A_2)} \; \to\text{R}$$

$$\frac{}{A_1 \supset A_2, \underline{A_1} \vdash \underline{A_2}} \; \supset\text{A} \qquad\qquad \frac{}{a{:}A_1 \to A_2, a_1{:}A_1 \vdash \mathsf{read}\, a\, \langle a_1, a_2\rangle :: (a_2 : A_2)} \; \to\text{A}$$

3.3 Adding recursion to SNAX

Recursion is added to SNAX in the same way as for SAX: We use recursive type and process definitions, $t = A$ and $\mathsf{proc}\, p\, (z{:}C)\, (x_1{:}A_1)\cdots(x_n{:}A_n) = P$, respectively. As in SAX, the SNAX type theory requires that these definitions are contractive. In addition, SNAX requires that all recursion in type definitions be *guarded* by a $\downarrow$ shift or a negative type constructor (only $\to$ in this paper), as in $boollist = \oplus\{\mathtt{nil}{:}\, \mathbf{1}, \mathtt{cons}{:}\, bool \times \downarrow boollist\}$, for example. The unguarded type $boollist = \oplus\{\mathtt{nil}{:}\, \mathbf{1}, \mathtt{cons}{:}\, bool \times boollist\}$ is forbidden because storing a value of type *boollist* according to this unguarded definition would require an unbounded amount of space.

3.4 Extended example: Mapping a function across a linked list of booleans

We can revisit the example of mapping a function across a list of booleans in SNAX. Here we choose to use the flattest of the layouts for lists of booleans, which corresponds to a type definition for *boollist* that uses only the $\downarrow$ shift necessary to guard the recursion. A common idiom is for both arguments and destinations of processes to be addresses (that is, pointers), a small departure from similar code in SAX.

$bool = \oplus\{\mathtt{tt}{:}\, \mathbf{1},\, \mathtt{ff}{:}\, \mathbf{1}\}$
$boollist = \oplus\{\mathtt{nil}{:}\, \mathbf{1},\, \mathtt{cons}{:}\, bool \times \downarrow boollist\}$

```
proc map (ysp : ↓boollist) (f : bool → bool) (xsp : ↓boollist) =
  read xsp (⟨xs⟩ ⇒                                    % read address xs from xsp
  read xs (                                           % read and branch on tag at xs
    nil⟨_⟩ ⇒ copy ysp xsp                             % copy input list pointer xsp to dest. ysp
  | cons⟨_⟩ ⇒ read (xs · cons) (⟨_, _⟩ ⇒              % read the pair at xs · cons
           ys ← (read f ⟨xs · cons · π₁, ys · cons · π₁⟩;   % alloc. ys; call f with dest. ys · cons · π₁
                 call map (ys · cons · π₂) f (xs · cons · π₂);  % call map with dest. ys · cons · π₂
                 write (ys · cons) ⟨_, _⟩;            % "write" pair to ys · cons
                 write ys cons⟨_⟩);                   % write tag cons to ys
         write ysp ⟨ys⟩)))                            % write pointer to ys at destination ysp
```

After reading the pointer *xsp* to access *xs*:*boollist*, the SNAX version of *map* generally follows the pattern of the SAX *map*, with a few essential deviations. First, here there is only a single point at which memory is allocated: the cut indicated by the $ys \leftarrow (\cdots);$ syntax. Second, the projections such as $xs \cdot \overline{cons} \cdot \pi_1$ are used to refer to memory cells within the allocated block as laid out by the type *boollist*. Third, because the projections are locally calculable, they can be elided from some parts of the syntax. Last, because the type now uses $ysp : \downarrow boollist$, a final $\mathsf{write}\, ysp\, \langle ys\rangle$ is needed.

3.5 Details of the SNAX type theory

Types and signatures are exactly as they were in SAX, so we do not repeat the details here.

As compared to SAX, SNAX processes have one additional form: $P;Q$ for concurrent composition of processes P and Q that does not allocate memory. Storables S and co-storables T have slightly different forms than in SAX, owing to SNAX's elision of eligible addresses from process syntax.

$$
\begin{array}{rl}
\textit{Processes} & P, Q ::= x \leftarrow P; Q \mid P; Q \mid \mathsf{copy}\ a\ b \mid \mathsf{write}\ a\ S \mid \mathsf{read}\ a\ T \\
\textit{Storables} & S ::= \langle \_, \_ \rangle \mid \langle \rangle \mid k\langle \_ \rangle \mid \langle a \rangle \mid (\langle x, z \rangle \Rightarrow P) \\
\textit{Co-storables} & T ::= (\langle \_, \_ \rangle \Rightarrow P) \mid (\langle \rangle \Rightarrow P) \mid (\ell\langle \_ \rangle \Rightarrow P_\ell)_{\ell \in L} \mid (\langle x \rangle \Rightarrow P) \mid \langle a_1, a_2 \rangle
\end{array}
$$

Once again, we use an operational semantics based on multiset rewriting with semantic objects of the forms $\mathsf{thread}(a, P)$, $\mathsf{cell}(a, \square)$, and $!\mathsf{cell}(a, S)$ that represent running processes, empty cells, and filled cells, respectively.

Configurations $\mathcal{C}$ and configuration contexts Φ are the same as in SAX; once again, configuration contexts are not subject to contraction and their addresses are presumed to be distinct, an invariant that will be preserved by the configuration typing rules. Also, we will continue to write a, b, and c in configurations and configuration contexts, but these runtime addresses may not contain static variables x.

The configuration typing rules are quite similar to those of SAX, but include one twist. Unlike SNAX process contexts Γ, configuration contexts Φ do not track eligibility. To mediate the two in rules THREAD and CELL, we therefore define a judgment $\Phi \vDash_c \Gamma$ that holds exactly when three conditions are met: *(i)* $a{:}A \in \Gamma$ only if $a{:}A \in \Phi$; *(ii)* $\underline{a{:}A} \in \Gamma$ only if $a{:}A \in \Phi$ and $a \succ c$; and *(iii)* $a{:}A \in \Phi$ and $a \succ c$ only if either $\underline{a{:}A} \in \Gamma$ or $a \succ b$ for some $\underline{b{:}B} \in \Gamma$. The process typing premises in rules THREAD and CELL then further guarantee that the choice of eligible antecedents is consistent with the process.

$$
\frac{}{\Phi \vDash (\cdot) :: \Phi}\ \text{EMP}
\qquad
\frac{\Phi \vDash \mathcal{C}_1 :: \Phi' \quad \Phi' \vDash \mathcal{C}_2 :: \Phi''}{\Phi \vDash \mathcal{C}_1\ \mathcal{C}_2 :: \Phi''}\ \text{JOIN}
$$

$$
\frac{(a \notin \mathrm{dom}\ \Phi) \quad \Phi \vDash_a \Gamma \quad \Gamma \vdash P :: (a : A)}{\Phi \vDash \mathsf{thread}(a, P)\ \mathsf{cell}(a, \square) :: (\Phi, a{:}A)}\ \text{THREAD}
\qquad
\frac{(a \notin \mathrm{dom}\ \Phi) \quad \Phi \vDash_a \Gamma \quad \Gamma \vdash \mathsf{write}\ a\ S :: (a : A)}{\Phi \vDash\ !\mathsf{cell}(a, S) :: (\Phi, a{:}A)}\ \text{CELL}
$$

The rewriting rules for futures, writes, reads, and calls are essentially the same as in SAX, but the definition of $S \triangleright T$ changes slightly. Because some addresses are locally calculable and elided from the syntax, substitution is no longer needed in some cases; however, substitution is still needed for the cases for pointers and functions.

$$
\begin{array}{ll}
\mathsf{thread}(c, (x \leftarrow P; Q)) \quad\quad (\alpha\ \text{fresh}) & \text{where } S \triangleright T \text{ is given by} \\
\quad \longmapsto \mathsf{thread}(\alpha, [\alpha/x]P)\ \mathsf{cell}(\alpha, \square)\ \mathsf{thread}(c, [\alpha/x]Q) & \quad\quad \langle \_, \_ \rangle \triangleright (\langle \_, \_ \rangle \Rightarrow P) = P \\
\mathsf{thread}(a, \mathsf{write}\ a\ S)\ \mathsf{cell}(a, \square) \longmapsto\ !\mathsf{cell}(a, S) & \quad\quad \langle \rangle \triangleright (\langle \rangle \Rightarrow P) = P \\
\mathsf{thread}(c, \mathsf{read}\ a\ T)\ !\mathsf{cell}(a, S) \longmapsto \mathsf{thread}(c, S \triangleright T) & \quad\quad k\langle \_ \rangle \triangleright (\ell\langle \_ \rangle \Rightarrow P_\ell)_{\ell \in L} = P_k \quad (k \in L) \\
\mathsf{thread}(c, \mathsf{call}\ p\ c\ a_1 \cdots a_n) & \quad\quad \langle a \rangle \triangleright (\langle x \rangle \Rightarrow P) = [a/x]P \\
\quad \longmapsto \mathsf{thread}(c, [c/z, a_1/x_1, \ldots, a_n/x_n]P) & \quad\quad \langle a_1, a_2 \rangle \triangleright (\langle x, z \rangle \Rightarrow P) = [a_1/x, a_2/z]P \\
\quad (\text{where } \mathsf{proc}\ p\ z\ x_1 \cdots x_n = P) &
\end{array}
$$

Snips are the essential difference between SAX and SNAX. The rewriting rule for a snip is broadly similar to that for futures, with the key difference that an address a is used instead of choosing a fresh α.

$$
\mathsf{thread}(c, (P; Q)) \longmapsto \mathsf{thread}(a, P)\ \mathsf{cell}(a, \square)\ \mathsf{thread}(c, Q) \text{ if } \mathsf{dest}(P) = \{a\}
$$

Because the address a written by P is not made locally explicit in the snip construct $P;Q$, the function $\mathsf{dest}(P)$ (short for "destination") traverses the process P to extract that address. This function returns a set of addresses, but for a well-typed process P, the set will always be a singleton. The definition of $\mathsf{dest}(P)$ can be found in appendix A.

Copying data must be handled differently in SNAX than in SAX. In SAX, we could simply copy a storable from one address to another; the implicit sharing would take care of a type's subformulas. In

SNAX, all sharing is made explicit through the $\downarrow$ shifts, and those may or may not appear in a given type's subformulas. However, at types $\downarrow A$ and $A_1 \to A_2$, simply copying the storable suffices.

$$\mathsf{thread}(a, \mathsf{copy}\ a\ b)\ !\mathsf{cell}(b, S) \longmapsto !\mathsf{cell}(a, S)$$

Before a process may be executed, we require that copys at other types are expanded, using reads and writes, so that only copys at types $\downarrow A$ and $A_1 \to A_2$ remain; this is reminiscent of η-expansion. The expansion of copys is shown in appendix A.

3.6 Type safety for SNAX

SNAX satisfies type safety, in the form of type preservation and progress results. Preservation is a bit subtle, but ultimately not difficult, to prove; it relies on various lemmas surrounding eligibility, as well as the definition of $\Phi \vDash_c \Gamma$.

Theorem 3.1 (Preservation) *If* $\Phi_0 \vDash \mathcal{C} :: \Phi$ *and* $\mathcal{C} \longmapsto \mathcal{C}'$, *then* $\Phi_0 \vDash \mathcal{C}' :: \Phi'$ *for some* $\Phi' \supseteq \Phi$.

Proof. By induction on the given derivation, using a few lemmas about eligibility; see appendix B. □

Theorem 3.2 (Progress) *If* $\vDash \mathcal{C} :: \Phi$, *then either* $\mathcal{C}$ *is final or* $\mathcal{C} \longmapsto \mathcal{C}'$ *for some* $\mathcal{C}'$.

Proof. By right-to-left induction on the structure of the given derivation; see appendix B. □

4   Compiling SAX processes to SNAX processes

When eligibility is viewed as a property of proofs, the eligibility of an antecedent may be silently forgotten at any time. On the other hand, when eligibility is a structural component of the logic, the eligibility of an antecedent cannot simply be dropped. However, we can mediate an eligible $\underline{A}$ and an ordinary A by using the ID and SNIP$^+$ rules:

$$\frac{\Gamma, \underline{A} \vdash C}{\Gamma, A \vdash C} \quad = \quad \frac{\overline{\Gamma, A \vdash A}\ ^{\mathrm{ID}} \quad \Gamma, \underline{A} \vdash C}{\Gamma, A \vdash C}\ \mathrm{SNIP}^+ .$$

This proof-theoretic equation suggests a type-theoretic procedure for compiling SAX processes into SNAX processes: We can examine a SAX process and determine which addresses would be eligible in its SNAX transliteration. Then, wherever there is a mismatch between an ordinary $b':A$ and an eligible $\underline{a':A}$, the above combination of ID and SNIP$^+$ is inserted as $\mathsf{copy}\ a'\ b'; -$.

One complication is that SAX addresses might possibly appear in several positions that correspond to distinct eligible addresses in a SNAX transliteration. There is therefore no single best translation to assign a given address, requiring the translation of processes to be parameterized by a translation of addresses. For example, $x{:}A \vdash \mathsf{write}\ a\ \langle x, x \rangle :: (a : A \times A)$ is a valid SAX process. Assuming that we choose to translate the SAX addresses a and x to the SNAX address a' and b', respectively, three translations are possible, depending on the identity of b':

- $\underline{b':A} \vdash \mathsf{copy}\ (a' \cdot \pi_2)\ b'; \mathsf{write}\ a'\ \langle \_, \_ \rangle :: (a' : A \times A)$ if $b' = a' \cdot \pi_1$;
- $\underline{b':A} \vdash \mathsf{copy}\ (a' \cdot \pi_1)\ b'; \mathsf{write}\ a'\ \langle \_, \_ \rangle :: (a' : A \times A)$ if $b' = a' \cdot \pi_2$; or
- $b':A \vdash \mathsf{copy}\ (a' \cdot \pi_1)\ b'; \mathsf{copy}\ (a' \cdot \pi_2)\ b'; \mathsf{write}\ a'\ \langle \_, \_ \rangle :: (a' : A \times A)$ if $b' \neq a' \cdot \pi_1$ and $b' \neq a' \cdot \pi_2$.

Along these lines, we define a translation $[\![P]\!]_\theta^{c'}$ of SAX processes P that yields a SNAX process that writes to c' and may read from $\theta(x)$ whenever P may read from x. This translation can be lifted to configurations by defining $[\![\mathcal{C}]\!]_\theta$. The definitions of these translations can be found in appendix C.

The translations are type-preserving and operationally correct.

Theorem 4.1 *If* $\theta : \Phi \to \Phi'$ *and* $\Phi \supseteq \Gamma$ *and* $\Gamma \vdash P :: (a : A)$ *and* $a' \notin \mathrm{dom}\ \Phi'$, *then* $\Phi' \vDash_{a'} \Gamma'$ *and* $\Gamma' \vdash [\![P]\!]_\theta^{a'} :: (a' : A)$ *for some* Γ'.

Proof. By induction on the structure of the given typing derivation. For details, refer to appendix C. □

Theorem 4.2 *For all final SAX configurations $\mathcal{M}_0$ and $\mathcal{M}$, if $\mathcal{M}_0$ thread(a, P) cell$(a, \square) \longmapsto^* \mathcal{M}$, then $[\![\mathcal{M}_0]\!]_\theta$ thread$(a', [\![P]\!]_\theta^{a'})$ cell$(a', \square) \longmapsto^* [\![\mathcal{M}]\!]_{\theta'}$ for some θ' such that $\theta' \supseteq (\theta, a'/a)$.*

Proof. By induction on the structure of the given process P. For details, refer to appendix C. □

5   Related work

Besides the aforementioned work on the semi-axiomatic sequent calculus and SAX [6], another item of related work is Smullyan's classical sequent calculus in which cuts must be analytic and all other inference rules are replaced by axioms [20]. Because all cuts are analytic, there is no direct cut elimination procedure and, consequently, the calculus does not seem to lend itself to computational interpretation.

From a computational standpoint, most closely related is perhaps the work on data layout using *ordered types* [16]. Ordered types were suitable to capture the original allocation and layout of data, but not the whole state of memory during computation since ordered logic has only a single ordered context thus cannot directly model many blocks of memory connected by pointers. The current design overcomes both of these limitations with a very different approach: our logic (and therefore the type theory) is not substructural at all.

Another point of comparison is Typed Assembly Language (TAL) [15]. We view TAL as a low-level type system that can reflect high level abstractions, but it does not seem to correspond to any particular proof system for intuitionistic logic. Furthermore, while TAL by necessity works with concrete data layouts, the compilation from the λ-calculus to TAL chooses a particular one among them rather than providing a choice to the programmer. Another point of difference is that in SNAX, functions receive *destinations* (that is, memory locations) for their results, while in TAL they receive *continuations* to be called with the result. TAL also resolves some issues that we leave to future work. Among them are parametric polymorphism and representation of closures.

6   Conclusion

We have shown how elevating notions of eligibility and snips that arise in the semi-axiomatic sequent calculus's cut elimination proof from refinement properties to first-class logical concepts yields a Curry–Howard explanation of (abstract) data layout in futures-based shared memory concurrency. Moreover, we have proved type preservation and progress for the resulting SNAX type theory. We have also sketched a compilation of SAX processes into SNAX processes.

In future work, we plan to extend SNAX to support parametric polymorphism, as well as adjoint layers for integrating a treatment of linear data with SNAX's existing treatment of persistent, write-once data. In designing both extensions, we will be able to lean on SNAX's strong logical foundations. Studying code optimization in the SNAX setting is another avenue for future work that we are pursuing.

References

[1] Andreoli, J.-M., *Logic programming with focusing proofs in linear logic*, Journal of Logic and Computation **2** (1992), pp. 197–347.
URL https://doi.org/10.1093/logcom/2.3.297

[2] Baker, H. C. and C. Hewitt, *The incremental garbage collection of processes*, SIGPLAN Notices **12** (1977), p. 55–59.
URL https://doi.org/10.1145/872734.806932

[3] Benton, N., *A mixed linear and non-linear logic: Proofs, terms and models*, in: L. Pacholski and J. Tiuryn, editors, *Selected Papers from the 8th International Workshop on Computer Science Logic (CSL'94)* (1994), pp. 121–135, an extended version appears as Technical Report UCAM-CL-TR-352, University of Cambridge.
URL https://doi.org/10.1007/BFb0022251

[4] Curry, H. B., *Functionality in combinatory logic*, Proceedings of the National Academy of Sciences, U.S.A. **20** (1934), pp. 584–590.
URL https://doi.org/10.1073/pnas.20.11.584

[5] DeYoung, H. and F. Pfenning, *Data layout from a type-theoretic perspective*, CoRR **abs/2212.06321** (2022), to appear in a special issue of Electronic Notes in Theoretical Informatics and Computer Science devoted to MFPS 2022.
URL https://arxiv.org/abs/2212.06321

[6] DeYoung, H., F. Pfenning and K. Pruiksma, *Semi-axiomatic sequent calculus*, in: Z. M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, LIPIcs **167**, Paris, France, 2020, pp. 29:1–29:22.
URL https://doi.org/10.4230/LIPIcs.FSCD.2020.29

[7] Dummett, M., "The Logical Basis of Metaphysics," Harvard University Press, Cambridge, Massachusetts, 1991, the William James Lectures, 1976.

[8] Gentzen, G., *Untersuchungen über das logische Schließen*, Mathematische Zeitschrift **39** (1935), pp. 176–210, 405–431, english translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.
URL https://doi.org/10.1007/BF01201353

[9] Girard, J.-Y., *On the unity of logic*, Annals of Pure and Applied Logic **59** (1993), pp. 201–217.
URL https://doi.org/10.1016/0168-0072(93)90093-S

[10] Halstead, R. H., *Multilisp: A language for concurrent symbolic computation*, ACM Transactions on Programming Languages and Systems **7** (1985), pp. 501–538.
URL https://doi.org/10.1145/4472.4478

[11] Herbelin, H., *A lambda-calculus structure isomorphic to Gentzen-style sequent calculus structure*, in: L. Pacholski and J. Tiuryn, editors, *8th International Workshop on Computer Science Logic* (1994), pp. 61–75.
URL https://doi.org/10.1007/BFb0022247

[12] Howard, W. A., *The formulae-as-types notion of construction* (1969), unpublished note. An annotated version appeared in: *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 479–490, Academic Press (1980).

[13] Levy, P. B., "Call-by-Push-Value," Ph.D. thesis, University of London (2001).
URL https://www.cs.bham.ac.uk/~pbl/papers/thesisqmwphd.pdf

[14] Martin-Löf, P., *On the meanings of the logical constants and the justifications of the logical laws* (1983), notes for three lectures given in Siena, Italy. Published in *Nordic Journal of Philosophical Logic*, 1(1):11-60, 1996.
URL http://www.hf.uio.no/ifikk/forskning/publikasjoner/tidsskrifter/njpl/vol1no1/meaning.pdf

[15] Morrisett, J. G., D. Walker, K. Crary and N. Glew, *From system F to typed assembly language*, ACM Transactions on Programming Languages and Systems **21** (1999), pp. 527–568.
URL https://doi.org/10.1145/319301.319345

[16] Petersen, L., R. Harper, K. Crary and F. Pfenning, *A type theory for memory allocation and data layout*, in: G. Morrisett, editor, *Conference Record of the 30th Annual Symposium on Principles of Programming Languages (POPL'03)* (2003), pp. 172–184, extended version available as Technical Report CMU-CS-02-171, December 2002.
URL https://doi.org/10.1145/604131.604147

[17] Pruiksma, K., W. Chargin, F. Pfenning and J. Reed, *Adjoint logic and its concurrent operational interpretation* (2018), unpublished manuscript.
URL http://www.cs.cmu.edu/~fp/papers/adjoint18.pdf

[18] Pruiksma, K. and F. Pfenning, *Back to futures*, Journal of Functional Programming **32** (2022), p. e6.
URL https://doi.org/10.1017/S0956796822000016

[19] Reed, J., *A judgmental deconstruction of modal logic* (2009), unpublished manuscript.
URL http://www.cs.cmu.edu/~jcreed/papers/jdml2.pdf

[20] Smullyan, R. M., *Analytic cut*, Journal of Symbolic Logic **33** (1968), pp. 560–564.
URL https://doi.org/10.2307/2271362

A   Auxiliary definitions for SNAX operational semantics

The definition of $\mathsf{dest}(P)$, which is used in the operational semantics of snips, is as follows.

$$
\begin{aligned}
\mathsf{dest}(x \leftarrow P; Q) &= \mathsf{dest}(Q) \\
\mathsf{dest}(P; Q) &= \mathsf{dest}(Q) \\
\mathsf{dest}(\mathsf{copy}\ a\ b) &= \{a\} \\
\mathsf{dest}(\mathsf{write}\ a\ S) &= \{a\} \\
\mathsf{dest}(\mathsf{read}\ a\ (\langle \_ , \_ \rangle \Rightarrow P)) &= \mathsf{dest}(P) \\
\mathsf{dest}(\mathsf{read}\ a\ (\langle \rangle \Rightarrow P)) &= \mathsf{dest}(P) \\
\mathsf{dest}(\mathsf{read}\ a\ (\ell\langle \_ \rangle \Rightarrow P_\ell)_{\ell \in L}) &= \textstyle\bigcup_{\ell \in L} \mathsf{dest}(P_\ell) \\
\mathsf{dest}(\mathsf{read}\ a\ (\langle x \rangle \Rightarrow P)) &= \mathsf{dest}(P) \\
\mathsf{dest}(\mathsf{read}\ a\ \langle a_1, a_2 \rangle) &= \{a_2\}
\end{aligned}
$$

Expansion of copys down to types $\downarrow A$ and $A_1 \to A_2$ is accomplished by the following function.

$$
\begin{aligned}
\eta(\mathsf{copy}\ a\ b : A_1 \times A_2) &= \mathsf{read}\ b\ (\langle \_ , \_ \rangle \Rightarrow \eta(\mathsf{copy}\ (a\pi_1)\ (b\pi_1) : A_1); \eta(\mathsf{copy}\ (a\pi_2)\ (b\pi_2) : A_2); \mathsf{write}\ a\ \langle \_ , \_ \rangle) \\
\eta(\mathsf{copy}\ a\ b : \mathbf{1}) &= \mathsf{read}\ b\ (\langle \rangle \Rightarrow \mathsf{write}\ a\ \langle \rangle) \\
\eta(\mathsf{copy}\ a\ b : \oplus\{\ell : A_\ell\}_{\ell \in L}) &= \mathsf{read}\ b\ (\ell\langle \_ \rangle \Rightarrow \eta(\mathsf{copy}\ (a{\cdot}\overline{\ell})\ (b{\cdot}\overline{\ell}) : A_\ell); \mathsf{write}\ a\ \ell\langle \_ \rangle)_{\ell \in L} \\
\eta(\mathsf{copy}\ a\ b : \downarrow A) &= \mathsf{copy}\ a\ b \\
\eta(\mathsf{copy}\ a\ b : A \to B) &= \mathsf{copy}\ a\ b
\end{aligned}
$$

B   SNAX Metatheorems

Lemma B.1 *If $a \succ b$ and $a \succ c$, then either $b \succcurlyeq c$ or $c \succcurlyeq b$.*

Proof. By proving by simultaneous induction on p_1 and p_2 that $b \cdot p_1 = c \cdot p_2$ implies $b \succcurlyeq c$ or $c \succcurlyeq b$. □

Lemma B.2 *If $\Gamma, \underline{a{:}A} \vdash P :: (c : C)$, then $a \succ c$.*

Proof. By induction on the structure of the given derivation. The two interesting cases are as follows.

Case:

$$
\frac{\Gamma_1, \underline{a{:}A} \vdash P :: (b : B) \quad \Gamma_2, \underline{b{:}B} \vdash Q :: (c : C)}{\Gamma_1, \Gamma_2, \underline{a{:}A} \vdash P; Q :: (c : C)} \ \ \mathrm{SNIP^+}
$$

Appealing to the inductive hypothesis on the first premise, we know that $a \succ b$. Similarly, appealing to the inductive hypothesis on the second premise, we know that $b \succ c$. So $a \succ c$ follows from transitivity.

Case:

$$
\frac{\Gamma_1, \underline{a{:}A} \vdash P :: (x : B) \quad \Gamma_2, x{:}B \vdash Q :: (c : C) \quad (x \text{ fresh})}{\Gamma_1, \Gamma_2, \underline{a{:}A} \vdash x \leftarrow P; Q :: (c : C)} \ \ \mathrm{CUT}
$$

By the inductive hypothesis on the first premise, we know that $a \succ x$. However, then having $\underline{a{:}A}$ in the rule's conclusion contradicts the freshness of x. (The case for the $\to$R rule is similar.) □

Lemma B.3 *If $\Gamma, \underline{a{:}A}, \underline{b{:}B} \vdash P :: (c : C)$, then $a \not\succcurlyeq b$ and $b \not\succcurlyeq a$.*

Proof. By induction on the structure of the given derivation. The two interesting cases are as follows.

Case:

$$\frac{\Gamma_1, \underline{a{:}A} \vdash P :: (c' : C') \quad \Gamma_2, \underline{b{:}B}, \underline{c'{:}C'} \vdash Q :: (c : C)}{\Gamma_1, \Gamma_2, \underline{a{:}A}, \underline{b{:}B} \vdash P; Q :: (c : C)} \text{ SNIP}^+$$

We must show that $a \neq b$ and $a \not\succ b$ and $b \not\succ a$.

- Suppose that $a = b$. We know from the first premise above and Lemma B.2 that $a \succ c'$. So $b \succ c'$ as well. By the inductive hypothesis on the second premise above, $b \not\succ c'$, yielding a contradiction. Therefore $a \neq b$.

- Suppose that $a \succ b$. Once again, we know from the first premise above and Lemma B.2 that $a \succ c'$. By Lemma B.1, either $b \succcurlyeq c'$ or $c' \succcurlyeq b$. Appealing to the inductive hypothesis on the second premise above, $b \not\succ c'$ and $c' \not\succ b$. This is a contradiction, so $a \not\succ b$.

- Suppose that $b \succ a$. Once again, we know from the first premise above and Lemma B.2 that $a \succ c'$. So $b \succ c'$ follows by transitivity of $\succ$. Appealing to the inductive hypothesis on the second premise above, $b \not\succ c'$ and $c' \not\succ b$. This is a contradiction, so $b \not\succ a$.

Case:

$$\frac{\Gamma_1, \underline{a{:}A} \vdash P :: (x : C') \quad \Gamma_2, \underline{b{:}B}, x{:}C' \vdash Q :: (c : C) \quad (x \text{ fresh})}{\Gamma_1, \Gamma_2, \underline{a{:}A}, \underline{b{:}B} \vdash x \leftarrow P; Q :: (c : C)} \text{ CUT}$$

By Lemma B.2 on the first premise, we know that $a \succ x$. However, then having $\underline{a{:}A}$ in the rule's conclusion contradicts the freshness of x. (The case for the $\rightarrow$R rule is similar.) □

Lemma B.4 *If $\Phi \vDash_c \Gamma, a{:}A$, then $\Phi \vDash_c \Gamma$.*

Proof. Assume $\Phi \vDash_c \Gamma, a{:}A$. To establish $\Phi \vDash_c \Gamma$, there are three parts.

- Assume $b{:}B \in \Gamma$. Then $b{:}B \in \Gamma, a{:}A$ as well. It follows from $\Phi \vDash_c \Gamma, a{:}A$ that $b{:}B \in \Phi$.

- Assume that $\underline{b{:}B} \in \Gamma$. Then $\underline{b{:}B} \in \Gamma, a{:}A$ as well. It follows from $\Phi \vDash_c \Gamma, a{:}A$ that $b{:}B \in \Phi$ and $b \succ c$.

- Assume that $b{:}B \in \Phi$ and $b \succ c$. It follows from $\Phi \vDash_c \Gamma, a{:}A$ that either $\underline{b{:}B} \in \Gamma, a{:}A$ or $b \succ b'$ for some $\underline{b'{:}B'} \in \Gamma, a{:}A$. Because $a{:}A$ is ordinary, either $\underline{b{:}B} \in \Gamma$ or $b \succ b'$ for some $\underline{b'{:}B'} \in \Gamma$.

□

Theorem B.5 (Preservation) *If $\Phi_0 \vDash \mathcal{C} :: \Phi$ and $\mathcal{C} \longmapsto \mathcal{C}'$, then $\Phi_0 \vDash \mathcal{C}' :: \Phi'$ for some $\Phi' \supseteq \Phi$.*

Proof. By induction on the structure of the given derivation, appealing to the preceding lemmas about eligibility. The most interesting case is as follows.

Case:

$$\frac{(c \notin \operatorname{dom}\Phi) \quad \Phi \vDash_c \Gamma_1, \Gamma_2 \quad \dfrac{\dfrac{\Gamma_1 \vdash P :: (a : A) \quad \Gamma_2, \underline{a{:}A} \vdash Q :: (c : C)}{\Gamma_1, \Gamma_2 \vdash P; Q :: (c : C)} \text{ SNIP}^+}{}}{\Phi \vDash \mathsf{thread}(c, P; Q)\, \mathsf{cell}(c, \square) :: (\Phi, c{:}C)} \text{ THREAD}$$

$$\longmapsto$$

$$\frac{\dfrac{(a \notin \operatorname{dom}\Phi) \quad \Phi \vDash_a \Gamma_1 \quad \Gamma_1 \vdash P :: (a : A)}{\Phi \vDash \mathsf{thread}(a, P)\, \mathsf{cell}(a, \square) :: (\Phi, a{:}A)} \quad \dfrac{(c \notin \operatorname{dom}(\Phi, a{:}A)) \quad \Phi, a{:}A \vDash_c \Gamma_2, \underline{a{:}A} \quad \Gamma_2, \underline{a{:}A} \vdash Q :: (c : C)}{\Phi, a{:}A \vDash \mathsf{thread}(c, Q)\, \mathsf{cell}(c, \square) :: (\Phi, a{:}A, c{:}C)}}{\Phi \vDash \mathsf{thread}(a, P)\, \mathsf{cell}(a, \square)\, \mathsf{thread}(c, Q)\, \mathsf{cell}(c, \square) :: (\Phi, a{:}A, c{:}C)} \text{ JOIN}$$

First, we must show that $a \notin \operatorname{dom}\Phi$.

- Suppose that $a \in \operatorname{dom}\Phi$. From the snip's second premise, we know that $a \succ c$ (Lemma B.2). Because $\Phi \vDash_c \Gamma_1, \Gamma_2$, either: $\underline{a{:}A} \in \Gamma_1$; $\underline{a{:}A} \in \Gamma_2$; or $a \succ b$ for some $\underline{b{:}B} \in \Gamma_1, \Gamma_2$.

· If $\underline{a{:}A} \in \Gamma_1$, then Lemma B.2 on the first premise yields $a \succ a$, which is impossible.

· If $\underline{a{:}A} \in \Gamma_2$, then Lemma B.3 on the second premise yields $a \not\succ a$, which is impossible.

· Otherwise, $a \succ b$ for some $\underline{b{:}B} \in \Gamma_1, \Gamma_2$. If $\underline{b{:}B} \in \Gamma_1$, then Lemma B.2 yields $b \succ a$, which contradicts $a \succ b$. If $\underline{b{:}B} \in \Gamma_2$, then Lemma B.3 yields $a \not\succ b$, which contradicts $a \succ b$.

Second, we must show that $\Phi \vDash_a \Gamma_1$.

- Assume that $b{:}B \in \Gamma_1$. From $\Phi \vDash_c \Gamma_1, \Gamma_2$, we therefore know that $b{:}B \in \Phi$, as required.

- Assume that $\underline{b{:}B} \in \Gamma_1$. From $\Phi \vDash_c \Gamma_1, \Gamma_2$, we therefore know that $b{:}B \in \Phi$ (and $b \succ c$). Because $\underline{b{:}B} \in \Gamma_1$, Lemma B.2 on the first premise yields $b \succ a$, as required.

- Assume that $b{:}B \in \Phi$ and $b \succ a$. From $\Phi \vDash_c \Gamma_1, \Gamma_2$, we therefore know that either $\underline{b{:}B} \in \Gamma_1, \Gamma_2$ or $b \succ b'$ for some $\underline{b'{:}B'} \in \Gamma_1, \Gamma_2$.
  · Suppose that $\underline{b{:}B} \in \Gamma_2$. By Lemma B.3 on the second premise, $b \not\succ a$, which contradicts $b \succ a$.
  · Suppose that $b \succ b'$ for some $\underline{b'{:}B'} \in \Gamma_2$. Because both $b \succ a$ and $b \succ b'$, Lemma B.1 yields either $a \not\succ b'$ or $b' \not\succ a$. However, by Lemma B.3 and the second premise, neither of these can be true.
  The only remaining possibility is that either $\underline{b{:}B} \in \Gamma_1$ or $b \succ b'$ for some $\underline{b'{:}B'} \in \Gamma_1$, as required.

Third, we must show that $c \notin \mathrm{dom}\,(\Phi, a{:}A)$.

- We are given that $c \notin \mathrm{dom}\,\Phi$. From Lemma B.2 and the snip's second premise, we know that $a \succ c$. This also implies that $c \neq a$, so we may indeed conclude that $c \notin \mathrm{dom}\,(\Phi, a{:}A)$.

Fourth, we must show that $\Phi, a{:}A \vDash_c \Gamma_2, \underline{a{:}A}$.

- Assume that $b{:}B \in \Gamma_2, \underline{a{:}A}$. More precisely, $b{:}B \in \Gamma_2$. Because $\Phi \vDash_c \Gamma_1, \Gamma_2$, it follows that $b{:}B \in \Phi$.

- Assume that $\underline{b{:}B} \in \Gamma_2, \underline{a{:}A}$.
  · If $\underline{b{:}B} \in \Gamma_2$, then it follows from $\Phi \vDash_c \Gamma_1, \Gamma_2$, it follows that $b{:}B \in \Phi$ and $b \succ c$, as required.
  · Otherwise, $b = a$ and $B = A$. Then $b{:}B \in \Phi, a{:}A$. Also, by Lemma B.2 on the first premise, $a \succ c$. So $b \succ c$, as required.

- Assume that $b{:}B \in \Phi, a{:}A$ and $b \succ c$. If $b = a$ and $B = A$, then $\underline{b{:}B} \in \Gamma_2, \underline{a{:}A}$. Otherwise, $b{:}B \in \Phi$. From $\Phi \vDash_c \Gamma_1, \Gamma_2$, we therefore know that either $\underline{b{:}B} \in \Gamma_1, \Gamma_2$ or $b \succ b'$ for some $\underline{b'{:}B'} \in \Gamma_1, \Gamma_2$.
  · Suppose that $\underline{b{:}B} \in \Gamma_1$. By Lemma B.2 on the first premise, $b \succ a$. And $\underline{a{:}A} \in \Gamma_2, \underline{a{:}A}$, as required.
  · Suppose that $b \succ b'$ for some $\underline{b'{:}B'} \in \Gamma_1$. By Lemma B.2 and the first premise, $b' \succ a$. By transitivity, $b \succ a$. And $\underline{a{:}A} \in \Gamma_2, \underline{a{:}A}$, as required.
  Therefore, in all cases, either $\underline{b{:}B} \in \Gamma_2, \underline{a{:}A}$ or $b \succ b'$ for some $\underline{b'{:}B'} \in \Gamma_2, \underline{a{:}A}$, as required.

Case:

$$\frac{(c \notin \mathrm{dom}\,\Phi_0) \quad \Phi_0 \vDash_c \Gamma_1, \Gamma_2 \quad \dfrac{\Gamma_1 \vdash P :: (x : A) \quad \Gamma_2, x{:}A \vdash Q :: (c : C) \quad (x \text{ fresh})}{\Gamma_1, \Gamma_2 \vdash x \leftarrow P; Q :: (c : C)} \text{ THREAD}}{\Phi_0 \vDash \mathsf{thread}(c, (x \leftarrow P; Q))\,\mathsf{cell}(c, \Box) :: (\Phi_0, c{:}C)} \text{ CUT}$$

$$\longmapsto$$

$$\frac{\overset{\mathcal{D}}{\Phi_0 \vDash \mathsf{thread}(\alpha, [\alpha/x]P)\,\mathsf{cell}(\alpha, \Box) :: (\Phi_0, \alpha{:}A)} \quad \overset{\mathcal{E}}{\Phi_0, \alpha{:}A \vDash \mathsf{thread}(c, [\alpha/x]Q)\,\mathsf{cell}(c, \Box) :: (\Phi_0, \alpha{:}A, c{:}C)}}{\Phi_0 \vDash \mathsf{thread}(\alpha, [\alpha/x]P)\,\mathsf{cell}(\alpha, \Box)\,\mathsf{thread}(c, [\alpha/x]Q)\,\mathsf{cell}(c, \Box) :: (\Phi_0, \alpha{:}A, c{:}C)}$$

where

$$\mathcal{D} = \frac{(\alpha \notin \mathrm{dom}\,\Phi_0) \quad \Phi_0 \vDash_\alpha \Gamma_1 \quad \dfrac{\Gamma_1 \vdash P :: (x : A)}{\Gamma_1 \vdash [\alpha/x]P :: (\alpha : A)}}{\Phi_0 \vDash \mathsf{thread}(\alpha, [\alpha/x]P)\,\mathsf{cell}(\alpha, \Box) :: (\Phi_0, \alpha{:}A)}$$

and

$$\mathcal{E} = \frac{(c \notin \mathrm{dom}\,(\Phi_0, \alpha{:}A)) \quad \dfrac{\dfrac{\Phi_0 \vDash_c \Gamma_2}{\Phi_0, \alpha{:}A \vDash_c \Gamma_2}}{\Phi_0, \alpha{:}A \vDash_c \Gamma_2, \alpha{:}A} \quad \dfrac{\Gamma_2, x{:}A \vdash Q :: (c : C)}{\Gamma_2, \alpha{:}A \vdash [\alpha/x]Q :: (c : C)}}{\Phi_0, \alpha{:}A \vDash \mathsf{thread}(c, [\alpha/x]Q)\,\mathsf{cell}(c, \square) :: (\Phi_0, \alpha{:}A, c{:}C)}$$

- $\alpha \notin \mathrm{dom}\,\Phi_0$ because α is chosen to be fresh.
- Because $c \notin \mathrm{dom}\,\Phi_0$ is given and α is fresh, $c \notin \mathrm{dom}\,(\Phi_0, \alpha{:}A)$ follows.
- Because $\Gamma_1 \vdash P :: (x : A)$ for a fresh x, the context Γ_1 must not contain any eligible addresses. Fortunately, because α is fresh, no address in $\mathrm{dom}\,\Phi_0$ will have the form $\alpha \cdot p$. Therefore $\Phi_0 \vDash_\alpha \Gamma_1$. Moreover, because Γ_1 contains no eligible addresses, a lemma gives $\Phi_0 \vDash_c \Gamma_2$ from $\Phi_0 \vDash_c \Gamma_1, \Gamma_2$.   □

Lemma B.6 *If* $\Gamma \vdash P :: (a : A)$, *then* $\mathsf{dest}(P) = \{a\}$.

Proof. By induction on the structure of the given derivation. The most interesting case is as follows.

Case:

$$\frac{\forall \ell \in L \colon \Gamma, a{\cdot}\overline{\ell}{:}A_\ell \vdash P_\ell :: (c : C)}{\Gamma, a{:}{\oplus}\{\ell \colon A_\ell\}_{\ell \in L} \vdash \mathsf{read}\ a\ (\ell\langle \_ \rangle \Rightarrow P_\ell)_{\ell \in L} :: (c : C)} \ \oplus\mathrm{L}$$

By the inductive hypothesis, $\mathsf{dest}(P_\ell) = \{c\}$ for all $\ell \in L$. Then $\bigcup_{\ell \in L} \mathsf{dest}(P_\ell) = \{c\}$, as required.   □

Theorem B.7 (Progress) *If* $\vDash \mathcal{C} :: \Phi$, *then either* $\mathcal{C}$ *is final or* $\mathcal{C} \longmapsto \mathcal{C}'$ *for some* $\mathcal{C}'$.

Proof. By right-to-left induction on the structure of the given derivation.

Case:

$$\frac{\vDash \mathcal{C} :: \Phi \quad \dfrac{(c \notin \mathrm{dom}\,\Phi) \quad \Phi \vDash_c \Gamma_1, \Gamma_2 \quad \dfrac{\Gamma_1 \vdash P :: (a : A) \quad \Gamma_2, \underline{a{:}A} \vdash Q :: (c : C)}{\Gamma_1, \Gamma_2 \vdash P; Q :: (c : C)}\ \mathrm{SNIP}^+}{\Phi \vDash \mathsf{thread}(c, (P; Q))\,\mathsf{cell}(c, \square) :: (\Phi, c{:}C)}}{\vDash \mathcal{C}\ \mathsf{thread}(c, (P; Q))\,\mathsf{cell}(c, \square) :: (\Phi, c{:}C)}\ \mathrm{JOIN}$$

By Lemma B.6, we have $\mathsf{dest}(P) = \{a\}$. Therefore,

$$\mathcal{C}\ \mathsf{thread}(c, (P; Q))\,\mathsf{cell}(c, \square) \longmapsto \mathcal{C}\ \mathsf{thread}(a, P)\,\mathsf{cell}(a, \square)\,\mathsf{thread}(c, Q)\,\mathsf{cell}(c, \square),$$

as required.

Case:

$$\frac{\vDash \mathcal{C} :: \Phi \quad \dfrac{(c \notin \mathrm{dom}\,\Phi) \quad \Phi \vDash_c \Gamma, a{:}{\oplus}\{\ell \colon A_\ell\}_{\ell \in L} \quad \dfrac{\forall \ell \in L \colon \Gamma, a{\cdot}\overline{\ell}{:}A_\ell \vdash P_\ell :: (c : C)}{\Gamma, a{:}{\oplus}\{\ell \colon A_\ell\}_{\ell \in L} \vdash \mathsf{read}\ a\ (\ell\langle \_ \rangle \Rightarrow P_\ell)_{\ell \in L} :: (c : C)}\ \oplus\mathrm{L}}{\Phi \vDash \mathsf{thread}(c, \mathsf{read}\ a\ (\ell\langle \_ \rangle \Rightarrow P_\ell)_{\ell \in L})\,\mathsf{cell}(c, \square) :: (\Phi, c{:}C)}}{\vDash \mathcal{C}\ \mathsf{thread}(c, \mathsf{read}\ a\ (\ell\langle \_ \rangle \Rightarrow P_\ell)_{\ell \in L})\,\mathsf{cell}(c, \square) :: (\Phi, c{:}C)}\ \mathrm{JOIN}$$

By the inductive hypothesis, either $\mathcal{C}$ is final or $\mathcal{C} \longmapsto \mathcal{C}'$ for some $\mathcal{C}'$.

- Suppose that $\mathcal{C} \longmapsto \mathcal{C}'$ for some $\mathcal{C}'$. Then $\mathcal{C}$ thread$(c, \text{read } a\ (\ell\langle\_\rangle \Rightarrow P_\ell)_{\ell\in L})$ cell$(c, \Box) \longmapsto \mathcal{C}'$ thread$(c, \text{read } a\ (\ell\langle\_\rangle \Rightarrow P_\ell)_{\ell\in L})$ cell$(c, \Box)$.

- Suppose that $\mathcal{C}$ is final. Because $\Phi \vDash_c \Gamma, a{:}\oplus\{\ell\colon A_\ell\}_{\ell\in L}$, we also have $a{:}\oplus\{\ell\colon A_\ell\}_{\ell\in L} \in \Phi$. Then, by inversion on the derivation of $\vDash \mathcal{C} :: \Phi$, it follows that $a{\cdot}\overline{k}{:}A_k \in \Phi$ and $\mathcal{C} = \mathcal{C}_0$!cell$(a{\cdot}\overline{k}, k\langle\_\rangle)$ for some $\mathcal{C}_0$ and $k \in L$. Therefore $\mathcal{C}$ thread$(c, \text{read } a\ (\ell\langle\_\rangle \Rightarrow P_\ell)_{\ell\in L})$ cell$(c, \Box) \longmapsto \mathcal{C}$ thread(c, P_k) cell$(c, \Box)$, as required.

$\Box$

C Compiling SAX to SNAX

First, we need to define $[x.P]_{a'}$, the set of eligible SNAX addresses to which SAX address x could potentially be translated, assuming that the destination to which P writes is translated to a'.

$$
\begin{aligned}
[x.(y{:}B \leftarrow P; Q)]_{a'} &= [x.Q]_{a'} \\
[x.\text{write } a\ \langle a_1, a_2\rangle]_{a'} &= \{a'\pi_1, a'\pi_2\} && \text{if } a_1 = x \text{ and } a_2 = x \\
[x.\text{write } a\ \langle a_1, a_2\rangle]_{a'} &= \{a'\pi_i\} && \text{if } a_i = x \text{ and } a_{3-i} \neq x \text{ (for } i \in \{1,2\}) \\
[x.\text{write } a\ k\langle a_k\rangle]_{a'} &= \{a'{\cdot}\overline{k}\} && \text{if } a_k = x \\
[x.\text{read } b\ (\langle y_1, y_2\rangle \Rightarrow P)]_{a'} &= [x.P]_{a'} \\
[x.\text{read } b\ (\langle\rangle \Rightarrow P)]_{a'} &= [x.P]_{a'} \\
[x.\text{read } b\ (\ell\langle y_\ell\rangle \Rightarrow P_\ell)_{\ell\in L}]_{a'} &= \textstyle\bigcup_{\ell\in L} [x.P_\ell]_{a'} \\
[x.\text{read } b\ (\langle y\rangle \Rightarrow P)]_{c'} &= [x.P]_{a'} \\
[x.P]_{a'} &= \emptyset && \text{otherwise}
\end{aligned}
$$

This function satisfies several properties.

Lemma C.1 *If $a' \in [x.P]_{c'}$, then $a' \succ c'$.*

Proof. By induction on the structure of the given process. $\Box$

Next, we need to define the nondeterministic function $[\![P]\!]_\theta^{c'}$. This translation takes a translated destination c' and a translation θ for all of the addresses that may be read by P. The translation relies on $[x.P]_{c'}$, as well as a function $\langle\!\langle a' \leftarrow b'\rangle\!\rangle P'$ that prepends onto P' a snipped copy from b' to a' whenever $a' \neq b'$.

$$
\begin{aligned}
[\![x \leftarrow P_1; P_2]\!]_\theta^{c'} &= \begin{cases} x' \leftarrow [\![P_1]\!]_\theta^{x'}; [\![P_2]\!]_{\theta'}^{c'} & \text{if } [x.P_2]_{c'} = \emptyset, \text{ where } \theta' = \theta, x'/x \\ [\![P_1]\!]_\theta^{a'}; [\![P_2]\!]_{\theta'}^{c'} & \text{if } a' \in [x.P_2]_{c'}, \text{ where } \theta' = \theta, a'/x \end{cases} \\
[\![\text{copy } a\ b]\!]_\theta^{a'} &= \text{copy } a'\ \theta(b) \\
[\![\text{write } a\ \langle a_1, a_2\rangle]\!]_\theta^{a'} &= \langle\!\langle a'\pi_1 \leftarrow \theta(a_1)\rangle\!\rangle (\langle\!\langle a'\pi_2 \leftarrow \theta(a_2)\rangle\!\rangle (\text{write } a'\ \langle\_,\_\rangle)) \\
[\![\text{write } a\ \langle\rangle]\!]_\theta^{a'} &= \text{write } a'\ \langle\rangle \\
[\![\text{write } a\ k\langle a_k\rangle]\!]_\theta^{a'} &= \langle\!\langle a'{\cdot}\overline{k} \leftarrow \theta(a_k)\rangle\!\rangle (\text{write } a'\ k\langle\_\rangle) \\
[\![\text{write } a\ \langle b\rangle]\!]_\theta^{a'} &= \text{write } a'\ \langle\theta(b)\rangle \\
[\![\text{read } a\ (\langle x_1, x_2\rangle \Rightarrow P)]\!]_\theta^{c'} &= \text{read } \theta(a)\ (\langle\_,\_\rangle \Rightarrow [\![P]\!]_{\theta'}^{c'}) \text{ where } \theta' = \theta, \theta(a)\pi_1/x_1, \theta(a)\pi_2/x_2 \\
[\![\text{read } a\ (\langle\rangle \Rightarrow P)]\!]_\theta^{c'} &= \text{read } \theta(a)\ (\langle\rangle \Rightarrow [\![P]\!]_\theta^{c'}) \\
[\![\text{read } a\ (\ell\langle x_\ell\rangle \Rightarrow P_\ell)_{\ell\in L}]\!]_\theta^{c'} &= \text{read } \theta(a)\ (\ell\langle\_\rangle \Rightarrow [\![P_\ell]\!]_{\theta'_\ell}^{c'})_{\ell\in L} \text{ where } \theta'_\ell = \theta, \theta(a){\cdot}\overline{\ell}/x_\ell \\
[\![\text{read } a\ (\langle x\rangle \Rightarrow P)]\!]_\theta^{c'} &= \text{read } \theta(a)\ (\langle x'\rangle \Rightarrow [\![P]\!]_{\theta'}^{c'}) \text{ where } \theta' = \theta, x'/x \\
[\![\text{write } a\ (\langle x, z\rangle \Rightarrow P)]\!]_\theta^{a'} &= \text{write } a'\ (\langle x', z'\rangle \Rightarrow [\![P]\!]_{\theta'}^{z'}) \text{ where } \theta' = \theta, x'/x \\
[\![\text{read } a\ \langle a_1, a_2\rangle]\!]_\theta^{a'_2} &= \text{read } \theta(a)\ \langle\theta(a_1), a'_2\rangle
\end{aligned}
$$

where

$$\langle\!\langle a' \leftarrow b' \rangle\!\rangle P' = \begin{cases} P' & \text{if } a' = b' \\ (\text{copy } a'\ b'); P' & \text{if } a' \neq b' \end{cases}$$

Here we will use the notation $\Phi' \vDash \theta :: \Phi$ to stand for $\theta : \Phi \to \Phi'$.

Theorem C.2

- If $\Phi' \vDash \theta :: \Phi \supseteq \Gamma$ and $\Gamma \vdash P :: (a : A)$ and $a' \notin \text{dom}\,\Phi'$, then $\Phi' \vDash_{a'} \Gamma'$ and $\Gamma' \vdash \llbracket P \rrbracket_\theta^{a'} :: (a' : A)$ for some Γ'.

Proof. By induction on the structure of the given process P. Here are two representative cases.

Case:

$$\Phi' \vDash \theta :: \Phi \supseteq \Gamma_1, \Gamma_2 \quad \text{and} \quad \frac{\Gamma_1 \vdash P :: (x : A) \quad \Gamma_2, x{:}A \vdash Q :: (c : C)}{\Gamma_1, \Gamma_2 \vdash x \leftarrow P; Q :: (c : C)}\ \text{CUT}$$

- Suppose that $[x.Q]_{c'} = \emptyset$. Let x' be fresh. Because $\Phi' \vDash \theta :: \Phi \supseteq \Gamma_1$, we may appeal to the inductive hypothesis on the first premise to deduce $\Phi' \vDash_{x'} \Gamma'_1$ and $\Gamma'_1 \vdash \llbracket P \rrbracket_\theta^{x'} :: (x' : A)$ for some Γ'_1. Also, $\Phi', x'{:}A \vDash (\theta, x'/x) :: (\Phi, x{:}A) \supseteq (\Gamma_2, x{:}A)$. Appealing to the inductive hypothesis on the second premise, we have $\Phi', x'{:}A \vDash_{c'} \Gamma''_2$ and $\Gamma''_2 \vdash \llbracket Q \rrbracket_{(\theta, x'/x)}^{c'} :: (c' : C)$ for some Γ''_2. Because $[x.Q]_{c'} = \emptyset$, we have $\Gamma''_2 = \Gamma'_2, x'{:}A$ for some Γ'_2. Applying the CUT rule, $\Gamma'_1, \Gamma'_2 \vdash x' \leftarrow \llbracket P \rrbracket_\theta^{x'}; \llbracket Q \rrbracket_{(\theta, x'/x)}^{c'} :: (c' : C)$.

- Otherwise, suppose that $a' \in [x.Q]_{c'}$. Because $\Phi' \vDash \theta :: \Phi \supseteq \Gamma_1$, we may appeal to the inductive hypothesis on the first premise to deduce $\Phi' \vDash_{a'} \Gamma'_1$ and $\Gamma'_1 \vdash \llbracket P \rrbracket_\theta^{a'} :: (a' : A)$ for some Γ'_1. Also, $\Phi', a'{:}A \vDash (\theta, a'/x) :: (\Phi, x{:}A) \supseteq (\Gamma_2, x{:}A)$. Appealing to the inductive hypothesis on the second premise, we have $\Phi', a'{:}A \vDash_{c'} \Gamma''_2$ and $\Gamma''_2 \vdash \llbracket Q \rrbracket_{(\theta, a'/x)}^{c'} :: (c' : C)$ for some Γ''_2. Because $a' \in [x.Q]_{c'}$, we have $a' \succ c'$ and $a' \in \text{dom}\,\Gamma''_2$. By definition of $\Phi', a'{:}A \vDash_{c'} \Gamma''_2$, it therefore follows that $\Gamma''_2 = \Gamma'_2, \underline{a'{:}A}$. Applying the SNIP$^+$ rule, we have $\Gamma'_1, \Gamma'_2 \vdash \llbracket P \rrbracket_\theta^{a'}; \llbracket Q \rrbracket_{(\theta, a'/x)}^{c'} :: (c' : C)$.

Case:

$$\Phi' \vDash \theta :: \Phi \supseteq (a_k{:}A_k) \quad \text{and} \quad \frac{(k \in L)}{a_k{:}A_k \vdash \text{write } a\ k\langle a_k \rangle :: (a : \oplus\{\ell : A_\ell\}_{\ell \in L})}\ \oplus\text{A}_k$$

We must show that $\Phi' \vDash_{a'} \Gamma'$ and $\Gamma' \vdash \langle\!\langle a'{\cdot}\overline{k} \leftarrow \theta(a_k) \rangle\!\rangle (\text{write } a'\ k\langle\_\rangle) :: (a' : \oplus\{\ell : A_\ell\}_{\ell \in L})$ for some Γ'. There are two possibilities:

- If $\theta(a_k) = a'{\cdot}\overline{k}$, then $\underline{a'{\cdot}\overline{k}{:}A_k} \vdash \langle\!\langle a'{\cdot}\overline{k} \leftarrow \theta(a_k) \rangle\!\rangle (\text{write } a'\ k\langle\_\rangle) :: (a' : \oplus\{\ell : A_\ell\}_{\ell \in L})$ follows from the SNAX $\oplus\text{A}_k$ axiom. So it suffices to show that $\Phi' \vDash_{a'} \underline{a'{\cdot}\overline{k}{:}A_k}$. Because $\Phi' \vDash \theta :: \Phi \supseteq (a_k{:}A_k)$ and $\theta(a_k) = a'{\cdot}\overline{k}$, it follows that $a'{\cdot}\overline{k}{:}A_k \in \Phi'$.

- Otherwise, if $\theta(a_k) \neq a'{\cdot}\overline{k}$, then $\theta(a_k){:}A_k \vdash \langle\!\langle a'{\cdot}\overline{k} \leftarrow \theta(a_k) \rangle\!\rangle (\text{write } a'\ k\langle\_\rangle) :: (a' : \oplus\{\ell : A_\ell\}_{\ell \in L})$ follows from the SNAX $\oplus\text{A}_k$ axiom and ID rule. So it suffices to show that $\Phi' \vDash_{a'} \theta(a_k){:}A_k$. Because $\Phi' \vDash \theta :: \Phi \supseteq (a_k{:}A_k)$, we have $\theta(a_k){:}A_k \in \Phi'$; we need only check that there is no $b' \in \text{dom}\,\Phi'$ such that $b' \succ a'$.

$\square$

We can also lift the preceding translation to *final* configurations $\mathcal{M}$ as $\llbracket \mathcal{M} \rrbracket_\theta$. This relies on $\langle\!\langle \mathcal{M} \rangle\!\rangle_\theta^{a \to a'}$, which are the additional cells that may be needed to bridge the gap between $\theta(a)$ and a' that can arise

24

because of snipped copies that are inserted.

$$\llbracket \cdot \rrbracket_\theta = (\cdot)$$

$$\llbracket \mathcal{M} \,!\text{cell}(a, S) \rrbracket_\theta = \begin{cases} \llbracket \mathcal{M} \rrbracket_\theta \,!\text{cell}(\theta(a), \langle\rangle) & \text{if } S = \langle\rangle \\ \llbracket \mathcal{M} \rrbracket_\theta \,!\text{cell}(\theta(a), \langle\theta(b)\rangle) & \text{if } S = \langle b \rangle \\ \llbracket \mathcal{M} \rrbracket_\theta \,!\text{cell}(\theta(a), (\langle x', z'\rangle \Rightarrow \llbracket P \rrbracket^{z'}_{(\theta, x'/x)})) & \text{if } S = (\langle x, z\rangle \Rightarrow P) \\ \llbracket \mathcal{M} \rrbracket_\theta \,\langle\!\langle \mathcal{M} \rangle\!\rangle^{a_1 \mapsto a'\pi_1}_\theta \,\langle\!\langle \mathcal{M} \rangle\!\rangle^{a_2 \mapsto a'\pi_2}_\theta \,!\text{cell}(\theta(a), \langle\_,\_\rangle) & \text{if } S = \langle a_1, a_2 \rangle \\ \llbracket \mathcal{M} \rrbracket_\theta \,\langle\!\langle \mathcal{M} \rangle\!\rangle^{a_k \mapsto \theta(a)\cdot\overline{k}}_\theta \,!\text{cell}(\theta(a), k\langle\_\rangle) & \text{if } S = k\langle a_k \rangle \end{cases}$$

where

$$\langle\!\langle \mathcal{M} \rangle\!\rangle^{a \mapsto a'}_\theta = \begin{cases} (\cdot) & \text{if } \theta(a) = a' \\ !\text{cell}(a', \langle\rangle) & \text{if } \mathcal{M}(a) = \langle\rangle \text{ and } \theta(a) \neq a' \\ !\text{cell}(a', \langle\theta(b)\rangle) & \text{if } \mathcal{M}(a) = \langle b\rangle \text{ and } \theta(a) \neq a' \\ !\text{cell}(a', \llbracket P \rrbracket^{z'}_{(\theta, x'/x)}) & \text{if } \mathcal{M}(a) = (\langle x, z\rangle \Rightarrow P) \text{ and } \theta(a) \neq a' \\ \langle\!\langle \mathcal{M} \rangle\!\rangle^{a_1 \mapsto a'\pi_1}_\theta \,\langle\!\langle \mathcal{M} \rangle\!\rangle^{a_2 \mapsto a'\pi_2}_\theta \,!\text{cell}(a', \langle\_,\_\rangle) & \text{if } \mathcal{M}(a) = \langle a_1, a_2 \rangle \text{ and } \theta(a) \neq a' \\ \langle\!\langle \mathcal{M} \rangle\!\rangle^{a_k \mapsto a'\cdot\overline{k}}_\theta \,!\text{cell}(a', k\langle\_\rangle) & \text{if } \mathcal{M}(a) = k\langle a_k \rangle \text{ and } \theta(a) \neq a' \end{cases}$$

and $\mathcal{M}(a)$ is the SAX storable S found in the cell at address a within the final configuration $\mathcal{M}$.

Theorem C.3

- *For all final SAX configurations $\mathcal{M}_0$ and $\mathcal{M}$, if $\mathcal{M}_0 \,\text{thread}(a, P) \,\text{cell}(a, \square) \longmapsto^* \mathcal{M}$, then $\llbracket \mathcal{M}_0 \rrbracket_\theta \,\text{thread}(a', \llbracket P \rrbracket^{a'}_\theta) \,\text{cell}(a', \square) \longmapsto^* \llbracket \mathcal{M} \rrbracket_{\theta'}$ for some θ' such that $\theta' \supseteq (\theta, a'/a)$.*
- *For all final SAX configurations $\mathcal{M}_0$ and $\mathcal{M}$, $\llbracket \mathcal{M} \rrbracket_\theta \,\text{thread}(c', \langle\!\langle a' \leftarrow \theta(b)\rangle\!\rangle P') \,\text{cell}(c', \square) \longmapsto^* \llbracket \mathcal{M} \rrbracket_\theta \,\langle\!\langle \mathcal{M} \rangle\!\rangle^{b \mapsto a'}_\theta \,\text{thread}(c', P') \,\text{cell}(c', \square).*$

Proof. For the first part, by induction on the structure of the given process P; for the second part, by induction on the type of b. Here are several interesting cases for the first part.

Case:
Suppose that

$$\mathcal{M}_0 \,\text{thread}(a, \text{write } a\ k\langle a_k\rangle) \,\text{cell}(a, \square) \longmapsto^* \mathcal{M}.$$

In this case, inversion shows that $\mathcal{M} = \mathcal{M}_0 \,!\text{cell}(a, k\langle a_k\rangle)$. We must therefore show that:

$$\llbracket \mathcal{M}_0 \rrbracket_\theta \,\text{thread}(a', \langle\!\langle a'\cdot\overline{k} \leftarrow \theta(a_k)\rangle\!\rangle(\text{write } a'\ k\langle\_\rangle)) \,\text{cell}(a', \square) \longmapsto^* \llbracket \mathcal{M}_0 \rrbracket_{\theta'} \,\langle\!\langle \mathcal{M}_0 \rangle\!\rangle^{a_k \mapsto a'\cdot\overline{k}}_{\theta'} \,!\text{cell}(a', k\langle\_\rangle),$$

for some θ' such that $\theta' \supseteq (\theta, a'/a)$. Notice that

$$\llbracket \mathcal{M}_0 \rrbracket_\theta \,\text{thread}(a', \langle\!\langle a'\cdot\overline{k} \leftarrow \theta(a_k)\rangle\!\rangle(\text{write } a'\ k\langle\_\rangle)) \,\text{cell}(a', \square)$$
$$\longmapsto^* \llbracket \mathcal{M}_0 \rrbracket_\theta \,\langle\!\langle \mathcal{M}_0 \rangle\!\rangle^{a_k \mapsto a'\cdot\overline{k}}_\theta \,\text{thread}(a', \text{write } a'\ k\langle\_\rangle) \,\text{cell}(a', \square)$$
$$\longmapsto \llbracket \mathcal{M}_0 \rrbracket_\theta \,\langle\!\langle \mathcal{M}_0 \rangle\!\rangle^{a_k \mapsto a'\cdot\overline{k}}_\theta \,!\text{cell}(a', k\langle\_\rangle),$$

as required.

Case:
Suppose that

$$\mathcal{M}_0 \,\text{thread}(c, (x \leftarrow P; Q)) \,\text{cell}(c, \square) \longmapsto^* \mathcal{M}.$$

In this case, inversion shows that $\mathcal{M}_0 \,\text{thread}(\alpha, [\alpha/x]P) \,\text{cell}(\alpha, \square) \longmapsto^* \mathcal{M}_1$ and $\mathcal{M}_1 \,\text{thread}(c, [\alpha/x]Q) \,\text{cell}(c, \square) \longmapsto^* \mathcal{M}$.

25

- Suppose that $[x.Q]_{c'} = \emptyset$. By the inductive hypothesis, $[\![\mathcal{M}_0]\!]_\theta \; \mathsf{thread}(\alpha', [\![P]\!]_\theta^{\alpha'}) \; \mathsf{cell}(\alpha', \square) \longmapsto^*$ $[\![\mathcal{M}_1]\!]_{\theta'_1}$ for some θ'_1 such that $\theta'_1 \supseteq (\theta, \alpha'/\alpha)$. Moreover, by the inductive hypothesis, $[\![\mathcal{M}_1]\!]_{\theta'_1} \; \mathsf{thread}(c', [\![Q]\!]_{\theta'_1}^{c'}) \; \mathsf{cell}(c', \square) \longmapsto^* [\![\mathcal{M}]\!]_{\theta'}$ for some θ' such that $\theta' \supseteq (\theta'_1, c'/c)$. It follows that

$$
\begin{aligned}
&[\![\mathcal{M}_0]\!]_\theta \; \mathsf{thread}(c', (x' \leftarrow [\![P]\!]_\theta^{x'}; [\![Q]\!]_{(\theta,x'/x)}^{c'})) \; \mathsf{cell}(c', \square) \\
&\longmapsto [\![\mathcal{M}_0]\!]_\theta \; \mathsf{thread}(\alpha', [\![P]\!]_\theta^{\alpha'}) \; \mathsf{cell}(\alpha', \square) \; \mathsf{thread}(c', [\![Q]\!]_{(\theta,\alpha'/x)}^{c'}) \; \mathsf{cell}(c', \square) \\
&\longmapsto^* [\![\mathcal{M}_1]\!]_{\theta'_1} \; \mathsf{thread}(c', [\![Q]\!]_{(\theta,\alpha'/x)}^{c'}) \; \mathsf{cell}(c', \square) \\
&\longmapsto^* [\![\mathcal{M}]\!]_{\theta'}
\end{aligned}
$$

- Suppose that $a' \in [x.Q]_{c'}$. By the inductive hypothesis, $[\![\mathcal{M}_0]\!]_\theta \; \mathsf{thread}(a', [\![P]\!]_\theta^{a'}) \; \mathsf{cell}(a', \square) \longmapsto^*$ $[\![\mathcal{M}_1]\!]_{\theta'_1}$ for some θ'_1 such that $\theta'_1 \supseteq (\theta, a'/x)$. Moreover, by the inductive hypothesis, $[\![\mathcal{M}_1]\!]_{\theta'_1} \; \mathsf{thread}(c', [\![Q]\!]_{\theta'_1}^{c'}) \; \mathsf{cell}(c', \square) \longmapsto^* [\![\mathcal{M}]\!]_{\theta'}$ for some θ' such that $\theta' \supseteq (\theta'_1, c'/c)$. It follows that

$$
\begin{aligned}
&[\![\mathcal{M}_0]\!]_\theta \; \mathsf{thread}(c', ([\![P]\!]_\theta^{a'}; [\![Q]\!]_{(\theta,a'/x)}^{c'})) \; \mathsf{cell}(c', \square) \\
&\longmapsto [\![\mathcal{M}_0]\!]_\theta \; \mathsf{thread}(a', [\![P]\!]_\theta^{a'}) \; \mathsf{cell}(a', \square) \; \mathsf{thread}(c', [\![Q]\!]_{(\theta,a'/x)}^{c'}) \; \mathsf{cell}(c', \square) \\
&\longmapsto^* [\![\mathcal{M}_1]\!]_{\theta'_1} \; \mathsf{thread}(c', [\![Q]\!]_{(\theta,a'/x)}^{c'}) \; \mathsf{cell}(c', \square) \\
&\longmapsto^* [\![\mathcal{M}]\!]_{\theta'}
\end{aligned}
$$

$\square$