Memory Reuse in Linear Functional Computation

Daniel Ng, advised by Prof. Frank Pfenning

Abstract

The semi-axiomatic sequent calculus, or SAX, offers an alternative way to represent proofs in the sequent calculus. SAX also corresponds to a process calculus, where processes interact by writing and reading from memory cells. Improvements were then made to the memory layout of data structures to create the SNAX language, which acts similarly to SAX but uses fewer pointer dereferences. We have now added a system of memory reuse to the existing linear system provided by SNAX, further minimizing the time spent performing memory allocations. We have also updated both the typing rules and the dynamic rules for SNAX to accommodate reuse. Unlike the original SNAX, which is concurrent, SNAX with reuse runs under a sequential semantics. With memory reuse, progress and preservation still hold, meaning that well-typed programs in SNAX will never cause a memory error. Our changes allow SNAX to more closely match other functional languages that are capable of updating their single-threaded data structures in-place.

1 Introduction

1.1 Background

In 2020, Henry DeYoung, Frank Pfenning, and Klaas Pruiksma developed the semi-axiomatic sequent calculus (SAX). SAX is an alternate formulation of the sequent calculus, which is a logical system where each part of a proof is a truthful implication [2]. When interpreted in a proofs-as-programs sense, SAX can be interpreted as a process calculus – a programming language where the state of a program at any given time is a configuration of processes and memory cells. Processes can communicate by writing and reading to the memory cells, allowing for messages to be passed between processes.

The version of SAX that we use stipulates that memory cells are also *linear*, meaning that they can be written to once, then read once, and cannot be used again. This style of memory management is of particular interest for a couple of reasons. Firstly, in the parallel version of SAX, these memory cells can be used as linear futures [8]. Once a process has written a value to a cell, it stays there until it is read by another process. In parallel SAX, if the reader process reaches the read point before the writer has finished writing, then it must wait for the value to be written, much like the semantics of futures dictate in other languages. Additionally, this linearity allows for easy garbage collection, as was suggested by Girard and Lafont [5]. Once a cell is read, its memory can be deallocated – there is no need for more complex garbage collection procedures in linear code.

My previous research covered ways that linearity could also be exploited to reuse the memory allocated to a particular cell after it had been read. The key result was that if a SAX program is executed sequentially – we always allow the leftmost process to execute to completion before stepping any other processes forward – then reusing memory according to the system that we had created was type-safe and could never lead to a state where the leftmost process is unable to step forward.

However, SAX comes with a notable weakness in its implementation. All cells, no matter the type, are represented by a block of memory made up of one or two words. In the case of a pair, each word is used to store a pointer to each element of the pair. In the case of a sum, the first word is used to store the label of the sum, while the second word points to the actual contents. This leads to significant inefficiencies when unpacking the components of practical data structures such as binary search trees, which store several pieces of information at each node and therefore require several pointer dereferences. To make SAX more practical, Pfenning and DeYoung introduced SNAX in 2022. The SNAX language takes SAX and improves the memory layout, so that a value of a sum or product type is represented in a single contiguous block rather than being split up over several blocks [1]. This, in turn, means that not all blocks are the same size, making memory management harder. The trade-off is that accessing projections of a block can be done via simple address arithmetic rather than dereferencing pointers, making SNAX more efficient overall.

SNAX has a front-end that corresponds to the system of natural deduction, which is a much older logical system proposed by Gentzen in his 1935 paper on constructive logic [4]. This makes it easier for users to write programs in our language. One step in our compilation pipeline is to translate these natural deduction programs to a SNAX intermediate representation, which enables us to reap the benefits of memory reuse without forcing the user to directly interact with memory cells and risk writing an invalid program. Having natural deduction as a front-end also allows us to make our compilation more modular. Translating it into SNAX with reuse is only one of several intermediate representations that can be chosen – there are also options for translation to SNAX without reuse, or SAX.

2 Motivation

While the number of pointer dereferences has been significantly reduced in comparison to SAX, SNAX still suffers from another memory-related problem that the original SAX did – it is difficult to modify or otherwise use a data structure. This is because a function that modifies a data structure must first read its contents via pattern matching and perform its modifications. It must then recreate the node from the components that it has read, as well as the modified values. This process involves freeing a block of memory, which held the old value of the data structure, and then allocating an identically sized block. Recursive data structures, such as lists and binary search trees, acutely suffer from this memory inefficiency, since operations on these data structures often involve the manipulation of several nodes, all of which must be reallocated after modification. We therefore hypothesized that adding memory reuse to SNAX would significantly speed up these operations, since the block of memory that previously held the input data structure could be reused to store the value of the data structure that is being returned.

This is actually inspired by the way data structures are handled in imperative languages such as C and Java. Reusing the same block of memory for the updated data structure is reminiscent of an in-place update, which is the semantics with which these languages choose to modify their structures. This does lead to these data structures also being single-threaded, as past versions of a data structure cannot be easily recovered since they are not stored. Most applications of data structures, especially in imperative languages, already work around this sort of semantics, since they are so common. There have already been some attempts at adding similar in-place updates to functional languages, such as Microsoft's FP2 lorenzen2023fp, as well as Reinking and Xie's Perceus [9]. Perceus is one of the closest languages to SNAX with reuse, featuring both in-place updates as well as a more efficient form of garbage collection through reference counting. However, neither of these languages rely on linearity, which is at the center of SNAX.

In real-world applications, writing fully linear programs proves to be challenging, if not impossible. Thus, we make linear SNAX with reuse more practical to use by embedding it into a *adjoint* language, which has both linear and nonlinear components. This language was developed by Jang, Pfenning, Roshal, and Pietntka in their 2024 paper [6]. Programs written in this language can enjoy the benefits of memory reuse for their linear portions, while also containing the nonlinear components that are necessary for them to function. Data structures in the linear portions of our language already behaved in a single-threaded manner – this was not a disadvantage that was added by reuse. We hypothesize that memory reuse in the linear portions of programs will then allow SNAX's performance to improve without losing any of its expressivity.

3 Summary of Changes

To implement memory reuse in SNAX, we made several improvements to the language on various levels. In the front end, we added a pattern matching system to read an entire memory cell all at once, rather than having to make several calls to the read construct to extract its contents. This might seem like a small change to the concrete syntax for our language, but it is actually essential for effective memory reuse. Previously in SNAX, a memory cell is broken down through a series of read statements. If it is processed in this manner, then it cannot be reused until the final read has finished. This makes it much more difficult for the user to find opportunities for reuse. This problem is compounded because it can be unclear when a memory cell has been completely read in code with multiple branches. As such, we entirely removed the read construct from the language, and replace it with pattern matching.

Once we have established the match construct, we are able to identify the point in a program where each memory cell becomes available for reuse. To facilitate this reuse, we make several further modifications to linear SNAX. The biggest change is the addition of four new constructs, which are used to free a cell after it is read, dispose of a read cell that cannot be used anymore, reuse a free cell, and deallocate a free cell that is no longer needed. Furthermore, the type system is modified to account for the fact that a cell is not destroyed after it has been fully read, and instead can be referenced in a later reuse statement. This requires us to introduce new typing judgments for both individual processes and entire configurations in our modified SNAX system. Most of the static rules have also been modified to match these new judgements. Additionally, the dynamic rules have been changed to account for reuse in our linear system. These changes still obey the theorems of progress and preservation, ensuring that our language is type-safe.

4 Pattern Matching and Translation

The first change that we make is to remove the read construct from our language, and replace it with a match construct. Threads can no longer be of the form (read $a(\cdots \Rightarrow \dots)$). Instead, a thread that reads a memory cell A must be of the form match $a(\cdots \Rightarrow \dots)$. If we are pattern-matching the contents of a memory cell $a : A$, then the inner parentheses will contain a rule of the form $p \Rightarrow P$ for every pattern p that matches memory cells of type A, as defined below. The contents of a will match with one of the patterns when it is read, and it will then execute the corresponding branch. This branch P might reference some variables that are bound from the pattern p.

Definition (Patterns)
We define a pattern p as follows:

$$p ::= \langle\rangle \mid x \mid \langle x\rangle \mid \langle p_1, p_2\rangle \mid \ell\langle p\rangle$$

The three base cases are a unit pattern $\langle\rangle$, a variable x, and a pointer $\langle x\rangle$. The two recursive cases correspond to a pair of patterns $\langle p_1, p_2\rangle$ and a labeled injection of a pattern $\ell\langle p\rangle$.
We next define the judgement $\Delta \Vdash p : A$, which states that pattern p matches some of the values of type A. The context Δ is used as an output in this case. It holds the set of variables used in the pattern, as well as their types. If a memory cell's value matches a particular pattern, then the variables in Δ will be bound and made available for use in the corresponding branch. This is similar by the judgement described by Zeilberger in his 2008 thesis [10].
The rules for this judgement are as follows:

$$\frac{}{\cdot \Vdash \langle\rangle : 1}\ \textsc{P-Unit} \qquad \frac{}{x : A^- \Vdash x : A^-}\ \textsc{P-Var} \qquad \frac{}{x : A \Vdash \langle x\rangle : {\downarrow}A}\ \textsc{P-Pointer}$$

$$\frac{\Delta_1 \Vdash p_1 : A \quad \Delta_2 \Vdash p_2 : B \quad \Delta_1 \cap \Delta_2 = \emptyset}{\Delta_1, \Delta_2 \Vdash \langle p_1, p_2\rangle : A \otimes B}\ \textsc{P-Pair} \qquad \frac{\Delta_\ell \Vdash p_\ell : A_\ell \quad \ell \in L}{\Delta_\ell \Vdash \ell\langle p_\ell\rangle : \oplus\{\ell : A_\ell\}_{\ell \in L}}\ \textsc{P-Sum}$$

The variable and pointer rules allow us to bind variables. P-Var allows us to take a value of negative type from a memory cell and store it in the variable x, while P-Pointer lets us take a pointer to a cell of type A and extract the cell into a variable. Note that the P-Sum rule is not deterministic – there are several patterns that could apply to a variable with an $\oplus$ type. Additionally, a single-variable pattern is only allowed to match against a negative type. This means that unlike Standard ML, our patterns must fully match the contents of a memory cell. We are not allowed to match a positive type, such as a pair, to a single variable – we must break it down all the way to ensure that we are finished with the cell.

We can then determine whether a process using a match statement type-checks using the following rule:

$$\frac{\forall \Delta, p \text{ s.t. } (\Delta \Vdash p : A) \;.\; p \in \{p_1, p_2, \ldots\} \text{ and } \Delta, \Gamma \vdash Q_p :: (c : C)}{\Gamma, a : A \vdash \text{match } a(p_1 \Rightarrow Q_1 \mid p_2 \Rightarrow Q_2 \mid \ldots) :: (c : C)}$$

In this rule, we verify that a process using a match statement is valid. This match statement reads a value of type A from memory cell a, and matches it against the patterns in the parentheses. Multiple patterns and branches will exist between the parentheses if there are multiple patterns that match values of type A. The first part of the premise states that every possible pattern for type A needs to be included in the set of patterns $\{p_1, p_2, \ldots\}$ that are included in the match statement. The second portion states that each branch needs to be well-typed after the variables in the pattern are bound. Together, these guarantee that the process using this match statement is well-typed and covers every possible pattern.

In SNAX with reuse, our programs will be written using this match construct. This allows us to guarantee that all of the contents of the cell are read, and that no references to the cell remain after the match completes. This, in turn, means that a cell can be reused after it has been read via a pattern-match. During compilation, we will translate each match statement to a series of read statements. These statements will be equivalent to the match statement, in that they will read the contents of a block and bind the necessary variables. After this series of statements executes, the block will also be freed using a free statement (described in the next section), which makes it ready for reuse. By abstracting these read and free statements away from the user, we guarantee that they will be used in a reuse-friendly manner that allows for progress and preservation to hold.

The translation we introduce begins with a match statement in the form match $\alpha(p_1 \Rightarrow Q_1 \mid \ldots)$. We use α here to denote the memory cell that is being matched, since we know that it is the root of a block. At the end of our translation, we will need to free α after all parts of it have been read. The judgement $P \rightsquigarrow P'$ indicates that the process P, which may use match statements, translates to P', which uses read and free statements. In doing so, we will make use of two intermediate constructs. $\text{match}_F\ (\alpha, a)(p_1 \Rightarrow Q_1 \mid \ldots)$ will match a against the appropriate branch, and then free α before Q begins execution. $\text{match}_N\ a(p_1 \Rightarrow Q_1 \mid \ldots)$ simply matches a with its pattern and executes the corresponding branch.

Translation Rules

Initializing the translation:

$$\frac{\text{match}_F\ (\alpha, \alpha)(p_1 \Rightarrow Q_1 \mid \ldots) \rightsquigarrow P}{\text{match } \alpha(p_1 \Rightarrow Q_1 \mid \ldots) \rightsquigarrow P}\ \textsc{Init}$$

Base cases:

$$\frac{}{\text{match}_F\ (\alpha, a)(\langle\rangle \Rightarrow Q) \rightsquigarrow \text{read } a(\langle\rangle \Rightarrow \text{free } \alpha; Q)}\ \textsc{Unit-F}$$

$$\frac{Q \rightsquigarrow Q'}{\text{match}_N\ a(\langle\rangle \Rightarrow Q) \rightsquigarrow \text{read } a(\langle\rangle \Rightarrow Q')}\ \textsc{Unit-N}$$

$$\frac{}{\text{match}_F\ (\alpha, a)(x \Rightarrow Q) \rightsquigarrow \text{let } x = a \text{ in free } \alpha; Q}\ \textsc{Var-F} \qquad \frac{Q \rightsquigarrow Q'}{\text{match}_N\ a(x \Rightarrow Q) \rightsquigarrow \text{let } x = a \text{ in } Q'}\ \textsc{Var-N}$$

$$\frac{}{\text{match}_F\ (\alpha, a)(\langle x\rangle \Rightarrow Q) \rightsquigarrow \text{read } a(\langle x\rangle \Rightarrow \text{free } \alpha; Q)}\ \textsc{Ptr-F} \qquad \frac{Q \rightsquigarrow Q'}{\text{match}_N\ a(\langle x\rangle \Rightarrow Q) \rightsquigarrow \text{read } a(\langle x\rangle \Rightarrow Q')}\ \textsc{Ptr-N}$$

Inductive cases:

$$\frac{\text{match}_F\ (\alpha, a \cdot \overline{\ell_i})(p_i \Rightarrow Q_i) \rightsquigarrow Q_i'}{\text{match}_F\ (\alpha, a)(\ell_1\langle p_1 \rangle \Rightarrow Q_1 \mid \dots) \rightsquigarrow \text{read } a(\ell_1\langle \_ \rangle \Rightarrow Q_1' \mid \dots)}\ \textsc{Sum-F}$$

$$\frac{\text{match}_N\ a \cdot \overline{\ell_i}(p_i \Rightarrow Q_i) \rightsquigarrow Q_i'}{\text{match}_N\ a(\ell_1\langle p_1 \rangle \Rightarrow Q_1 \mid \dots) \rightsquigarrow \text{read } a(\ell_1\langle \_ \rangle \Rightarrow Q_1' \mid \dots)}\ \textsc{Sum-N}$$

$$\frac{\text{match}_N\ a \cdot \pi_1(p_{1,1} \Rightarrow (\text{match}_F\ (\alpha, a \cdot \pi_2)(p_{2,1} \Rightarrow Q_{1,1} \mid \dots)) \mid \dots) \rightsquigarrow Q'}{\text{match}_F\ (\alpha, a)(\langle p_{1,1}, p_{2,1} \rangle \Rightarrow Q_{1,1} \mid \dots) \rightsquigarrow \text{read } a(\langle \_, \_ \rangle \Rightarrow Q')}\ \textsc{Prod-F}$$

$$\frac{\text{match}_N\ a \cdot \pi_1(p_{1,1} \Rightarrow (\text{match}_N\ a \cdot \pi_2(p_{2,1} \Rightarrow Q_{1,1} \mid \dots)) \mid \dots) \rightsquigarrow Q'}{\text{match}_N\ a(\langle p_{1,1}, p_{2,1} \rangle \Rightarrow Q_{1,1} \mid \dots) \rightsquigarrow \text{read } a(\langle \_, \_ \rangle \Rightarrow Q')}\ \textsc{Prod-N}$$

The product rules have to support multiple cases because the left element of the pair could match to any of several different patterns $\{p_{1,1}, p_{1,2}, \dots\}$, while the right element also can match to one of the patterns in $\{p_{2,1}, p_{2,2}, \dots\}$. As such, when matching a pair, we need a branch for every choice of left pattern and right pattern. We translate this by first matching the left element. In each of the branches for left elements, we then match the right element, and execute the corresponding code.

This translation makes use of a key invariant, which states that after we initialize the translation, there is exactly one instance of match$_F$ and no instances of match$_N$ can occur after it. This allows the last base case to safely free the root α after matching, since no further accesses to it will occur. This invariant is preserved in the recursive cases – the Sum-F rule passes the responsibility of freeing α into each branch, while Prod-F makes sure $a \cdot \pi_2$ frees the whole cell once it is read. The rules for both sums and products also account for the fact that multiple patterns might match a. In particular, the product rule first matches $a \cdot \pi_1$ against one of its possible matches $p_{1a}, p_{1b}, \dots$. Then, in the case for each pattern, it matches $a \cdot \pi_2$ with one of its possible matches $p_{2a}, p_{2b}, \dots$.

Translating the match construct to a series of read and free operations makes compilation easier, since it becomes more similar to the existing linear SNAX language without reuse. Moreover, this similarity allows us to more easily prove progress and preservation.

5 Changes to SNAX

Now that we have established the pattern-matching syntax that makes memory reuse possible, we can begin to modify the language to add the relevant constructs. Previously in SNAX, a thread could take on several forms – it could either write to a memory cell, read a memory cell and use its contents, or copy data from one memory cell to another. Threads could also use the cut rule, which split a thread into two threads, and gave them a memory cell on which they could communicate. When we add memory reuse, we add four forms to processes, as follows:

$$P ::= \cdots \mid \text{free } \alpha; P \mid \text{dealloc } \alpha; P \mid \text{dispose } a; P \mid (\text{reuse } \alpha \leftarrow P_1; P_2)$$

The first new form simply frees the root of a memory cell, making it available for reuse. The cell still exist in the configuration, but is in a free state. The second construct is used to deallocate a memory cell that is no longer needed, which removes it entirely – it no longer exists in the configuration at all. The third new form is used to dispose of a memory cell that is no longer being used, but still needs to remain in the configuration as other cells might rely on it. The last new construct is similar to the cut rule, in that it spawns two processes. Instead of creating a new memory cell for the processes to communicate, it reuses the existing free cell a.

We additionally need to modify the semantics for memory cells. Previously, any memory cells that appeared in the configuration were of the form cell x S, which stated that the cell x held the storable S. Moreover, this implied that the cell was available for a process to read. Once the cell was read, it would be removed entirely from the configuration. Adding memory reuse to the system requires us to not destroy memory cells once they are read, since they might be referenced again later. Thus, we define memory cells as follows:

$$M ::= \text{cell } a \ \square \mid \text{cell } a \ S \mid \text{read } a \ S \mid \text{free } a \ S \mid \text{referenced } a \ S$$

The first two constructors in this grammar are the same as before. The first one represents a cell that a process is going to write to, but has not written to yet. The second one represents a cell named a that holds a storable S, which is available for use. Once it is used, it becomes a read cell, instead of being destroyed. Read cells use the third constructor, read a S. They still contain their previous contents, so that we are able to find the type of a read cell. However, they cannot be used in any way other than being freed, which creates a free cell, free a S. These free cells can either be deallocated, which removes them entirely from the configuration, or they can be reused. Finally, a cell can be referenced, meaning that a pointer to it exists elsewhere in memory. A referenced cell, referenced a S, still exists, but cannot be used in any way – until the cell containing the corresponding pointer $\langle a \rangle$ is read, at which point the cell becomes free. The referenced state for a cell is therefore used to indicate that a cell is not currently usable, but needs to remain in the configuration for some reason. The transitions between these states are governed by the dynamic rules, which will be described later.

The grammar for programs remains largely unchanged. The current state of a program is defined by its configuration, which is a set of memory cells and processes ordered from left to right. Thus, we get:

$$C ::= \cdot \mid M \mid \text{thread } (a, P) \mid C_1 \ C_2$$

A configuration can be empty, or made up of a single memory cell M. It can also consist of a thread, which is made up of a process P and a memory cell a which it will write its results to. Finally, a configuration can be the concatenation of two smaller configurations, which allows for more complex programs to be written.

5.1 Type System

In this language, we need a type system which properly models the states that a memory cell is in. Memory cells no longer carry just a type in the context – rather, they carry both the type of the data they store, and their current status. Thus, the typing judgement for memory cells is of the form $M : \tau[s]$, where the *status* $[s]$ is either [available], [read], [free], or [referenced]. This allows the type system to account for the current state of each memory cell at any time. As a result, the context Γ must hold all of the memory cells, as well as their types and statuses. It is no longer limited to the cells that are available.

Processes must now contend with the new typing judgements and contexts as well. Previously, we wrote $\Gamma \vdash P :: (a : A)$ to denote that a process P used the values in Γ to write a value of type A to the memory cell a, which would be available for use. However, the typing judgements in our new calculus must contend with the fact that a process P will make use of some memory cells in Γ, and that these memory cells might still exist after the process is done executing! Thus, we instead write $\Gamma \vdash P :: (a : A); \Gamma'$ for our typing judgement. This states that given the memory cells in Γ with their provided types and statuses, the process P will execute. After P is done executing, a will be available and contain a value of type A, while the remaining cells will now match the types and statuses in Γ', the output context. These cells' types should not change, however, their statuses likely change from operations that P carries out as it executes.

Finally, we have the matter of typing a configuration. In nonlinear SNAX, a distinction was made between configuration contexts Φ and process contexts Γ. However, in linear SNAX with reuse, no such distinction exists. Configuration contexts were notable in that they did not support contraction, and that their addresses were presumed to be distinct [1]. However, contraction is already not supported in a linear process calculus. Moreover, our type system prohibits two pointers from pointing to the same address. Therefore, we can use the same kinds of contexts Γ for typing processes and configurations.

It follows that the configuration typing judgements $\Phi \vDash C :: \Phi'$ can largely remain unchanged. In linear SNAX with reuse, we write these judgements as $\Gamma \vDash C :: \Gamma'$ to express that C, when given the memory cells in Γ, will execute and leave behind a configuration of type Γ'. This is very similar to both the original style of the configuration typing judgement and the typing judgement for a single process.

5.2 Statics

5.2.1 Process Typing Rules

Many of the process typing rules are similar to those presented in the original SNAX paper, with linearity added to match the requirements of our language. The judgments have been converted to match the new $\Gamma \vdash P :: (a : \tau); \Gamma'$ format as well. We begin by looking at reading and writing basic types to memory.

Units

Processes can read and write unit values of type 1 as follows:

$$\frac{}{\Gamma \vdash \text{write } a\ \langle\rangle :: (a : 1); \Gamma}\ 1\text{A} \qquad \frac{\Gamma, a : 1[\text{read}] \vdash P :: (c : C); \Gamma'}{\Gamma, a : 1[\text{available}] \vdash \text{read } a(\langle\rangle \Rightarrow P) :: (c : C); \Gamma'}\ 1\text{L}$$

The axiomatic rule for a unit remains largely unchanged. Writing a unit to a memory cell a makes it available with type 1, while not affecting any other memory cells in the process. The main change to the rule comes from making it work with our new judgement format. The changes to the 1L rule are a bit more interesting. We take a context that has a cell a of type 1[available] and read it, as before. However, rather than entirely eliminating a, we allow the process P to have access to it with the [read] status. This allows P to continue interacting with a, and reflects the fact that it is not deallocated entirely.

Pairs

$$\frac{}{\Gamma, \underline{a \cdot \pi_1 : A_1}[\text{available}], \underline{a \cdot \pi_2 : A_2}[\text{available}] \vdash \text{write } a\ \langle\_,\_\rangle :: (a : A_1 \otimes A_2); \Gamma}\ \otimes\text{A}$$

$$\frac{\Gamma, \underline{a \cdot \pi_1 : A_1}[\text{available}], \underline{a \cdot \pi_2 : A_2}[\text{available}], a : A_1 \otimes A_2[\text{read}] \vdash P :: (c : C); \Gamma'}{\Gamma, a : A_1 \otimes A_2[\text{available}] \vdash \text{read } a(\langle\_,\_\rangle \Rightarrow P) :: (c : C); \Gamma'}\ \otimes\text{L}$$

Here, the two cells $a \cdot \pi_1$ and $a \cdot \pi_2$ can be consumed by the write operation to write the value of type $A_1 \otimes A_2$ to the memory cell a. Thanks to the memory layout of SNAX, no actual copying needs to occur. Instead, we lose access to $a \cdot \pi_1$ and $a \cdot \pi_2$, and can treat them as one memory cell a which is available. The read operation on pairs splits up a into its two parts, and makes them available for the rest of P to use. As P operates, it changes the remainder of the context from Γ into Γ', which is the output context, and also makes c into a cell of type C[available].

Labeled Sums

$$\frac{(k \in L)}{\Gamma, \underline{a \cdot \overline{k}} : A_k[\text{available}] \vdash \text{write } a\ k\langle\_\rangle :: (a : \oplus\{\ell : A_\ell\}_{\ell \in L}); \Gamma} \oplus\text{A}$$

$$\frac{\forall \ell \in L : \Gamma, \underline{a \cdot \overline{\ell}} : A_\ell[\text{available}], a : \oplus\{\ell : A_\ell\}_{\ell \in L}[\text{read}] \vdash P_\ell :: (c : C); \Gamma'}{\Gamma, a : \oplus\{\ell : A_\ell\}_{\ell \in L}[\text{available}] \vdash \text{read } a\ (\ell\langle\_\rangle \Rightarrow P_\ell)_{\ell \in L} :: (c : C); \Gamma'} \oplus\text{L}$$

Much like the product rules, the sum rules are very similar to the existing SNAX ones. The rule for writing a sum is axiomatic, requiring only that the contents already be written and that a valid tag is selected. The most important change comes from the rule for eliminating sums. In this rule, every branch P_ℓ must write a value of type C to the cell c, as before. This allows us to state that the overall process has type $(c : C)$. In SNAX with reuse, an additional condition is added – all of these branches must also leave the remainder of the configuration in the state corresponding to Γ'. This ensures that no matter which branch is taken, a predictable set of resources are left behind for the next process to operate with.

Pointers

$$\frac{}{\Gamma, b : A[\text{available}], \text{proj}(b)[\text{available}] \vdash \text{write } a\ \langle b\rangle :: (a : \downarrow A); \Gamma, b : A[\text{referenced}], \text{proj}(b)[\text{referenced}]} \downarrow\text{A}$$

$$\frac{\Gamma, x : A[\text{available}], a : \downarrow A[\text{read}] \vdash P :: (c : C); \Gamma' \quad x \notin \Gamma'}{\Gamma, a : \downarrow A[\text{available}] \vdash \text{read } a\ (\langle x\rangle \Rightarrow P) :: (c : C); \Gamma'} \downarrow\text{L}$$

Here, we introduce the [referenced] status into our judgements to represent what happens to a cell b when it is referenced by a pointer, which is written to the cell a. Since this cell is already being referenced, we cannot allow it to be used as an available cell – otherwise, this use could lead to us violating the linearity requirement of our language. In SNAX without reuse, we solved this problem by entirely removing the cell from the configuration (and therefore the context), however, we cannot do this in SNAX with reuse since this memory cell still needs to be accessible. As such, we mark a memory cell as [referenced] when its pointer is written.

Once the pointer is dereferenced through a read statement, the cell x that a was pointing to becomes available for use. However, we need to ensure that x does not escape its scope, and we therefore require that the process P that uses x removes it from the context as necessary. This is reflected in the $\downarrow$L rule, which allows us to read a pointer and have its contents available for use in the remainder of the process P – but not in the resulting context Γ'.

Functions

Functions are different from the previously mentioned types in that their axiomatic rule comes when they are used, rather than when they are created.

$$\frac{\Gamma_2, x : A_1[\text{available}] \vdash P :: (z : A_2); \Gamma_2', x : A_1[\text{read}] \quad \Gamma_2^{\text{refd}} \subseteq \Gamma_2'}{\Gamma_1, \Gamma_2 \vdash \text{write } a(\langle x, z\rangle \Rightarrow P) :: (a : A_1 \to A_2); \Gamma_1} \to\text{R}$$

$$\frac{}{\Gamma, a : A_1 \to A_2[\text{available}], a_1 : A_1[\text{available}] \vdash \text{read } a\ \langle a_1, a_2\rangle :: (a_2 : A_2); \Gamma, a : A_1 \to A_2[\text{read}], a_1 : A_1[\text{read}]} \to\text{A}$$

As in SNAX, a function of type $A_1 \to A_2$ is represented as something which takes two addresses – the first is the address of the input, while the second is the address that it is supposed to write its results to. Since functions might be written at one time and then be called at another, we need some way to ensure that the resources from the context that the function requires are still there when it gets called. To do this, we split up the context into two parts, Γ_1 and Γ_2. Γ_1 continues to be usable after the function has been written. Γ_2 contains the resources used by the function body, which will use them later when it is called. As such, we remove these resources from the context so that further processes do not alter them. The function body is also required to leave every memory cell from Γ_2 in the [referenced] state when it concludes. This is expressed by the condition $\Gamma_2^{\text{refd}} \subseteq \Gamma_2'$ – we use Γ_2^{refd} to denote the set of cells in Γ_2, with all of their statuses changed to [referenced]. Γ_2' may also contain other memory cells allocated by the process, since z, the output of the function, might rely on them. We also require that the input to the function is left in a [read] state to ensure consistency with the $\to\textsc{a}$ rule.

Since functions are a negative type, the rule to invoke a function is the axiomatic one. Here, we require that both the function being called and its input are available. They are both read, and the body of the function will write a value of type A_2 to the provided destination cell a_2. No other memory cells in the configuration are affected, so the remainder of the context Γ remains the same.

Lazy Products

$$\frac{\forall \ell \in L : \Gamma_2 \vdash P_\ell :: (y : A_m^\ell); \Gamma_2' \quad \Gamma_2^{\text{refd}} \subseteq \Gamma_2'}{\Gamma_1, \Gamma_2 \vdash \text{write } a\ (\ell(y) \Rightarrow P_\ell)_{\ell \in L} :: (a : \&_m\{\ell : A_m^\ell\}_{\ell \in L}); \Gamma_1}\ \&\textsc{r}$$

$$\frac{(k \in L)}{\Gamma; a : \&_m\{\ell : A_m^\ell\}_{\ell \in L}[\text{available}] \vdash \text{read } a\langle k, y\rangle :: (y : A_m^k); \Gamma, a : \&_m\{\ell : A_m^\ell\}_{\ell \in L}[\text{read}]}\ \&\textsc{a}$$

The type of lazy products, also known as lazy records, is similar to both the sum and function types. Lazy products do not perform any computations when they are created (since they are lazy), thus, writing them only requires that each of the branches is appropriately typed. As with functions, we split the context into two parts. Γ_1 contains the resources that the lazy product does not interact with, while Γ_2 contains the resources that every branch is allowed to use when it is invoked. Importantly, every branch is given the same Γ_2, rather than having different contexts for separate branches. Invoking a lazy product requires passing in the corresponding label, as well as a destination cell. At this point, the result is written to the destination, and the lazy product is marked as being read. The same condition of $\Gamma_2^{\text{refd}} \subseteq \Gamma_2'$ is required for each branch of the lazy product.

Calling Processes

$$\frac{}{\Gamma, a_1 : A_1[\text{available}], \ldots, a_n : A_n[\text{available}] \vdash \text{call } p\ c\ a_1 \ldots a_n :: (c : C); \Gamma}\ \textsc{Call}$$

where the process p is defined as proc $p\ (z : C)\ (x_1 : A_1) \ldots (x_n : A_n) = P$. This rule has no premises as the definitions of processes are instead checked separately. We can check a function definition proc $p\ (z : C)\ (x_1 : A_1) \ldots (x_n : A_n) = P$ by checking that $x_1 : A_1[\text{available}], \ldots, x_n : A_n[\text{available}] \vdash P :: (z : C); \Gamma$ such that $x_1 : A_1[\text{referenced}], \ldots, x_n : A_n[\text{referenced}] \in \Gamma$, in the same style as type-checking a closure. We do this using the type declarations for all function definitions. This allows us to handle recursion without having the typechecker risk nontermination.

As in SNAX, we also have a way to invoke a recursive function to write to a particular destination. We call such a recursive function a ‘process’. We can make use of a process by calling it with a destination and the required arguments. This causes the process to take over the existing thread, taking control of the arguments which were passed in. The remaining resources stay in Γ and are not touched at all. Since we have no way of reasoning about the behavior of the called process p, we instead force it to deallocate the arguments and any cells that it happens to create.

Identity

$$\frac{}{\Gamma, b : A^{-}[\text{available}] \vdash \text{move } a\ b :: (a : A^{-}); \Gamma, b : A^{-}[\text{read}]}\ \text{ID}$$

The identity rule is responsible for moving data from a memory cell b to another one of the same type a, marking b as being read in the process. It is similar to its counterpart in SNAX without reuse in that it performs this move without affecting the statuses of any other memory cells. It is also similar in that these move operations are only available for variables of type $\downarrow A$, $A_1 \to A_2$, and $\&_m\{\ell : A_m^{\ell}\}$. Variables with unit, sum, and product types must be read from the old cell and written to the new cell, as in SNAX.

Cuts and Snips

$$\frac{\Gamma_1 \vdash P :: (x : A); \Gamma_2 \quad \Gamma_2, x : A[\text{available}] \vdash Q :: (c : C); \Gamma_3 \quad (x \text{ fresh})}{\Gamma_1 \vdash x \leftarrow P; Q :: (c : C); \Gamma_3}\ \text{CUT}$$

$$\frac{\Gamma_1 \vdash P :: (a : A); \Gamma_2 \quad \Gamma_2, \underline{a : A}[\text{available}] \vdash Q :: (c : C); \Gamma_3}{\Gamma_1 \vdash P; Q :: (c : C); \Gamma_3}\ \text{SNIP}$$

The cut rule allows us to spawn two processes, P and Q, and gives them a memory cell x on which they can communicate. P writes a value to x when it is completed, and Q reads this value from x to create a value of type C in memory cell C. The remainder of the configuration can be modified by both P and Q. In particular, the starting context Γ_1 is modified by P to create Γ_2, as the execution of P may read or write other cells. Γ_2 is therefore the context that Q has access to during its execution, which eventually results in Γ_3, which is also the overall resulting context of the process as a whole.

The snip rule is similar to the cut rule, but it does not allocate a new memory cell at all. Instead, the process P writes its result to an address which already exists within a previously allocated memory cell, and Q reads from that address. This rule is also largely unchanged, apart from modifying it to work with the new structure of the typing judgements.

Free

$$\frac{\Gamma, \alpha : A[\text{free}] \vdash P :: (c : C); \Gamma', \text{proj } (\alpha)[\text{free}]}{\Gamma, \alpha : A[\text{read}], \text{proj}(\alpha)[\text{read}] \vdash \text{free } \alpha; P :: (c : C); \Gamma'}\ \text{FREE}$$

In this rule, we use proj(α) to refer to a set of addresses related α. Every element in this set of projections must be read before we can free the cell. We define this set as follows:

- For a cell a of type $A_1 \otimes A_2$, proj(a) is defined to be $\{a \cdot \pi_1, a \cdot \pi_2\} \cup \text{proj}(a \cdot \pi_1) \cup \text{proj}(a \cdot \pi_2)$.
- For a cell a of type $\oplus\{\ell : A_\ell\}_{\ell \in L}$, proj($a$) is defined as any one of the sets $\{a \cdot \overline{k}\} \cup \text{proj}(A_k)$ for any $k \in L$.
- For a cell a of any other type, proj(a) is empty as a has no projections.

This means that freeing a product requires us to have read both its π_1 and π_2 components fully, while freeing a sum requires us to have read exactly one of the summands.

There is an important distinction between two types of addresses in this rule – there are *roots* α, which are guaranteed to not have any projections in their addresses, and general addresses a which can have projections. We only allow a root to be freed. Other addresses can be marked as read, however, they cannot be explicitly freed – instead, they are freed when their corresponding root is freed. When a memory cell is freed, it is made available for the rest of the process to interact with. These interactions can come either in the form of reuse or deallocation.

Reuse

$$\frac{\Gamma_1 \vdash P :: (\alpha : A); \Gamma_2 \quad \Gamma_2, \alpha : A[\text{available}] \vdash Q :: (c : C); \Gamma_3}{\Gamma_1, \alpha : A[\text{free}], \text{proj}(\alpha)[\text{free}] \vdash \text{reuse } (\alpha \leftarrow P; Q) :: (c : C); \Gamma_3} \text{ REUSE}$$

The reuse rule is extremely similar to the cut rule. Both cut and reuse cause two processes to be spawned, and provide them with a memory cell with which they can communicate. The primary difference between the two is that the cut rule creates an entirely new memory cell x with a fresh label, while the reuse rule simply uses an existing free cell α for this purpose. Once this occurs, P and Q are free to use α to communicate as before. α can also occur in Γ_3, which allows further processes in the configuration can refer to this cell as a after the name x has gone out of scope.

This reuse rule is slightly different from the one that was used to add reuse to SAX. In SAX, all memory cells had the same size. Therefore, we were free to reuse any cell that had already been read to store values of any other type. This no longer applies to SNAX, where the size of a memory cell is dependent on the type of the values it holds. As a result, we restrict reuse to only be legal on values of the same type. This means that a memory cell of type A can only be reused as a memory cell of type A. Even if type B is the same size as A when laid out in memory, we will not allow a memory cell of type A to be reused to hold a value of type B, which helps ensure memory safety.

Deallocation

$$\frac{\Gamma \vdash P :: (c : C); \Gamma'}{\Gamma, \alpha : A[\text{free}], \text{proj}(\alpha)[\text{free}] \vdash \text{deallocate } \alpha; P :: (c : C); \Gamma'} \text{ DEALLOC}$$

The deallocation rule allows us to entirely remove a memory cell once we no longer need it. We require that a cell which we are deallocating has first been fully read and freed using the free(α) construct. Once this has happened, we can remove it, and its projections, from the configuration and the context, and the rest of the process P operates without access to this cell. This is particularly important for ensuring functions and lazy products can type-check, since they are required to finish executing with an empty context. This means that any cells they allocate must be cleaned up via this deallocate construct.

Disposal

$$\frac{\Gamma, a : A[\text{referenced}], \text{proj}(a)[\text{referenced}] \vdash P :: (c : C); \Gamma'}{\Gamma, a : A[\text{read}], \text{proj}(a)[\text{read}] \vdash \text{dispose}(a); P :: (c : C); \Gamma'} \text{ DISPOSE}$$

Our last new construct is disposal. The disposal construct takes a read cell and changes the status of it, and its projections, to [referenced]. These cells are marked as referenced to denote the fact that they cannot be used anymore, but still might be needed in the configuration to ensure other cells type-check. Since no pointers to these cells exist, their state will remain as [referenced] for the remainder of the program. This disposal construct is useful for functions that read an argument cell a and need to set a's status to [referenced] to comply with the $\rightarrow$ R.

5.2.2 Configuration Typing Rules

Now that we have established how we can find the type of a particular process, we turn our attention to how we can find the type of an entire configuration. As mentioned earlier, a configuration is a set of memory cells and processes, ordered from left to right. As such, we have rules to type configurations made up of individual threads, as well as configurations made up of multiple parts. However, our linear system allows us to use the same contexts Γ for both individual processes and entire configurations, rather than having a special kind of configuration contexts Φ.

Configuration Typing Rules

$$\frac{}{\Gamma \vDash \cdot :: \Gamma}\ \textsc{Empty} \qquad \frac{\Gamma_0 \vDash C_1 :: \Gamma_1 \quad \Gamma_1 \vDash C_2 :: \Gamma_2}{\Gamma_0 \vDash C_1 C_2 :: \Gamma_2}\ \textsc{Join} \qquad \frac{\Gamma \vdash P :: (a : A); \Gamma'}{\Gamma \vDash \text{thread } (a, P), \text{cell } a\ \square :: \Gamma', a : A[\text{available}]}\ \textsc{Thread}$$

$$\frac{\Gamma^a \vdash \text{write}(a, S) :: (a : A); \Gamma'}{\Gamma \vDash \text{cell } a\ S :: R(\Gamma, \Gamma'), a : A[\text{available}]}\ \textsc{Cell} \qquad \frac{\Gamma^a \vdash \text{write}(a, S) :: (a : A); \Gamma'}{\Gamma \vDash \text{read } a\ S :: \Gamma, a : A[\text{read}]}\ \textsc{Read}$$

$$\frac{\Gamma^a \vdash \text{write}(a, S) :: (a : A); \Gamma'}{\Gamma \vDash \text{free } a\ S :: \Gamma, a : A[\text{free}]}\ \textsc{Free} \qquad \frac{\Gamma^a \vdash \text{write}(a, S) :: (a : A); \Gamma'}{\Gamma \vDash \text{ref } a\ S :: \Gamma, a : A[\text{referenced}]}\ \textsc{Referenced}$$

Each of these configuration typing rules takes the form of $\Gamma \vDash C :: \Gamma'$. Γ is the context, while Γ' is the result type. Both of these are sets of typing judgements, which map memory cells to their eventual types and statuses after execution has completed. A configuration is initially typed with an empty context. However, the Join rule allows us to find the result type of the left-hand portion of a configuration, which can then be used as the context for typing the right-hand portion.

The remaining rules are used as base cases for typing individual threads or memory cells. When we type a thread thread (a, P), we give it access to Γ in its current state, which corresponds to the resources that the process P is allowed to use as it computes a value to be written into cell a. Typing a cell in any status – available, read, free, or referenced – simply requires us to find the type of its contents. To accomplish this, we find the type of the storable S that is stored in this cell. This is accomplished by considering what the output type of a process writing S to a would be.

However, there is a problem with trying to derive this judgement using Γ, the current state of the context. S might rely on some variables that are currently in a referenced state, which would prevent the process write(x, S) from interacting with them. Additionally, if S is stored in a read cell, then it might rely on other cells that have also been read. The same principle applies to free cells. As such, when typing cells, we use a version of Γ where the status of every cell is set to [available], which we call Γ^a. This allows us to type the corresponding cell.

The context Γ' that comes after using Γ^a to type a read, free, or referenced cell can be thrown away, because these cells do not currently contain a usable value S. In these cases, Γ^a existed purely to find the type of the cell. However, in the case of an available cell, the resulting context does matter. Γ' no longer contains cells that have been consumed while writing the value S, which should not exist in the context afterward. They still exist in the configuration to the left of the cell a, however, they are not accessible to any type-safe processes to the right of a. As such, we need to use Γ' to get our resulting context, by *restoring* the statuses of all cells in Γ' from Γ. We call this $R(\Gamma, \Gamma')$, and it is defined as follows:

- If a cell exists in Γ but no longer exists in Γ' (because it has been consumed), then it does not exist in $R(\Gamma, \Gamma')$.
- If a cell exists in Γ with status s and still exists in Γ', then it exists in $R(\Gamma, \Gamma')$ and inherits its previous status s from Γ.

In practice, the only cells that meet the first condition are the projections of a, which are consumed if a has a sum or pair type. If a is a pointer to some other cell b, then b will exist in Γ' as a referenced cell, which should match its state in Γ. Writing the value S should not interact with any other cells, meaning that their statuses will be restored from their original ones in Γ.

5.3 Dynamics

In SNAX with reuse, the dynamic rules are used to state how a configuration can step forward to another configuration. The threads in a configuration are *ordered.* Unlike SNAX, which was designed to run concurrently, our version of linear SNAX with reuse runs sequentially and deterministically. Dynamic rules that step a thread forward only can be used if the thread stepping forward is the leftmost one in the configuration. If a thread is not the leftmost one, then it must wait for all threads to the left of it to finish executing before it can begin to step forward. Beyond this ordering requirement, many of these dynamic rules are the same as in linear SNAX, but also account for the fact that the status of a cell is often changed when it is read.

Dynamic Rules

Writing storables S and pointers $\langle x\rangle$:

$$\text{thread } (a, \text{write } a\ S), \text{cell } a\ \square \longmapsto \text{cell } a\ S$$

$$\text{thread } (a, \text{write } a\ \langle x\rangle), \text{cell } x\ S, \text{cell proj}(x), \text{cell } a\ \square \longmapsto \text{cell } a\ \langle x\rangle, \text{referenced } x\ S, \text{referenced proj}(x)$$

Reading general storables S and pointers $\langle x\rangle$:

$$\text{thread } (c, \text{read } a\ T), \text{cell } a\ S \longmapsto \text{thread } (c, S \triangleright T), \text{read } a\ S$$

$$\text{thread } (c, \text{read } a\ (\langle x\rangle \Rightarrow P)), \text{referenced } y\ S, \text{referenced proj}(y), \text{cell } a\ \langle y\rangle$$

$$\longmapsto$$

$$\text{thread } (c, [y/x]P), \text{cell } y\ S, \text{cell proj}(y), \text{read } a\ S$$

Calling recursive functions (where proc $p\ z\ x_1 \ldots x_n = P$)

$$\text{cell } a_1\ S_1, \ldots, \text{cell } a_n\ S_n, \text{cell } c\ \square, \text{thread } (c, \text{call } p\ c\ a_1 \ldots a_n)$$

$$\longmapsto$$

$$\text{cell } a_1\ S_1, \ldots, \text{cell } a_n\ S_n, \text{cell } c\ \square, \text{thread } (c, [c/z, a_1/x_1, \ldots a_n/x_n]P)$$

Moving values between cells (only for negative types):

$$\text{thread } (a, \text{move } a\ b), \text{cell } b\ S \longmapsto \text{cell } a\ S, \text{read } b\ S$$

The cut, snip, and reuse rules split a process into two processes:

$$\text{thread } (c, (x \leftarrow P; Q)) \longmapsto \text{thread } (\alpha, [\alpha/x]P), \text{cell } a\ \square, \text{thread } (c, [\alpha/x]Q) \qquad (\alpha \text{ fresh})$$

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

$$\text{thread } (c, \text{reuse } \alpha(x \leftarrow P; Q)), \text{free } \alpha\ S, \text{free proj}(\alpha) \longmapsto \text{thread } (\alpha, [\alpha/x]P), \text{cell } a\ \square, \text{thread } (c, [\alpha/x]Q)$$

Freeing, deallocating, and disposing read cells:

$$\text{thread } (c, \text{free } \alpha; P), \text{read } \alpha\ S, \text{read proj}(\alpha) \longmapsto \text{thread } (c, P), \text{free } \alpha\ S, \text{free proj}(\alpha)$$

$$\text{thread } (c, \text{deallocate } \alpha; P), \text{free } \alpha\ S, \text{free proj}(\alpha) \longmapsto \text{thread } (c, P)$$

$$\text{thread } (c, \text{dispose } a; P), \text{read } a\ S, \text{read proj}(a) \longmapsto \text{thread } (c, P), \text{referenced } a\ S, \text{referenced proj}(a)$$

The judgement $S \triangleright T$ is used in the rules for reading general storables S out of memory cells. It is defined as:

$$\langle\_,\_\rangle \triangleright (\langle\_,\_\rangle \Rightarrow P) = P$$

$$\langle\rangle \triangleright (\langle\rangle \Rightarrow P) = P$$

$$k\langle\_\rangle \triangleright (\ell\langle\_\rangle P_\ell)_{\ell\in L} = P_k \quad (k \in L)$$

$$(\langle x, z\rangle \Rightarrow P) \triangleright \langle a_1, a_2\rangle = [a_1/x, a_2/z]P$$

$$(\ell(x) \Rightarrow P_\ell)_{\ell\in L} \triangleright \langle k, y\rangle = [y/x]P_k \qquad (k \in L)$$

Each of these rules states how a specific subset of the cells and threads within the configuration can step forward, transitioning from their previous state to a new one. Many of these rules are the same, or very similar to the corresponding dynamic rules in SNAX without reuse. Specifically, the rules for writing general storables, calling recursive functions, and splitting a process into two (via the cut rule) remain the same. The snip rule also remains the same as before, with the $\text{dest}(P)$ judgement being defined in the same manner. A few cases need to be added to account for our four new constructs, however:

Additional Cases of dest

$$\text{dest}(\text{free } a; P) = \text{dest}(P)$$

$$\text{dest}(\text{dispose } a; P) = \text{dest}(P)$$

$$\text{dest}(\text{deallocate } a; P) = \text{dest}(P)$$

$$\text{dest}(\text{reuse } a(x \leftarrow P; Q)) = \text{dest}(Q)$$

Lastly, the rule for copying values from one cell to another also remains the same – including the constraint that the values being copied must be a pointer, function, or lazy product. As stated previously, copy operations for other values must be η-expanded into series of reads and writes [1].

The first modified set of rules are those pertaining to reading and writing pointers, $\langle x\rangle$. In the write case, we must not only write the pointer $\langle x\rangle$ to the corresponding cell – we must also change the state of the cell x to reflect the fact that it is now being referenced by a pointer. The type system uses this state to prevent x from being used again. Likewise, when reading, we must ensure that the previously-referenced cell x is changed back to an available state, so that the process P that reads the pointer can access x's contents. This is done in a manner consistent with the definition of $S \triangleright T$ for pointers that we had in SNAX.

The rules for reading memory cells are also modified to accommodate the fact that cells stay in the configuration after they are read, rather than being entirely removed. Reading the data causes the necessary changes to existing threads to be made from $S \triangleright T$, and also marks the cell as being read. The configuration thus accumulates read cells, each in the form read $a\ S$, which stay there until it is time for them to be freed. Additionally, the storable S is kept in the cell, which both allows the cell to be typed later and allows the reading process to read addresses from this cell.

Finally, there is the matter of the new dynamics that we have added to support the new language features. The dynamic rule for reusing a cell is nearly identical to the dynamic rule for cut – the only change is that we consume the free cell, and all of its projections, that we are reusing in place of allocating an entirely new cell. Freeing a cell requires that the cell, and all of its projections, have been read. We slightly abuse notation by stating in the premise for this rule that read $\text{proj}(\alpha)$, which is used to express that all of the projections in $\text{proj}(\alpha)$ have also been marked as read. This ensures that no further reads to this cell will be attempted, allowing us to safely put it in the free state. Like reuse, deallocating a cell α also requires that it is free, however, this construct simply removes α from the configuration entirely and does not perform further operations on it. Disposing a cell a is similar to freeing a, in that it requires both a and $\mathrm{proj}(a)$ to be read. The difference is that their statuses are set to referenced, rather than read. This prevents them from being used further – no pointers exist to a, and therefore it cannot be moved out of the read state by using the pointer-reading dynamic rule.

Freeing, reuse, and deallocation all operates on the roots of cells α. This requires that the address being freed, reused, or deallocated cannot contain any projections, ensuring that we do not attempt to free or reuse only part of a memory cell. It is possible that we might later be able to split up a free cell into multiple parts, each of which could be individually reused. For example, a cell of type $A_1 \otimes A_2$ could be split into two cells, one of type A_1 and one of type A_2. They could then each be reused or deallocated separately. However, the language does not currently support this. Instead, we mandate that every cell be reused to store a value of its original type. This allows us to easily guarantee that it is the correct size for its data.

5.4 Progress

We define a configuration to be final if it has no threads and is entirely made up of cells. Given this definition, we are able to prove a sequential form of the progress theorem for SNAX with reuse:

Progress: If $\cdot \vDash C :: \Gamma$, then either C is final or there exists some C' such that $C \longmapsto C'$.

More specifically, we show that the leftmost thread in C can always step forward. In a similar manner to how progress was shown for the original SNAX language, we show that this holds by right-to-left induction on the structure of the derivation $\cdot \vDash C :: \Gamma$, which is given to us. We begin by reproducing a lemma from the original proof, which remains mostly unchanged when memory reuse is present.

Lemma

If $\Gamma \vdash P :: (a : A); \Gamma'$, then $\mathrm{dest}(P) = A$.

The definition of $\mathrm{dest}(P)$ has not changed from the original SNAX language, and our statics remain mostly the same. As such, we proceed in a similar manner, using induction on the derivation of the typing judgement $\Gamma \vdash P :: (a : A); \Gamma$. The most interesting case once again comes from reading a sum:

$$\frac{\forall \ell \in L : \Gamma, a \cdot \overline{\ell} : A_\ell[\text{available}], a : \oplus\{\ell : A_\ell\}_{\ell \in L}[\text{read}] \vdash P_\ell :: (c : C); \Gamma'}{\Gamma, a : \oplus\{\ell : A_\ell\}_{\ell \in L}[\text{available}] \vdash \text{read } a \ (\ell\langle\_\rangle \Rightarrow P_\ell)_{\ell \in L} :: (c : C); \Gamma'} \ \oplus\text{L}$$

The inductive hypothesis still applies, giving us that $\mathrm{dest}(P_\ell) = \{c\}$ for all choices of $\ell \in L$. Then $\bigcup_{\ell \in L} \mathrm{dest}(P_\ell) = \{c\}$. The proof of this lemma for the reuse rule is identical to the one for the cut rule. The free construct is handled as follows:

$$\frac{\Gamma, \alpha : A[\text{free}] \vdash P :: (c : C); \Gamma'}{\Gamma, \alpha : A[\text{read}], \mathrm{proj}(\alpha)[\text{read}] \vdash \text{free } \alpha; P :: (c : C); \Gamma'} \ \text{FREE}$$

By the inductive hypothesis, we know that $\mathrm{dest}(P) = \{c\}$. The provided definition for $\mathrm{dest}(\text{free } \alpha; P)$ then states that the destination is $\{c\}$, as desired. The cases for dispose $a; P$ and deallocate $\alpha; P$ are identical. Thus, the lemma holds, even when the language is extended with memory reuse.

With this lemma, we are now ready to show that the progress theorem holds. We proceed by induction on the given typing judgement, which states that $\cdot \vDash C :: \Gamma$. We then show that any well-typed configuration C either is final or can step forward to some C'. This requires us to strengthen the inductive hypothesis with an additional clause, which states that every cell a in a configuration C has the correct status in Γ if $\cdot \vDash C :: \Gamma$ – and that this property is preserved when stepping forward. We begin by reproducing the snip case from the original paper on SNAX, to show that the proof of progress still holds in the same form:

$$\dfrac{\cdot \vDash C : \Gamma_1 \qquad \dfrac{\dfrac{\Gamma_1 \vdash P :: (a : A); \Gamma_2 \quad \Gamma_2, a : A[\text{available}] \vdash Q :: (c : C); \Gamma}{\Gamma_1 \vdash (P; Q) :: (c : C); \Gamma}\ \textsc{snip}^+}{\Gamma_1 \vDash \text{thread } (c, (P; Q)), \text{cell } c\ \square :: \Gamma, c : C[\text{available}]}\ \textsc{thread}}{\cdot \vDash C, \text{thread } (c, (P; Q)), \text{cell } c\ \square :: \Gamma, c : C[\text{available}]}\ \textsc{join}$$

By the inductive hypothesis, either C is final or it can step forward to some new configuration C'. In the former case, we note that $\text{dest}(P)$ is $\{a\}$, using the earlier lemma. Our dynamic rule for snips still applies in this case:

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

On the other hand, if the remainder of the configuration can step forward $C \longmapsto C'$, then we allow it to do so:

$$C, \text{thread } (c, (P; Q)), \text{cell } c\ \square \longmapsto C', \text{thread } (c, (P; Q)), \text{cell } c\ \square$$

It follows that any configuration whose typing judgement is derived in this manner can step forward. Progress can be proved for other constructs in the original SNAX language by translating their progress proofs to work with the new typing judgements from SNAX with reuse, in a similar manner to the snip example shown here. We now show that progress holds for the reuse, free, and deallocate rules, which are the three newly added cases that cannot be directly converted from a SNAX equivalent. To do this, we need another lemma to ensure that the statuses in Γ are accurate:

Lemma (Status Accuracy)

Let C be a configuration made up entirely of cells, with no threads – that is, C is final. If $\cdot \vDash C :: \Gamma$ and $a \in \Gamma$ with some status s, then a exists as a cell in C with the status s.

We proceed by right-to-left induction on the configuration C. C cannot be empty, as it contains at least one cell. Our base cases therefore come when C contains one cell. In this case, C is typed with one of the single-cell rules. We can proceed by inversion on the configuration typing rule corresponding to status s to conclude that C is made up of a single cell with the desired status.

In the inductive case, we can type the configuration using the Join rule. In particular, we can select C_2 as the rightmost cell, while C_1 is the remaining cells in the configuration. We now consider the address of the cell in C_2. There are two cases – either this cell has address a or it does not. In either case, we consider the judgement $\Gamma' \vDash C_2 :: \Gamma$, which is derived by one of the single-cell typing rules.

If the cell in C_2 is a, then it follows that the single-cell typing rule added a to Γ with the status s. Thus, a must have status s in the configuration, since no other status could lead to it being added to the context with this status.

If the cell in C_2 has some address $b \neq a$, then we know that the single-cell typing rule used for C_2 added $b \neq a$ to the context. It follows that $a \in \Gamma'$ with status s. Additionally, we can observe that $\cdot \vDash C_1 :: \Gamma$ from the other premise of the Join rule. We can therefore apply the inductive hypothesis to conclude that a is in C_1 (and therefore in C) with the status s. $\square$

Now that we have our lemma, we can begin with the reuse:

$$\dfrac{\cdot \vDash C : \Gamma_1, a : A[\text{free}] \qquad \dfrac{\dfrac{\Gamma_1 \vdash P :: (x : A); \Gamma_2 \quad \Gamma_2, x : A[\text{available}] \vdash Q :: (c : C); \Gamma}{\Gamma_1, a : A[\text{free}] \vdash \text{reuse } a(x \leftarrow P; Q) :: (c : C); \Gamma}\ \textsc{reuse}}{\Gamma_1, a : A[\text{free}] \vDash \text{thread } (c, \text{reuse } a(x \leftarrow P; Q)), \text{cell } c\ \square :: \Gamma, c : C[\text{available}]}\ \textsc{thread}}{\cdot \vDash C, \text{thread } (c, \text{reuse } a(x \leftarrow P; Q)), \text{cell } c\ \square :: \Gamma, c : C[\text{available}]}\ \textsc{join}$$

Once again, we note that either C is final or $C \longmapsto C'$ for some other configuration C'. In the first case, it must hold that C contains the cell free $a\ S$ for some storable S of type A, and it must also contain the cells $\overline{\text{free proj}(a)}$ for any applicable projections. This is because C is final, meaning our reusing thread is the leftmost process in the configuration, and we can therefore apply our lemma on a for all of these cells. Therefore, we can rewrite C as the configuration C'', free $a\ S$, $\overline{\text{free proj}(a)}$. It then follows that we can step forward as follows:

$$C'', \text{free } a\ S, \overline{\text{free proj}(a)}, \text{thread } (c, \text{reuse } a(x \leftarrow P; Q)), \text{cell } c\ \square \longmapsto C'', \text{thread } (a, P), \text{cell } a\ \square, \text{thread } (c, Q), \text{cell } c\ \square$$

In the second case, we allow C to step forward to C', thus making the configuration as a whole step forward as before.

The case for deallocating a freed cell is very similar, albeit somewhat simpler:

$$\dfrac{\cdot \vDash C : \Gamma_1, a : A[\text{free}] \qquad \dfrac{\dfrac{\Gamma_1 \vdash P :: (c : C); \Gamma}{\Gamma_1, a : A[\text{free}] \vdash \text{deallocate } a; P :: (c : C); \Gamma}\ \textsc{dealloc}}{\Gamma_1, a : A[\text{free}] \vDash \text{thread } (c, \text{deallocate } a; P), \text{cell } c\ \square :: \Gamma, c : C[\text{available}]}\ \textsc{thread}}{\cdot \vDash C, \text{thread } (c, \text{deallocate } a; P), \text{cell } c\ \square :: \Gamma, c : C[\text{available}]}\ \textsc{join}$$

Once again, we note that C is either final or $C \longmapsto C'$. If C is final, then it contains free $a\ S$ and $\overline{\text{free proj}(a)}$ by the lemma. This enables the dynamic rule for deallocation to take effect. If $C \longmapsto C'$, then we can step the entire configuration forward by stepping C forward.

The final cases are for freeing or disposing a cell that has previously been read. Progress in these cases is proved in a very similar manner to how we showed progress for the deallocate construct. The sole change is that instead of looking for the cell free $a\ S$ in C, we instead look for read $a\ S$ and its read projections so that we can transition them to either free $a\ S$ or referenced $a\ S$. Indeed, these cells must exist in C using the same lemma as before. It follows that the progress theorem holds for both the previously existing forms for processes and the newly added ones. Thus, progress holds for SNAX with reuse as a whole.

5.5 Preservation

Preservation: If $\Gamma \vDash C :: \Gamma'$ and $C \longmapsto C'$, then $\Gamma \vDash C' :: \Gamma''$, where $\Gamma' \subseteq \Gamma''$.

Most cases of preservation actually show a stricter claim, where $\Gamma' = \Gamma''$. We previously allowed $\Gamma' \subseteq \Gamma''$ because cells introduced by the step forward persisted until the end of the program. This comes from the fact that the original SNAX language was not linear. Now, this requirement comes about because certain steps, such as functions and lazy products, cause additional garbage to be added to the configuration that will not be freed and deallocated.

To show this, we begin by observing that our contexts are no longer separated into configuration contexts Φ and process contexts Γ – rather, the general form of contexts Γ is used for typing all of these things. The lemmas B.1 through B.3 still hold in a slightly modified form. Lemma B.4 is irrelevant, since it describes a form of weakening which is not used in linear SNAX. The proofs are identical to those in the SNAX paper, replacing Φ with Γ where appropriate [1].

Lemma

Lemma 1: If $a > b$ and $a > c$, then either $b \geq c$ or $c \geq b$.

Lemma 2: If $\Gamma, \underline{a : A}[\text{available}] \vdash P :: (c : C); \Gamma'$, then $a > c$.

Lemma 3: If $\Gamma, \underline{a : A}[\text{available}], \underline{b : B}[\text{available}] \vdash P :: (c : C); \Gamma'$, then $a \ngeq b$ and $b \ngeq a$.

The preservation theorem can be shown by induction on the dynamic rule used to step $C \longmapsto C'$. These lemmas are relevant to show that the preservation theorem holds in some of the cases presented in the original SNAX paper. The dynamics for these cases remain unmodified after our changes, and their proofs of preservation therefore remain identical.

The interesting cases come from the new dynamics that we have modified and added to the language. The first such dynamic is the read rule, which has now been changed to mark a cell as read after reading it, rather than removing it entirely from the configuration. Reads can appear in several forms, each corresponding to the type of the value that is being read. Two cases of preservation for the read rule appear in Appendix A.

We now show preservation for the free construct. To do this, we need to show that the initial configuration can be typed as follows:

$$\Gamma_0 \vDash \text{read } \alpha\ S, \text{read proj}(\alpha)\ , \text{thread } (a, \text{free } \alpha; P), \text{cell } a \,\square :: \Gamma_1, a : A[\text{available}]$$

The types of the cells in this configuration can be found using the following derivation:

$$\mathcal{A} = \dfrac{\dfrac{\overset{\mathcal{D}}{\Gamma_0^a \vdash \text{write } (\alpha, S) :: (\alpha : \tau); \Gamma_0'}}{\Gamma_0 \vDash \text{read } \alpha, S \ :: \Gamma_0, \alpha : \tau[\text{read}]}\ \textsc{Read} \qquad \dfrac{\dots}{\Gamma_0, \alpha : \tau[\text{read}] \vDash \text{read proj}(\alpha) \ :: \Gamma_0, \alpha : \tau[\text{read}], \text{proj}(\alpha)[\text{read}]}\ \textsc{Join}}{\Gamma_0 \vDash \text{read } \alpha\ S, \text{read proj}(\alpha) \ :: \Gamma_0, \alpha : \tau[\text{read}], \text{proj}(\alpha)[\text{read}]}\ \textsc{Join}$$

Here, the ... represents the things that are needed to conclude that the projections of α are all in a read state. This can be done by applying the Join rule several times to type each of the individual read cells represented by the read proj(α) in the configuration. A similar technique can be used to find the type of α and its projections when they are all freed:

$$\mathcal{A}' = \dfrac{\dfrac{\overset{\mathcal{D}}{\Gamma_0^a \vdash \text{write } (\alpha, S) :: (\alpha : \tau); \Gamma_0'}}{\Gamma_0 \vDash \text{free } \alpha, S \ :: \Gamma_0, \alpha : \tau[\text{free}]}\ \textsc{Free} \qquad \dfrac{\dots}{\Gamma_0, \alpha : \tau[\text{read}] \vDash \text{free proj}(\alpha) \ :: \Gamma_0, \alpha : \tau[\text{read}], \text{proj}(\alpha)[\text{free}]}\ \textsc{Join}}{\Gamma_0 \vDash \text{free } \alpha\ S, \text{free proj}(\alpha) \ :: \Gamma_0, \alpha : \tau[\text{free}], \text{proj}(\alpha)[\text{free}]}\ \textsc{Join}$$

Then we get the following:

$$\dfrac{\mathcal{A} \qquad \dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma_0, \alpha : \tau[\text{free}], \text{proj}(\alpha)[\text{free}] \vdash P : (a : A); \Gamma_1}}{\Gamma_0, \alpha : \tau[\text{read}], \text{proj}(\alpha)[\text{read}] \vdash \text{free } \alpha; P :: (a : A); \Gamma_1}\ \textsc{Free}}{\Gamma_0, \alpha : \tau[\text{read}], \text{proj}(\alpha)[\text{read}] \vDash \text{thread } (a, \text{free } \alpha; P), \text{cell } a \,\square :: \Gamma_1, a : A[\text{available}]}\ \textsc{Thread}}{\Gamma_0 \vDash \text{read } (\alpha, S), \text{read proj}(\alpha), \text{thread } (a, \text{free } \alpha; P), \text{cell } a \,\square :: \Gamma_1, a : A[\text{available}]}\ \textsc{Join}$$

$$\longmapsto$$

$$\dfrac{\mathcal{A}' \qquad \dfrac{\overset{\mathcal{E}}{\Gamma_0, \alpha : \tau[\text{free}], \text{proj}(\alpha)[\text{free}] \vdash P :: (a : A); \Gamma_1}}{\Gamma_0, \alpha : \tau[\text{free}], \text{proj}(\alpha)[\text{free}] \vDash \text{thread } (a, P), \text{cell } a \,\square :: \Gamma_1, a : A[\text{available}]}\ \textsc{Thread}}{\Gamma_0 \vDash \text{free } (\alpha, S), \text{free proj}(\alpha), \text{thread } (a, P), \text{cell } a \,\square :: \Gamma_1, a : A[\text{available}]}\ \textsc{Join}$$

This proof of preservation also works for the dispose construct, if we replace the instances of “free” with “referenced” in the proof trees and allow it to be applied to a general address a instead of a root α. After a cell has been freed, we can reuse it to allow two processes to communicate. This step also obeys the preservation theorem, which uses the same derivation $\mathcal{A}'$ that we used in the previous proof to type a free cell and its free projections. We use [a] as an abbreviation for [available] in this proof, where necessary.

$$\dfrac{\mathcal{A}' \qquad \dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma_0 \vdash P :: (\alpha : A); \Gamma_1} \quad \overset{\mathcal{F}}{\Gamma_1, \alpha : A[\text{available}] \vdash Q :: (b : B); \Gamma_2}}{\Gamma_0, \alpha : \tau[\text{free}], \text{proj}(\alpha)[\text{free}] \vdash \text{reuse } \alpha \leftarrow P; Q :: (b : B); \Gamma_2}\ \textsc{Reuse}}{\Gamma_0, \alpha : \tau[\text{free}], \text{proj}(\alpha)[\text{free}] \vDash \text{thread } (b, \text{reuse } \alpha \leftarrow P; Q), \text{cell } b \ \square :: \Gamma_2, b : B[\text{available}]}\ \textsc{Thread}}{\Gamma_0 \vDash \text{free } \alpha \ S, \text{free proj}(\alpha), \text{thread } (b, \text{reuse } \alpha \leftarrow P; Q), \text{cell } b \ \square :: \Gamma_2, b : B[\text{available}]}\ \textsc{Join}$$

$$\longmapsto$$

$$\dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma_0 \vdash P :: (\alpha : A); \Gamma_1}}{\Gamma_0 \vDash \text{thread } (\alpha, P), \text{cell } \alpha \ \square :: \Gamma_1, \alpha : \tau[\text{a}]}\ \textsc{Thread} \qquad \dfrac{\overset{\mathcal{F}}{\Gamma_1, \alpha : \tau[\text{a}] \vdash Q :: (b : B); \Gamma_2}}{\Gamma_1, \alpha : \tau[\text{a}] \vDash \text{thread } (b, Q), \text{cell } b \ \square :: \Gamma_2, b : B[\text{a}]}\ \textsc{Thread}}{\Gamma_0 \vDash \text{thread } (\alpha, P), \text{cell } \alpha \ \square, \text{thread } (b, Q), \text{cell } b \ \square :: \Gamma_2, b : B[\text{a}]}\ \textsc{Join}$$

Finally, there is the matter of deallocating a cell that has already been freed:

$$\dfrac{\mathcal{A}' \qquad \dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma_0 \vdash P :: (a : A); \Gamma_1}}{\Gamma_0, \alpha : \tau[\text{free}], \text{proj}(\alpha)[\text{free}] \vdash \ \text{dealloc } \alpha; P :: (a : A); \Gamma_1}\ \textsc{Dealloc}}{\Gamma_0, \alpha : \tau[\text{free}], \text{proj}(\alpha)[\text{free}] \vDash \text{thread } (a, \ \text{dealloc } \alpha; P), \text{cell } a \ \square :: \Gamma_1, a : A[\text{available}]}\ \textsc{Thread}}{\Gamma_0 \vDash \text{free } \alpha \ S, \text{free proj}(\alpha), \text{thread } (a, \ \text{dealloc } \alpha; P), \text{cell } a \ \square :: \Gamma_1, a : A[\text{available}]}\ \textsc{Join}$$

$$\longmapsto$$

$$\dfrac{\overset{\mathcal{E}}{\Gamma_0 \vdash P :: (a : A); \Gamma_1}}{\Gamma_0 \vDash \text{thread } (a, P), \text{cell } a \ \square :: \Gamma_1, a : A[\text{available}]}\ \textsc{Thread}$$

It follows that both progress and preservation hold for our language when it is executed sequentially – that is, SNAX with reuse still remains type safe.

6 Compilation

The grammar of SNAX is fairly low-level. Programs directly written in the SNAX language are mostly made up of read and write operations to memory cells, rather than higher level computations. This allows a fine-grained level of control over what the program is doing in memory. However, it also makes SNAX programs difficult to write – much like assembly, each individual move operation must be controlled by the user. Thus, we do not anticipate that many programs will be written directly into SNAX. Instead, users will likely write code in some front-end language, which is a higher-level language which is more user-friendly.

In our examples, we use a front-end language, which is based off of natural deduction as described by Gentzen in his paper [4]. This language is being developed by Jang et al. in their unpublished work on adjoint natural deduction [6]. It operates in a similar manner to existing functional languages, such as Standard ML and OCaml. Users are able to define their own custom types and datatypes, and can assign them names. They can also define functions, which have access to features such as pattern matching and recursion. This natural deduction framework is only one possible front-end for our language – other languages can also be made to compile down to SNAX.

We chose this particular front-end because it is fairly versatile in that it supports several *modes* for each function.

The two defining traits of a particular mode are whether it supports weakening and whether it supports contraction. The first criterion, weakening, specifies whether we are allowed to remove a variable from the context. In practice, this corresponds to dropping a variable without using it in any way. The second criterion, contraction, specifies whether we are allowed to make multiple “copies” of a variable within the context. In a real program, this would correspond to using a variable multiple times – if we use a variable twice, this is equivalent to creating two copies of that variable and consuming each one.

Linear logic, which has been used throughout our description of SNAX with reuse, supports neither weakening nor contraction. The front-end supports linear logic by allowing the user to specify that both weakening and contraction are disallowed for a particular function’s inputs and outputs. Likewise, the user can specify that weakening and contraction are allowed to get the standard form of constructive logic, or specify that only weakening is allowed to get affine logic. Moreover, the user can create mode variables to represent the modes of particular things. These variables are filled in by the type system, and can result in functions which are polymorphic across modes. The result is an *adjoint* system, which combines multiple logical modes into a single language. Our memory reuse optimizations are applied to the linear and affine types of this adjoint code, which both do not support contraction. This means that we can immediately free any cell that is read in these modes, allowing us to reuse their memory.

The compiler and interpreter for this language are both written in Standard ML, which is compiled with MLton to produce the runtime. The first step in compiling functions which are polymorphic across modes is to monomorphize them. This means that a function which is used at two modes M and N gets elaborated into two functions – one at mode M and one at mode N. Each of the two functions is used in the places where its mode matches the mode in which it is being called. The two functions perform identical computations, but are treated as separate functions by the compiler. After the monomorphization of functions occurs, the compiler then performs closure conversion and hoisting. Functions that are defined with reference to variables in their environments – that is, closures – are adapted to no longer reference their environments. They are then moved to the top level to make code generation easier.

It is at this point that the compilation branches to allow compilation to various middle-end languages. Currently, our compiler allows us to use either SAX or SNAX as the next intermediate language. This means that the memory optimizations provided by SNAX can be toggled on and off as necessary. Moreover, the SAX and SNAX code produced can be written with or without reuse, which allows us to have a further degree of control over the compilation process. At this point, the program has been translated to a series of SAX or SNAX functions, each of which is made up of read, write, cut, and identity operations. Some optimizations are added here – one notable optimization removes unnecessary move operations. From this point, the only thing left to do is to translate this code to C. The C code is then handled by GCC to create the final executable.

7 Memory Reuse Example

To illustrate our compilation process, as well as show where reuses can be inserted into a SNAX program, we consider the process of repeatedly decrementing a binary number. This code was generated by our binary counter benchmark, which measures memory efficiency by starting with the binary number for 2^{16} and repeatedly decrements it until it reaches 0. We begin by defining our type of binary numbers. In our natural deduction syntax, the code to do this looks like:

```
type std[m] = +{'b0 : <pos[m]>, 'b1 : <std[m]>, 'e : 1}
type pos[m] = +{'b0 : <pos[m]>, 'b1 : <std[m]>        }
```

Here, we define `std` to be the type of standard binary numbers that we will use. We define it as a recursive sum with three constructors. `'e` is the constructor for an empty binary number, which is equal to 0. `'b0` and `'b1` are both constructors, which represent that the least significant bit of the binary number is 0 or 1 respectively. They each hold a pointer to the remaining portion of the binary number as well, which is represented by the `<pos[m]>` – note the angle brackets to denote that this is a pointer to a `pos`, and not the `pos` value itself. The `pos` type is used to represent a positive (i.e. non-zero) binary number. A number with a least significant bit of 0 is required to have the remainder of the number be positive, to prevent leading zeroes. The `[m]` attached to each `bin` is a variable representing the mode. In SNAX, this compiles to the following type declaration:

```
type std[m] = +{'b0 : down[m] pos[m], 'b1 : down[m] std[m], 'e : 1}
type pos[m] = +{'b0 : down[m] pos[m], 'b1 : down[m] std[m]}
```

This type is defined similarly, with the only change being that the type of a pointer to a standard binary number is written as `down[m] std[m]` instead of as `<std[m]>`. In memory, a cell of type `std` would then be made up of two words – the first is the label for the sum, and the second is the pointer to the rest of the number. In the `'e` case, the second word would remain unused.

Now that we have established the type of a binary number, we are ready to write code using these binary numbers. The first function we will present decrements a binary number with a simple algorithm. If the number is already empty, then we return the empty number again – our binary number type does not support negative numbers, so there is no way to decrement 0. Otherwise, we look at the least significant bit of the number. If it is 0, then we change it to 1, and decrement the rest of the number. If it is 1, then we have to check what the rest of the number is. If there are no further digits, then decrementing results in an empty number. However, if there are further digits, we simply set the least significant bit to 0 and return.

```
defn dec x = match x with
    | 'b0 <x> => 'b1 <dec x>
    | 'b1 <'e()> => 'e()
    | 'b1 <'b0 <x>> => 'b0 <'b0 <x>>
    | 'b1 <'b1 <x>> => 'b0 <'b1 <x>>
    | 'e() => 'e()
```

The first line shows the syntax for declaring a function, which takes a single variable `x` as its input. We then match `x` against one of five patterns, which each correspond to one of the possible labels for a value of type `bin`. In each of these cases, we change the number according to our algorithm, and then return the resulting binary number. This includes a recursive call in the `'b0` case, where we must decrement the remaining bits of the number. Note that we have to explicitly write out the `'b1 <'b0 <x>>` and `'b1 <'b1 <x>>` cases, rather than matching the inner pointer with a variable. This is because we are not allowed to match one branch (the `'e` branch) with an explicit constructor, and then use a variable in the other cases.

This code compiles to the following SNAX code, after we monomorphize it, perform closure conversion, and translate match statements to read statements:

```
 1  proc dec/0 ($0:std[L]) (x:std[L]) =
 2      read x =>
 3      | 'b0(_) =>
 4          read x.b0 <$1> =>
 5          cut $2 = x : std[L]  % reuse
 6              call dec/0 $2 $1
 7          write $0.b1 <$2>
 8          write $0 'b1(_)
 9      | 'b1(_) =>
10          read x.b1 <$4> =>
11          read $4 =>
12          | 'b0(_) =>
13              read $4.b0 <$5> =>
14              cut $6 = $4 : pos[L]  % reuse
15                  write $6.b0 <$5>
16                  write $6 'b0(_)
17              write $0.b0 <$6>
18              write $0 'b0(_)
19          | 'b1(_) =>
20              read $4.b1 <$8> =>
21              cut $9 = $4 : pos[L]  % reuse
22                  write $9.b1 <$8>
23                  write $9 'b1(_)
24              write $0.b0 <$9>
25              write $0 'b0(_)
26          | 'e(_) =>
27              read $4.e () =>
28              write $0.e ()
29              write $0 'e(_)
30      | 'e(_) =>
31          read x.e () =>
32          write $0.e ()
33          write $0 'e(_)
```

The declaration of functions in SNAX is somewhat different from that of our natural deduction syntax. The process that we define, named `dec/0`, now takes two arguments. This comes about because SNAX functions are written in *destination-passing style*, where the first argument to the function is the location to which it should write its output. The second argument, `x`, is the actual input to the function. The `[L]` markings on both arguments denote that the function has been monomorphized to take a linear argument and write a linear output.

The program begins by casing on the label in `x`. In each case, it reads the projections of `x` as well, before proceeding according to the specification. Of particular note are the `cut` operations on lines 5, 14, and 21. These operations use a special form of cut to denote the fact that we are reusing memory. On line 5, we reuse `x` to store the result of the recursive call to `dec`, while on line 14, the memory cell `$4` is reused to hold the upper bits of the number.

The concrete syntax here is slightly different from the abstract syntax that we have previously used for SNAX programs. While we still read the projections of a cell all at once, we do not have an explicit `free` statement at the end of the series of read operations. The purpose of this statement in the abstract syntax was to flag the cell as being available for reuse. However, this availability is tracked internally by the compiler, and therefore does not need to be emitted to the SNAX code. Furthermore, `cut` is used to represent both cut operations and reuse operations. If an already-read linear cell is available when a cut of the same type is occurring, then it is automatically reused.

This compiled code has also taken advantage of another opportunity for memory reuse on lines 14 and 21. The cell `$4` has type `std`, however, we are reusing it to store a value of type `pos`. While this is not explicitly permitted by our type theory, the compiler can conclude that this reuse is safe by noting that `pos` is actually a subtype of `std`! Since this is the case, we are able to take advantage of the similar memory layouts of the two types and reuse a memory cell from one type to store a value from the other.

Now that we have compiled the decrement function, we can write a recursive function that repeatedly decrements a number until it has reached 0. In our front-end, the code is:

```
defn count_down x = match x with
    | 'e() => ()
    | 'b0 <x> => count_down (dec ('b0 <x>))
    | 'b1 <x> => count_down (dec ('b1 <x>))
```

This compiles to the following SNAX code:

```
proc count_down/0 ($0:1) (x:std[L]) =
    read x =>
    | 'b0(_) =>
        read x.b0 <$1> =>
        cut $2 = x : std[L]  % reuse
            cut $3:std[L]
                write $3.b0 <$1>
                write $3 'b0(_)
            call dec/0 $2 $3
        call count_down/0 $0 $2
    | 'b1(_) =>
        read x.b1 <$5> =>
        cut $6 = x : std[L]  % reuse
            cut $7:std[L]
                write $7.b1 <$5>
                write $7 'b1(_)
            call dec/0 $6 $7
        call count_down/0 $0 $6
```

Benchmark Name	Description
count	Repeatedly subtracts 1 from the binary number 2^{16} until it reaches 0.
bst	Performs operations on a binary search tree such as insertion and deletion.
dequeue	Inserts elements into a double-ended queue, which is represented by two lists.
escardo	Performs arithmetic on real numbers using Escardó's representation [3].
fibsum	Performs arithmetic on integers represented using Fibonacci encodings.
list	Tests list creation and deletion, as well as operations like mapping and folding.
perms	Enumerates the permutations of the input list.
trie	Insertions and deletions for a set of binary numbers that is represented using a trie.

Figure 1: The list of benchmarks and what computations each one performs.

```
19      | 'e(_) =>
20          read x.e () =>
21          write $0 ()
```

Once again, SNAX is able to reuse the read cell `x` to hold another value of type `std`. In the first branch, `$2` is used as the argument to the recursive call. It is reused on line 5 after `x` is completely read, and gets used on lines 9 and 10 where it receives the output from `dec` and passes this output into `count_down`. As such, the SNAX runtime identifies this as an opportunity for reuse, and uses `x` here rather than allocating a new cell. The same holds for `$6` on line 13, which is reused in a similar manner.

8 Impacts of Memory Reuse

To measure the impacts of memory reuse, the compiler that we previously described was instrumented. The programs it generates, in both SAX and SNAX, track the number of allocations they make and what the total size of those allocations are, in machine words. When reuse is enabled, these programs also track how many words are reused. To test the effectiveness of our changes, we then ran a set of benchmarks, which are described in Figure 1.

Each of these benchmarks was run in both SAX and SNAX with reuse enabled. In both languages, we tracked the number of words allocated, as well as the number of words that were reused. Summing these gives the number of allocations that would have been used if the program was running without reuse, since the reuses in our system replace allocations that otherwise would have required a fresh cell to be created.

The tables in Figure 2 and 3 summarize our results. In Figure 2, the first three columns for each benchmark state the number of allocations and reuses in SAX with reuse (which we abbreviate to rSAX), as well as the total number of allocations that would have been necessary when running the benchmark in SAX without reuse. The fourth column states the percentage of SAX allocations that still occur in the rSAX program. Figure 3 states the same data for rSNAX and SNAX.

Benchmark	rSAX allocations	rSAX reuses	SAX allocations	% of original
count	196724	589878	786602	25.01%
bst	261988	491732	753720	34.76%
dequeue	52397	51658	104055	50.36%
escardo	1561258	397699	1958957	79.70%
list	6743	394752	401495	1.68%
perms	230630	140444	371074	62.15%
trie	785946	338537	1124483	69.89%

Figure 2: The results from the SAX versions of the benchmarks. The total sizes of allocations and reuses given here are measured in words. The last column measures the percentage of the original program's allocations that are still used in the program with memory reuse – a lower percentage is better.

Benchmark	rSNAX allocations	rSNAX reuses	SNAX allocations	% of original (lower is better)
count	131152	393266	524418	25.01%
bst	278142	135152	413294	67.30%
dequeue	52410	10746	63156	82.98%
escardo	1029802	38653	1068455	96.38%
list	5960	197376	203336	2.93%
perms	149421	71970	221391	67.49%
trie	550822	241856	792678	69.49%

Figure 3: The same metrics from the SNAX versions of the benchmarks.

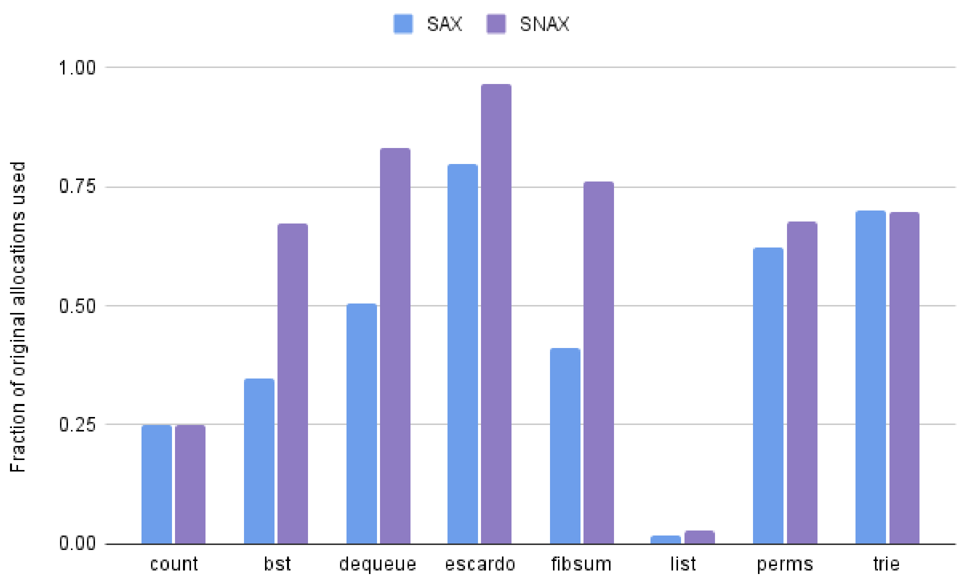

Figure 4: The fraction of the original number of allocations that are needed after enabling reuse in both SAX and SNAX.

As expected, the data indicates that the overall number of words allocated decreases when using SNAX instead of SAX. This can be attributed to the lower number of pointers that are allocated when the program is run in SNAX instead. In all of the benchmarks, SNAX allocated fewer words than SAX. Moreover, rSAX allocated significantly fewer words than SAX on most of the benchmarks. The same relationship holds between rSNAX and SNAX, as expected.

Figure 4 provides a more interesting comparison. By dividing the number of words allocated by rSAX on a particular benchmark by the number of words allocated by SAX on the same program, we are able to determine the fraction of allocations that are still necessary. These fractions form the blue series in the column chart, while the purple series represents the coresponding metrics for SNAX. On almost every benchmark, we can see that enabling reuse for SAX is more impactful than enabling reuse for SNAX. This is likely because the original SAX language inefficiently uses pointers, leading to more opportunities for reuse arising.

We can also see that escardo is a particularly bad test case for memory reuse in SNAX, as it offers nearly no advantage when compared to running escardo without memory reuse. Strangely, enabling memory reuse for tries in SNAX is actually more impactful than enabling it in SAX. This could indicate an inefficiency in the SNAX layout for tries. However, this gap is only 0.4%, which could be considered negligible.

Figure 5 demonstrates the overall impacts of both the memory reuse and memory layout optimizations. For each benchmark, we used the number of allocations from SAX without reuse as our baseline. The number of allocations needed by each optimized language wa divided by the baseline value to find the fraction of allocations that it required – a lower fraction indicates that less memory was needed, and that it was more efficiently used.

Adding reuse to SNAX was effective in most cases. rSNAX generally outperformed both rSAX, which only had memory reuse, and SNAX, which only had the improved memory layout. The two exceptions were BST and fibsum, where rSAX used less memory than rSNAX in total. There are a few reasons why this might be the case. The first is because cells in SNAX of type A can only be reused to store values of type A, or any of its subtypes. In contrast, SAX cells can be

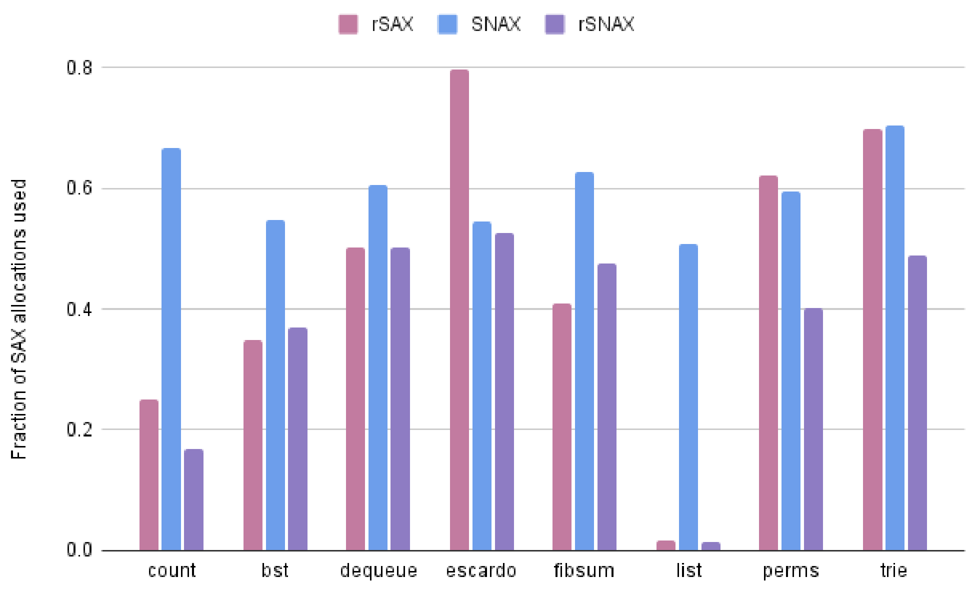

Figure 5: The relative impacts of only reuse optimizations (red), only layout optimizations (blue), and both types (purple).

reused to store values of any type, since they are all the same size. This means that SNAX might have to allocate new cells in places where SAX could reuse a cell that previously stored a different type. The second reason this might be happening relates to function calls. If function f frees a cell and then calls function g, the free cell is not available to g to reuse. If the output of g is part of some large SNAX memory cell, then it will have to rewrite the entire cell. In contrast, SAX programs can reuse most of the cell, and have g return only the part that it has changed, leading to better memory efficiency.

rSNAX performed particularly well in benchmarks that involved repeated operations, such as counting, generating permutations, and trie operations. In each of these three cases, it outperformed both rSAX and SNAX by a significant fraction. It also provided significant improvements on the list benchmark, much like rSAX did – lists allocate many nodes, creating many reuse opportunities.

Tests such as dequeue and escardo were less successful for rSNAX, but they still demonstrated that rSNAX was no worse than rSAX and SNAX. Part of the reason that all three languages still need at least 50% of the memory that SAX needs on these cases is that these test cases are *adjoint* – they contain both linear and nonlinear components. Memory reuse only affects the linear components, so it is less effective on adjoint tests.

From these benchmarks, we can conclude that SNAX with reuse generally performs better SAX with reuse, as it uses fewer allocations on most of the benchmarks. Likewise, it allocates significantly less memory than SNAX in most cases. As such, it appears that SNAX with reuse has successfully unified the performance gains from memory reuse with the improved memory layout that is provided by SNAX. The relative impact of adding memory reuse to SNAX is less than that of SAX, but this is because many cells that get reused in SAX store pointers which are optimized out in SNAX. Thus, memory reuse in SNAX is still impactful and improves the efficiency of programs by reducing their number of allocations.

9 Comparison with Existing Work

The linear process calculus that we have described represents one way to solve the problem of memory reuse in functional computation. This is a particularly interesting problem because it offers a way to reduce the number of allocations that functional programs perform. The nature of functional programs is that they will write many persistent, immutable values to memory, where they are stored until the end of the program whether they are needed or not. This causes them to perform many memory allocations, which can create a significant impact on performance in a few ways. The most obvious one comes from the fact that the memory allocator is being repeatedly called, adding to the runtime. Another problem that can arise comes when the program runs out of heap space. When this occurs, the program is forced to perform a process known as *garbage collection*, where values that are no longer needed are deallocated to free up memory for further allocations. Researchers seeking to optimize both memory reuse and garbage collection have therefore investigated systems similar to SNAX with reuse.

One similar system is Perceus, which was described by Reinking et al. in their 2021 paper [9]. Like SNAX, Perceus aims to use memory reuse to avoid garbage collection. However, the way that they go about doing this is different from our approach. Instead of making memory cells linear, Perceus keeps a reference count for every memory cell, which represents the number of pointers to each that cell which currently exist. If this count ever reaches 0 for a cell, then the cell is no longer needed and can be deallocated or reused. In this way, Perceus allows a more flexible form of programming since values can be referenced multiple times.

In contrast, our adjoint approach requires that each cells must be used at most once if they are also being reused. From a reference-counting perspective, this can be seen as forcing every cell to always have a reference count of either 0 or 1. This invariant further simplifies the language, for what might initially appear to be no useful gain. However, it also makes programming easier for the end user, as they do not have to infer when memory is deallocated and reused based on their high-level code. Furthermore, using a linear system introduces the potential to run processes in the configuration concurrently. With linearity, a process can immediately free a cell after it is used.

In contrast, in a reference-counting language, a process would have to decrement the reference count for a cell after it is done reading it. However, this has the potential to cause concurrency issues, such as the ABA problem, to arise. The concurrent program itself is safe, because values are only read from cells and are not mutated. However, any process that is a reader for a particular cell then becomes a writer for its reference count! Thus, if two concurrent reads target the same cell, then it is possible that the reference count could be inaccurate after the reads have completed. This could cause garbage to accumulate, or even lead to a cell being freed when it is still needed. It is possible to resolve this problem by ensuring that the changes to reference counts are atomic, however, linearity provides a simpler solution to this problem.

Another system that investigates similar problems to our work is Microsoft Research's FP2, which was was published in Lorenzen, Leijen, and Swierstra's 2023 technical report [7]. FP2 approaches the problem of memory reuse in a similar manner to our language – it is linear and aims to use said linearity to reduce the number of allocations overall. It frames its memory reuse in the context of in-place updates. The central idea behind these updates is that a common way to interact with a (possibly recursive) data structure of some type τ is to read it, perform some small modification, and then write a new value of type τ to the same location. Indeed, this pattern of in-place updates was one of the motivations for SNAX with reuse.

However, the similarities between FP2 and SNAX end at this point. The implementation of FP2 that the authors present is based on reference counting, much like Perceus. Functions that perform in-place updates on a data structure then perform the update in-place if and only if they hold the last remaining reference to that block of memory – otherwise, they allocate new values since the old data is still being used. Moreover, FP2's memory reuses are limited in comparison to SNAX, since it only reuses memory in the case where a data structure is being updated in-place. In contrast, SNAX can reuse a cell of type A for any other value of type A, regardless of whether the new value is related to the old one. While many instances of reuse fall into the category of in-place updates, there are certainly still other reuse opportunities that FP2 cannot take advantage of.

As such, SNAX represents another language that solves the problems of memory reuse in functional programming, with the intent of reducing memory allocation overall. This idea is closely connected to the problems of reducing the overhead of garbage collection and performing in-place updates on data structures. Linear SNAX with memory reuse addresses both of these problems in different ways from the existing solutions of Perceus and FP2. Linearity allows for easy garbage collection since a cell must be freed as soon as it is read, entirely eliminating the need for complex garbage collection routines. Additionally, the in-place update patterns that FP2 optimizes translate to clear opportunities for reuse in our language as well. SNAX with reuse therefore provides a similar, but not identical, approach to these problems.

10 Future Work

One potential research direction is to further optimize the generated code when rewriting certain memory cells. Suppose we are reading a cell a that belongs to some labeled sum type, where the potential labels are ℓ_1 and ℓ_2. If we reuse a along the ℓ_1 branch, then we know that a contains the label ℓ_1. However, the compiler will overwrite all of a when writing the new value into it after a reuse. This can lead to unnecessary writes, as we might overwrite the word containing a label with the same label that it held previously. Likewise, updates to data structures can leave many of the values in a memory cell unchanged, despite having to rewrite them. In the future, it might be useful to instrument the compiler with a way to detect what values are stored in a cell after it is freed, and use this information to minimize the amount of words that are written when the cell is eventually reused.

Another improvement that could be made is to allow a more diverse set of reuse operations. Currently, we stipulate that reuses must be done on the root of a cell $\alpha : A$, and that the new value must also be of type A. The most obvious way to add more reuse opportunities to the language is to allow α to be reused to other types. More specifically, we could find the set of types whose representations are the same size as type A, and allow any of these to be stored in α after reuse. However, this comes with a potential problem – this reuse must be kept strictly internal to the compiler, since the user should not need to be aware of the internal implementation of types.

A different way to increase the number of reuse opportunities is to allow a free cell to be split into multiple free cells, each of which can be reused individually. The primary impact of this would be on cells of type $A_1 \otimes A_2$. In the flat memory layout of SNAX, these cells are represented as a cell of type A_1 directly before a cell of type A_2. The first portion could thus be reused to store a value of type A_1, while the second portion could be reused to store a separate value of type A_2 at a different time. Similar opportunities come up with sums and pointers. Removing the label from a cell that stores a sum creates a cell that can store a value of one of the summand types. Pointers are all the same size anyhow, and therefore always present these reuse opportunities. These improvements could bring significant speedups to the compiled executables, and also further reduce their memory usage.

A third way to increase the number of reuse opportunities comes around function boundaries, like the ones described in the fibsum benchmark. If a function f calls another function g using the call construct, then g does not have access to any resources beyond its arguments. This can be problematic for memory reuse if f has freed a cell that g would be able to reuse. Thus, another optimization that we would like to investigate is adding the ability for a process to take free cells as inputs. An argument that is given in a free state could then be reused, giving the callee access to a memory cell from the caller. This would allow them to reuse these cells that would otherwise be wasted.

Function boundaries also cause problems because they can generate garbage in the form of [referenced] cells which have no actual references. When we were defining functions, we stated that a function that uses up the memory cells in Γ must conclude with these cells somehow ending in a referenced state. This sometimes naturally happens because a pointer to something in Γ is written to the output. However, it also can come about because the function disposes the cell to comply with the requirements from its typing rule. Disposed cells have no references and therefore cannot ever be freed or reused, losing out on potential reuse opportunities. The same problem applies to lazy products, which are the other negative type in our language.

The final potential research direction is to add concurrency to our language. The original SNAX language, in both its linear and nonlinear forms, permitted concurrency. This stemmed from the fact that memory cells could only be read after they were initially written, preventing race conditions from occurring. In the linear setting, a cell a can only be read once, and then it no longer exists. The type system therefore requries that a is written exactly once, and that it is read exactly once. The process reading a blocks until a is written, therefore, no uninitialized reads can occur either. In the nonlinear setting, a is written exactly once, and then other processes can read a an unlimited number of times – the cell a persists to the end of the program. However, since a's contents do not change, no race conditions on a can occur. After all, every read from cell a reads identical data, ensuring that the program outputs a consistent result.

Throughout this paper, we have been presenting a sequential form of SNAX, where the leftmost process in the configuration is always the one to step forward. If we lift this restriction and allow any thread to step forward, we get a concurrent form of SNAX. Unfortunately, adding concurrency to SNAX with memory reuse can create race conditions in certain programs. This problem becomes evident when considering a simple configuration like the following:

$$\begin{gathered}
\text{cell } a \ \langle\rangle \\
\text{thread } (b, \text{read } a (n \Rightarrow \text{free } a; \text{reuse } (a \leftarrow (\text{write } x \ (2 \cdot n); \text{write } b \ \langle\rangle)))) \\
\text{cell } b \ \square \\
\text{thread } (c, \text{read } a (n \Rightarrow \text{free } a; \text{reuse } (a \leftarrow (\text{write } x \ (n+1); \text{write } c \ \langle\rangle)))) \\
\text{cell } c \ \square
\end{gathered}$$

This configuration type-checks under our linear system – the first cell a is available, with type 1. The thread writing to b performs three operations. First, it reads the contents of a. It then doubles its contents, and writes the new value back into a, which can be thought of as an in-place update for a. Finally, it writes a unit to the cell b. The context that was used to type this thread is $a : 1[\text{available}]$. If we were to apply the Thread rule to this thread, we would get that the new context is $a : 1[\text{available}], b : 1[\text{available}]$ – meaning that a is available to the next thread as well. This thread, which writes to c, performs a similar in-place update, except it adds 1 to the value of a instead of doubling it.

Interpreted concurrently, this configuration contains a race condition depending on which of the two processes steps forward first! If the thread writing to b steps forward first, then we will multiply the value in a by 2 before making it available to the other process, which will then add 1. Thus, once these two processes have finished executing, a will contain the value 4. However, if these two processes execute in the opposite order, then a will contain the value 3 instead. In this example, the race condition is benign, as the value of a is never used. Still, the value stored in a can differ between runs of our program, making it nondeterministic. If we add a fourth process that reads the value of a and does something meaningful with it, then the unpredictability of the value of a can cause actual problems as well. Thus, a program that type-checks under our current set of rules is not guaranteed to be race-free.

These sorts of problems can be generalized into a larger class of programs. Suppose we have a cell b, which stores a unit. After this cell, we have a series of processes, each of which reads $\langle\rangle$ out of b and then restores $\langle\rangle$ back to b through a reuse once it is done executing. In this manner, we have essentially created a mutual exclusion lock in SNAX, such that only one process can continue executing at any given time. However, the manner in which these processes might execute varies, which could once again lead to the results stored in the other cells differing between runs of the program. Combining multiple of these "locks" can also lead to other problems, such as deadlock.

Due to this problem, integrating memory reuse into the current SNAX language will require more work in the future. The statics and dynamics presented here only suffice to ensure that sequential execution of SNAX programs will not deadlock or otherwise exhibit undesirable behavior. To ensure that concurrent programs are valid, there are two options. The first is to make the statics stricter, so that problematic programs like the above one will not type-check. The second is to make the dynamics more specific, so that these programs can only step forward in a limited set of ways which end up being equivalent. Both of these approaches will require significant changes to the language in the future.

11 Bibliography

References

[1] Henry DeYoung and Frank Pfenning. "Data Layout from a Type-Theoretic Perspective". In: *Electronic Notes in TICS* Vol. 1 (Feb. 2023). DOI: 10.46298/entics.10507.

[2] Henry DeYoung, Frank Pfenning, and Klaas Pruiksma. "Semi-Axiomatic Sequent Calculus". In: *FSCD 2020*. Vol. 167. 2020. DOI: 10.4230/LIPIcs.FSCD.2020.29.

[3] Martin Hötzel Escardó. "PCF extended with real numbers". In: *Theoretical Computer Science* 162.1 (1996), pp. 79–115. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(95)00250-2.

[4] G. Gentzen. "Untersuchungen über das logische Schließen I". In: *Mathematische Zeitschrift* 39 (1935), pp. 176–210.

[5] Jean-Yves Girard and Yves Lafont. "Linear Logic and Lazy Computation". In: *TAPSOFT, Vol.2*. 1987.

[6] Junyoung Jang et al. *Adjoint Natural Deduction (Extended Version)*. 2024. arXiv: 2402.01428 [cs.LO].

[7] Anton Lorenzen, Daan Leijen, and Wouter Swierstra. *FP2: Fully in-Place Functional Programming*. Tech. rep. MSR-TR-2023-19. Extended version of the ICFP'23 publication. Microsoft, May 2023.

[8] Klaas Pruiksma and Frank Pfenning. "Back to Futures". In: *CoRR* abs/2002.04607 (2020). arXiv: 2002.04607.

[9] Alex Reinking et al. "Perceus: Garbage Free Reference Counting with Reuse". In: *PLDI 2021*. Virtual, Canada: Association for Computing Machinery, 2021, pp. 96–111. ISBN: 9781450383912. DOI: 10.1145/3453483.3454032.

[10] Noam Zeilberger. "Focusing and higher-order abstract syntax". In: *POPL 2008*, pp. 359–369.

Appendix A: Additional Cases of Preservation

We once again consider the preservation theorem, and show two cases for the modified read rules: one for the positive type $A_1 \otimes A_2$, and one for the negative type $A_1 \to A_2$. We start with the proof of preservation in the case where we are reading a value of type $A_1 \otimes A_2$ out of a memory cell. For brevity, we abbreviate [available] as [a] where necessary.

The typing judgement for the relevant portion of the configuration is of the form:

$$\Gamma_0 \vDash \text{cell } x.\pi_1\ S_1, \text{cell } x.\pi_2\ S_2, \text{cell } x\ \langle \_, \_ \rangle, \text{thread } (a, \text{read } x(\langle \_, \_ \rangle \Rightarrow P)), \text{cell } a\ \square :: \Gamma_3, a : A[\text{available}]$$

To find the type of this before it steps forward, we need to first add the cells on the left-hand side to the context, and then use them to type the process on the right-hand side. We begin by finding the type of the individual component cells, which we call derivation $\mathcal{A}$:

$$\mathcal{A} = \dfrac{\dfrac{\overset{\mathcal{D}}{\Gamma_0^a \vdash \text{write } (x \cdot \pi_1, S_1) :: (x \cdot \pi_1 : A_1); \Gamma_0'}}{\Gamma_0 \vDash \text{cell } x \cdot \pi_1\ S_1 :: \Gamma_1, \underline{x \cdot \pi_1 : A_1}[\text{a}]}\ \textsc{Cell} \qquad \dfrac{\overset{\mathcal{E}}{\Gamma_1^a, \underline{x \cdot \pi_1 : A_1}[\text{a}] \vdash \text{write } (x \cdot \pi_2, S_2) :: (x \cdot \pi_2 : A_2); \Gamma_1', \underline{x \cdot \pi_1 : A_1}[\text{a}]}}{\Gamma_1, \underline{x \cdot \pi_1 : A_1}[\text{a}] \vDash \text{cell } x \cdot \pi_2\ S_2 :: \Gamma_2, \underline{x \cdot \pi_1 : A_1}[\text{a}], \underline{x \cdot \pi_2 : A_2}[\text{a}]}\ \textsc{Cell}}{\Gamma_0 \vDash \text{cell } x \cdot \pi_1\ S_1, \text{cell } x \cdot \pi_2\ S_2 :: \Gamma_0, \underline{x \cdot \pi_1 : A_1}[\text{a}], \underline{x \cdot \pi_2 : A_2}[\text{a}]}\ \textsc{Join}$$

We write Γ_1 for $R(\Gamma_0, \Gamma_0')$ and Γ_2 for $R(\Gamma_1, \Gamma_1')$ here, which allows us to account for any cells that S_1 and S_2 consume. With the conclusion from $\mathcal{A}$, we can find the type of the overall cell x, which we call derivation $\mathcal{B}$:

$$\mathcal{B} = \dfrac{\mathcal{A} \qquad \dfrac{\dfrac{}{\Gamma_2, \underline{x \cdot \pi_1 : A_1}[\text{a}], \underline{x \cdot \pi_2 : A_2}[\text{a}] \vdash \text{write } x\ \langle \_, \_ \rangle : \Gamma_2, x : A_1 \otimes A_2[\text{a}]}\ \otimes\text{A}}{\Gamma_2, \underline{x \cdot \pi_1 : A_1}[\text{a}], \underline{x \cdot \pi_2 : A_2}[\text{a}] \vDash \text{cell } x\ \langle \_, \_ \rangle :: \Gamma_2, x : A_1 \otimes A_2[\text{a}]}\ \textsc{Cell}}{\Gamma_0 \vDash \text{cell } x \cdot \pi_1\ S_1, \text{cell } x \cdot \pi_2\ S_2, \text{cell } x\ \langle \_, \_ \rangle :: \Gamma_2, x : A_1 \otimes A_2[\text{a}]}\ \textsc{Join}$$

Here, Γ_2 stays the same on both the left and right-hand contexts when typing the cell x because x only consumes the two cells $x.\pi_1$ and $x.\pi_2$, and nothing else. With derivation $\mathcal{B}$ in hand, we can now find the type of the configuration as a whole:

$$\dfrac{\mathcal{B} \qquad \dfrac{\dfrac{\overset{\mathcal{F}}{\Gamma_2, \underline{x \cdot \pi_1 : A_1}[\text{a}], \underline{x \cdot \pi_2 : A_2}[\text{a}], x : A_1 \otimes A_2[\text{read}] \vdash P :: (a : A); \Gamma_3}}{\Gamma_2, x : A_1 \otimes A_2[\text{a}] \vdash \text{read } x(\langle \_, \_ \rangle \Rightarrow P :: (a : A); \Gamma_3}\ \otimes\text{L}}{\Gamma_2, x : A_1 \otimes A_2[\text{a}] \vDash \text{thread } (a, \text{read } x(\langle \_, \_ \rangle \Rightarrow P)), \text{cell } a\ \square :: \Gamma_3, a : A[\text{a}]}\ \textsc{Thread}}{\Gamma_0 \vDash \text{cell } x \cdot \pi_1\ S_1, \text{cell } x \cdot \pi_2\ S_2, \text{cell } x\ \langle \_, \_ \rangle, \text{thread } (a, \text{read } x(\langle \_, \_ \rangle \Rightarrow P)), \text{cell } a\ \square :: \Gamma_3, a : A[\text{a}]}\ \textsc{Join}$$

Once we step this configuration forward using the read rule, we want to show:

$$\Gamma_0 \vDash \text{cell } x.\pi_1\ S_1, \text{cell } x.\pi_2\ S_2, \text{read } x\ \langle \_, \_ \rangle, \text{thread } (a, P), \text{cell } a\ \square :: \Gamma_3, a : A[\text{a}]$$

The type of the leftmost three cells can once again be found using the Join rule. Since the first two have not changed, we can reuse the derivation $\mathcal{A}$ and get derivation $\mathcal{B}'$:

$$\mathcal{B}' = \dfrac{\mathcal{A} \qquad \dfrac{\dfrac{}{\Gamma_2, \underline{x \cdot \pi_1 : A_1}[\text{a}], \underline{x \cdot \pi_2 : A_2}[\text{a}] \vdash \text{write } x\ \langle \_, \_ \rangle : \Gamma_2, x : A_1 \otimes A_2[\text{a}]}\ \otimes\text{A}}{\Gamma_2, \underline{x \cdot \pi_1 : A_1}[\text{a}], \underline{x \cdot \pi_2 : A_2}[\text{a}] \vDash \text{read } x\ \langle \_, \_ \rangle :: \Gamma_2, \underline{x \cdot \pi_1 : A_1}[\text{a}], \underline{x \cdot \pi_2 : A_2}[\text{a}], x : A_1 \otimes A_2[\text{read}]}\ \textsc{Read}}{\Gamma_0 \vDash \text{cell } x \cdot \pi_1\ S_1, \text{cell } x \cdot \pi_2\ S_2, \text{read } x\ \langle \_, \_ \rangle :: \Gamma_2, \underline{x \cdot \pi_1 : A_1}[\text{a}], \underline{x \cdot \pi_2 : A_2}[\text{a}], x : A_1 \otimes A_2[\text{read}]}\ \textsc{Join}$$

And finally, we can prove the desired result as follows:

$$\dfrac{\mathcal{B}' \qquad \dfrac{\overset{\mathcal{F}}{\Gamma_2, \underline{x \cdot \pi_1 : A_1}[\mathrm{a}], \underline{x \cdot \pi_2 : A_2}[\mathrm{a}], x : A_1 \otimes A_2[\mathrm{read}] \vdash P :: (a : A); \Gamma_3}}{\Gamma_2, \underline{x \cdot \pi_1 : A_1}[\mathrm{a}], \underline{x \cdot \pi_2 : A_2}[\mathrm{a}], x : A_1 \otimes A_2[\mathrm{read}] \vDash \mathrm{thread}\ (a, P), \mathrm{cell}\ a\ \square :: \Gamma_3, a : A[\mathrm{available}]}\ \textsc{Thread}}{\Gamma_0 \vDash \mathrm{cell}\ x \cdot \pi_1\ S_1, \mathrm{cell}\ x \cdot \pi_2\ S_2, \mathrm{read}\ x\ \langle \_, \_ \rangle, \mathrm{thread}\ (a, P), \mathrm{cell}\ a\ \square :: \Gamma_3, a : A[\mathrm{a}]}\ \textsc{Join}$$

The other cases for the use of the read dynamic for positive types are similar to this one – we find the types of the relevant cells to the left of the process. This consists of any new resources that are available to the thread after the read executes, as well as the type of the cell being read. We then step forward, marking the relevant cell as read in the configuration. The resulting context should then match the remainder of the process, allowing us to show that preservation holds.

We now consider the read dynamic for the case of invoking a closure of type $A_1 \to A_2$, and show that preservation holds in this case as well.

First, we show a lemma, which is similar to weakening but for linear programs:

Lemma

If $\Gamma_1 \vdash P :: (a : A); \Gamma_1'$, then $\Gamma_1, \Gamma_2 \vdash P :: (a : A); \Gamma_1', \Gamma_2$.

This lemma essentially states that we can add memory cells to the context of a well-typed process, and when the process finishes, those cells will remain untouched. We can show this by induction on the structure of the given typing judgement. The base cases come from axiomatic rules, where there are no premises. For example:

$$\dfrac{}{\Gamma_1 \vdash \mathrm{write}\ a\langle\rangle :: (a : 1); \Gamma_1}\ 1\textsc{a} \longrightarrow \dfrac{}{\Gamma_1, \Gamma_2 \vdash \mathrm{write}\ a\langle\rangle :: (a : 1); \Gamma_1, \Gamma_2}\ 1\textsc{a}$$

Cases with premises can be resolved largely by appealing to the inductive hypothesis. One interesting example of this is with the cut rule:

$$\dfrac{\Gamma_1 \vdash P :: (x : A); \Gamma_2 \quad \Gamma_2, x : A[\mathrm{available}] \vdash Q :: (c : C); \Gamma_3 \quad (x\ \mathrm{fresh})}{\Gamma_1 \vdash x \leftarrow P; Q :: (c : C); \Gamma_3}\ \textsc{cut}$$

$$\longrightarrow$$

$$\dfrac{\Gamma_1, \Gamma_4 \vdash P :: (x : A); \Gamma_2, \Gamma_4 \quad \Gamma_2, \Gamma_4, x : A[\mathrm{available}] \vdash Q :: (c : C); \Gamma_3, \Gamma_4 \quad (x\ \mathrm{fresh})}{\Gamma_1, \Gamma_4 \vdash x \leftarrow P; Q :: (c : C); \Gamma_3, \Gamma_4}\ \textsc{cut}$$

The second derivation here is a direct consequence of applying the inductive hypothesis to both premises. □

We now consider the following configuration:

$$\Gamma_0 \vDash \mathrm{cell}\ x\ S, \mathrm{cell}\ a\ (\langle z, d\rangle \Rightarrow P), \mathrm{thread}\ (c, \mathrm{read}\ a\langle x, c\rangle), \mathrm{cell}\ c\ \square :: \Gamma_2, c : A_2[\mathrm{available}], a : A_1 \to A_2[\mathrm{read}], y : A_1[\mathrm{read}]$$

We begin by finding the types of the two cells on the left-hand side. We define Γ_2, Γ_3 to be some partition of the cells in Γ_1, where Γ_3 are the cells needed by the function, and Γ_2 are the cells that can still be used after the closure is typed.

$$\mathcal{A} = \dfrac{\dfrac{\overset{\mathcal{D}}{\Gamma_0 \vdash \mathrm{write}\ (x, S) :: (x : A_1); \Gamma_1}}{\Gamma_0 \vDash \mathrm{cell}\ x\ S :: \Gamma_1, x : A_1[\mathrm{a}]}\ \textsc{Cell} \quad \dfrac{\dfrac{\overset{\mathcal{E}}{\Gamma_3, z : A_1[\mathrm{a}] \vdash P :: (d : A_2); \Gamma_4, z : A_1[\mathrm{read}] \quad (\Gamma_3^{\mathrm{refd}} \subseteq \Gamma_4)}}{\Gamma_2, \Gamma_3, x : A_1[\mathrm{a}] \vdash \mathrm{write}\ (a, (\langle d, z\rangle \Rightarrow P)) :: (a : A_1 \to A_2); \Gamma_2, x : A_1[\mathrm{a}]}\ \to\textsc{r}}{\Gamma_2, \Gamma_3, x : A_1[\mathrm{a}] \vDash \mathrm{cell}\ a\ (\langle z, d\rangle \Rightarrow P) :: \Gamma_2, x : A_1[\mathrm{a}], a : A_1 \to A_2[\mathrm{a}]}\ \textsc{Cell}}{\Gamma_0 \vDash \mathrm{cell}\ x\ S, \mathrm{cell}\ a\ (\langle z, d\rangle \Rightarrow P) :: \Gamma_2, x : A_1[\mathrm{a}], a : A_1 \to A_2[\mathrm{a}]}\ \textsc{Join}$$

Using this, we can find the type of the thread, and therefore the overall configuration:

$$\dfrac{\mathcal{A} \quad \dfrac{\overline{\Gamma_2, x : A_1[\mathrm{a}], a : A_1 \to A_2[\mathrm{a}] \vdash \text{read } a\langle x, c\rangle :: (c : A_2); \Gamma_2, a : A_1 \to A_2[\text{read}], x : A_1[\text{read}]}\ {\to}\textsc{a}}{\Gamma_2, x : A_1[\mathrm{a}], a : A_1 \to A_2[\mathrm{a}] \vDash \text{thread }(c, \text{read } a\langle x,c\rangle), \text{cell } c\ \square :: \Gamma_2, c : A_2[\mathrm{a}], a : A_1 \to A_2[\text{read}], x : A_1[\text{read}]}\ \textsc{Thread}}{\Gamma_0 \vDash \text{cell } x\ S, \text{cell } a\ (\langle z,d\rangle \Rightarrow P), \text{thread }(c, \text{read } a\langle x,c\rangle), \text{cell } c\ \square :: \Gamma_2, c : A_2[\mathrm{a}], a : A_1 \to A_2[\text{read}], x : A_1[\text{read}]}\ \textsc{Join}$$

It follows that the resulting type of the configuration is $\Gamma_2, a : A_1 \to A_2[\text{read}], x : A_1[\text{read}]$. We now show that once the configuration takes a step forward, it will have a type that is a superset of this type. After applying the read dynamic, we get:

$$\Gamma_0 \vDash \text{cell } y\ S, \text{read } a(\langle z,d\rangle \Rightarrow P), \text{thread }(c, [x/z, c/d]P), \text{cell } c\ \square :: \Gamma_2, \Gamma_4, c : A_2[\mathrm{a}], a : A_1 \to A_2[\text{read}], x : A_1[\text{read}]$$

We now want to show that this type holds after stepping forward. Since $\Gamma_2 \subseteq \Gamma_2, \Gamma_4$, showing this will prove that preservation holds in this case. We once again begin by finding the type of the first two cells:

$$\mathcal{A}' = \dfrac{\dfrac{\begin{array}{c}\mathcal{D}\\ \Gamma_0 \vdash \text{write }(x, S) :: (x : A_1); \Gamma_1\end{array}}{\Gamma_0 \vDash \text{cell } x\ S :: \Gamma_1, x : A_1[\mathrm{a}]}\ \textsc{Cell} \quad \dfrac{\dfrac{\begin{array}{c}\mathcal{E}\\ \Gamma_3, z : A_1[\mathrm{a}] \vdash P :: (d : A_2); \Gamma_4, z : A_1[\text{read}] \quad (\Gamma_3^{\text{refd}} \subseteq \Gamma_4)\end{array}}{\Gamma_2, \Gamma_3, x : A_1[\mathrm{a}] \vdash \text{write }(a, (\langle d, z\rangle \Rightarrow P)) :: (a : A_1 \to A_2); \Gamma_2, x : A_1[\mathrm{a}]}\ {\to}\textsc{r}}{\Gamma_2, \Gamma_3, x : A_1[\mathrm{a}] \vDash \text{cell } a\ (\langle z,d\rangle \Rightarrow P) :: \Gamma_2, \Gamma_3, x : A_1[\mathrm{a}], a : A_1 \to A_2[\text{read}]}\ \textsc{Read}}{\Gamma_0 \vDash \text{cell } x\ S, \text{read } a\ (\langle z,d\rangle \Rightarrow P) :: \Gamma_2, \Gamma_3, x : A_1[\mathrm{a}], a : A_1 \to A_2[\text{read}]}\ \textsc{Join}$$

With the information in $\mathcal{A}'$ and our earlier lemma, we are ready to prove what the overall type of the configuration is.

$$\dfrac{\mathcal{A}' \quad \dfrac{\dfrac{\begin{array}{c}[c/d, x/z]\mathcal{E}\\ \Gamma_3, x : A_1[\mathrm{a}] \vdash [x/z, c/d]P :: (c : A_2); \Gamma_4, x : A_1[\text{read}]\end{array}}{\Gamma_2, \Gamma_3, x : A_1[\mathrm{a}], a : A_1 \to A_2[\text{read}] \vdash [x/z, c/d]P :: (c : A_2); \Gamma_2, \Gamma_4, a : A_1 \to A_2[\text{read}], x : A_1[\text{read}]}\ \text{Lemma}}{\Gamma_2, \Gamma_3, x : A_1[\mathrm{a}], a : A_1 \to A_2[\text{read}] \vDash \text{thread }(c, [x/z, c/d]P), \text{cell } c\ \square :: \Gamma_2, \Gamma_4, c : A_2[\mathrm{a}], a : A_1 \to A_2[\text{read}], x : A_1[\text{read}]}\ \textsc{Thread}}{\Gamma_0 \vDash \text{cell } y\ S, \text{read } a(\langle z,d\rangle \Rightarrow P), \text{thread }(c, [x/z, c/d]P), \text{cell } c\ \square :: \Gamma_2, \Gamma_4, c : A_2[\mathrm{a}], a : A_1 \to A_2[\text{read}], x : A_1[\text{read}]}\ \textsc{Join}$$

Here, we are able to apply the lemma on $\Gamma_2, a : A_1 \to A_2[\text{read}]$ to obtain the desired result. A similar proof works for lazy products, which are the other negative type in our language.

It is also important to note that as this configuration continues to step forward, we will continue to be able to find the type of $(\langle z,d\rangle \Rightarrow P)$, which are the contents of the read cell a. This is because the function needs to know the types of the cells in Γ_3 in order to type-check. We also know that in Γ_4, each such cell has a [referenced] status. Thus, none of these cells can be deallocated by the function body P, allowing us to continue using them for our typing judgement.