Ergometric Multilinear Futures

Aditi Gupta

Advisor: Frank Pfenning

Carnegie Mellon University

School of Computer Science

Senior Honors Thesis

Abstract

In functional languages, it can be challenging to implement parallelism through futures efficiently. Two significant bottlenecks are memory management and granularity control in scheduling of parallel tasks. In this thesis, we provide the theoretical basis for addressing both of these challenges.

Our language is based on a mixed linear/nonlinear language, formulated in a semi-axiomatic sequent calculus so that its natural computational interpretation encompasses futures. Linearity allows eager deallocation, easing garbage collection. We identify multilinear types that may be shared among threads while retaining the advantages of linearity; parallel garbage collection on multilinear data uses simple reference counting. Finally, we augment this language with ergometric types to capture the total work of program execution under a flexible model of amortized cost based on potentials. This allows for informed scheduling decisions and partially-automated granularity control in both purely linear and multilinear settings. The potential inherent in a multilinear data structure is statically known and split among the threads accessing it. We prove type safety, which includes accuracy of reference counts and accounting of potential, and illustrate our language with small examples.

Acknowledgements

First and foremost, I would like to express my deepest gratitude towards my advisor, Frank Pfenning. Without his guidance, I would not be the computer scientist or researcher that I am now. Thank you for introducing me to programming languages research, for helping me navigate the challenges of this project, and most of all for believing in me.

I also want to thank Mark Stehlik for his patience and advice throughout our countless conversations about my frustrations, doubts, and anything else that was on my mind.

Finally, I wish to thank my family and friends for their constant love and support. Their encouragement and willingness to listen has meant the world to me.

Contents

List of Figures

Chapter 1

Introduction

Futures are a highly expressive form of parallel programming involving spawning multiple threads of computation that execute simultaneously, but they can be difficult to implement efficiently [5, 16]. Futures are more general than the more widely-studied fork-join parallelism, in which computations split into branches and later synchronize, since their use is not constrained to synchronous join points. While fork-join parallelism does not allow data dependencies between concurrently-executing threads, futures have no such restrictions.

Two of the primary obstacles in efficiently implementing futures seem to appear in garbage collection [3] and in scheduling decisions [32]; our language minimizes the need for complex garbage collection and presents opportunities to automate certain parts of scheduling. In this work, we present a core language for efficient functional parallelism with futures that takes advantage of linearity to enable straightforward memory management and static resource tracking. Specifically, enforcing notions of linearity eases garbage collection, as noted by Girard and Lafont [15]. Additionally, resource-aware types, which become possible in linear languages, can allow us to partially automate granularity decisions based on cost [1].

Overall, we present a language in which linearity, non-linearity, and an intermediate notion of *multilinearity* coexist to attain the benefits of linear futures without the restrictiveness. This research represents the convergence of many disparate avenues of prior work; we draw inspiration from work on futures, linearity, mixed linear/nonlinear systems, and ergometric types for the purpose of achieving practically efficient and expressive parallelism through futures. Our model of futures [16] is founded upon a semi-axiomatic sequent calculus with a shared-memory semantics [11, 28], inspired by linear session types [6, 22, 30]. Focusing primarily on linear futures affords us both practical and asymptotic efficiency through straightforward

memory management [15], algorithmic strategies [5], and granularity control automation.

However, because pervasive linearity restrictions are impractical, we allow both a linear and a nonlinear mode, as in LNL [4, 29]; the nonlinear mode enables natural functional programming, while the linear mode preserves efficiency benefits. We begin by presenting this mixed linear/nonlinear language with futures. For simplicity, we omit resource-tracking types in this and the next language and reintroduce them near the end of the thesis.

To further allow programmers to write ordinary functional code while gaining efficiency advantages, we next introduce a notion of *multilinearity*, which allows us to reuse and discard certain kinds of data in a limited way. Specifically, we identify a class of types that can be shared. Data with these types *can* be copied or dropped even in a linear language through functions of type $A \multimap A \otimes A$ or $A \multimap \mathbf{1}$ [25], but doing so is costly and tedious. We introduce reference counting for multilinear types into the operational semantics of our language; an address may have some finite number of references representing its number of clients, with no additional cost incurred. Programmers can then make use of three classes of data: an unrestricted layer, in which programmers can write ordinary functional programs; pure linearity, which enforces fully linear types; and multilinearity, which allows programmers to write programs as though data can be shared and discarded while still achieving the guarantees and benefits of linearity.

We lastly introduce the tracking of potential through ergometric types, as in work on Rast [9, 8]. We enable resource tracking for both linear and multilinear types [20, 21]; an address storing potential that has multiple clients can simply share its potential across its many clients. We describe the nuances and challenges of simultaneously introducing multilinearity and resource-tracking. Finally, we prove the nontrivial properties of progress and preservation.

Our primary contribution in this thesis is a basis for an efficient, general, and expressive implementation of functional futures. In Chapter 2, we provide background on linear futures, including a discussion of preliminary experiments regarding efficiency. In Chapter 3, we present a concurrent language that supports linear futures and add a nonlinear layer for greater expressivity and ease of use. Then, in Chapter 4, we introduce multilinear types that can be aliased or dropped through reference counting; we no longer incur the cost of explicitly copying or discarding them, and we retain both the efficiency of linearity and the generality of ordinary functional programs. Finally, in Chapter 5, we augment this system with ergometric types that allow us to track cost and simplify granularity control, thus enabling the full efficiency benefits of linearity.

We see our type system as an intermediate language in between a high-level language with a standard

functional syntax and machine code. We leave the details of the syntax, type-checking, elaboration, and efficient low-level implementation to future work.

Chapter 2

Linear Futures

2.1 Futures

Futures are a powerful source of parallelism in several classes of programs, relying on the concept that a computation need not be completed until its value is used. A future is an expression that begins a computation and immediately moves on to the next steps of a program; not until the value stored inside the future is needed will the program wait until the expression has been fully evaluated. This allows a program to compute both the expression in the future and the next instructions at the same time.

Introduced first for Multilisp [16], an extension of the Lisp programming language, futures have been adopted in many modern languages, including Python [27], Java [23], and Scala [12]. In Standard ML (SML), futures may be implemented with the signature shown in Figure 2.1.

```
signature FUTURE =
sig
  (* a future yielding a value of type 'a *)
  type 'a t

  (* create a new parallel future *)
  val future : (unit -> 'a) -> 'a t

  (* force the completion of a future *)
  val touch : 'a t -> 'a
end
```

Figure 2.1: Signature for Futures in SML

Futures are especially powerful for a technique known as pipelining [5], in which we parallelize at the instruction level by starting tasks that rely on subtasks of previous tasks before those tasks have been evaluated in full. As a running example throughout this thesis, we will consider the insertion of elements into a trie, which is highly parallelizable using futures; we can simply start a future that inserts one number while continuing to perform other operations on the trie simultaneously. Insertion into a large trie can be expensive, since we must traverse through the various levels of the trie, following branches depending on the structure of our element. If we instead begin a process that inserts one element and immediately continue with the rest of our computations, the insertion can occur in the background while we perform other operations simultaneously.

Futures can be modeled naturally using languages that resemble session-typed programming, which describes protocols between communicating processes [6, 22, 30].

2.2 Linear Types

In linear systems, we enforce the restriction that each object must be used exactly once. This means that we cannot write programs that duplicate or discard data. For example, if SML enforced a linearity restriction, we would not be able to write either of the following functions:

```
fun f x = (x, x)
fun g (x, y) = x
```

The first function, `f`, uses `x` more than once, while the second function `g`, leaves `y` unused. In a proof-theoretic setting, linearity corresponds to the removal of *weakening* (in which we are permitted to leave resources unused) and *contraction* (in which resources may be reused without restriction). Though linearity is highly restrictive, linear futures enjoy a wide variety of efficiency benefits over nonlinear futures.

Blelloch and Reid-Miller have shown that the asymptotic complexity of a pipelining algorithm can be improved with the knowledge of linearity [5]. Additionally, linear programs afford straightforward garbage collection [33]; since each object is accessed exactly once, it can be deallocated immediately after use. Finally, we observe that linearity allows us to gain granularity control benefits through ergometric types.

2.2.1 Granularity Control Using Ergometric Types

While algorithmic speedups and garbage collection have been studied previously, the automation of granularity control is somewhat non-obvious. Prior research has suggested that granularity control has a sig-

Processors	Time (µs)
Sequential	65921026
4	27364798
8	15867453
12	11902278
16	10036928
20	8808359
24	8309490
28	8376619

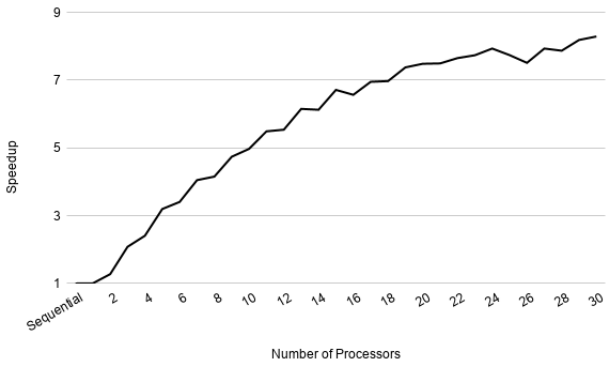


Figure 2.2: Speedup of Prime Sieve Program in Rast (Input 2,048)

nificant impact on efficiency in that it prevents overheads in scheduling from outweighing parallelism benefits [26]. Additionally, recent work has approached automation in granularity control from an execution time/systems perspective, based on runtime observations and experiments [1].

Here, we instead use programming language techniques to express cost at the type level without the need for experimental evaluation. Ergometric types allow us to track statically how much work will be done by each process; in executing a given computation, we use up some amount of "potential" that was originally available to our program. Without sufficient potential, a program will fail to typecheck. With ergometric types, we can keep track of the precise, amortized cost of computations, and use these values to make scheduling decisions. For instance, we could use ergometric types to automate the decision to execute a computation in parallel if its cost is high enough.

Based on preliminary experiments in Rast, a linear, session-typed, resource-aware language that can support futures, automating granularity control using ergometric types seems to be promising. Rast employs an ergometric type system using arithmetic refinements [10, 8]. This allows us to shift the decision of when to execute something in parallel to the interpreter rather than forcing the programmer to do it manually. Since the parallelism in Rast processes tends to be very fine-grained, we aggregate the work performed by multiple sequentially-executing processes. When enough potential will be expended, we execute computations in parallel rather than sequentially. As an example, we show the speedup of a prime sieve program in Rast in Fig. 2.2.

Though Rast is not practically usable due to its linearity restriction, it serves as a "proof-of-concept" research language, and these experiments provide motivation for extending ergometric types to a non-linear setting.

Standard ML	Rast	New Language
Nonlinear	Linearity restriction	Linear, nonlinear, and multilinear
Manual granularity control	Automated granularity control based on potentials	Automated granularity control based on potentials for most programs
Complex garbage collection	Easy garbage collection	Easy garbage collection for most programs
Futures not inherent in the language	Natural concurrency	Natural concurrency

Figure 2.3: Comparison of SML, Rast, and New Language

2.3 Summary of Motivation

In this work, we introduce a language that has the expressiveness of SML with the linearity advantages of Rast. If augmented with a functional, usable syntax, we anticipate that this language would present opportunities for efficient and easy-to-achieve parallelism through futures. A table summarizing the contributions and advantages of this language can be found in Fig. 2.3.

Chapter 3

A Mixed Linear/Nonlinear Type System for Futures

Because linearity severely limits functionality, we loosen linearity restrictions to allow an additional "mode" of the language for nonlinear data, based on adjoint logic [29]. We begin by presenting a preliminary linear/nonlinear type system, onto which we later add multilinearity and ergometric types. Since "potentials" represented by ergometric types are inherently linear, we omit ergometric types in this section and the next for the sake of simplicity, reintroducing them in Chapter 5.

3.1 Language Design

Our language is inspired in large part by Seax, a concurrent language supporting linear futures, based on session-typed systems [30]. In our shared-memory semantics, we read from and write to addresses in memory using concurrently-executing processes. Addresses are represented by cells, while processes are denoted by running threads. Futures are expressed through a *cut* operator, represented $x \leftarrow P\ ;\ Q$; this spawns a new process P, writing to x, and executes P and Q concurrently. If Q tries to read from x, it will block until P has finished computing its result.

With our naturally concurrent language, spawned processes automatically execute in parallel. However, it is straightforward to introduce a sequential composition operator that programmers can selectively use instead; we would add a *sequential cut*, represented by $x \Leftarrow P\ ;\ Q$. This sequential composition would be typed and executed almost identically to our existing concurrent spawn, except that Q would block until

A_m, B_m	$::=$	$\mathbf{1}_m$	true	unit
	$\mid$	$\oplus_m\{l : A^l_m\}_{l\in L}$	disjunction	variant records
	$\mid$	$\&_m\{l : A^l_m\}_{l\in L}$	additive conjunction	lazy records
	$\mid$	$A_m \otimes_m B_m$	multiplicative conj.	eager pairs
	$\mid$	$A_m \multimap_m B_m$	implication	functions
	$\mid$	$\uparrow A_{\mathsf{L}}\ (m = \mathsf{U})$	shift L to U	coerce to nonlinear
	$\mid$	$\downarrow A_{\mathsf{U}}\ (m = \mathsf{L})$	shift U to L	coerce to linear
	$\mid$	t		defined (recursive) type
P, Q	$::=$	$x_m \leftarrow y_m$	identity	move ($m = \mathsf{L}$) or copy ($m = \mathsf{U}$)
	$\mid$	$x_m \leftarrow P\ ;\ Q$	cut	allocate x and spawn P
	$\mid$	$x_k \leftarrow p\ \overline{y_m}$		call p
	$\mid$	$x_m.V$	$(\mathbf{1}R, \oplus R, \otimes R, \downarrow R)$	write V to x
	$\mid$	$\mathsf{case}\ x_m\ \{K\}$	$(\mathbf{1}L, \oplus L, \otimes L, \downarrow L)$	read V from x, pass V to K
	$\mid$	$x_m.V$	$(\&L, \multimap L, \uparrow L)$	read K from x, pass V to K
	$\mid$	$\mathsf{case}\ x_m\ \{K\}$	$(\&R, \multimap R, \uparrow R)$	write K to x

V	$::=$	$l(y)$	K	$::=$	$(l(y) \Rightarrow P_l)_{l\in L}$	injection ($\oplus$) selection ($\&$)
	$\mid$	$\langle w, y\rangle$		$\mid$	$\langle w, y\rangle \Rightarrow P$	pair ($\otimes$) function ($\multimap$)
	$\mid$	$\langle\rangle$		$\mid$	$\langle\rangle \Rightarrow P$	unit ($\mathbf{1}$)
	$\mid$	$\mathsf{shift}(y)$		$\mid$	$\mathsf{shift}(y) \Rightarrow P$	to linear ($\downarrow$) to nonlinear ($\uparrow$)

Figure 3.1: Types and Process Expressions

x is written. Because this is merely a matter of scheduling, sequential composition is directly supported in our language, type system, and metatheory. If we were to translate a functional source language to our intermediate language, most constructs would become sequential.

3.1.1 Grammar and Statics

The types and processes in this initial language follow LNL and Seax. The grammar for the language can be found in Fig. 3.1. Types are denoted by A_m and B_m, while process expressions are represented by P and Q.

Because pure linearity is highly restrictive, our language supports two distinct layers, inspired by LNL, which has both linear and nonlinear modes [4], represented via subscripts on types and variables. The linear mode, denoted L, admits neither weakening nor contraction, and the unrestricted mode, denoted U, admits both and thus represents ordinary functional programs. Here, we use the terms "unrestricted" and "nonlinear" interchangeably. The modes are ordered based on $\mathsf{U} > \mathsf{L}$. We enforce an Independence Principle that prevents unrestricted data from relying on linear data.

Below, we provide brief explanations of the various components of this language.

Types The types in this language are the standard propositions of adjoint logic, extended with recursive types. Most types, except for the shifts between modes, are the same as those from session-typed systems

[22] and from intuitionistic linear logic [15]. As in Seax [30], we support equirecursive types through a fixed global signature Σ, meaning that we can implicitly replace a type variable with its definition. For any type definition $t = A$, A is contractively defined, allowing us to treat types equirecursively based on a coinductive definition and an efficient algorithm for type equality, which is at the core of a typechecker [13].

As a running example, we will use binary numbers, which are represented by bit strings of `b0` or `b1`, ending with `e`. They are expressed using a disjunctive choice $\oplus$:

```
type bin = +{ b0 : bin, b1 : bin, e : 1 }
```

We envision binary numbers to be linear. Later, when we introduce multilinearity, they will be seen as multilinear; because they are purely positive types, they may be inductively copied or discarded, as demonstrated in Appendix A.1.1.

Processes Most of our process expressions are straighforward, and we explain them in detail below. The typing for a process P that reads from a context consisting of addresses $x_1, ..., x_n$, of types $A^1, ..., A^n$ and modes $m_1, ..., m_n$, and writes to address x is as follows:

$$(x_1 : A^1_{m_1}), ..., (x_n : A^n_{m_n}) \vdash P :: (x : A_k)$$

Note that each address has a unique writer, but it may have multiple readers, also described as clients, depending on its mode. Where an address represents a future, the readers will block until the cell has been written to.

The full typing rules can be found in Fig. 3.2. To account for both unrestricted and linear variables, we define a notion of $+$ on contexts, used in the typing rules to combine variables in contexts based on their modes:

$$\frac{}{\Gamma_1 = \Gamma_1 + (\cdot)} \qquad \frac{}{\Gamma_2 = (\cdot) + \Gamma_2}$$

$$\frac{\Gamma = \Gamma_1 + \Gamma_2}{\Gamma, (a : A_{\mathsf{L}}) = (\Gamma_1, (a : A_{\mathsf{L}})) + \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 + \Gamma_2}{\Gamma, (a : A_{\mathsf{L}}) = \Gamma_1 + (\Gamma_2, (a : A_{\mathsf{L}}))}$$

$$\frac{\Gamma = \Gamma_1 + \Gamma_2}{\Gamma, (a : A_{\mathsf{U}}) = (\Gamma_1, (a : A_{\mathsf{U}})) + (\Gamma_2, (a : A_{\mathsf{U}}))}$$

We also use Γ_W to denote a fully weakenable context, i.e., a context comprised of only unrestricted addresses.

$$\frac{m \geq r \qquad \Gamma_1 \vdash P :: (x : A_m) \qquad \Gamma_2, (x : A_m) \vdash Q :: (y : B_r) \qquad \Gamma = \Gamma_1 + \Gamma_2}{\Gamma \vdash x \leftarrow P \,;\, Q :: (y : B_r)}\ \textsc{Cut}$$

$$\frac{\overline{w : B_m} \vdash P :: (x : A_m) \in \Sigma \qquad \Gamma \vdash \overline{y : B_m}}{\Gamma \vdash z \leftarrow p\ \overline{y} :: (z : A_m)}\ \textsc{Call} \qquad \frac{}{\Gamma_W + (y : A_m) \vdash x \leftarrow y :: (x : A_m)}\ \textsc{Id}$$

$$\frac{i \in L}{\Gamma_W + (y : A^i_m) \vdash x.i(y) :: (x : \oplus_m \{l : A^l_m\}_{l \in L})}\ \oplus\mathrm{R}^0$$

$$\frac{\Gamma, (y : A^l_m) \vdash Q_{\mathsf{L}} :: (z : C_r)\ (\text{for all } l \in L)}{\Gamma + (x : \oplus_m \{l : A^l_m\}_{l \in L}) \vdash \mathsf{case}\ x\ \{l(y) \Rightarrow Q_{\mathsf{L}}\}_{l \in L} :: (z : C_r)}\ \oplus\mathrm{L}$$

$$\frac{}{\Gamma_W + (w : A_m) + (y : B_m) \vdash x.\langle w, y\rangle :: (x : A_m \otimes_m B_m)}\ \otimes\mathrm{R}^0$$

$$\frac{\Gamma, (w : A_m), (y : B_m) \vdash Q :: (z : C_r)}{\Gamma + (x : A_m \otimes_m B_m) \vdash \mathsf{case}\ x\ \{\langle w, y\rangle \Rightarrow Q\} :: (z : C_r)}\ \otimes\mathrm{L}$$

$$\frac{\Gamma \vdash Q_{\mathsf{L}} :: (y : A^l_m)\ (\text{for all } l \in L)}{\Gamma \vdash \mathsf{case}\ x\ \{l(y) \Rightarrow Q_{\mathsf{L}}\}_{l \in L} :: (x : \&_m \{l : A^l_m\}_{l \in L})}\ \&\mathrm{R}$$

$$\frac{i \in L}{\Gamma_W + (x : \&_m \{l : A^l_m\}_{l \in L}) \vdash x.i(y) :: (y : A^i_m)}\ \&\mathrm{L}^0 \qquad \frac{\Gamma, (w : A_m) \vdash Q :: (y : B_m)}{\Gamma \vdash \mathsf{case}\ x\ \{\langle w, y\rangle \Rightarrow Q\} :: (x : A_m \multimap_m B_m)}\ \multimap\mathrm{R}$$

$$\frac{}{\Gamma_W + (w : A_m) + (x : A_m \multimap_m B_m) \vdash x.\langle w, y\rangle :: (y : B_m)}\ \multimap\mathrm{L}^0$$

$$\frac{}{\Gamma_W \vdash x.\langle\rangle :: (x : \mathbf{1}_m)}\ \mathbf{1}\mathrm{R}^0 \qquad \frac{\Gamma \vdash Q :: (z : C_r)}{\Gamma + (x : \mathbf{1}_m) \vdash \mathsf{case}\ x\ \{\langle\rangle \Rightarrow Q\} :: (z : C_r)}\ \mathbf{1}\mathrm{L}$$

$$\frac{\Gamma \vdash Q :: (y : A_{\mathsf{L}})}{\Gamma \vdash \mathsf{case}\ x\ \{\mathsf{shift}(y) \Rightarrow Q\} :: (x : \uparrow A_{\mathsf{L}})}\ \uparrow\mathrm{R} \qquad \frac{}{\Gamma_W + (x : \uparrow A_{\mathsf{L}}) \vdash x.\mathsf{shift}(y) :: (y : A_{\mathsf{L}})}\ \uparrow\mathrm{L}^0$$

$$\frac{}{\Gamma_W + (y : A_{\mathsf{U}}) \vdash x.\mathsf{shift}(y) :: (x : \downarrow A_{\mathsf{U}})}\ \downarrow\mathrm{R}^0 \qquad \frac{\Gamma, (y : A_{\mathsf{U}}) \vdash Q :: (z : C_r)}{\Gamma, (x : \downarrow A_{\mathsf{U}}) \vdash \mathsf{case}\ x\ \{\mathsf{shift}(y) \Rightarrow Q\} :: (z : C_r)}\ \downarrow\mathrm{L}$$

Figure 3.2: Process Typing Rules for LNL

The first expression in our process grammar serves as the identity, either moving (in the case of linear addresses) or copying (in the case of unrestricted addresses) the contents of one address into another. The second is a *cut*, which allows us to spawn two concurrently-executing processes, as discussed previously. We additionally have a way of calling named processes, which, like recursive types, are defined in the fixed signature Σ.

The remaining process terms represent reading from and writing to addresses. Notably, we distinguish between values V and continuations K based on polarities of types [24, 2, 25, 14]. A value represents a *positive* type, which is defined by its structure; positive types are inductively defined and values can be seen as finite data structures in memory. In contrast, a continuation represents a *negative* type, which is defined by its behavior rather than its structure; continuations may be thought of as "lazy" data structures that await values and eventually proceed with their continuing processes. For instance, $\oplus$ represents a positive choice, while & represents a negative choice; both are expressed using $l(y)$. To write a label l to a variable x with positive type, we would use a process like $x.l(y)$, where y already exists in the context. Reading from a variable x with type $\oplus\{l : A^l\}_{l \in L}$ would involve a process $\mathsf{case}\ x\ \{l(y) \Rightarrow P_l\}$. Meanwhile, writing a continuation to a variable x with negative type & would look like $\mathsf{case}\ x\ \{l(y) \Rightarrow P_l\}$; to read a continuation (i.e., pass it a value), we would run a process $x.l(y)$. The apparent ambiguity on $x.V$ and $\mathsf{case}\ x\ \{K\}$ is resolved based on whether x has already been written; during typechecking, we distinguish reads and writes from the position of x in the antecedent (read) or succedent (write) of the typing judgment.

Our values and continuations are fairly straightforward: we have labels, pairs (which also represent functions), units, and shifts. The $\mathsf{shift}(x)$ operator, modeled after general adjoint systems, is used to move between the linear and unrestricted modes.

As an example, we demonstrate a process that takes two binary numbers (of equal length) and computes the bitwise XOR:

```
decl xor : bin * bin -o bin
proc b <- xor (x1, x2) =
  case x1 ( b0(x1') => case x2 ( b0(x2') => b' <- xor (x1', x2') ;
                                            b.b0(b')
                               | b1(x2') => b' <- xor (x1', x2') ;
                                            b.b1(b')
                               | e() => % invalid
                               )
          | b1(x1') => case x2 ( b0(x2') => b' <- xor (x1', x2') ;
                                            b.b1(b')
                               | b1(x2') => b' <- xor (x1', x2') ;
                                            b.b0(b')
```

```
                              | e() => % invalid
                              )
        | e() => case x2 ( b0(x2') => % invalid
                         | b1(x2') => % invalid
                         | e() => b.e()
                         )
        )
```

Note that we have used abbreviated syntax here for simplicity.

Configurations In our shared-memory semantics, the state of a program is represented by a multiset containing threads and cells, called a configuration.

The grammar for a configuration $\mathcal{C}$ is shown below:

$\mathcal{C} ::= \cdot$	empty
$\mid \mathsf{thread}(c, P), \mathsf{cell}(c, -)$	thread running P, writing to cell c
$\mid \mathsf{cell}(c, W)$	written cell c containing W
$\mid \mathcal{C}_1, \mathcal{C}_2$	combination of two configurations

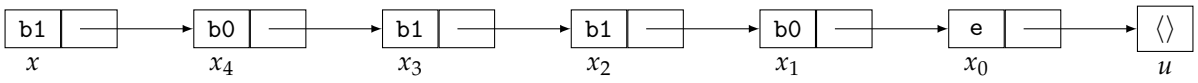
A thread describes a running process that writes to some (currently empty) cell. Meanwhile, a written cell contains either a value V or a continuation K. We define $W := V \mid K$; then, $\mathsf{cell}(c, W)$ represents cell c containing data W.

If we were to lay out the binary number `0b10110` (22 in decimal) in memory, it would be represented by the following configuration:

$$\mathsf{cell}(u, \langle\rangle), \mathsf{cell}(x_0, \mathtt{e}(u)), \mathsf{cell}(x_1, \mathtt{b0}(x_0)), \mathsf{cell}(x_2, \mathtt{b1}(x_1)), \mathsf{cell}(x_3, \mathtt{b1}(x_2)), \mathsf{cell}(x_4, \mathtt{b0}(x_3)), \mathsf{cell}(x, \mathtt{b1}(x_4))$$

Notably, each bit points to an address in memory containing the "previous" (i.e., lower) bit.

Below, we display a visual representation of what this binary number would look like in memory:



Similarly, `0b11110` (30 in decimal) would be represented by

$$\mathsf{cell}(v, \langle\rangle), \mathsf{cell}(y_0, \mathtt{e}(v)), \mathsf{cell}(y_1, \mathtt{b0}(y_0)), \mathsf{cell}(y_2, \mathtt{b1}(y_1)), \mathsf{cell}(y_3, \mathtt{b1}(y_2)), \mathsf{cell}(y_4, \mathtt{b1}(y_3)), \mathsf{cell}(y, \mathtt{b1}(y_4))$$

$$\frac{}{\Gamma \Vdash (\cdot) :: \Gamma}\ \text{C:Empty} \qquad \frac{\Gamma_1 \Vdash \mathcal{C}_1 :: \Gamma_2 \qquad \Gamma_2 \Vdash \mathcal{C}_2 :: \Gamma}{\Gamma_1 \Vdash \mathcal{C}_1, \mathcal{C}_2 :: \Gamma}\ \text{C:Join}$$

$$\frac{\Gamma_1 \vdash x.V :: (x : A_m) \qquad \Gamma = \Gamma_1 + \Gamma_2}{\Gamma \Vdash \mathsf{cell}(x, V) :: \Gamma_2, (x : A_m)}\ \text{C:Val} \qquad \frac{\Gamma_1 \vdash \mathsf{case}\ x\ \{K\} :: (x : A_m) \qquad \Gamma = \Gamma_1 + \Gamma_2}{\Gamma \Vdash \mathsf{cell}(x, K) :: \Gamma_2, (x : A_m)}\ \text{C:Cont}$$

$$\frac{\Gamma \vdash P :: (x_m : A_m) \qquad \Gamma = \Gamma_1 + \Gamma_2}{\Gamma \Vdash \mathsf{thread}(x_m, P), \mathsf{cell}(x_m, -) :: \Gamma_2, (x_m : A_m)}\ \text{C:Thread}$$

Figure 3.3: Configuration Typing Rules

To compute the XOR of these two numbers, our configuration would include the thread

$$\mathsf{thread}(c, c \leftarrow \mathtt{xor}\ \overline{(x, y)}), \mathsf{cell}(c, -)$$

The typing rules for configurations in this setting are shown in Fig. 3.3; we write values and continuations, run processes as threads, and allow for empty configurations and combinations of configurations.

3.1.2 Dynamics

In this system, we transition configurations via multiset rewriting, in which rules can be applied to any subconfiguration without modifying the remainder of the configuration. A configuration is *final* when it consists only of written cells and has no running threads.

To simplify the rules in which we read from a cell, we define what it means to pass a value V to a continuation K:

$$\begin{array}{lll} i(x) \circ (l(y) \Rightarrow P_l)_{l \in L} & = [x/y]P_i & (\oplus, \&) \\ \langle a, b \rangle \circ (\langle w, y \rangle \Rightarrow P) & = [a/w, b/y]P & (\otimes, \multimap) \\ \langle\rangle \circ (\langle\rangle \Rightarrow P) & = P & (\mathbf{1}) \\ \mathsf{shift}(x) \circ (\mathsf{shift}(y) \Rightarrow P) & = [x/y]P & (\uparrow, \downarrow) \end{array}$$

Also, to allow for unrestricted cells, we define

$$\mathcal{C}, [\mathsf{cell}(c_{\mathsf{U}}, D)] = \mathcal{C}, \mathsf{cell}(c_{\mathsf{U}}, D)$$
$$\mathcal{C}, [\mathsf{cell}(c_{\mathsf{L}}, D)] = \mathcal{C}$$

The full rules may be found in Fig. 3.4.

$$\mathsf{thread}(y, x \leftarrow P\ ;\ Q) \longmapsto \mathsf{thread}(x, P), \mathsf{cell}(x, -), \mathsf{thread}(y, Q)\ (y\ \mathsf{fresh}) \qquad \textit{cut}$$
$$\mathsf{thread}(z, z \leftarrow p\ \overline{y}) \longmapsto \mathsf{thread}(z, [z/x, \overline{y}/\overline{w}]P)\ (\text{given } P = x \leftarrow p\ \overline{w} \in \Sigma) \qquad \textit{call}$$
$$\mathsf{cell}(x_m, W), \mathsf{thread}(d_m, y_m \leftarrow x_m), \mathsf{cell}(d_m, -) \longmapsto [\mathsf{cell}(x_m, W)], \mathsf{cell}(y_m, W) \qquad \textit{id}$$
$$\mathsf{thread}(x, x.V), \mathsf{cell}(x, -) \longmapsto \mathsf{cell}(x, V) \qquad (\oplus R^0, \otimes R^0, \mathbf{1}_m R^0, \downarrow R^0)$$
$$\mathsf{cell}(x, V), \mathsf{thread}(y, \mathsf{case}\ x\ \{K\}), \mathsf{cell}(y, -) \longmapsto [\mathsf{cell}(x, V)], \mathsf{cell}(y, V \circ K) \qquad (\oplus L, \otimes L, \mathbf{1}_m L, \downarrow L)$$
$$\mathsf{thread}(x, \mathsf{case}\ x\ \{K\}), \mathsf{cell}(x, -) \longmapsto \mathsf{cell}(x, K) \qquad (\multimap R, \& R, \uparrow R)$$
$$\mathsf{cell}(x, K), \mathsf{thread}(y, x.V), \mathsf{cell}(y, -) \longmapsto [\mathsf{cell}(x, K)], \mathsf{cell}(y, V \circ K) \qquad (\multimap L^0, \& L^0, \uparrow L^0)$$

Figure 3.4: Concurrent Stepping Rules

If we were to step through the previous example of XOR-ing two binary numbers, we would get the trace shown in Fig. 3.5. We omit some of the later steps, which are analogous to the first steps.

3.2 Example: Tries of Linear Binary Numbers

With this initial language, we present the example of binary tries. We define a type `trie` as an interface allowing insertion and deletion of a binary number; because tries are defined in terms of their behavior, we use a negative choice &.

```
type trie = &{ update : bin * bool -o trie, ... }
```

A trie of this type is a continuation which, upon being passed an `update` label, takes in a binary number and a boolean representing whether to insert or delete. It then returns an updated trie. This interface would contain other fields we elide, such as `lookup`. A trie is somewhat like an object in an object-oriented language, with `update` being a method. We envision a trie to be purely linear due to the way in which tries are used; every time we update a trie, we should no longer have access to the original trie, and we should instead replace it with the returned value for our new trie.

We have recursive definitions for the `leaf` and `node` cases of a trie, each of which implement the shown interface. The full implementation (including potential annotations, as discussed in Section 5.3) may be found in Appendix A.

Booleans are represented by labels of `true` or `false`. Both booleans and binary numbers are notably positive choices ($\oplus$), representing data structures in memory. We imagine both of these types, like tries, to be linear.

```
type bool = +{ true : 1, false : 1 }
```

$\mathsf{cell}(u, \langle\rangle), \mathsf{cell}(x_0, \mathtt{e}(u)), \mathsf{cell}(x_1, \mathtt{b0}(x_0)), \mathsf{cell}(x_2, \mathtt{b1}(x_1)), \mathsf{cell}(x_3, \mathtt{b1}(x_2)), \mathsf{cell}(x_4, \mathtt{b0}(x_3)), \mathsf{cell}(x, \mathtt{b1}(x_4)),$
$\mathsf{cell}(v, \langle\rangle), \mathsf{cell}(y_0, \mathtt{e}(v)), \mathsf{cell}(y_1, \mathtt{b0}(y_0)), \mathsf{cell}(y_2, \mathtt{b1}(y_1)), \mathsf{cell}(y_3, \mathtt{b1}(y_2)), \mathsf{cell}(y_4, \mathtt{b1}(y_3)), \mathsf{cell}(y, \mathtt{b1}(y_4)),$
$\mathsf{thread}(c, c \leftarrow \mathtt{xor}\ \overline{(x, y)}), \mathsf{cell}(c, -)$

$\longmapsto \mathsf{cell}(u, \langle\rangle), \mathsf{cell}(x_0, \mathtt{e}(u)), \mathsf{cell}(x_1, \mathtt{b0}(x_0)), \mathsf{cell}(x_2, \mathtt{b1}(x_1)), \mathsf{cell}(x_3, \mathtt{b1}(x_2)), \mathsf{cell}(x_4, \mathtt{b0}(x_3)), \mathsf{cell}(x, \mathtt{b1}(x_4))$,
$\mathsf{cell}(v, \langle\rangle), \mathsf{cell}(y_0, \mathtt{e}(v)), \mathsf{cell}(y_1, \mathtt{b0}(y_0)), \mathsf{cell}(y_2, \mathtt{b1}(y_1)), \mathsf{cell}(y_3, \mathtt{b1}(y_2)), \mathsf{cell}(y_4, \mathtt{b1}(y_3)), \mathsf{cell}(y, \mathtt{b1}(y_4)),$
$\mathsf{thread}(c, \mathsf{case}\ x\ \{...\}), \mathsf{cell}(c, -)$

$\longmapsto \mathsf{cell}(u, \langle\rangle), \mathsf{cell}(x_0, \mathtt{e}(u)), \mathsf{cell}(x_1, \mathtt{b0}(x_0)), \mathsf{cell}(x_2, \mathtt{b1}(x_1)), \mathsf{cell}(x_3, \mathtt{b1}(x_2)), \mathsf{cell}(x_4, \mathtt{b0}(x_3)),$
$\mathsf{cell}(v, \langle\rangle), \mathsf{cell}(y_0, \mathtt{e}(v)), \mathsf{cell}(y_1, \mathtt{b0}(y_0)), \mathsf{cell}(y_2, \mathtt{b1}(y_1)), \mathsf{cell}(y_3, \mathtt{b1}(y_2)), \mathsf{cell}(y_4, \mathtt{b1}(y_3)), \mathsf{cell}(y, \mathtt{b1}(y_4))$,
$\mathsf{thread}(c, \mathsf{case}\ y\ \{...(\mathtt{b1}\ \text{branch})...\}), \mathsf{cell}(c, -)$

$\longmapsto \mathsf{cell}(u, \langle\rangle), \mathsf{cell}(x_0, \mathtt{e}(u)), \mathsf{cell}(x_1, \mathtt{b0}(x_0)), \mathsf{cell}(x_2, \mathtt{b1}(x_1)), \mathsf{cell}(x_3, \mathtt{b1}(x_2)), \mathsf{cell}(x_4, \mathtt{b0}(x_3)),$
$\mathsf{cell}(v, \langle\rangle), \mathsf{cell}(y_0, \mathtt{e}(v)), \mathsf{cell}(y_1, \mathtt{b0}(y_0)), \mathsf{cell}(y_2, \mathtt{b1}(y_1)), \mathsf{cell}(y_3, \mathtt{b1}(y_2)), \mathsf{cell}(y_4, \mathtt{b1}(y_3)),$
$\mathsf{thread}(c, b' \leftarrow b' \leftarrow \mathtt{xor}\ \overline{(x_4, y_4)}\ ;\ c.\mathtt{b0}(b')), \mathsf{cell}(c, -)$

$\longmapsto \ldots$

$\longmapsto \mathsf{thread}(c, b_0 \leftarrow b_0.\mathtt{e}()\ ;\ b_1 \leftarrow b_1.\mathtt{b0}(b_0)\ ;\ b_2 \leftarrow b_2.\mathtt{b0}(b_1)\ ;$
$b_3 \leftarrow b_3.\mathtt{b0}(b_2)\ ;\ b_4 \leftarrow b_4.\mathtt{b1}(b_3)\ ;\ c.\mathtt{b0}(b_4)), \mathsf{cell}(c, -)$

$\longmapsto \mathsf{cell}(u', \langle\rangle), \mathsf{cell}(b_0, \mathtt{e}(u')), \mathsf{cell}(b_1, \mathtt{b0}(b_0)), \mathsf{cell}(b_2, \mathtt{b0}(b_1)), \mathsf{cell}(b_3, \mathtt{b0}(b_2)), \mathsf{cell}(b_4, \mathtt{b1}(b_3)), \mathsf{cell}(c, \mathtt{b0}(b_4))$

In each configuration, we highlight the thread in green and any cells that are removed in the next step (due to linearity) in red.

Figure 3.5: Trace of Linear XOR of Binary Numbers

To insert a binary number into a trie, we case on the first bit and then recursively insert the remaining bits of our number into the appropriate subtrie, and similarly for deletion. When the end of bit sequence is reached, the node in the trie will hold the value `true` (the number is present) or `false` (the number is not present). To define insertion into a trie, we write `leaf` and `node` processes describing the behavior for each kind of trie, as shown below:

```
decl (l : trie) (b : bool) (r : trie) |- node : (t : trie)
decl . |- leaf : (t : trie)

proc t <- node l b r =
  case t ( update ((x,c),t') => case x (
    b0(y) => l' <- l.update((y,c),l') ; t' <- node l' b r
  | b1(y) => r' <- r.update((y,c),r') ; t' <- node l b r'
  | e() => t' <- node l c r
  ))

proc t <- leaf =
  case t ( update ((x,c),t') => case x (
    b0(y) => l <- leaf ; r <- leaf{1} ; l' <- l.update((y,c),l') ;
             t' <- node l' (false ()) r
  | b1(y) => ...symmetric...
  | e() => l <- leaf ; r <- leaf ;
           t' <- node l c r
  ))
```

Now, to insert a binary number `x` into a trie `t`, we would write

```
t1 <- t.update(t1) ; u <- u.() ; b <- b.true(u) ;
p <- p.(x,b) ; t2 <- t1.(p,t2) ;
```

This code begins with a trie `t`, which it then "updates" by reading from the trie and passing it the label `update`; the resulting trie will be called `t1`. The next two processes create a boolean value of `true`; since booleans are not built-in types, they are defined as labels on units (`1`). In particular, we allocate a variable `u` that contains a unit, and label it with `true`, producing variable `b`. Then, we pair up `x` and `b` by writing the pair to a new variable, `p`. Finally, we pass `p` and the continuation channel `t2` into our trie `t1`, which is awaiting an update.

Hiding the internal allocations, we can abbreviate this as

```
t2 <- t.update((x, true()), t2) ;
```

This passes the label `update` to the trie, and then proceeds to pass it a pair of a binary number and a boolean `true`. Since `t` is linear, this can be seen as an update to `t`, subsequently called `t2`. Notably, we must provide `t2`

as an input on the right-hand side of the arrow because the `update` label represents an implication; because of the way implication is defined, when we have $x.\langle w, y\rangle$ where x has an arrow type $A \multimap B$ and w has type A, y represents the resulting variable, with type B.

3.2.1 Copying

Given such a binary trie, a programmer might wish to reuse `x`. In our current system, however, an operation that involves reusing a value is quite cumbersome; once we use `x` by inserting it, any code that refers to it would be ill-typed.

To account for this, we might write a recursive `copy` function on binary numbers of type `bin -o bin * bin`. However, this would would require us to traverse the entire data structure and duplicate each bit individually:

```
proc p <- copy x =
  case x ( b0(x') => p' <- copy x' ;
                     case p' ((p1, p2) => p.(b0(p1), b0(p2)))
         | b1(x') => p' <- copy x' ;
                     case p' ((p1, p2) => p.(b1(p1), b1(p2)))
         | e() => p.((), ())
         )
```

The only way to avoid this extra cost is to make the binary number nonlinear. In doing so, however, we lose the benefits gained by linearity. Without the linearity restriction, we no longer achieve straightforward memory management. Though we have omitted ergometric types in this preliminary language, nonlinearity also causes us to lose the ability to track cost (see Chapter 5 for the version with ergometric types).

Chapter 4

Multilinearity

We now augment our language with a notion of multilinearity. Multilinear types are the subset of linear types A_{L} such that (possibly recursive) functions to copy ($A_{\mathsf{L}} \multimap A_{\mathsf{L}} \otimes A_{\mathsf{L}}$) and discard them ($A_{\mathsf{L}} \multimap \mathbf{1}$) can be defined [25]. Our operational semantics *shares* addresses of such types between an arbitrary number of readers, so no explicit copying or discarding is required. This allows us to retain the benefits of linearity without the need for extensive and expensive copying and dropping operations.

We first notice that any purely positive type (with no reliance on negative types) should be multilinear [25]. Furthermore, we can also easily reuse or discard shifted unrestricted types ($\downarrow A_{\mathsf{U}}$), since nonlinear cells may inherently have multiple readers. In a proof-theoretic sense, these types admit both weakening and contraction. This gives rise to the following grammar for multilinear types as a subset of the linear mode of the type system:

$$Q, R ::= \mathbf{1}_{\mathsf{L}} \mid \oplus_{\mathsf{L}}\{l : Q^l_{\mathsf{L}}\}_{l \in L} \mid Q_{\mathsf{L}} \otimes_{\mathsf{L}} R_{\mathsf{L}} \mid \downarrow A_{\mathsf{U}}$$

An alternative approach would to treat multilinearity as a third mode M in addition to the existing linear and unrestricted modes, with $\mathsf{L} < \mathsf{M} < \mathsf{U}$, instead of leaving multilinearity as a property on certain types within the linear mode. This decision is a matter of style and does not affect the essence of the type system or its theory.

4.1 Reference Counting

In the dynamics, we associate an address with a reference count representing its number of clients. Within the linear mode, addresses are no longer deallocated as soon as they are used; rather, they are deallocated

when their reference count becomes zero. As we alias and drop references to addresses, the reference counts of the underlying addresses increase and decrease, respectively. Specifically, we augment our process grammar with two new terms:

$$\begin{array}{rll} P, Q ::= \ldots \mid \mathsf{alias}\ x\ \mathsf{as}\ y, z\ ;\ P & & \text{alias a variable} \\ \mid \mathsf{drop}\ x\ ;\ P & & \text{drop a variable} \end{array}$$

We now need to differentiate between addresses (entirely dynamic artifacts), which have reference counts, and variables (in the statics), which are just linear or nonlinear. Our nonlinear mode remains mostly unchanged; we have unrestricted addresses that correspond to unrestricted variables. Within the linear mode, we continue to use variables uniquely, but allow multiple variables to refer to the same address. While previously variables and addresses were interchangeable, we now maintain environments that substitute addresses for variables at runtime, where an address's reference count denotes the number of variables for which it can be substituted.

In a thread, a closing environment η is defined as

$$\eta ::= \cdot \mid \eta, a/x$$

where a is an address and x is a variable. Threads substitute addresses for variables at runtime based on their associated closing environments. During compilation, however, we use variables so that we can take advantage of linearity; though addresses are shared, variables remain purely linear or unrestricted. Closing environments mediate between runtime and compile time, which are distinguished in this updated system; we discuss the typing for environments in more detail later in this section.

We update our configurations so that each address (whether it has already been written or not) is associated with a reference count:

$$\begin{array}{lll} \mathcal{C} ::= \cdot & & \text{empty} \\ \quad \mid \mathsf{thread}(c, [\eta], P), \mathsf{cell}(c, -, n) & & \text{thread, writing to cell } c \text{ with reference count } n \\ \quad \mid \mathsf{cell}(c, W, n) & & \text{written cell with reference count } n \\ \quad \mid \mathcal{C}_1, \mathcal{C}_2 & & \text{combination of two configurations} \end{array}$$

Purely linear addresses always have a reference count of 1, and nonlinear addresses have a reference count of ω, representing their ability to have any number of clients. For simplicity since we often refer to both linear and nonlinear addresses in unified rules, we define $\sigma ::= 1 \mid \omega$. We let $\sigma = 1$ if the mode is

linear, and $\sigma = \omega$ if the mode is nonlinear. Meanwhile, multilinear addresses may have a reference count of $n \in \mathbb{N}$, which can change based on aliasing and dropping of their references.

For example, consider what our two binary numbers 22 and 30 could look like laid out in memory, if we were to take advantage of multilinearity. Since both numbers have common bits at the end, they can take advantage of the same substructure in memory. In particular, both numbers share the following bits:

$$\mathsf{cell}(u, \langle\rangle, 1), \mathsf{cell}(b_0, \mathtt{e}(u), 1), \mathsf{cell}(b_1, \mathtt{b0}(b_0), 1), \mathsf{cell}(b_2, \mathtt{b1}(b_1), 1), \mathsf{cell}(b_3, \mathtt{b1}(b_2), 2)$$

Then, we can define 22 as

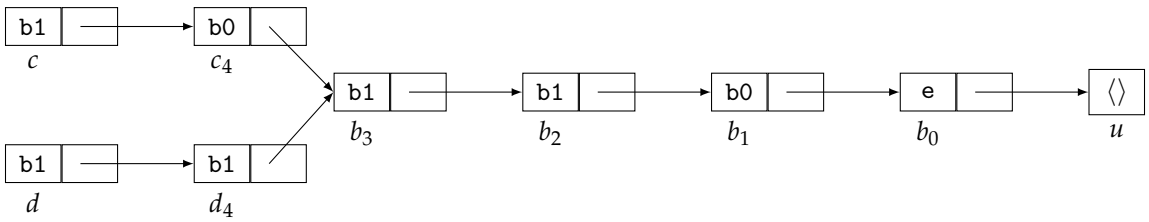
$$\mathsf{cell}(c_4, \mathtt{b0}(b_3), 1), \mathsf{cell}(c, \mathtt{b1}(c_4), 1)$$

and 30 as

$$\mathsf{cell}(d_4, \mathtt{b0}(b_3), 1), \mathsf{cell}(d, \mathtt{b1}(d_4), 1)$$

Notably, b_3 has a reference count of 2 because each of c_4 and d_4 must access it. All other addresses are only used once each.

We display the configuration containing both numbers visually in memory below:



4.1.1 Statics

Introducing reference counting necessitates some significant modifications to our type system for dynamic artifacts. By differentiating between addresses and variables, we can retain the same process typing as we would have in the original linear/nonlinear setting, with $\Gamma \vdash P :: (x : A)$ representing a process that reads from variables in Γ and writes the variable x of type A. Addresses are substituted in for the variables in Γ at runtime based on the environments associated with running processes.

Most of the rules are the same as before, but we add process terms for aliasing and dropping variables to enable sharing.

$$\frac{}{\Psi_1 = \Psi_1 + (\cdot)}\ \text{P1} \qquad \frac{}{\Psi_2 = (\cdot) + \Psi_2}\ \text{P2}$$

$$\frac{\Psi = \Psi_1 + \Psi_2}{\Psi, (a^1 : A_{\mathsf{L}}) = (\Psi_1, (a^1 : A_{\mathsf{L}})) + \Psi_2}\ \text{P3} \qquad \frac{\Psi = \Psi_1 + \Psi_2}{\Psi, (a^1 : A_{\mathsf{L}}) = \Psi_1 + (\Psi_2, (a^1 : A_{\mathsf{L}}))}\ \text{P4}$$

$$\frac{\Psi = \Psi_1 + \Psi_2}{\Psi, (a^{k+n} : Q_{\mathsf{L}}) = (\Psi_1, (a^k : Q_{\mathsf{L}})) + (\Psi_2, (a^n : Q_{\mathsf{L}}))}\ \text{P5}$$

$$\frac{\Psi = \Psi_1 + \Psi_2}{\Psi, (a^{\omega} : A_{\mathsf{U}}) = (\Psi_1, (a^{\omega} : A_{\mathsf{U}})) + (\Psi_2, (a^{\omega} : A_{\mathsf{U}}))}\ \text{P6}$$

Figure 4.1: Judgment for Combining Address Contexts

$$\frac{\Gamma, (y : Q_{\mathsf{L}}), (z : Q_{\mathsf{L}}) \vdash P :: (w : C_r)}{\Gamma, (x : Q_{\mathsf{L}}) \vdash \mathsf{alias}\ x\ \mathsf{as}\ y, z\ ;\ P :: (w : C_r)}\ \textsc{Alias} \qquad \frac{\Gamma \vdash P :: (w : C_r)}{\Gamma, (x : Q_{\mathsf{L}}) \vdash \mathsf{drop}\ x\ ;\ P :: (w : C_r)}\ \textsc{Drop}$$

Both of these rules only apply for variables of multilinear type. In both rules, we know by the Independence Principle that $r = \mathsf{L}$, since w relies on linear channels; however, we use a mode r in $(w : C_r)$ to allow changes to the type system without requiring substantial changes to this rule. When we alias, we replace a single variable by two variables that refer to the same address; similarly, dropping a variable simply removes it from the context. In the shared-memory semantics of our language, these process terms will increment and decrement reference counts of addresses.

Thus far, Γ has represented a context of variables, used in process typing and defined as $\Gamma = \cdot \mid \Gamma, (x_m : A)$. However, we now need an additional way to discuss the typing of addresses, which may have multiple references. We introduce a second kind of context containing addresses and their reference counts: $\Psi = \cdot \mid \Psi, (a_m^n : A)$. Here, $(a_m^n : A)$ indicates that address a_m has type A with n clients.

Now that addresses may have multiple references, we must provide a new way to combine two contexts, represented by $\Psi = \Psi_1 + \Psi_2$, defined in Fig. 4.1. This is particularly nuanced when it comes to multilinear addresses. Specifically, a multilinear address may have multiple references; to account for this, we allow reference counts to be added when two contexts are combined, as seen in Rule P5. For instance, $(a^2 : Q) = (a^1 : Q) + (a^1 : Q)$ when a is multilinear. For purely linear and unrestricted addresses, this definition is almost identical to $+$ on variable contexts, defined in Chapter 3.

We define a corresponding notion of $+$ on environments, which mirrors that on variable contexts; an unrestricted substitution persists in both segments and a linear substitution may be found only in one. The

$$\frac{}{\eta_1 = \eta_1 + []} \qquad \frac{}{\eta_2 = [] + \eta_2} \qquad \frac{\eta = \eta_1 + \eta_2}{\eta, a_{\mathsf{L}}/x_{\mathsf{L}} = (\eta_1, a_{\mathsf{L}}/x_{\mathsf{L}}) + \eta_2} \qquad \frac{\eta = \eta_1 + \eta_2}{\eta, a_{\mathsf{L}}/x_{\mathsf{L}} = \eta_1 + (\eta_2, a_{\mathsf{L}}/x_{\mathsf{L}})}$$

$$\frac{\eta = \eta_1 + \eta_2}{\eta, a_{\mathsf{U}}/x_{\mathsf{U}} = (\eta_1, a_{\mathsf{U}}/x_{\mathsf{U}}) + (\eta_2, a_{\mathsf{U}}/x_{\mathsf{U}})}$$

Figure 4.2: Judgment for Combining Environments

definition may be found in Fig. 4.2. Here, we do not need to consider reference counts, since each variable is purely linear or unrestricted.

We also define substitution through environments, where $\Psi \vdash \eta : \Gamma$ indicates that the addresses in Ψ are substituted through η for the variables in Γ:[1]

$$\frac{}{\Psi_W \vdash (\cdot) : (\cdot)}\ \text{E1} \qquad \frac{\Psi \vdash \eta : \Gamma}{\Psi + (a^{\sigma} : A) \vdash (\eta + a/x) : (\Gamma + (x : A))}\ \text{E2}$$

Note that Ψ_W represents a fully weakenable context, i.e., a context of only nonlinear addresses. This typing enforces the policy that an address's reference count exactly matches the number of clients it has. In particular, if σ is 1, we can create only one substitution of a. As an example, we consider the case of creating three variables that refer to a single address a. Based on the rules for combining contexts, if $\Psi = (a^3 : A)$, then $\Psi = (a^1 : A) + (a^1 : A) + (a^1 : A)$. Then, based on applications of Rules E1 and E2, we have $(a^1 : A) \vdash a/x : (x : A)$. We can again apply E2 to get $(a^1 : A) + (a^1 : A) \vdash a/x, a/y : (x : A), (y : A)$. Finally, we can apply E2 once more to get $\Psi \vdash a/x, a/y, a/z : (x : A), (y : A), (z : A)$. At this time, we are not permitted any more references to a, so we cannot substitute it for any other variables; thus, the reference count of 3 has been enforced.

Given these definitions, we can now provide the typing for full configurations in Fig. 4.3; $\Psi \Vdash \mathcal{C} :: \Psi'$ represents the typing of $\mathcal{C}$, where the configuration may read from addresses in Ψ and will write to those in Ψ'. The C:Empty and C:Join rules remain the same as before, but the value, continuation, and thread rules change.

Notably, there are key differences between these three rules. Since values V are intended to be directly observable, they contain only addresses, not variables (Rule C:Val); this also means that they do not need to be associated with environments. We therefore need a specific judgment $\Psi \vdash V : A$ referring directly to

[1]Now that we have introduced substitutions of addresses for variables, it also becomes straightforward to formally define substitutions of variables for other variables in order to more precisely define the call process. This does not affect the core of our language, so we do not discuss it in detail, but it makes the definition of call somewhat more elegant.

$$\frac{\Psi_1 \vdash V : A_m \qquad \Psi = \Psi_1 + \Psi_2}{\Psi \Vdash \mathsf{cell}(c, V, n) :: \Psi_2, (c^n : A_m)}\ \text{C:VAL}$$

$$\frac{\Psi_1 \vdash \eta : \Gamma \qquad \Gamma \vdash \mathsf{case}\ x\ \{K\} :: (x : A_m) \qquad \Psi = \Psi_1 + \Psi_2}{\Psi \Vdash \mathsf{cell}(c, [\eta]K, \sigma) :: \Psi_2, (c^\sigma : A_m)}\ \text{C:CONT}$$

$$\frac{\Psi_1 \vdash \eta : \Gamma \qquad \Gamma \vdash P :: (x_m : A_m) \qquad \Psi = \Psi_1 + \Psi_2}{\Psi \Vdash \mathsf{thread}(c_m, [\eta, c_m/x_m], P), \mathsf{cell}(c_m, -, n) :: \Psi_2, (c_m^n : A_m)}\ \text{C:THREAD}$$

Figure 4.3: Configuration Typing Rules

$$\frac{}{\Psi_W \vdash \langle\rangle : \mathbf{1}_m}\ \text{VAL-}\mathbf{1}\text{R}^0 \qquad \frac{i \in L}{\Psi_W + (y^\sigma : A_m^i) \vdash i(y) : \oplus_m\{l : A_m^l\}_{l \in L}}\ \text{VAL-}\oplus\text{R}^0$$

$$\frac{}{\Psi_W + (w^\sigma : A_m) + (y^1 : B_m) \vdash \langle w, y\rangle : A_m \otimes_m B_m}\ \text{VAL-}\otimes\text{R}^0 \qquad \frac{}{\Psi_W + (y^\sigma : A_\mathsf{U}) \vdash \mathsf{shift}(y) : \downarrow A_\mathsf{U}}\ \text{VAL-}\downarrow\text{R}^0$$

Figure 4.4: Typing Rules for Values Containing Addresses

addresses, shown in Fig. 4.4. These rules follow the same structure as the process typing rules for writing positively-typed values, but they act on addresses rather than variables. For instance, consider the configuration $\mathsf{cell}(u_\mathsf{L}, \langle\rangle, n), \mathsf{cell}(b_\mathsf{L}, \mathsf{true}(u_\mathsf{L}), k)$, which includes an address b_L representing the boolean value true. This configuration would be typed according to Rules C:Join and C:Val, along with Val-$\oplus$R^0, since the written cell b_L refers directly to an address.

Meanwhile, continuations and threads include process expressions with variables, so they are associated with environments. Continuations now become closures, which means that we define data in cells as $W := V \mid [\eta]K$, rather than simply $W := V \mid K$ as before. In C:Cont, we also restrict our cell to have a reference count of either 1 or ω, since continuation cells cannot be multilinear. On the other hand, in C:Thread, we allow multiple references to the result of the thread; even before a cell is written, we can refer to it, and if it is a multilinear address, we may have more than one reference.

4.1.2 Dynamics

Multilinearity also requires significant changes to the dynamics of the original LNL-based system. We still call configurations final when they consist only of cells and no threads, while transitions rely on multiset rewriting. The primary change to the rules is that we must now consider reference counts and substitutions in each of our steps. Most rules remain fairly standard, so we omit them here, leaving them to Appendix C

and focusing instead on the most illuminating.

As a first example, we begin with the rule that writes a label to a cell, representing the right rule for $\oplus$.

$$\mathsf{thread}(c_m, [\eta, d_m/y, c_m/x], x_m.i(y)), \mathsf{cell}(c_m, -, n) \longmapsto \mathsf{cell}(c_m, i(d_m), n)$$

Here, we must be careful with our closing environment; in our environment, the destination address c_m is substituted for x, the written variable in the process, while d_m, which should already exist in the configuration, is substituted for y. When we make the step, we dynamically substitute the addresses c_m and d_m for x and y to achieve a cell c_m containing a label on address d_m. The typing of this resulting configuration is reflected in Rule C:Val, defined in Fig. 4.3. The remaining right rules follow similarly.

Next, we examine the rule for cut:

$$\mathsf{thread}(c_m, [\eta_1 + \eta_2], x \leftarrow P\ ;\ Q) \longmapsto \mathsf{thread}(a, [\eta_1, a/x], P), \mathsf{cell}(a, -, \sigma), \mathsf{thread}(c_m, [\eta_2, a/x], Q)\ (a\ \mathsf{fresh})$$

In this rule, we must split up the environment to mirror linearity and create a new binding as we allocate a new address a. This new address, notably, has a reference count of 1 if the variable x is intended to be linear, or a reference count of ω if it should be unrestricted. The split of $\eta_1 + \eta_2$ is identified based on which variables are used by which process, which is determined during typechecking; this information can then be propagated to runtime.

Finally, we consider the rules for aliasing and dropping, and their implications for the rest of the dynamics. These steps are implemented directly by updating reference counts. Specifically, aliasing a cell simply increases its reference count; no inductive traversals need to be performed. Meanwhile, dropping a cell decreases its reference count; this rule only applies if the reference count is already greater than 0. We know by inversion on the typing rules for the alias and drop processes that these rules will only apply to multilinear addresses.

$$\mathsf{cell}(c, V, n), \mathsf{thread}(d, [\eta, c/x], \mathsf{alias}\ x\ \mathsf{as}\ y, z\ ;\ P) \longmapsto \mathsf{cell}(c, V, n+1), \mathsf{thread}(d, [\eta, c/y, c/z], P) \quad \textit{alias}$$
$$\mathsf{cell}(c, V, n), \mathsf{thread}(d, [\eta, c/x], \mathsf{drop}\ x\ ;\ P) \longmapsto \mathsf{cell}(c, V, n-1), \mathsf{thread}(d, [\eta], P)\ (\text{if } n > 0) \quad \textit{drop}$$

Based on these rules, our rules for reading from addresses with multilinear type become slightly more complicated. We use the standard stepping for any address with a reference count of 1 or ω. However, we also have a second set of rules for reading from multilinear cells that have a reference count of more than 1.

In our discussion, we focus on this latter set of rules.

As an example, we consider the following rule, in which we read from a multilinear cell containing a label, with a reference count of greater than 1:

$$\begin{aligned} &\mathsf{cell}(a, M, n'), \mathsf{cell}(c_m, i(a), n), \mathsf{thread}(d_k, [\eta, c_m/x_m, d_k/z_k], \mathsf{case}\ x_m\ \{(l(y) \Rightarrow P_l)_{l \in L}\}) \\ \longmapsto\ &\mathsf{cell}(a, M, n' \oplus 1), \mathsf{cell}(c_m, i(a), n-1), \mathsf{thread}(d_k, [\eta, a/y, d_k/z_k], P_i) \end{aligned}$$

with $\omega \oplus 1 := \omega$ if the address a is unrestricted.

In the above rule, we are reading from cell c_m, which is a label i on address a. Notably, we let $M := W \mid -$ to express that the address a might not be written when c_m refers to it. When we case on x_m, we lose one reference to c_m, meaning that we decrease its reference count by 1. However, c_m does not disappear from the configuration, as it would if it were purely linear; instead, it retains a reference to a. Now, a (which may or may not be written, but will be allocated) is also substituted in for y in the process P_i; we have created one extra reference to a that we need to account for.

This means that, if a is linear, we must increase its reference count. The other rules for reading from multilinear cells with reference count greater than 1 follow similarly, and all rules may be found in Fig. 4.5. For simplicity in expressing our rules, we elaborate forwarding for positive types into reading and writing cells, as follows:

$$\begin{aligned} &\mathsf{cell}(c_m, V, n), \mathsf{thread}(d_m, 0, [\eta, c_m/x_m, d_m/y_m], y_m \leftarrow x_m) \\ =\ &\mathsf{cell}(c_m, V, n), \mathsf{thread}(d_m, 0, [\eta, c_m/x_m, d_m/y_m], \mathsf{case}\ x_m\ \{V' \Rightarrow y_m.V'\}) \end{aligned}$$

where

$$\begin{aligned} V' &= i(z)\ (z\ \mathit{fresh}) && \text{if } V = i(c') \\ &= \langle w, z \rangle\ (w\ \mathit{fresh}, z\ \mathit{fresh}) && \text{if } V = \langle c', c'' \rangle \\ &= \langle\rangle && \text{if } V = \langle\rangle \\ &= \mathsf{shift}(z)\ (z\ \mathit{fresh}) && \text{if } V = \mathsf{shift}(c') \\ &= \mathsf{pot}\ q\ (z)\ (z\ \mathit{fresh}) && \text{if } V = \mathsf{pot}\ q\ (c') \end{aligned}$$

Thus, we do not have a specific dynamics rule for forwarding on positive cells.

With these updated dynamics, we can examine a trace for XOR-ing binary numbers, using the configuration in which both numbers share the same tail. This trace is shown in Fig. 4.6. Due to the length and complexity of the trace, we omit the early steps, which are equivalent to those shown in the purely linear trace, and the later steps, which follow the same pattern as the steps that are shown. In this trace, we use

General Rules

$\mathsf{thread}(c_m, [\eta_1 + \eta_2, c_m/y], x \leftarrow P\ ;\ Q)$
$\quad\longmapsto\ \mathsf{thread}(a, [\eta_1, a/x], P), \mathsf{cell}(a, -, \sigma), \mathsf{thread}(c_m, [\eta_2, a/x, c_m/y], Q)\ (a\ \mathsf{fresh})$ *cut*

$\mathsf{thread}(c_m, [\eta, c_m/z], z \leftarrow p[\zeta]), \mathsf{cell}(c_m, -, n)$
$\quad\longmapsto\ \mathsf{thread}(c_m, [\eta \circ \zeta, c_m/x], P), \mathsf{cell}(c_m, -, n)\ (\text{given } P = x \leftarrow p[\zeta]\ \in \Sigma)$ *call*

$\mathsf{cell}(c_m, [\eta']\{p\}K, \sigma), \mathsf{thread}(d_m, [(\eta + c_m/x_m), d_m/y_m], y_m \leftarrow x_m), \mathsf{cell}(d_m, -, \sigma)$
$\quad\longmapsto\ [\mathsf{cell}(c_m, [\eta']K, \sigma)], \mathsf{cell}(d_m, [\eta' + \eta]K, \sigma)$ *idK*

Positive Right Rules

$\mathsf{thread}(c_m, [\eta, d_m/y, c_m/x], x_m.i(y)), \mathsf{cell}(c_m, -, n) \longmapsto\ \mathsf{cell}(c_m, i(d_m), n)$ $(\oplus R^0)$

$\mathsf{thread}(c_m, [\eta, b_m/w, d_m/y, c_m/x], x.\langle w, y\rangle), \mathsf{cell}(c_m, -, n) \longmapsto\ \mathsf{cell}(c_m, \langle b_m, d_m\rangle, n)$ $(\otimes R^0)$

$\mathsf{thread}(c_m, [\eta, c_m/x], x.\langle\rangle), \mathsf{cell}(c_m, -, n) \longmapsto\ \mathsf{cell}(c_m, \langle\rangle, n)$ $(\mathbf{1}R^0)$

$\mathsf{thread}(c, [\eta, d/y, c/x], x.\mathsf{shift}(y)), \mathsf{cell}(c, -, n) \longmapsto\ \mathsf{cell}(c, \mathsf{shift}(d), n)$ $(\downarrow R^0)$

Negative Right Rules

$\mathsf{thread}(c_m, [\eta, c_m/x_m], \mathsf{case}\ x_m\ \{K\}), \mathsf{cell}(c_m, -, \sigma) \longmapsto\ \mathsf{cell}(c_m, [\eta]K, \sigma)$ $(\multimap R, \&R, \uparrow R)$

Linear and Nonlinear Left Rules

$\mathsf{cell}(c_m, i(c'), \sigma), \mathsf{thread}(d_k, [(\eta + c_m/x_m), d_k/z], \mathsf{case}\ x_m\ \{(l(y) \Rightarrow P_l)_{l\in L}\})$
$\quad\longmapsto\ [\mathsf{cell}(c_m, i(c'), \sigma)], \mathsf{thread}(d_k, [\eta, c'/y, d_k/z], P_i)$ $(\oplus L2)$

$\mathsf{cell}(c_m, \langle c', d'\rangle, \sigma), \mathsf{thread}(d_k, [(\eta + c_m/x_m), d_k/z], \mathsf{case}\ x_m\ \{\langle y, w\rangle \Rightarrow P\})$
$\quad\longmapsto\ [\mathsf{cell}(c_m, \langle c', d'\rangle, \sigma)], \mathsf{thread}(d_k, [\eta, c'/y, d'/w, d_k/z], P)$ $(\otimes L2)$

$\mathsf{cell}(c_m, \langle\rangle, \sigma), \mathsf{thread}(d_k, [(\eta + c_m/x_m), d_k/z], \mathsf{case}\ x_m\ \{\langle\rangle \Rightarrow P\})$
$\quad\longmapsto\ [\mathsf{cell}(c_m, \langle\rangle, \sigma)], \mathsf{thread}(d_k, [\eta, d_k/z], P)$ $(\mathbf{1}L2)$

$\mathsf{cell}(c_m, \mathsf{shift}(c'), 1), \mathsf{thread}(d_k, [(\eta + c_m/x_m), d_k/z], \mathsf{case}\ x_m\ \{\mathsf{shift}(y) \Rightarrow P\})$
$\quad\longmapsto\ \mathsf{thread}(d_k, [\eta, c'/y, d_k/z], P)$ $(\downarrow L2)$

Multilinear Left Rules

Note: All of these rules only apply when $n > 1$.

$\mathsf{cell}(a, D, n'), \mathsf{cell}(c_m, i(a), n), \mathsf{thread}(d_k, [\eta, c_m/x_m, d_k/z_k], \mathsf{case}\ x_m\ \{(l(y) \Rightarrow P_l)_{l\in L}\})$
$\quad\longmapsto\ \mathsf{cell}(a, D, n' \oplus 1), \mathsf{cell}(c_m, i(a), n-1), \mathsf{thread}(d_k, [\eta, a/y, d_k/z_k], P_i)$ $(\oplus L)$

$\mathsf{cell}(a, D_1, n_1'), \mathsf{cell}(b, D_2, n_2'), \mathsf{cell}(c_m, \langle a, b\rangle, n), \mathsf{thread}(d_k, [\eta, c_m/x_m, d_k/z_k], \mathsf{case}\ x_m\ \{\langle y, z\rangle \Rightarrow P\})$
$\quad\longmapsto\ \mathsf{cell}(a, D_1, n_1' \oplus 1), \mathsf{cell}(b, D_2, n_2' \oplus 1), \mathsf{cell}(c_m, \langle a, b\rangle, n-1),$
$\qquad \mathsf{thread}(d_k, [\eta, a/y, b/z, d_k/z_k], P)$ $(\otimes L)$

$\mathsf{cell}(c_m, \langle\rangle, n), \mathsf{thread}(d_k, [\eta, c_m/x_m, d_k/z_k], \mathsf{case}\ x_m\ \{\langle\rangle \Rightarrow P\})$
$\quad\longmapsto\ \mathsf{cell}(c_m, \langle\rangle, n-1), \mathsf{thread}(d_k, [\eta, d_k/z_k], P)$ $(\mathbf{1}L)$

$\mathsf{cell}(a, D, n'), \mathsf{cell}(c_m, \mathsf{shift}(a), n), \mathsf{thread}(d_k, [\eta, c_m/x_m, d_k/z_k], \mathsf{case}\ x_m\ \{\mathsf{shift}(y) \Rightarrow P\})$
$\quad\longmapsto\ \mathsf{cell}(a, D, n' \oplus 1), \mathsf{cell}(c_m, \mathsf{shift}(a), n-1), \mathsf{thread}(d_k, [\eta, a/y, d_k/z_k], P)$ $(\downarrow L)$

Negative Left Rules

$\mathsf{cell}(c_m, [\eta](l(y) \Rightarrow P_l)_{l\in L}, \sigma), \mathsf{thread}(d_k, [(\eta' + c_m/x_m), d_k/z], x_m.i(z))$
$\quad\longmapsto\ [\mathsf{cell}(c_m, [\eta]\{p\}(l(y) \Rightarrow P_l)_{l\in L}, \sigma)], \mathsf{thread}(d_k, [\eta' + \eta, d_k/y], P_i)$ $(\&L^0)$

$\mathsf{cell}(c_m, [\eta](\langle w, y\rangle \Rightarrow P), \sigma), \mathsf{thread}(d_k, [(\eta' + c_m/x_m), e_k/z_1, d_k/z_2], x_m.\langle z_1, z_2\rangle)$
$\quad\longmapsto\ [\mathsf{cell}(c_m, [\eta]\{p\}(\langle w, y\rangle \Rightarrow P), \sigma)], \mathsf{thread}(d_k, [\eta' + \eta, e_k/z_1, d_k/z_2], P)$ $(\multimap L^0)$

$\mathsf{cell}(c_m, [\eta](\mathsf{shift}(y) \Rightarrow P), \sigma), \mathsf{thread}(d_k, [(\eta' + c_m/x_m), d_k/z], x_m.\mathsf{shift}(z))$
$\quad\longmapsto\ [\mathsf{cell}(c_m, [\eta]\{p\}(\mathsf{shift}(y) \Rightarrow P), \sigma)], \mathsf{thread}(d_k, [\eta' + \eta, d_k/y], P)$ $(\uparrow L^0)$

Figure 4.5: Multilinear Dynamics Rules

the first bits of our numbers linearly, as before, but when we come to the shared substructure, we refrain from deallocating immediately and instead modify reference counts.

4.2 Example: Tries of Multilinear Binary Numbers

With multilinearity, our example becomes more straightforward. For the purposes of this example, we focus on reusing data; discarding variables by decrementing reference counts follows a very similar pattern.

Assuming again that we have a variable `x : bin`, we can write the following code, avoiding the need to traverse x to copy it.

```
alias x as y, z ; P
```

Internally, we store only one address for both y and z. We begin with a cell c_{L} with reference count 1, which is substituted for x in the thread:

$$\mathsf{cell}(c_{\mathsf{L}}, \mathtt{bin}, 1), \mathsf{thread}(d_{\mathsf{L}}, [c_{\mathsf{L}}/x], \mathsf{alias}\ x\ \mathsf{as}\ y, z\ ;\ P)$$

Now, by the dynamics rule for `alias`, this configuration steps to

$$\mathsf{cell}(c_{\mathsf{L}}, \mathtt{bin}, 2), \mathsf{thread}(d_{\mathsf{L}}, [c_{\mathsf{L}}/y, c_{\mathsf{L}}/z], P)$$

Notably, c_{L} now has a reference count of 2 because it has two variables accessing it: y and z, both of which can be used by P.

Using our two references to the binary number, we can now write code that reuses the number. For instance, we can insert the number and later delete it.

```
alias x as y, z ;
% insert into trie
t1 <- t.update((y,true()), t1) ;
% ... other operations ...
t2 <- t1.update((z,false()), t2) ;
```

Another possible use case would be deconstructing one reference to a number, and inserting the other into a trie. For example, let P be:

```
alias x as y, z ;
w <- case y { b0(y') => Q0 | b1(y') => Q1 | e(y')  => QE } ;
```

$\mathsf{cell}(u, \langle\rangle, 1), \mathsf{cell}(b_0, \mathtt{e}(u), 1), \mathsf{cell}(b_1, \mathtt{b0}(b_0), 1), \mathsf{cell}(b_2, \mathtt{b1}(b_1), 1), \mathsf{cell}(b_3, \mathtt{b1}(b_2), 2),$
$\mathsf{cell}(c_4, \mathtt{b0}(b_3), 1), \mathsf{cell}(c, \mathtt{b1}(c_4), 1), \mathsf{cell}(d_4, \mathtt{b0}(b_3), 1), \mathsf{cell}(d, \mathtt{b1}(d_4), 1),$
$\mathsf{thread}(a, [c/x, d/y, a/r], r \leftarrow \mathtt{xor}[x/x_1, y/x_2])$

$\longmapsto$ ...

$\longmapsto \mathsf{cell}(u, \langle\rangle, 1), \mathsf{cell}(b_0, \mathtt{e}(u), 1), \mathsf{cell}(b_1, \mathtt{b0}(b_0), 1), \mathsf{cell}(b_2, \mathtt{b1}(b_1), 1), \mathsf{cell}(b_3, \mathtt{b1}(b_2), 2),$
$\mathsf{thread}(a, [b_3/x_1', b_3/x_2', a/r], z_4 \leftarrow (z_3 \leftarrow z_3 \leftarrow \mathtt{xor}[x_1'/x_1, x_2'/x_2]\ ;\ z_4.\mathtt{b1}(z_3))\ ;\ r.\mathtt{b0}(z_4))$

$\longmapsto \mathsf{cell}(u, \langle\rangle, 1), \mathsf{cell}(b_0, \mathtt{e}(u), 1), \mathsf{cell}(b_1, \mathtt{b0}(b_0), 1),$ $\mathsf{cell}(b_2, \mathtt{b1}(b_1), 1)$, $\mathsf{cell}(b_3, \mathtt{b1}(b_2), 2)$,
$\mathsf{thread}(a, [b_3/x_1, b_3/x_2, a/r], z_4 \leftarrow (z_3 \leftarrow \mathsf{case}\ x_1\ \{...\}\ ;\ z_4.\mathtt{b1}(z_3))\ ;\ r.\mathtt{b0}(z_4))$

$\longmapsto \mathsf{cell}(u, \langle\rangle, 1), \mathsf{cell}(b_0, \mathtt{e}(u), 1), \mathsf{cell}(b_1, \mathtt{b0}(b_0), 1), \mathsf{cell}(b_2, \mathtt{b1}(b_1), 2),$ $\mathsf{cell}(b_3, \mathtt{b1}(b_2), 1)$,
$\mathsf{thread}(a, [b_2/x_1', b_3/x_2, a/r], z_4 \leftarrow (z_3 \leftarrow \mathsf{case}\ x_2\ \{...(\mathtt{b1}\ \text{branch})...\}\ ;\ z_4.\mathtt{b1}(z_3))\ ;\ r.\mathtt{b0}(z_4))$

$\longmapsto \mathsf{cell}(u, \langle\rangle, 1), \mathsf{cell}(b_0, \mathtt{e}(u), 1), \mathsf{cell}(b_1, \mathtt{b0}(b_0), 1), \mathsf{cell}(b_2, \mathtt{b1}(b_1), 2),$
$\mathsf{thread}(a, [b_2/x_1', b_2/x_2', a/r],$
$z_4 \leftarrow (z_3 \leftarrow (z_2 \leftarrow z_2 \leftarrow \mathtt{xor}[x_1'/x_1, x_2'/x_2]\ ;\ z_3.\mathtt{b0}(z_2))\ ;\ z_4.\mathtt{b1}(z_3))\ ;\ r.\mathtt{b0}(z_4))$

$\longmapsto \mathsf{cell}(u, \langle\rangle, 1), \mathsf{cell}(b_0, \mathtt{e}(u), 1),$ $\mathsf{cell}(b_1, \mathtt{b0}(b_0), 1)$, $\mathsf{cell}(b_2, \mathtt{b1}(b_1), 2)$,
$\mathsf{thread}(a, [b_2/x_1, b_2/x_2, a/r], z_4 \leftarrow (z_3 \leftarrow (z_2 \leftarrow \mathsf{case}\ x_1\ \{...\}\ ;\ z_3.\mathtt{b0}(z_2))\ ;\ z_4.\mathtt{b1}(z_3))\ ;\ r.\mathtt{b0}(z_4))$

$\longmapsto \mathsf{cell}(u, \langle\rangle, 1), \mathsf{cell}(b_0, \mathtt{e}(u), 1), \mathsf{cell}(b_1, \mathtt{b0}(b_0), 2),$ $\mathsf{cell}(b_2, \mathtt{b1}(b_1), 1)$,
$\mathsf{thread}(a, [b_1/x_1', b_2/x_2, a/r],$
$z_4 \leftarrow (z_3 \leftarrow (z_2 \leftarrow \mathsf{case}\ x_2\ \{...(\mathtt{b1}\ \text{branch})...\}\ ;\ z_3.\mathtt{b0}(z_2))\ ;\ z_4.\mathtt{b1}(z_3))\ ;\ r.\mathtt{b0}(z_4))$

$\longmapsto \mathsf{cell}(u, \langle\rangle, 1), \mathsf{cell}(b_0, \mathtt{e}(u), 1), \mathsf{cell}(b_1, \mathtt{b0}(b_0), 2),$
$\mathsf{thread}(a, [b_1/x_1', b_1/x_2', a/r], z_4 \leftarrow (z_3 \leftarrow (z_2 \leftarrow (z_1 \leftarrow z_1 \leftarrow \mathtt{xor}[x_1'/x_1, x_2'/x_2]\ ;$
$z_2.\mathtt{b0}(z_1))\ ;\ z_3.\mathtt{b0}(z_2)))\ ;\ z_4.\mathtt{b1}(z_3)))\ ;\ r.\mathtt{b0}(z_4)))$

$\longmapsto$ ...

$\longmapsto \mathsf{thread}(a, [a/r], z_0 \leftarrow z_0.\mathtt{e}()\ ;\ z_1 \leftarrow z_1.\mathtt{b0}(z_0)\ ;\ z_2 \leftarrow z_2.\mathtt{b0}(z_1)\ ;$
$z_3 \leftarrow z_3.\mathtt{b0}(z_2)\ ;\ z_4 \leftarrow z_4.\mathtt{b1}(z_3)\ ;\ r.\mathtt{b0}(z_4))$

$\longmapsto \mathsf{cell}(u', \langle\rangle), \mathsf{cell}(z_0, \mathtt{e}(u')), \mathsf{cell}(z_1, \mathtt{b0}(z_0)), \mathsf{cell}(z_2, \mathtt{b0}(z_1)), \mathsf{cell}(z_3, \mathtt{b0}(z_2)), \mathsf{cell}(z_4, \mathtt{b1}(z_3)), \mathsf{cell}(c, \mathtt{b0}(z_4))$

As in Fig. 3.5, we highlight the thread in green, and any cells that are removed in the next step (due to linearity) are shown in red. Cells whose reference count changes are highlighted in yellow before the change.

Figure 4.6: Trace of Multilinear XOR of Binary Numbers

```
% insert z into trie
t1 <- t.update((z,true()), t1) ;
```

If we had the initial configuration (prior to running P)

$$\mathsf{cell}(a, M, 1), \mathsf{cell}(c_{\mathsf{L}}, b1(a), 2), \mathsf{thread}(d_{\mathsf{L}}, [c_{\mathsf{L}}/y, c_{\mathsf{L}}/z], P)$$

then we would need to increment the reference count of a when we read from c_{L} and bind a new variable because it is being used both by `z` and by `y'`. The resulting configuration after extracting the first bit of `y` would be

$$\mathsf{cell}(a, M, 2), \mathsf{cell}(c_{\mathsf{L}}, b1(a), 1), \mathsf{thread}(d_{\mathsf{L}}, [c_{\mathsf{L}}/z, a/y'], w \leftarrow Q1\ ;\ P')$$

with P' representing everything after the case expression. By using multilinearity, this code never requires computing anything more than we would in a purely linear setting, nor does it incur any additional cost.

Chapter 5

Ergometric Types

Now that we have developed our multilinear system, we proceed to introduce ergometric types through potential annotations, which track the cost of computations at a static level [20, 21, 8]. In particular, since our type system is inspired by session types, we base our work on cost upon Rast, a purely linear, resource-aware, session-typed language [9]. Introducing ergometric types allows us to more easily discuss the cost of programs and provides benefits for parallelism. Specifically, we can use resource-tracking to specify and partially automate granularity control by making scheduling decisions based on cost annotations.

Our notion of cost is flexible and can be arbitrarily chosen by the programmer; for instance, we can use a model that incurs cost only when work is explicitly inserted by the user, or a model in which any read or write operation costs potential.

We annotate computations with the *potential* they require, or, in other words, their cost; each step of a program requires a specified amount of potential, accounting for any operations within it. Without sufficient potential, a program will not typecheck. We introduce two related types that express cost but do not incur any runtime overhead. Specifically, an address $(a : \triangleright^q A_{\mathsf{L}})$ stores q units of potential in memory for future use; when a client reads from this address, it harvests the potential for its own use [19]. Meanwhile, an address $(a : \triangleleft^q A_{\mathsf{L}})$ requires q units of potential in order to run the continuation of type A. We represent potential with the value V of the form $\mathsf{pot}\ q\ (a)$. The corresponding continuation is of the form $\mathsf{pot}\ q\ (y) \Rightarrow P$. Notably, $\triangleright^q A_{\mathsf{L}}$ is a positive type, while $\triangleleft^q A_{\mathsf{L}}$ is its dual negative type; addresses of the former type contain values, while addresses of the latter type contain continuations.

As an example, we introduce potential annotations to our type for binary numbers.

```
type bin{p} = +{ b0 : |{p}> bin{p}, b1 : |{p}> bin{p}, e : 1 }
```

Here, binary numbers are represented by `bin{p}` with `p` representing the potential stored in each bit. For simplicity, our language does not internally support arithmetic refinements that would allow in-language type definitions parameterized on numbers as in `bin{q}`, but adding these would be straightforward [10]. We instead consider binary numbers to be an infinite family of types, parameterized outside the language on the potential that they store. Adding these potential annotations allows us to track the cost of insertion based on the size of the binary number.

5.1 Statics and Dynamics

We now annotate each of our typing judgments with a potential value ($\vdash^q$ or $\Vdash^q$) to represent the cost that is used up. We introduce rules for writing (storing) and reading (harvesting) potential. Most of these rules are as expected:

$$\frac{}{\Gamma_W, (y : A_\mathsf{L}) \vdash^q x.\mathsf{pot}\ q\ (y) :: (x : \triangleright^q A_\mathsf{L})}\ \triangleright\mathrm{R}^0 \qquad \frac{\Gamma, (y : A_\mathsf{L}) \vdash^{q+p} Q :: (z : C_r)}{\Gamma, (x : \triangleright^p A_\mathsf{L}) \vdash^q \mathsf{case}\ x\ \{\mathsf{pot}\ p\ (y) \Rightarrow Q\} :: (z : C_r)}\ \triangleright\mathrm{L}$$

$$\frac{\Gamma \vdash^{q+p} P :: (x : A_\mathsf{L})}{\Gamma \vdash^p \mathsf{case}\ y\ \{\mathsf{pot}\ q\ (x) \Rightarrow P\} :: (y : \triangleleft^q A_\mathsf{L})}\ \triangleleft\mathrm{R} \qquad \frac{}{\Gamma_W, (x : \triangleleft^q A_\mathsf{L}) \vdash^q x.\mathsf{pot}\ q\ (y) :: (y : A_\mathsf{L})}\ \triangleleft\mathrm{L}^0$$

Storing potential q in a variable costs us q potential; meanwhile, reading from a variable storing potential allows us to use that potential in our continuing process. Similarly, when we write a continuation requiring potential, we can later pay that potential to execute the continuing process by reading from the continuation variable.

Most of the dynamics follow similarly:

$$\begin{array}{ll}
\mathsf{thread}(c_\mathsf{L}, q, [\eta, d_\mathsf{L}/y_\mathsf{L}, c_\mathsf{L}/x_\mathsf{L}], x_\mathsf{L}.\mathsf{pot}\ q\ (y_\mathsf{L})), \mathsf{cell}(c_\mathsf{L}, -, n) \longmapsto \mathsf{cell}(c_\mathsf{L}, \mathsf{pot}\ q\ (d_\mathsf{L}), n) & (\triangleright R^0) \\
\mathsf{thread}(c_m, q, [\eta, c_m/x_m], \mathsf{case}\ x_m\ \{\mathsf{pot}\ q\ (y) \Rightarrow P\}), \mathsf{cell}(c_m, -, \sigma) & \\
\quad \longmapsto \mathsf{cell}(c_m, [\eta]\{q\}(\mathsf{pot}\ q\ (y) \Rightarrow P), \sigma) & (\triangleleft R) \\
\mathsf{cell}(c_\mathsf{L}, \mathsf{pot}\ q'\ (c'), 1), \mathsf{thread}(d_k, q, [\eta, c_\mathsf{L}/x_\mathsf{L}, d_k/z], \mathsf{case}\ x_\mathsf{L}\ \{\mathsf{pot}\ q'\ (y) \Rightarrow P\}) & \\
\quad \longmapsto \mathsf{thread}(d_k, q + q', [\eta, c'/y, d_k/z], P) & (\triangleright L) \\
\mathsf{cell}(c_\mathsf{L}, [\eta]\{p\}(\mathsf{pot}\ q\ (c'_\mathsf{L}) \Rightarrow P), 1), \mathsf{thread}(d_\mathsf{L}, q, [\eta', c_\mathsf{L}/y_\mathsf{L}, d_\mathsf{L}/x_\mathsf{L}], y_\mathsf{L}.\mathsf{pot}\ q\ (x_\mathsf{L})) & \\
\quad \longmapsto \mathsf{thread}(d_\mathsf{L}, q + p, [\eta' + \eta, d_\mathsf{L}/c'_\mathsf{L}], P) & (\triangleleft L^0)
\end{array}$$

For now, we omit the case for reading from a cell $\mathsf{cell}(c_\mathsf{L}, \mathsf{pot}\ q\ (c'_\mathsf{L}), n)$ for $n > 1$; this situation is more complex and will be discussed later in this chapter.

In addition to allowing potential to be stored in addresses or required by a continuation, we also have a *work* construct that allows us to note when work is performed or cost expended. Work can either be manually inserted using $\mathsf{work}\ \{q\}\ ;\ P$ or implicitly performed based on the cost model. In most cases, it will be automatically computed from the cost model. However, the programmer can also choose to manually insert work if they want to describe additional cost. For instance, one could have a cost model in which everything is free except for a few certain computations, which require user-specified potential; in such a case, the user could use the work construct to specify the cost of these computations. We have the following rules for work in the statics and dynamics:

$$\frac{\Gamma \vdash^q P :: (x : A_m)}{\Gamma \vdash^{r+q} \mathsf{work}\ \{r\}\ ;\ P :: (x : A_m)}\ \textsc{Work}$$

$$\mathsf{thread}(d_m, w, [\eta], \mathsf{work}\ \{r\}\ ;\ P) \longmapsto^r\ \mathsf{thread}(d_m, w - r, [\eta], P)$$

Note that $\mathcal{C} \longmapsto^q\ \mathcal{C}'$ represents that the step costs q potential; we assume that $\mathcal{C} \longmapsto\ \mathcal{C}'$ represents a step that does not cost potential.

In incorporating ergometric types, processes and continuations now need to be annotated simultaneously with potential and with environments. We generalize the contents of our cells to $W := V \mid [\eta]\{q\}K$, where both the environment η and the potential annotation q are passed to the continuing process at runtime. Meanwhile, threads are now defined as $\mathsf{thread}(c_m, q, [\eta], P), \mathsf{cell}(c_m, -, n)$, where q represents the potential available to the thread and η is the environment substituting addresses for variables. These updates require minor and straightforward changes to our configuration typing rules for continuations and threads:

$$\frac{\Psi_1 \vdash \eta : \Gamma \qquad \Gamma \vdash^q \mathsf{case}\ x\ \{K\} :: (x : A_m) \qquad \Psi = \Psi_1 + \Psi_2 \qquad q = 0\ \text{if}\ m = \mathsf{U}}{\Psi \Vdash^q \mathsf{cell}(c, [\eta]\{q\}K, \sigma) :: \Psi_2, (c^\sigma : A_m)}\ \text{C:}\textsc{Cont}$$

$$\frac{\Psi_1 \vdash \eta : \Gamma \qquad \Gamma \vdash^q P :: (x_m : A_m) \qquad \Psi = \Psi_1 + \Psi_2}{\Psi \Vdash^q \mathsf{thread}(c_m, q, [\eta, c_m/x_m], P), \mathsf{cell}(c_m, -, n) :: \Psi_2, (c_m^n : A_m)}\ \text{C:}\textsc{Thread}$$

In the C:Cont rule, we enforce that the potential associated with a continuation K must be exactly the potential needed in its typing. We also require that unrestricted continuations may not require any potential, since this would violate the inherent linearity of potential. In the C:Thread rule, we simply annotate our threads with the cost that they require.

We must also augment our value typing rules with a case for potential, since we may have cells storing

values like pot $q\ (a)$ with a being a direct address:

$$\frac{}{\Psi_W + (a^1 : A_\mathsf{L}) \vdash^q \mathsf{pot}\ q\ (a) : \triangleright^q A_\mathsf{L}}\ \textsc{Val-}\triangleright\textsc{R}^0$$

For the most part, other rules in our multilinear type system do not change substantially, apart from annotating typing judgments and continuations with potential. For instance, the rules for cut require us to split up potential between the two resulting processes, just as we split up contexts and environments; the dynamics step becomes

$$\begin{aligned} &\mathsf{thread}(c_m, w_1 + w_2, [\eta_1 + \eta_2, c_m/y], x \leftarrow^{w_1} P\ ;\ Q) \\ \longmapsto\ &\mathsf{thread}(a, w_1, [\eta_1, a/x], P), \mathsf{cell}(a, -, \sigma), \mathsf{thread}(c_m, w_2, [\eta_2, a/x, c_m/y], Q)\ (a\ \mathsf{fresh}) \end{aligned}$$

We modify other rules similarly, with the full dynamics shown in Appendix C.

5.2 Multilinear Potentials

The most significant challenge in introducing ergometric types to a multilinear system is that potential is inherently linear; it cannot be reused if we hope to track cost accurately [20, 21]. Since cost is linear, we restrict potential to the linear mode of our language; the nonlinear mode cannot involve any notions of cost.

We extend our type grammar for *multilinear types* with the one that stores potential:

$$Q, R ::= \ldots \mid \triangleright^q Q_\mathsf{L} \qquad\qquad \text{value with potential } q$$

Data structures of this type may be inductively copied and discarded, as long as we are precise about tracking the potential; thus, we can also alias and drop them through reference counting. Notably, we do not include $\triangleleft^q A_\mathsf{L}$ in our multilinear grammar; because it is a negative type, it would not make sense in our current model to consider this a multilinear type. It cannot be inductively copied or dropped, nor can the address be automatically shared.

The primary complexity involved in introducing multilinear types with potential is that an address containing potential should be able to share its potential across its many references, but we treat potential as purely linear. This means that we cannot always alias an address simply by creating two references of identical type; instead, we must split the available potential between them. To account for this, we introduce a relation between types that allows us to appropriately split a single type into two (possibly different) types

$$\frac{}{\mathbf{1} = (\mathbf{1} + \mathbf{1})} \qquad \frac{Q = (R + S) \qquad Q' = (R' + S')}{Q \otimes Q' = (R \otimes R' + S \otimes S')} \qquad \frac{Q = (R + S) \text{ for all } l \in L}{\oplus\{l : Q^l\}_{l \in L} = (\oplus\{l : R^l\}_{l \in L} + \oplus\{l : S^l\}_{l \in L})}$$

$$\frac{}{\downarrow A_{\mathsf{U}} = (\downarrow A_{\mathsf{U}} + \downarrow A_{\mathsf{U}})} \qquad \frac{Q = (R + S) \qquad q_1 = q_2 + q_3}{\triangleright^{q_1} Q = (\triangleright^{q_2} R + \triangleright^{q_3} S)}$$

Figure 5.1: Cyclic Proof System for Splitting Types

[17, 18]. Specifically, $Q = (R + S)$ means that a variable with type Q may be aliased as two variables with types R and S. The definition is provided in Fig. 5.1. This relation is defined only on multilinear types. The crux of this judgment is in the splitting of potential type: a type carrying potential is required to exactly divide the potential between its two new references. As an example, $\triangleright^8 \mathbf{1}_{\mathsf{L}} = \triangleright^2 \mathbf{1}_{\mathsf{L}} + \triangleright^6 \mathbf{1}_{\mathsf{L}} = \triangleright^2 \mathbf{1}_{\mathsf{L}} + \triangleright^1 \mathbf{1}_{\mathsf{L}} + \triangleright^5 \mathbf{1}_{\mathsf{L}}$. We could also perform this kind of splitting on a recursive type; for instance, `bin{5} = bin{3} + bin{2}`, allowing us to split up the potential in each bit of a binary number. Notably, we allow circular derivations to account for the structure of equirecursive type definitions.

We then need to update our definition of combining address contexts correspondingly; it is no longer the case that multiple occurrences of the same address will have the same type. In particular, the rule for joining contexts with multilinear addresses becomes

$$\frac{\Psi = \Psi_1 + \Psi_2 \qquad Q_{\mathsf{L}} = (R_{\mathsf{L}} + S_{\mathsf{L}}) \qquad Q_{\mathsf{L}} = R_{\mathsf{L}} \text{ if } n = 0 \qquad Q_{\mathsf{L}} = S_{\mathsf{L}} \text{ if } k = 0}{\Psi, (a^{k+n} : Q_{\mathsf{L}}) = (\Psi_1, (a^k : R_{\mathsf{L}})) + (\Psi_2, (a^n : S_{\mathsf{L}}))} \text{ P5}$$

Building on our previous example of types that can be added, we have that $(a^3 : \triangleright^8 \mathbf{1}_{\mathsf{L}}) = (a^1 : \triangleright^2 \mathbf{1}_{\mathsf{L}}) + (a^2 : \triangleright^6 \mathbf{1}_{\mathsf{L}}) = (a^1 : \triangleright^2 \mathbf{1}_{\mathsf{L}}) + (a^1 : \triangleright^1 \mathbf{1}_{\mathsf{L}}) + (a^1 : \triangleright^5 \mathbf{1}_{\mathsf{L}})$. In this rule, we notably enforce that if our reference count becomes 0, we must use up all the potential; this is because our system should not allow us to waste potential. If we allowed a cell with reference count 0 to store potential, this would violate the invariant that the sum of the potentials used by a cell's clients is equal to the potential it carries. Because the work construct allows us to explicitly use up potential, this should not be an issue in our system.

This same reasoning also means that we must require $x : Q_{\mathsf{L}}$ not to have potential in order to drop it; we do not want to lose potential when we drop a cell or when a process terminates. To enforce this, we introduce a nullary version of the splitting relation on types. This definition is straightforward (although, like for the splitting relation, derivations may be circular) and may be found in Fig. 5.2; any multilinear type may be dropped as long as it does not store potential.

$$\frac{}{\mathbf{1} = ()} \qquad \frac{Q = () \qquad Q' = ()}{Q \otimes Q' = ()} \qquad \frac{Q^l = () \text{ for all } l \in L}{\oplus\{l : Q^l\}_{l \in L} = ()} \qquad \frac{}{\downarrow A_{\mathsf{U}} = ()} \qquad \frac{Q = () \qquad q = 0}{\triangleright^q Q = ()}$$

Figure 5.2: Cyclic Proof System for Dropping Types

We can now update the alias and drop constructs by parameterizing them on types:

$$\frac{Q_{\mathsf{L}} = (R_{\mathsf{L}} + S_{\mathsf{L}}) \qquad \Gamma, (y : R_{\mathsf{L}}), (z : S_{\mathsf{L}}) \vdash^q P :: (w : C_r)}{\Gamma, (x : Q_{\mathsf{L}}) \vdash^q \mathsf{alias}\ x : Q_{\mathsf{L}}\ \mathsf{as}\ y : R_{\mathsf{L}}, z : S_{\mathsf{L}}\ ;\ P :: (w : C_r)}\ \textsc{Alias}$$

$$\frac{Q_{\mathsf{L}} = () \qquad \Gamma \vdash^q P :: (w : C_r)}{\Gamma, (x : Q_{\mathsf{L}}) \vdash^q \mathsf{drop}\ x : Q_{\mathsf{L}}\ ;\ P :: (w : C_r)}\ \textsc{Drop}$$

The dynamics for aliasing and dropping are largely unaffected by the introduction of potential.

Lastly, we revisit the dynamics for reading from a cell containing potential with a reference count of more than one. This rule is non-trivial in that we cannot assume that all of the potential will be immediately used by a single reader. Instead, we modify the potential stored in the cell by subtracting whatever potential is used up, as shown below:

$$\begin{array}{l} \quad \mathsf{cell}(c'_{\mathsf{L}}, D, n'), \mathsf{cell}(c_{\mathsf{L}}, \mathsf{pot}\ q\ (c'_{\mathsf{L}}), n), \mathsf{thread}(d_m, q', [\eta, c_{\mathsf{L}}/y_{\mathsf{L}}], \mathsf{case}\ y_{\mathsf{L}}\ \{\mathsf{pot}\ q''\ (x_{\mathsf{L}}) \Rightarrow P\}) \\ \longmapsto\ \mathsf{cell}(c'_{\mathsf{L}}, D, n' \oplus 1), \mathsf{cell}(c_{\mathsf{L}}, \mathsf{pot}\ (q - q'')\ (c'_{\mathsf{L}}), n - 1), \mathsf{thread}(d_m, q' + q'', [\eta, c'_{\mathsf{L}}/x_{\mathsf{L}}], P) \end{array}$$

The described modifications produce a system that supports resource tracking for both linear and multilinear types. The complete process typing and dynamics rules may be found in Appendix B and Appendix C. Notably, introducing ergometric types makes the system much more complex, as seen in the following example, but this syntax is intended as an internal intermediate representation rather than something a programmer would actually write. In the presented system, cost annotations are explicit, but we expect that there would be an easy-to-use functional syntax on top of this internal language. The user-facing language could support a variety of properties. One possibility would be a concrete syntax in which resource bounds are inferred automatically, as in Resource-Aware ML [18]; such a language would be an advancement over RAML in adding parallelism through futures and supporting a linear/multilinear/non-linear distinction. An alternate possibility would be a Rast-like syntax, in which the programmer supplies some limited information about resource usage [9]; however, this information would certainly not be as heavy-handed as

what is currently expected in this intermediate language.

5.3 Example: Tries of Multilinear, Ergometric Binary Numbers

We attempt once more to reuse a binary number, this time using both multilinear and ergometric types. For the purposes of this example, we will assume a cost model that requires 1 unit of potential per `read` or `write` operation.

Because binary numbers now store potential in each bit, they are costly to produce. For instance, storing the number 271 as `bin{p}` would require $9 * (p + 1) + 2$ units of potential, based on the number's nine digits. Though we assume that our binary numbers are already laid out in memory, we give the code for explicitly allocating a number with potential to demonstrate the exact cost.

```
decl b271{9*(p+1)+2} : . |- (x : bin{p})
proc x <- b271{9*(p+1)+2} =
  x1.e() ;   % 2 erg
  x2 <- x1.b1(x2) ; x2' <- x2.pot p (x2') ;
  x3 <- x2'.b0(x3) ; x3' <- x3.pot p (x3') ;
  ... allocate x4 through x8' ...
  x9 <- x8'.b1(x9) ; x9' <- x9.pot p (x9') ;
  x10 <- x9'.b1(x10) ; x.pot p (x10)
```

Here, we need two units of potential to write the closing digit (`e`), along with its associated unit. In addition, we need one unit of potential per bit to write a `b0` or `b1` label and p units of potential to store in each bit.

We also annotate our `trie` type with potential:

```
type trie = &{ update : <{7}| bin{11} * bool -o trie, ... }
```

Both node and leaf processes need an intrinsic potential of 1 unit so they can write their continuation to memory, which we write as `node{1}` and `leaf{1}`. After an `update` label is passed to the written continuation, we need to supply it with 7 units of potential. On the critical path, this accounts for 4 units to get the input and 3 units to create a new leaf. Furthermore, every bit must carry 11 units of potential: the critical path requires 3 to get the input, 5 to create a new node, and 3 to make a recursive call. Note that transfer of potential is "free"; since we prove that they cannot become negative, and no computationally relevant decisions depend on them, potentials can be erased before executing the program.

The code is shown below. Instead of nested match expressions we write nested patterns, and we similarly compress nested allocations. These can easily be expanded into our official syntax, but this would make the

code even less readable. All potentials are enclosed in {..}.

```
decl (l : trie) (b : bool) (r : trie) |- node{1} : (t : trie)
decl . |- leaf{1} : (t : trie)

proc t <- node{1} l b r =                    % +1
  case t ( update (pot{7} ((x,c),t')) =>  % +7 -3
    case x ( pot{11} b0(y)                  % +11 -1
           => l' <- l.update(pot{7} ((y,c),l')) ;
                                            % -7 -3
              work{4} ;                     % -4
              t' <- node{1} l' b r          % -1
           | pot{11} b1(y) => ...symmetric...
           | e() => drop b ;                % -2
                    work{2} ;               % -2
                    t' <- node{1} l c r     % -1
           ))

proc t <- leaf{1} =                          % +1
  case t ( update (pot{7} ((x,c),t')) =>  % +7 -3
    case x ( pot{11} b0(y)                  % +11 -1
           => l <- leaf{1} ;                % -1
              r <- leaf{1} ;                % -1
              l' <- l.update(pot{7} ((y,c),l')) ;
                                            % -7 -3
              t' <- node{1} l' (false ()) r
                                            % -1 -2
           | pot{11} b1(y) => ...symmetric...
           | e() => l <- leaf{1} ;          % -2 -1
                    r <- leaf{1} ;          % -1
                    t' <- node{1} l c r     % -1
           ))
```

Now, we assume that we have already expended $110 = 9 * (11 + 1) + 2$ units of potential to store the number 271 in memory with 11 units of potential per bit. We also assume that we have a trie `t : trie`. Consider a case in which we want to insert the number x into `t` and then remove it:

```
% x : bin{11}
% insert 271 into trie
t1 <- t.update((x,true()), t1) ;
% remove 271 from trie
t2 <- t1.update((x,false()), t2) ;
```

This code is notably invalid, since x should not be reused. To make this program possible, we must instead make two copies of the number 271 and insert one and delete the other.

Without multilinear types, we would first need to recursively copy x into y and z, as discussed in Section 3.2. However, this is expensive in our cost model, since we charge one unit of potential per read and

write. Given our potential annotations and our cost model, we would have a process for every `n` and `k` of type `copy{5} : bin{n+k+5} -o bin{n} * bin{k}`. Copying requires a constant intrinsic cost of 5 units of potential, as well as 5 extra units of potential per bit based on reading and writing costs and the cost of recursively calling `copy`. The full code is provided in Appendix A. In this case, we would take a binary number of potential $11 + 11 + 5 = 27$ per bit (which would cost 254 units of potential to allocate initially) and reference it with two variables of type `bin{11}`, incurring a total cost of 259 units of potential.

Using multilinearity, we can simply use aliasing to produce two references to the same binary number without incurring extraneous cost. Given a number of type `bin{22}` (which for 271 costs 209 units of potential to allocate), we can alias it into two variables, each of type `bin{11}`. Then, we can simply write the following code:

```
alias x : bin{22} as y : bin{11}, z : bin{11} ;
```

and we can perform both of our insertions using only one address:

```
% y : bin{11}, z : bin{11}
t1 <- t.update((y,true()), t1) ;
t2 <- t1.update((z,false()), t2) ;
```

Aliasing saves us the cost of traversing through the 9-bit number and explicitly copying it. This difference would be even more marked for longer bit strings and more complex data structures.

5.4 Soundness

We now state the soundness properties for our system. Representative cases of the proofs can be found in Appendix D. First, we state progress, or liveness, which expresses that any well-typed configuration is either final (i.e., it does not have any running threads) or can take a step. The step must require no more potential than the configuration has available to it.

Theorem 1 (Progress)**.** *If* $\cdot \Vdash^q \mathcal{C} :: \Psi$*, then either* $\mathcal{C}$ *final or* $\mathcal{C} \longmapsto^{q'} \mathcal{C}'$ *for some configuration* $\mathcal{C}'$ *and* $q' \leq q$*.*

We provide a proof sketch below:

Proof. We prove progress by right-to-left induction on $\mathcal{C}$.

Base Case Assume $\mathcal{C}$ is empty. Then, $\mathcal{C}$ is trivially final, since there are no threads.

Inductive Case Let $\mathcal{C} = \mathcal{C}_1, \mathcal{C}_2$ where either $\mathcal{C}_2 = \mathsf{cell}(d_m, W, n)$ or $\mathcal{C}_2 = \mathsf{thread}(d_m, q_2, [\eta], P), \mathsf{cell}(d_m, -, n)$.

We know by C:Join that $\cdot \Vdash^{q_1} \mathcal{C}_1 :: \Psi_1$ and $\Psi_1 \Vdash^{q_2} \mathcal{C}_2 :: \Psi$ for some Ψ_1, q_1, q_2 with $q_1 + q_2 = q$.

First, we apply the inductive hypothesis on $\mathcal{C}_1$. Either $\mathcal{C}_1 \longmapsto^{q_1'} \mathcal{C}_1'$ for some $\mathcal{C}_1'$ and $q_1' \leq q_1$ or $\mathcal{C}_1$ final. In the first case, by multiset rewriting, we know that $\mathcal{C} \longmapsto^{q_1'} \mathcal{C}_1', \mathcal{C}_2$; since $q_1' \leq q_1 \leq q$, the conclusion holds.

In the second case, $\mathcal{C}_1$ final. We proceed by casing on $\mathcal{C}_2$. If $\mathcal{C}_2 = \mathsf{cell}(d_m, W, n)$, then clearly $\mathcal{C}$ final, since $\mathcal{C}$ is made up only of cells.

If $\mathcal{C}_2 = \mathsf{thread}(d_m, q_2, [\eta], P), \mathsf{cell}(d_m, -, n)$, we have

$$\frac{\Psi_3 \vdash \eta : \Gamma \qquad \Gamma \vdash^{q_2} P :: (x_m : A_m) \qquad \Psi_1 = \Psi_3 + \Psi_2}{\Psi_1 \Vdash^{q_2} \mathsf{thread}(c_m, q_2, [\eta, c_m/x_m], P), \mathsf{cell}(c_m, -, n) :: \Psi_2, (c_m^n : A_m)} \text{ C:Thread}$$

We proceed by induction on the derivation of the second premise. In these cases of our proof, we make use of two key lemmas that express that in a final configuration $\mathcal{C}$, if $\cdot \Vdash^q \mathcal{C} :: \Psi$ with a channel $c \in \Psi$, then the cell c exists in the configuration $\mathcal{C}$ with an appropriate reference count.

□

Next, we state preservation, or safety, which expresses that if a well-typed configuration makes a step, the resulting configuration retains the same typing. Notably, if the step uses up some potential w and the original configuration had potential $q + w$ available to it, the resulting configuration has q potential remaining.

Theorem 2 (Preservation)**.** *If $\Psi \Vdash^{q+w} \mathcal{C} :: \Psi'$, and $\mathcal{C} \longmapsto^{w} \mathcal{C}'$ for some configuration $\mathcal{C}'$, then $\Psi \Vdash^{q} \mathcal{C}' :: \Psi'$.*

The proof of preservation is complex and tedious, so we provide key points below and formalize it in Appendix D.

Proof. Preservation may be proven by induction on $\mathcal{C} \longmapsto^{w} \mathcal{C}'$, using inversions on configuration and process typing. To prove preservation, we must prove and use lemmas about the shape of contexts. For instance, we prove that if we "add" or "remove" references to an address on both the left and right sides of a configuration typing judgment, the resulting judgment is still valid. Using these lemmas, we proceed by cases, considering each individual rule and the various possibilities for its reference count. □

Chapter 6

Conclusion

In this paper, we have presented a core language in which linear/multilinear/nonlinear types coexist to reap the benefits of linearity. In particular, this multilinear, resource-aware language offers possibilities for efficient functional programming through futures. We began with a concurrent language with two modes: an nonlinear mode and a linear mode. Next, we departed from prior work in introducing the idea of multilinearity, in which data of certain types may be aliased or dropped simply by modifying a reference count. In particular, we focused on the distinction between variables and addresses and on maintaining accurate reference counts that signify the number of clients that addresses have. Finally, we introduced cost tracking through potential, discussing how potential annotations are affected by multilinearity. We developed a technique to share potential across multiple references to an address. Throughout these different stages, we illustrated the effectiveness of our language using the example of binary numbers and tries.

6.1 Related Work

There has been substantial work in recent years on the benefits and challenges of linearity. The basis for our research is foundational work on linear logic, including that by Girard [14], Lafont [15], and Wadler [33]. We are also inspired substantially by work on traditional session types [22, 6]. The same syntax as we present here could be alternatively interpreted in terms of message passing so that, modulo terminology and actual cost of operations, the type system would apply; however, we would not a priori expect similar benefits for lower-level efficiency due to different relative costs of local and message-passing operations.

Recently, there have been investigations into futures, as presented by Halstead [16], as a technique for

greater functional efficiency. For instance, we draw from work by Pruiksma and Pfenning on an adjoint logic and its use for futures [30] for our preliminary language. Work by Blelloch and Reid-Miller has demonstrated several uses for linear and nonlinear futures and the asymptotic speedups that they can offer [5]. Futures have also been adopted in numerous modern programming languages, including Scala [12], Python [27], and Java [23].

Our work additionally draws upon to other research on potential and cost-tracking. Das et al. recently demonstrated the use of arithmetic refinements and work analysis for session-typed systems [8, 9, 10]. Meanwhile, Resource-Aware ML (RaML) automatically derives resource bounds for OCaml programs based on linearity and potential [17, 18]. The notion of potential in RaML is similar to the one we use here, but RaML implements fully automatic resource tracking with some limitations. Our work could be used either manually or in a RaML-like context; it may be possible to elaborate a language like RaML into our intermediate language. Significant work has also focused on the automation of granularity control based on execution times; these techniques could be used to further improve efficiency once cost bounds are identified. For example, some previous work has explored the use of machine learning to connect high-level cost bounds with actual execution time [7], and other research has experimented with "oracle-guided scheduling," in which granularity is controlled by an "oracle" that predicts actual execution times [1].

Finally, recent work has focused on using linearity to improve the efficiency of memory management. In particular, work on Perceus by Reinking et al. has demonstrated how reference counting inspired by linearity can enable efficient, garbage-free memory management [31]. Perceus focuses on optimizing the reuse of memory and freeing memory as soon as it has no more references. The dynamics for this language include "dup" and "drop" constructs that increment and decrement reference counts. When a reference count reaches 0, the cell is deallocated and drop is recursively called. Though Perceus does not focus on parallelism and does not include resource tracking, this work suggests that the multilinear language we have developed would enjoy benefits for garbage collection akin to those achieved by pure linearity.

6.2 Future Work

This research presents many opportunities for future exploration. The most critical next step is achieving a complete implementation of this language, including an elaboration from a more usable surface syntax. Specifically, since this is intended to represent an intermediate language, we anticipate that we will be able to compile a functional language down to this language (with most operations remaining sequential, but

some occurring in parallel), and then compile this language to machine code. A complete implementation would allow us to examine the potential efficiency benefits from a practical perspective. Currently, without a compiler, it is challenging to accurately assess runtime speed and cost; it becomes difficult to determine whether a noticed speedup can be attributed to the program itself or merely to the interpreter for the language. Introducing a compiler, though challenging due to the difficulty in scheduling futures, would allow us to gain a more precise understanding of cost and speedups.

We also hope to add support for general arithmetic refinements. This would allow us to add *recharge* and *discharge* primitives with no runtime cost that add to or remove potential from data structures without traversing them. This would be analogous to our step from explicit copying and discarding to aliasing and dropping through reference counting.

Another direction that we leave for later work is an investigation into greater automation of granularity control. Even in the presence of ergometric types, a programmer must determine an appropriate grain experimentally, based on the specific machine and architecture. Because such a decision is both program- and machine- dependent, it is not immediately clear how to automate it without experimental trials. Further examination of granularity control techniques could augment our work on efficient parallelism.

Bibliography

[1] Umut A. Acar, Vitaly Aksenov, Arthur Charguéraud, and Mike Rainey. Provably and practically efficient granularity control. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 214–228, New York, NY, USA, 2019. Association for Computing Machinery.

[2] Jean-Marc Andreoli. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation*, 2(3):297–347, June 1992.

[3] Jatin Arora, Sam Westrick, and Umut A. Acar. Provably space-efficient parallel functional programming. *Proceedings of the ACM on Programming Languages*, jan 2021.

[4] P. N. Benton. A mixed linear and non-linear logic: Proofs, terms and models. In Leszek Pacholski and Jerzy Tiuryn, editors, *Computer Science Logic*, pages 121–135, Berlin, Heidelberg, 09 1994. Springer Berlin Heidelberg.

[5] G. E. Blelloch and M. Reid-Miller. Pipelining with futures. *Theory of Computing Systems*, 32:213–239, 1999.

[6] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *Proceedings of the 21st International Conference on Concurrency Theory (CONCUR 2010)*, pages 222–236, Paris, France, August 2010. Springer LNCS 6269.

[7] Ankush Das and Jan Hoffmann. ML for ML: learning cost semantics by experiment. In A. Legay and T. Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference (TACAS 2017)*, volume 10205, pages 190–207, 2017.

[8] Ankush Das, Jan Hoffmann, and Frank Pfenning. Work analysis with resource-aware session types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, page 305–314, New York, NY, USA, 2018. Association for Computing Machinery.

[9] Ankush Das and Frank Pfenning. Rast: Resource-aware session types with arithmetic refinements. In Z. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, pages 4:1–4:17. LIPIcs 167, June 2020. System description.

[10] Ankush Das and Frank Pfenning. Session Types with Arithmetic Refinements. In Igor Konnov and Laura Kovács, editors, *31st International Conference on Concurrency Theory (CONCUR 2020)*, volume 171 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:18, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. URL: `https://drops.dagstuhl.de/opus/volltexte/2020/12825`, `doi:10.4230/LIPIcs.CONCUR.2020.13`.

[11] Henry DeYoung, Frank Pfenning, and Klaas Pruiksma. Semi-Axiomatic Sequent Calculus. In Z. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, volume 167 of *LIPIcs*, pages 29:1–29:22, 2020.

[12] Futures and Promises. URL: `https://docs.scala-lang.org/overviews/core/futures.html`.

[13] Simon Gay and Malcolm Hole. Subtyping for session types in the pi calculus. *Acta Inf.*, 42(2–3):191–225, November 2005.

[14] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.

[15] Jean-Yves Girard and Yves Lafont. Linear logic and lazy computation. In *TAPSOFT*, volume 2, page 52–66, 1987.

[16] Robert H. Halstead. Multilisp: A language for parallel symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–539, October 1985.

[17] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Resource Aware ML. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV '12, page 781–786, Berlin, Heidelberg, 2012. Springer-Verlag.

[18] Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for OCaml. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 359–373, New York, NY, USA, 2017. Association for Computing Machinery.

[19] Martin Hofmann and Steffen Jost. Static prediction of heap space usage for first-order functional programs. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2003, page 185–197, New York, NY, USA, 2003. Association for Computing Machinery.

[20] Martin Hofmann and Steffen Jost. Type-based amortised heap-space analysis. In Peter Sestoft, editor, *Programming Languages and Systems*, pages 22–37, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[21] Martin Hofmann and Dulma Rodriguez. Efficient type-checking for amortised heap-space analysis. In *Proceedings of the 23rd CSL International Conference and 18th EACSL Annual Conference on Computer Science Logic*, CSL'09/EACSL'09, page 317–331, Berlin, Heidelberg, 2009. Springer-Verlag.

[22] Kohei Honda. Types for dyadic interaction. In E. Best, editor, *4th International Conference on Concurrency Theory (CONCUR 1993)*, pages 509–523. Springer LNCS 715, 1993.

[23] Interface Future, Java Platform Standard Edition 7. URL: `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html`.

[24] Olivier Laurent. Polarized proof-nets: Proof-nets for lc. In *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications*, TLCA '99, page 213–227, Berlin, Heidelberg, 1999. Springer-Verlag.

[25] Olivier Laurent. Syntax vs. semantics: A polarized approach. *Theoretical Computer Science*, 343(1):177–206, 2005.

[26] José E. Moreira, Dale Schouten, and Constantine D. Polychronopoulos. The performance impact of granularity control and functional parallelism. In *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, LCPC '95, pages 581–597, Berlin, Heidelberg, 1995. Springer-Verlag.

[27] PEP 3148 – Futures, 2009. URL: `https://www.python.org/dev/peps/pep-3148/`.

[28] Klaas Pruiksma and Frank Pfenning. A shared-memory semantics for mixed linear and non-linear session types. Submitted, November 2018.

[29] Klaas Pruiksma and Frank Pfenning. A message-passing interpretation of adjoint logic. In F. Martins and D. Orchard, editors, *Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software*, pages 60–79, Prague, Czech Republic, April 2019. EPTCS 291.

[30] Klaas Pruiksma and Frank Pfenning. Back to futures. *Journal of Functional Programming*, 2022. To appear. URL: `http://arxiv.org/abs/2002.04607`.

[31] Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. Perceus: Garbage free reference counting with reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 96–111, New York, NY, USA, 2021. Association for Computing Machinery.

[32] Alexandros Tzannes, George C. Caragea, Uzi Vishkin, and Rajeev Barua. Lazy scheduling: A runtime adaptive scheduler for declarative parallelism. *ACM Transactions on Programming Languages and Systems*, 36(3), September 2014.

[33] Philip Wadler. Linear types can change the world! In *TC 2 Working Conference on Programming Concepts and Methods*, pages 546–566. North-Holland, 1990.

Appendix A

Tries and Binary Numbers

For this example, we use a simplified syntax, annotating certain condensed lines with their cost. We use `erg` to represent a unit of potential. For instance, something of the form

```
l1 <- l.update(l1) ;    % 1 erg
p <- p.(x, b) ;         % 1 erg
p' <- p.(pot{7} p') ;
l2 <- l1.(p', l2) ;     % 1 erg
```

in the conventional syntax is simplified into

```
l2 <- l.update(pot{7} (x, b), l2) ;    % 3 erg
```

In much of the following code, we track updates to potential in a comment on each line.

A.1 Binary Numbers

```
%%% bin{p} = binary numbers where each bit carries potential p
type bin{p} = +{ b0 : |{p}> bin{p},
                 b1 : |{p}> bin{p},
                 e : 1 }
```

A.1.1 Dropping and Copying

```
%%% discard{2} recursively deallocates a binary number
```

```
decl discard{2} : (x : bin{p+1}) |- (u : 1)
proc u <- discard{2} x =                          % +2
  case x ( pot{p+1} (b0(x')) =>                   % -1 +p+1
             work p ;                             % -p
             u <- discard{p} x'                   % -2
         | pot{p+1} (b1(x')) =>                   % -1 +p+1
             work p ;                             % -p
             u <- discard{p} x'                   % -2
         | e(x')  => u <-> x'                     % -2
         )

%%% copy{5} recursively copies a binary number,
%%% splitting the potential
decl copy{5} : (x : bin{n+k+5}) |- (p : bin{n} * bin{k})
proc p <- copy{5} x =                             % +5
  case x ( b0(pot{n+k+5} (x')) =>                 % -1 +n+k+5
           p' <- copy{5} x' ;                     % -5
           case p' ( (p1, p2) =>                  % -1
             p1' <- p1.(pot{n} (b0(p1'))) ;       % -1 -n
             p2' <- p2.(pot{k} (b0(p2'))) ;       % -1 -k
             p.(p1', p2')                         % -1
           )
         | b1(pot{n+k+5} (x')) =>                 % -1 +n+k+5
           p' <- copy{5} x' ;                     % -5
           case p' ( (p1, p2) =>                  % -1
             p1' <- p1.(pot{n} (b1(p1'))) ;       % -1
             p2' <- p2.(pot{k} (b1(p2'))) ;       % -1
             p.(p1', p2')                         % -1
           )
         | e() => p.((), ())                      % -2 -3
         )
```

A.2 Tries

```
type bool = +{ true : 1, false : 1 }
type trie = &{ update : <{7}| bin{11} * bool -o trie, ... }

decl (l : trie) (b : bool) (r : trie) |- node{1} : (t : trie)
decl . |- leaf{1} : (t : trie)

proc t <- node{1} l b r =                         % +1
  case t ( update (pot{7} ((x,c),t')) =>          % +7 -3
    case x ( pot{11} b0(y)                        % +11 -1
             => l' <- l.update(pot{7} ((y,c),l')) ;
                                                  % -7 -3
                work{4} ;                         % -4
                t' <- node{1} l' b r              % -1
           | pot{11} b1(y)                        % +11 -1
```

```
            => r' <- r.update(pot{7} ((y,c),r')) ;
                                          % -7 -3
               work{4} ;                  % -4
               t' <- node{1} l b r'       % -1
          | e() => drop b ;               % -2
                   work{2} ;              % -2
                   t' <- node{1} l c r    % -1
          )
  )

proc t <- leaf{1} =                       % +1
  case t ( update (pot{7} ((x,c),t')) =>  % +7 -3
    case x ( pot{11} b0(y)                % +11 -1
            => l <- leaf{1} ;             % -1
               r <- leaf{1} ;             % -1
               l' <- l.update(pot{q} ((y,c),l')) ;
                                          % -7 -3
               t' <- node{1} l' (false ()) r
                                          % -1 -2
          | pot{11} b1(y)                 % +11 -1
            => l <- leaf{1} ;             % -1
               r <- leaf{1} ;             % -1
               r' <- r.update(pot{q} ((y,c),r')) ;
                                          % -7 -3
               t' <- node{1} l (false ()) r'
                                          % -1 -2
          | e() => l <- leaf{1} ;         % -2 -1
                   r <- leaf{1} ;         % -1
                   t' <- node{1} l c r    % -1
          )
  )
```

Appendix B

Statics

B.1 Judgment for Combining Environments

$\boxed{\eta = \eta_1 + \eta_2}$

$$\frac{}{\eta_1 = \eta_1 + []} \qquad \frac{}{\eta_2 = [] + \eta_2} \qquad \frac{\eta = \eta_1 + \eta_2}{\eta, a_{\mathsf{L}}/x_{\mathsf{L}} = (\eta_1, a_{\mathsf{L}}/x_{\mathsf{L}}) + \eta_2} \qquad \frac{\eta = \eta_1 + \eta_2}{\eta, a_{\mathsf{L}}/x_{\mathsf{L}} = \eta_1 + (\eta_2, a_{\mathsf{L}}/x_{\mathsf{L}})}$$

$$\frac{\eta = \eta_1 + \eta_2}{\eta, a_{\mathsf{U}}/x_{\mathsf{U}} = (\eta_1, a_{\mathsf{U}}/x_{\mathsf{U}}) + (\eta_2, a_{\mathsf{U}}/x_{\mathsf{U}})}$$

B.2 Substitutions Used for Calling Processes

$\boxed{\Gamma \vdash \zeta : \Delta}$

$$\frac{}{\Gamma_W \vdash (\cdot) : (\cdot)}\ \text{Z1} \qquad \frac{\Gamma \vdash \zeta : \Delta}{\Gamma + (x_m : A_m) \vdash \zeta, x_m/x'_m : \Delta, (x'_m : A_m)}\ \text{Z2}$$

$\boxed{\Psi \vdash \eta \circ \zeta : \Delta}$

$$\frac{\Psi \vdash \eta : \Gamma \qquad \Gamma \vdash \zeta : \Delta}{\Psi \vdash \eta \circ \zeta : \Delta}\ \textsc{Clos-Comp}$$

B.3 Process Typing Rules

$$\frac{m \geq r \qquad \Gamma_1 \vdash^p P :: (x : A_m) \qquad \Gamma_2, (x : A_m) \vdash^q Q :: (y : B_r) \qquad \Gamma = \Gamma_1 + \Gamma_2}{\Gamma \vdash^{p+q} x \leftarrow^p P \; ; \; Q :: (y : B_r)} \textsc{Cut}$$

$$\frac{\Gamma \vdash \zeta : \Delta \qquad \Delta \vdash^q P :: (x : A_m) \in \Sigma}{\Gamma \vdash^q z \leftarrow p[\zeta] :: (z : A_m)} \textsc{Call} \qquad \frac{}{\Gamma_W + (y : A_m) \vdash^0 x \leftarrow y :: (x : A_m)} \textsc{Id}$$

$$\frac{Q_L = (R_L + S_L) \qquad \Gamma, (y : R_L), (z : S_L) \vdash^q P :: (w : C_r)}{\Gamma, (x : Q_L) \vdash^q \mathsf{alias}\ x : Q_L\ \mathsf{as}\ y : R_L, z : S_L \; ; \; P :: (w : C_r)} \textsc{Alias} \qquad \frac{Q_L = () \qquad \Gamma \vdash^q P :: (w : C_r)}{\Gamma, (x : A_L) \vdash^q \mathsf{drop}\ x : Q_L \; ; \; P :: (w : C_r)} \textsc{Drop}$$

$$\frac{i \in L}{\Gamma_W + (y : A^i_m) \vdash^0 x.i(y) :: (x : \oplus_m \{l : A^l_m\}_{l \in L})} \oplus\mathrm{R}^0$$

$$\frac{\Gamma, (y : A^l_m) \vdash^q Q_L :: (z : C_r)\ (\text{for all } l \in L)}{\Gamma + (x : \oplus_m \{l : A^l_m\}_{l \in L}) \vdash^q \mathsf{case}\ x\ \{l(y) \Rightarrow Q_L\}_{l \in L} :: (z : C_r)} \oplus\mathrm{L}$$

$$\frac{}{\Gamma_W + (w : A_m) + (y : B_m) \vdash^0 x.\langle w, y\rangle :: (x : A_m \otimes_m B_m)} \otimes\mathrm{R}^0$$

$$\frac{\Gamma, (w : A_m), (y : B_m) \vdash^q Q :: (z : C_r)}{\Gamma + (x : A_m \otimes_m B_m) \vdash^q \mathsf{case}\ x\ \{\langle w, y\rangle \Rightarrow Q\} :: (z : C_r)} \otimes\mathrm{L} \qquad \frac{\Gamma \vdash^q Q_L :: (y : A^l_m)\ (\text{for all } l \in L)}{\Gamma \vdash^q \mathsf{case}\ x\ \{l(y) \Rightarrow Q_L\}_{l \in L} :: (x : \&_m \{l : A^l_m\}_{l \in L})} \&\mathrm{R}$$

$$\frac{i \in L}{\Gamma_W + (x : \&_m \{l : A^l_m\}_{l \in L}) \vdash^0 x.i(y) :: (y : A^i_m)} \&\mathrm{L}^0 \qquad \frac{\Gamma, (w : A_m) \vdash^q Q :: (y : B_m)}{\Gamma \vdash^q \mathsf{case}\ x\ \{\langle w, y\rangle \Rightarrow Q\} :: (x : A_m \multimap_m B_m)} \multimap\mathrm{R}$$

$$\frac{}{\Gamma_W + (w : A_m) + (x : A_m \multimap_m B_m) \vdash^0 x.\langle w, y\rangle :: (y : B_m)} \multimap\mathrm{L}^0$$

$$\frac{}{\Gamma_W \vdash^0 x.\langle\rangle :: (x : \mathbf{1}_m)} \mathbf{1}\mathrm{R}^0 \qquad \frac{\Gamma \vdash^q Q :: (z : C_r)}{\Gamma + (x : \mathbf{1}_m) \vdash^q \mathsf{case}\ x\ \{\langle\rangle \Rightarrow Q\} :: (z : C_r)} \mathbf{1}\mathrm{L}$$

$$\frac{\Gamma \vdash^0 Q :: (y : A_L)}{\Gamma \vdash^0 \mathsf{case}\ x\ \{\mathsf{shift}(y) \Rightarrow Q\} :: (x : \uparrow A_L)} \uparrow\mathrm{R} \qquad \frac{}{\Gamma_W + (x : \uparrow A_L) \vdash^0 x.\mathsf{shift}(y) :: (y : A_L)} \uparrow\mathrm{L}^0$$

$$\frac{}{\Gamma_W + (y : A_U) \vdash^0 x.\mathsf{shift}(y) :: (x : \downarrow A_U)} \downarrow\mathrm{R}^0 \qquad \frac{\Gamma, (y : A_U) \vdash^q Q :: (z : C_r)}{\Gamma, (x : \downarrow A_U) \vdash^q \mathsf{case}\ x\ \{\mathsf{shift}(y) \Rightarrow Q\} :: (z : C_r)} \downarrow\mathrm{L}$$

$$\frac{}{\Gamma_W, (y : A_L) \vdash^q x.\mathsf{pot}\ q\ (y) :: (x : \triangleright^q A_L)} \triangleright\mathrm{R}^0 \qquad \frac{\Gamma, (y : A_L) \vdash^{q+p} Q :: (z : C_r)}{\Gamma, (x : \triangleright^p A_L) \vdash^q \mathsf{case}\ x\ \{\mathsf{pot}\ p\ (y) \Rightarrow Q\} :: (z : C_r)} \triangleright\mathrm{L}$$

$$\frac{\Gamma \vdash^{q+p} P :: (x : A_L)}{\Gamma \vdash^p \mathsf{case}\ y\ \{\mathsf{pot}\ q\ (x) \Rightarrow P\} :: (y : \triangleleft^q A_L)} \triangleleft\mathrm{R} \qquad \frac{}{\Gamma_W, (x : \triangleleft^q A_L) \vdash^q x.\mathsf{pot}\ q\ (y) :: (y : A_L)} \triangleleft\mathrm{L}^0$$

$$\frac{\Gamma \vdash^q P :: (x : A_m)}{\Gamma \vdash^{r+q} \mathsf{work}\ \{r\} \; ; \; P :: (x : A_m)} \textsc{Work}$$

Appendix C

Dynamics

$\boxed{\mathcal{C}\ \mathsf{final}}$

$$\frac{}{(\cdot)\ \mathsf{final}} \qquad \frac{\mathcal{C}\ \mathsf{final}}{\mathcal{C}, \mathsf{cell}(c_m, W, n)\ \mathsf{final}}$$

$\boxed{\mathcal{C} \longmapsto^{q} \mathcal{C}'}$

General Rules

$\mathsf{thread}(c_m, w_1 + w_2, [\eta_1 + \eta_2, c_m/y], x \leftarrow^{w_1} P\ ;\ Q), \mathsf{cell}(c_m, -, n)$
$\quad \longmapsto\ \mathsf{thread}(a, w_1, [\eta_1, a/x], P), \mathsf{cell}(a, -, \sigma),$
$\quad\quad \mathsf{thread}(c_m, w_2, [\eta_2, a/x, c_m/y], Q), \mathsf{cell}(c_m, -, n)$ (a fresh) *cut*

$\mathsf{thread}(c_m, q, [\eta, c_m/z], z \leftarrow p[\zeta]), \mathsf{cell}(c_m, -, n)$
$\quad \longmapsto\ \mathsf{thread}(c_m, q, [\eta \circ \zeta, c_m/x], P), \mathsf{cell}(c_m, -, n)$ (given $P = x \leftarrow p[\zeta]\ \in \Sigma$) *call*

$\mathsf{cell}(c_m, [\eta']\{p\}K, \sigma), \mathsf{thread}(d_m, 0, [(\eta + c_m/x_m), d_m/y_m], y_m \leftarrow x_m), \mathsf{cell}(d_m, -, \sigma)$
$\quad \longmapsto\ [\mathsf{cell}(c_m, [\eta']\{p\}K, \sigma)], \mathsf{cell}(d_m, [\eta' + \eta]\{p\}K, \sigma)$ *idK*

$\mathsf{thread}(d_m, w, [\eta], \mathsf{work}\ \{r\}\ ;\ P), \mathsf{cell}(d_m, -, n) \longmapsto^{r}\ \mathsf{thread}(d_m, w - r, [\eta], P), \mathsf{cell}(d_m, -, n)$ *work*

Alias and Drop

$\mathsf{cell}(c, V, n), \mathsf{thread}(d, q, [\eta, c/x], \mathsf{alias}\ x : \tau_1\ \mathsf{as}\ y : \tau_2, z : \tau_3\ ;\ P), \mathsf{cell}(d, -, n')$
$\quad \longmapsto\ \mathsf{cell}(c, V, n+1), \mathsf{thread}(d, q, [\eta, c/y, c/z], P), \mathsf{cell}(d, -, n')$ *alias*

$\mathsf{cell}(c, V, n), \mathsf{thread}(d, q, [\eta, c/x], \mathsf{drop}\ x : \tau\ ;\ P), \mathsf{cell}(d, -, n')$
$\quad \longmapsto\ \mathsf{cell}(c, V, n-1), \mathsf{thread}(d, q, [\eta], P), \mathsf{cell}(d, -, n')$ (if $n > 0$) *drop*

Positive Right Rules

$\mathsf{thread}(c_m, 0, [\eta, d_m/y, c_m/x], x_m.i(y)), \mathsf{cell}(c_m, -, n) \longmapsto \mathsf{cell}(c_m, i(d_m), n)$ $(\oplus R^0)$

$\mathsf{thread}(c_m, 0, [\eta, b_m/w, d_m/y, c_m/x], x.\langle w, y\rangle), \mathsf{cell}(c_m, -, n) \longmapsto \mathsf{cell}(c_m, \langle b_m, d_m\rangle, n)$ $(\otimes R^0)$

$\mathsf{thread}(c_m, 0, [\eta, c_m/x], x.\langle\rangle), \mathsf{cell}(c_m, -, n) \longmapsto \mathsf{cell}(c_m, \langle\rangle, n)$ $(\mathbf{1}R^0)$

$\mathsf{thread}(c, 0, [\eta, d/y, c/x], x.\mathsf{shift}(y)), \mathsf{cell}(c, -, n) \longmapsto \mathsf{cell}(c, \mathsf{shift}(d), n)$ $(\downarrow R^0)$

$\mathsf{thread}(c_L, q, [\eta, d_L/y_L, c_L/x_L], x_L.\mathsf{pot}\ q\ (y_L)), \mathsf{cell}(c_L, -, n) \longmapsto \mathsf{cell}(c_L, \mathsf{pot}\ q\ (d_L), n)$ $(\triangleright R^0)$

Negative Right Rules

$\mathsf{thread}(c_m, q, [\eta, c_m/x_m], \mathsf{case}\ x_m\ \{K\}), \mathsf{cell}(c_m, -, \sigma) \longmapsto \mathsf{cell}(c_m, [\eta]\{q\}K, \sigma)$ $(\multimap R, \& R, \uparrow R, \triangleleft R)$

Linear and Nonlinear Left Rules

$\mathsf{cell}(c_m, i(c'), \sigma), \mathsf{thread}(d_k, q, [(\eta + c_m/x_m), d_k/z], \mathsf{case}\ x_m\ \{(l(y) \Rightarrow P_l)_{l\in L}\})$
$\longmapsto [\mathsf{cell}(c_m, i(c'), \sigma)], \mathsf{thread}(d_k, q, [\eta, c'/y, d_k/z], P_i)$ $(\oplus L2)$

$\mathsf{cell}(c_m, \langle c', d'\rangle, \sigma), \mathsf{thread}(d_k, q, [(\eta + c_m/x_m), d_k/z], \mathsf{case}\ x_m\ \{\langle y, w\rangle \Rightarrow P\})$
$\longmapsto [\mathsf{cell}(c_m, \langle c', d'\rangle, \sigma)], \mathsf{thread}(d_k, q, [\eta, c'/y, d'/w, d_k/z], P)$ $(\otimes L2)$

$\mathsf{cell}(c_m, \langle\rangle, \sigma), \mathsf{thread}(d_k, q, [(\eta + c_m/x_m), d_k/z], \mathsf{case}\ x_m\ \{\langle\rangle \Rightarrow P\})$
$\longmapsto [\mathsf{cell}(c_m, \langle\rangle, \sigma)], \mathsf{thread}(d_k, q, [\eta, d_k/z], P)$ $(\mathbf{1}L2)$

$\mathsf{cell}(c_m, \mathsf{shift}(c'), 1), \mathsf{thread}(d_k, q, [(\eta + c_m/x_m), d_k/z], \mathsf{case}\ x_m\ \{\mathsf{shift}(y) \Rightarrow P\})$
$\longmapsto \mathsf{thread}(d_k, q, [\eta, c'/y, d_k/z], P)$ $(\downarrow L2)$

$\mathsf{cell}(c_L, \mathsf{pot}\ q'\ (c'), 1), \mathsf{thread}(d_k, q, [\eta, c_L/x_L, d_k/z], \mathsf{case}\ x_L\ \{\mathsf{pot}\ q'\ (y) \Rightarrow P\})$
$\longmapsto \mathsf{thread}(d_k, q + q', [\eta, c'/y, d_k/z], P)$ $(\triangleright L2)$

Quasilinear Left Rules

Note: All of these rules only apply when $n > 1$.

$\mathsf{cell}(a, D, n'), \mathsf{cell}(c_m, i(a), n), \mathsf{thread}(d_k, q, [\eta, c_m/x_m, d_k/z_k], \mathsf{case}\ x_m\ \{(l(y) \Rightarrow P_l)_{l\in L}\})$
$\longmapsto \mathsf{cell}(a, D, n' \oplus 1), \mathsf{cell}(c_m, i(a), n-1), \mathsf{thread}(d_k, q, [\eta, a/y, d_k/z_k], P_i)$ $(\oplus L)$

$\mathsf{cell}(a, D_1, n_1'), \mathsf{cell}(b, D_2, n_2'), \mathsf{cell}(c_m, \langle a, b\rangle, n), \mathsf{thread}(d_k, q, [\eta, c_m/x_m, d_k/z_k], \mathsf{case}\ x_m\ \{\langle y, z\rangle \Rightarrow P\})$
$\longmapsto \mathsf{cell}(a, D_1, n_1' \oplus 1), \mathsf{cell}(b, D_2, n_2' \oplus 1), \mathsf{cell}(c_m, \langle a, b\rangle, n-1),$
$\mathsf{thread}(d_k, q, [\eta, a/y, b/z, d_k/z_k], P)$ $(\otimes L)$

$\mathsf{cell}(c_m, \langle\rangle, n), \mathsf{thread}(d_k, q, [\eta, c_m/x_m, d_k/z_k], \mathsf{case}\ x_m\ \{\langle\rangle \Rightarrow P\})$
$\longmapsto \mathsf{cell}(c_m, \langle\rangle, n-1), \mathsf{thread}(d_k, q, [\eta, d_k/z_k], P)$ $(\mathbf{1}L)$

$\mathsf{cell}(a, D, n'), \mathsf{cell}(c_m, \mathsf{shift}(a), n), \mathsf{thread}(d_k, q, [\eta, c_m/x_m, d_k/z_k], \mathsf{case}\ x_m\ \{\mathsf{shift}(y) \Rightarrow P\})$
$\longmapsto \mathsf{cell}(a, D, n' \oplus 1), \mathsf{cell}(c_m, \mathsf{shift}(a), n-1), \mathsf{thread}(d_k, q, [\eta, a/y, d_k/z_k], P)$ $(\downarrow L)$

$\mathsf{cell}(c_L', D, n'), \mathsf{cell}(c_L, \mathsf{pot}\ q\ (c_L'), n), \mathsf{thread}(d_m, q', [\eta, c_L/y_L, d_m/z], \mathsf{case}\ y_L\ \{\mathsf{pot}\ q''\ (x_L) \Rightarrow P\})$
$\longmapsto \mathsf{cell}(c_L', D, n' \oplus 1), \mathsf{cell}(c_L, \mathsf{pot}\ (q - q'')\ (c_L'), n-1), \mathsf{thread}(d_m, q' + q'', [\eta, c_L'/x_L, d_m/z], P)$
(undefined if $q < q''$) $(\triangleright L)$

Negative Left Rules

$\mathsf{cell}(c_m, [\eta]\{p\}(l(y) \Rightarrow P_l)_{l\in L}, \sigma), \mathsf{thread}(d_k, q, [(\eta' + c_m/x_m), d_k/z], x_m.i(z))$
$\longmapsto [\mathsf{cell}(c_m, [\eta]\{p\}(l(y) \Rightarrow P_l)_{l\in L}, \sigma)], \mathsf{thread}(d_k, q + p, [\eta' + \eta, d_k/y], P_i)$ $(\& L^0)$

$\mathsf{cell}(c_m, [\eta]\{p\}(\langle w, y\rangle \Rightarrow P), \sigma), \mathsf{thread}(d_k, q, [(\eta' + c_m/x_m), e_k/z_1, d_k/z_2], x_m.\langle z_1, z_2\rangle)$
$\longmapsto [\mathsf{cell}(c_m, [\eta]\{p\}(\langle w, y\rangle \Rightarrow P), \sigma)], \mathsf{thread}(d_k, q + p, [\eta' + \eta, e_k/z_1, d_k/z_2], P)$ $(\multimap L^0)$

$\mathsf{cell}(c_m, [\eta]\{p\}(\mathsf{shift}(y) \Rightarrow P), \sigma), \mathsf{thread}(d_k, q, [(\eta' + c_m/x_m), d_k/z], x_m.\mathsf{shift}(z))$
$\longmapsto [\mathsf{cell}(c_m, [\eta]\{p\}(\mathsf{shift}(y) \Rightarrow P), \sigma)], \mathsf{thread}(d_k, q + p, [\eta' + \eta, d_k/y], P)$ $(\uparrow L^0)$

$\mathsf{cell}(c_L, [\eta]\{p\}\mathsf{pot}\ q\ (c_L') \Rightarrow P, 1), \mathsf{thread}(d_L, q, [\eta', c_L/y_L, d_L/x_L], y_L.\mathsf{pot}\ q\ (x_L))$
$\longmapsto \mathsf{thread}(d_L, q + p, [\eta' + \eta, d_L/c_L'], P)$ $(\triangleleft L^0)$

Appendix D

Proofs

D.1 Progress

Lemma 3 (Associativity of + Relation)**.** *If $A = (B + C)$ and $B = (D + E)$, then $A = (D + (E + C))$.*

Lemma 4 (Reading from Multilinear Cells)**.** *If $\mathcal{C}$ final, $\cdot \Vdash^{q_1+q_2+q_3} \mathcal{C} :: \Psi$, and $(c_m^{n'} : B) \in \Psi$, then $\mathcal{C} = \mathcal{C}_1, \mathsf{cell}(c_m, W, n), \mathcal{C}_2$ where:*

(i) $\cdot \Vdash^{q_1} \mathcal{C}_1 :: \Psi_1$ *for some* Ψ_1

(ii) $\Psi_1 \Vdash^{q_2} \mathsf{cell}(c_m, W, n) :: \Psi_1', (c_m^n : A)$ *for some* $\Psi_1 = \Psi_1' + \Psi_1''$ *for some* Ψ_2''

(iii) $\Psi_1', (c_m^n : A_m) \Vdash^{q_3} \mathcal{C}_2 :: \Psi$

(iv) $n \geq n'$

(v) $A = (B + C)$ *for some type* C

Proof. We proceed by right-to-left induction on $\mathcal{C}$.

Base Case Assume that $\mathcal{C}$ is empty. Then, the lemma holds vacuously, because Ψ would be empty by C:Empty.

Inductive Case Let $\mathcal{C} = \mathcal{C}', \mathsf{cell}(a_k, W', r)$, noting that $\mathcal{C}$ cannot end with a thread because $\mathcal{C}$ is final. Then we have

$$\frac{\cdot \Vdash^{q_1''} \mathcal{C}' :: \Psi' \qquad \Psi' \Vdash^{q_2''} \mathsf{cell}(a_k, W', r) :: \Psi}{\cdot \Vdash^{q_1''+q_2''} \mathcal{C}', \mathsf{cell}(c_k, W', r) :: \Psi}\ \text{C:Join}$$

By inversion on the second premise using C:Val and C:Cont, we have that $\Psi = \Psi'', (a_k^r : A_k')$ for some $A_k', \Psi' = \Psi'' + \Psi'''$.

Then, if $a_k = c_m$, we have the desired conclusion, obviously, noting that $r = n'$ and thus $r \geq n'$ and that $A_k' = B$ and thus there exists C such that $A_k' = (B + C)$.

Otherwise, we have that $(c_m^{n''} : B') \in \Psi'$. Here, $n'' \geq n'$ and $B' = (B + B'')$ for some B''. To see this, note that we have a final configuration, so all elements are cells. Then, by C:Val and C:Cont, we know that each individual cell cannot increase the reference count for any cell; this is because of the definition of $+$ on contexts. Then, by C:Join, we know that the reference count of any other cell cannot increase, so $n'' \geq n'$. Similarly, by the definition of $+$, we know that the only way the type of a cell can change is if it is split into two fragments with related types (i.e., $(d^2 : A)$ is split into $(d^1 : A')$ and $(d^1 : A'')$ where $A = (A' + A'')$).

By the inductive hypothesis on $\mathcal{C}'$, which we know is final, we have that $\mathcal{C}' = \mathcal{C}_1, \mathsf{cell}(c_m, W, n), \mathcal{C}_2$, with

(i) $\cdot \Vdash^{q_1'} \mathcal{C}_1 :: \Psi_1$ for some Ψ_1

(ii) $\Psi_1 \Vdash^{q_2'} \mathsf{cell}(c_m, W, n) :: \Psi_1', (c_m^n : A_m)$ for some $\Psi_1' \subseteq \Psi_1$

(iii) $\Psi_1', (c_m^n : A_m) \Vdash^{q_3'} \mathcal{C}_2 :: \Psi'$

(iv) $n \geq n''$

(v) $A_m = (B' + C')$ for some C'

where $q_1' + q_2' + q_3' = q_1''$. Thus, we have $\mathcal{C} = \mathcal{C}_1, \mathsf{cell}(c_m, W, n), \mathcal{C}_3$ where $\mathcal{C}_3 = \mathcal{C}_2, \mathsf{cell}(x_k, W', r)$. In addition, by C:Join, we have that $\Psi_1', (c_m^n : A_m) \Vdash^{q_3' + q_2''} \mathcal{C}_3 :: \Psi$ and $q_1' + q_2' + q_3' + q_2'' = q_1'' + q_2''$. Also, since $n \geq n''$ and $n'' \geq n'$, we have $n \geq n'$. Finally, since $A_m = (B' + C')$ and $B' = (B + B'')$, we can see that $A_m = (B + C)$ for some C by Lemma 3.

We conclude that the desired typing holds.

□

Lemma 5 (Reading from Non-Multilinear Cells)**.** *If $\mathcal{C}$ final, $\cdot \Vdash^{q_1+q_2+q_3} \mathcal{C} :: \Psi$, and $(x_m^\sigma : A_m) \in \Psi$, then $\mathcal{C} = \mathcal{C}_1, \mathsf{cell}(x_m, W, \sigma), \mathcal{C}_2$ where:*

(i) $\cdot \Vdash^{q_1} \mathcal{C}_1 :: \Psi_1$ *for some* Ψ_1

(ii) $\Psi_1 \Vdash^{q_2} \mathsf{cell}(x_m, W, \sigma) :: \Psi_1', (x_m^n : A_m)$ *for some* $\Psi_1 = \Psi_1' + \Psi_1''$

(iii) $\Psi_1', (x_m^\sigma : A_m) \Vdash^{q_3} \mathcal{C}_2 :: \Psi$

Proof. We proceed by right-to-left induction on $\mathcal{C}$.

Base Case Assume that $\mathcal{C}$ is empty. Then, the lemma holds vacuously, because Ψ would be empty by C:Empty.

Inductive Case Let $\mathcal{C} = \mathcal{C}', \mathsf{cell}(c_k, W', r)$, noting that $\mathcal{C}$ cannot end with a thread because $\mathcal{C}$ is final. Then we have

$$\frac{\cdot \Vdash^{q_1} \mathcal{C}' :: \Psi' \qquad \Psi' \Vdash^{q_2} \mathsf{cell}(c_k, W', r) :: \Psi}{\cdot \Vdash^{q_1+q_2} \mathcal{C}', \mathsf{cell}(c_k, W', r) :: \Psi}\ \text{C:Join}$$

By inversion on the second premise using C:Val and C:Cont, we have that $\Psi = \Psi'', (c_k^r : B_k)$ for some B_k, $\Psi' = \Psi'' + \Psi'''$. Then, if $c_k = x_m$, we have the desired conclusion, obviously.

Otherwise, we have that $(x_m^1 : A_m) \in \Psi'$. Notably, we know that the type and reference count must remain unchanged, since A_m is not a multilinear type and thus cannot be split in any way.

By the inductive hypothesis on $\mathcal{C}'$, which we know is final, we have that $\mathcal{C}' = \mathcal{C}_1, \mathsf{cell}(x_m, W, \sigma), \mathcal{C}_2$, with

(i) $\cdot \Vdash^{q_1'} \mathcal{C}_1 :: \Psi_1$ for some Ψ_1

(ii) $\Psi_1 \Vdash^{q_2'} \mathsf{cell}(x_m, W, \sigma) :: \Psi_1', (x_m^\sigma : A_m)$ for some $\Psi_1' \subseteq \Psi_1$

(iii) $\Psi_1', (x_m^\sigma : A_m) \Vdash^{q_3'} \mathcal{C}_2 :: \Psi'$

Thus, we have $\mathcal{C} = \mathcal{C}_1, \mathsf{cell}(x_m, W, \sigma), \mathcal{C}_3$ where $\mathcal{C}_3 = \mathcal{C}_2, \mathsf{cell}(c_k, W', r)$. In addition, by C:Join, we have that $\Psi_1', (x_m^\sigma : A_m) \Vdash^{q_3'+q_2} \mathcal{C}_3 :: \Psi$. We conclude that the desired typing holds.

□

Theorem 6 (Progress)**.** *If* $\cdot \Vdash^q \mathcal{C} :: \Psi$*, then either* $\mathcal{C}$ *final or* $\mathcal{C} \longmapsto^{q'} \mathcal{C}'$ *for some configuration* $\mathcal{C}'$ *and* $q' \leq q$*.*

Proof. We proceed by right-to-left induction on $\mathcal{C}$, using Lemmas 5 and 4 to show that a reading thread must be to the right of a valid cell.

Base Case Assume $\mathcal{C}$ is empty. Then, $\mathcal{C}$ is trivially final, since there are no threads.

Inductive Case Let $\mathcal{C} = \mathcal{C}_1, \mathcal{C}_2$ where either $\mathcal{C}_2 = \mathsf{cell}(d_m, W, n)$ or $\mathcal{C}_2 = \mathsf{thread}(d_m, q_2, [\eta], P), \mathsf{cell}(d_m, -, n)$. We know by C:Join that $\cdot \Vdash^{q_1} \mathcal{C}_1 :: \Psi_1$ and $\Psi_1 \Vdash^{q_2} \mathcal{C}_2 :: \Psi$ for some Ψ_1, q_1, q_2 with $q_1 + q_2 = q$.

First, we apply the inductive hypothesis on $\mathcal{C}_1$. Either $\mathcal{C}_1 \longmapsto^{q_1'} \mathcal{C}_1'$ for some $\mathcal{C}_1'$ and $q_1' \leq q_1$ or $\mathcal{C}_1$ final. In the first case, by multiset rewriting, we know that $\mathcal{C} \longmapsto^{q_1'} \mathcal{C}_1', \mathcal{C}_2$; since $q_1' \leq q_1 \leq q$, the conclusion holds.

In the second case, $\mathcal{C}_1$ final. We proceed by casing on $\mathcal{C}_2$. If $\mathcal{C}_2 = \mathsf{cell}(d_m, W, n)$, then clearly $\mathcal{C}$ final, since $\mathcal{C}$ is made up only of cells.

If $\mathcal{C}_2 = \mathsf{thread}(d_m, q_2, [\eta], P), \mathsf{cell}(d_m, -, n)$, we have

$$\frac{\Psi_3 \vdash \eta : \Gamma \qquad \Gamma \vdash^{q_2} P :: (x_m : A_m) \qquad \Psi_1 = \Psi_3 + \Psi_2}{\Psi_1 \Vdash^{q_2} \mathsf{thread}(c_m, q_2, [\eta, c_m/x_m], P), \mathsf{cell}(c_m, -, n) :: \Psi_2, (c_m^n : A_m)} \text{ C:Thread}$$

We proceed by induction on the derivation of the second premise.

$\oplus R^0$ First, note that if the second premise is derived by $\oplus R^0$, then $\Gamma = \Gamma_W + (y : A_m^i)$, and the potential is 0. Then, by $E2$, we have that $\eta = \eta' + d/y = \eta'', d/y$. Finally, $\mathcal{C}_2 \longmapsto \mathcal{C}_2'$ for some $\mathcal{C}_2'$ by dynamics rule $\oplus R^0$. Then, by multiset rewriting, $\mathcal{C} \longmapsto \mathcal{C}'$ for some $\mathcal{C}'$ and the conclusion holds.

$\otimes R^0, \mathbf{1}R^0, \downarrow R^0, \triangleright R^0$ Similarly to above (except that in $\triangleright R^0$, potential may be non-zero).

$\multimap R, \uparrow R, \triangleleft R$ Obviously, by the respective dynamics rules, $\mathcal{C}_2 \longmapsto \mathcal{C}_2'$ for some $\mathcal{C}_2'$. Then, by multiset rewriting, $\mathcal{C} \longmapsto \mathcal{C}'$ for some $\mathcal{C}'$ and the conclusion holds.

Cut Next, we consider a derivation via Cut. Obviously, $\mathcal{C}_2 \longmapsto \mathcal{C}_2'$ for some $\mathcal{C}_2'$, so $\mathcal{C} \longmapsto \mathcal{C}'$ for some $\mathcal{C}'$.

Call If Call is used, then by the Call dynamics rule, we obviously can make a step, since we assume that all processes in the fixed signature Σ are typechecked ahead of time.

$\oplus L$ We next consider the $\oplus L$ rule. In this case, we have $\Gamma' + (x : \oplus_k\{l : B_k^l\}_{l \in L}) \vdash^{q_2} \mathsf{case}\ x\ \{l(y) \Rightarrow Q\}_{l \in L} :: (y_m : A_m)$, where $\Gamma = \Gamma' + (x : \oplus_k\{l : B_k^l\}_{l \in L})$.

We consider two cases: when x is multilinear and when x is not.

First, we consider the multilinear case. By inversion on the first premise and E2, we get that $\Psi_3 = \Psi_3' + (a^1 : \oplus_k\{l : B_k^l\}_{l \in L})$ for some a with $\eta = \eta' + a/x$ for some η'. Then by + on contexts, we also have $(a^n : \oplus_k\{l : C_k^l\}_{l \in L}) \in \Psi_1$ for some $n \geq 1$, which is provided by $\mathcal{C}_1$, a

final configuration, where $C = (B + D)$ for some D. By Lemma 4 and inversions on C:Val and Val$- \oplus R^0$, we have that $\mathsf{cell}(a, i(y'), n') \in \mathcal{C}_1$ with $n' \geq n > 0$. By the fact that there is one reference to y' in cell a, we know that cell y' has been allocated and exists to the left of cell a. Then, by the $\oplus L$ dynamics rule, we have that $\mathcal{C} \longmapsto \mathcal{C}'$ for some $\mathcal{C}'$, as desired.

Next, we consider the non-multilinear case. By inversion on the first premise and E2, we get that $(a^\sigma : \oplus_k\{l : B_k^l\}_{l \in L}) \in \Psi_3$ for some a with $\eta = \eta' + a/x = \eta'', a/x$ for some η', η''. Then by + on contexts, we also have $(a^\sigma : \oplus_k\{l : B_k^l\}_{l \in L}) \in \Psi_1$, which is provided by $\mathcal{C}_1$, a final configuration. By Lemma 5 and inversions on C:Val and Val$- \oplus R^0$, we have that $\mathsf{cell}(a, i(y'), \sigma) \in \mathcal{C}_1$. By the fact that there is one reference to y' in cell a, we know that cell y' has been allocated and exists to the left of cell a. Then, by the $\oplus L2$ dynamics rule, we have that $\mathcal{C} \longmapsto \mathcal{C}'$ for some $\mathcal{C}'$, as desired.

$\multimap L^0$ Next, we consider the $\multimap L^0$ rule. In this case, we have $\Gamma' + (w_m : B_m) + (x : B_m \multimap_m A_m) \vdash^0 x.\langle w_m, y_m\rangle :: (y_m : A_m)$, where $\Gamma = \Gamma' + (w_m : B_m) + (x : B_m \multimap_m A_m)$. By inversion on the first premise and E2, we get that $(b^n : B_m), (a^\sigma : B_m \multimap_m A_m) \in \Psi_3$ for some a, b with $\eta = \eta' + a/x + b/w = \eta'', a/x, b/w$ for some η', η'', and $n > 0$. By + on contexts, this tells us that $(b^{n'} : C_m), (a^1 : B_m \multimap_m A_m) \in \Psi_1$, which is provided by $\mathcal{C}_1$, a final configuration, where $C_m = (B_m + D_m)$ and $n' \geq n > 0$. By Lemma 5 and inversions on C:Cont and $\multimap R$, we have that $\mathsf{cell}(a, \langle w'_m, y'_m\rangle \Rightarrow Q, \sigma) \in \mathcal{C}_1$ for some Q. Then, by the $\multimap L$ dynamics rule, we have that $\mathcal{C} \longmapsto \mathcal{C}'$ for some $\mathcal{C}'$, as desired.

$\triangleright L$ We next consider the $\triangleright L$ rule. In this case, we have $\Gamma', (x : \triangleright^p B_{\mathsf{L}}) \vdash^{q_2} \mathsf{case}\ x\ \{\mathsf{pot}\ p\ (y) \Rightarrow Q\} :: (z_m : A_m)$, where $\Gamma = \Gamma', (x : \triangleright^p B_{\mathsf{L}})$.

We again consider two cases: when x is multilinear, and when x is not.

First, we take the case where x is multilinear (it must be linear because it carries potential). By inversion on the first premise and E2, we get that $\Psi_3 = \Psi'_3 + (a^1 : \triangleright^p B_{\mathsf{L}})$ for some a with $\eta = \eta', a/x$ for some η, and $n > 0$. This tells us that $(a^n : \triangleright^{p'} C_{\mathsf{L}}) \in \Psi_1$, which is provided by $\mathcal{C}_1$, a final configuration, where $p' \geq p$ and $n > 0$ and $C = (B + D)$ for some D. By Lemma 4 and inversions on C:Val and Val$- \triangleright R^0$, we have that $\mathsf{cell}(a, \mathsf{pot}\ p''\ (y'), n') \in \mathcal{C}_1$, for $n' \geq n > 0$ and $p'' \geq p' \geq p$. Then, by the $\triangleright L$ dynamics rule, we have that $\mathcal{C} \longmapsto \mathcal{C}'$ for some $\mathcal{C}'$, as desired.

$\triangleleft L^0$ If our derivation used the $\triangleleft L^0$ rule, we have $\Gamma', (x : \triangleleft^r A_{\mathsf{L}}) \vdash^0 x.\mathsf{pot}\ r\ (y) :: (y : A_{\mathsf{L}})$ for $\Gamma = \Gamma', (x : \triangleleft^r A_{\mathsf{L}})$. By inversion on the first premise and E2, we get that $(x^1 : \triangleleft^r A_{\mathsf{L}}) \in \Psi_3$ for some a with $\eta = \eta', a/x$ for some η. This tells us that $(x^1 : \triangleleft^r A_{\mathsf{L}}) \in \Psi_1$, which is provided by $\mathcal{C}_1$, a final configuration. By Lemma 5 and inversions on C:Cont and $\triangleleft R$, we have that $\mathsf{cell}(x, \mathsf{pot}\ r\ (y') \Rightarrow Q, 1) \in \mathcal{C}_1$.

Then, by the $\triangleleft L$ dynamics rule, we have that $\mathcal{C} \longmapsto \mathcal{C}'$ for some $\mathcal{C}'$, as desired.

$\mathbf{1}L, \otimes L, \&L^0, \uparrow L^0, \downarrow L$ All other left rules follow similarly to those described above.

IdK If the IdK rule is used, we have $\Gamma' + (x_m : A_m) \vdash^0 d_m \leftarrow x_m :: (d_m : A_m)$, with $\Gamma = \Gamma' + (x_m : A_m)$ where A_m is a negative type. Then by inversion on the first premise and E2, we have $(c_m^\sigma : A_m) \in \Psi_3$ and thus $\in \Psi_1$, with $\eta = \eta', c_m/x_m$ for some η'. By Lemma 5, we have that $\mathsf{cell}(c_m, [\eta'']\{p\}K, \sigma) \in \mathcal{C}_1$ for some K; then, by the Id dynamics rule, we can conclude that $\mathcal{C} \longmapsto \mathcal{C}'$ for some $\mathcal{C}'$ and the conclusion holds.

Work Next, we consider the Work rule. If $P = \mathsf{work}\ \{r\}\ ;\ P'$, we know that $r \leq q$ by the typing rules. Then, $\mathcal{C} \longmapsto^r \mathcal{C}'$, with $r \leq q$, as desired.

Alias If the Alias rule is used, we have $\Gamma', (x_\mathsf{L} : A_\mathsf{L}) \vdash^q \mathsf{alias}\ x : A_\mathsf{L}\ \mathsf{as}\ y : B_\mathsf{L}, z : C_\mathsf{L}\ ;\ P :: (w : T_r)$, with $\Gamma = \Gamma', (x_\mathsf{L} : A_\mathsf{L})$. We also note that A_L must be multilinear. Then by inversion on the first premise and E2, $\Psi_3 = \Psi_3' + (c_\mathsf{L}^1 : A_\mathsf{L})$ for $n > 0$, with $\eta = \eta', c_\mathsf{L}/x_\mathsf{L}$ for some η'. Then, by the definition of $+$ on contexts, we have $(c_\mathsf{L}^n : B_\mathsf{L}) \in \Psi_1$, and $n > 0$ and $B_\mathsf{L} = (A_\mathsf{L} + C_\mathsf{L})$ for some C_L. By Lemma 4, we have that $\mathsf{cell}(c_\mathsf{L}, V, n') \in \mathcal{C}_1$ for some V and $n' \geq n > 0$; then, by the Copy dynamics rule, we can conclude that $\mathcal{C} \longmapsto \mathcal{C}'$ for some $\mathcal{C}'$ and the conclusion holds.

Drop Finally, we consider the Drop rule, in which case we have $\Gamma', (x_\mathsf{L} : A_\mathsf{L}) \vdash^q \mathsf{drop}\ x : A_\mathsf{L}\ ;\ P :: (y_r : C_r)$. We also note that A_L must be multilinear. Then by inversion on the first premise and E2, $\Psi_3 = \Psi_3' + (c_\mathsf{L}^1 : A_\mathsf{L})$, with $\eta = \eta' + c_\mathsf{L}/x_\mathsf{L}$ for some η'. Then by $+$ on contexts, $(c_\mathsf{L}^n : B_\mathsf{L}) \in \Psi_1$ with $n > 0$ and $B_\mathsf{L} = (A_\mathsf{L} + C_\mathsf{L})$ for some C_L. By Lemma 4, we have that $\mathsf{cell}(c_\mathsf{L}, V, n') \in \mathcal{C}_1$ for some V and $n' \geq n > 0$; so, by the Drop dynamics rule, we can conclude that $\mathcal{C} \longmapsto \mathcal{C}'$ for some $\mathcal{C}'$ and the conclusion holds.

Thus, the progress theorem holds for any configuration $\mathcal{C}$. □

D.2 Preservation

Lemma 7 (Weakenable Environments)**.** *If Γ is entirely weakenable (i.e., only contains of unrestricted variables) in $\Psi \vdash \eta : \Gamma$, then Ψ and η must both consist of exclusively unrestricted addresses and substitutions.*

Proof. Assume that $\Psi \vdash \eta : \Gamma$ where Γ is weakenable. We proceed by induction on η. In the base case, when $\eta = \cdot$, we have by inversion on E1 that Γ is empty and Ψ is weakenable, so the conclusion is satisfied.

In the inductive case, we have $\eta = \eta' + a/x$. Then, by inversion on E2, we have $\Psi' + (a^\sigma : A) \vdash \eta' + a/x : \Gamma' + (x : A)$. We know Γ' is fully weakenable, so Ψ' must also be weakenable, as must be η', by the inductive hypothesis. We also know that since x is weakenable and therefore must be unrestricted, a and A must also be unrestricted. This means that Ψ is unrestricted and $\eta = \eta' + a/x$ must have only unrestricted substitutions. □

Lemma 8 (Context Persistence)**.** *If* $\Psi \Vdash^q \mathcal{C} :: \Psi'$*, then* $\Psi + (a^n : A) \Vdash^q \mathcal{C} :: \Psi' + (a^n : A)$ *for any* $(a^n : A)$*.*

Proof. We proceed by induction on the judgment $\Psi \Vdash^q \mathcal{C} :: \Psi'$.

- If $\mathcal{C} = \cdot$, then $\Psi = \Psi'$ and we apply C:Empty to get $\Psi + (a^n : A) \Vdash^0 \cdot :: \Psi + (a^n : A)$.
- If $\mathcal{C} = \mathcal{C}_1, \mathcal{C}_2$, we have $\Psi \Vdash^{q_1} \mathcal{C}_1 :: \Psi_1$ and $\Psi_1 \Vdash^{q_2} \mathcal{C}_2 :: \Psi'$. By the inductive hypothesis on the first of these, we have that $\Psi + (a^n : A) \Vdash^{q_1} \mathcal{C}_1 :: \Psi_1 + (a^n : A)$. Then, by the inductive hypothesis on the second premise, we have $\Psi_1 + (a^n : A) \Vdash^{q_2} \mathcal{C}_2 :: \Psi' + (a^n : A)$. Finally, we apply C:Join to get $\Psi + (a^n : A) \Vdash^{q_1+q_2} \mathcal{C}_1, \mathcal{C}_2 :: \Psi' + (a^n : A)$.
- If $\mathcal{C} = \mathsf{cell}(c, W, n)$ or $\mathcal{C} = \mathsf{thread}(c, p, [\eta], P), \mathsf{cell}(c, -, n)$, then we know $\Psi = \Psi_1 + \Psi_2$ for some Ψ_1, Ψ_2, with $\Psi' = \Psi_2, (c^k : A_m)$ ($k = n$ unless $\mathcal{C}$ is a cell containing a continuation, in which case $k = 1$). Then, $\Psi + (a^n : A) = \Psi_1 + (\Psi_2 + (a^n : A))$. Then, we can apply C:Val, C:Cont, or C:Thread (depending on $\mathcal{C}$) to get that $\Psi + (a^n : A) \Vdash^q \mathcal{C} :: \Psi_2 + (a^n : A), (a^n : A)$, i.e. $\Psi + (a^n : A) \Vdash^q \mathcal{C} :: \Psi' + (a^n : A)$.

□

Lemma 9 (Reducing Reference Counts)**.** *If* $\Psi, (c_{\mathsf{L}}^n : A_{\mathsf{L}}) \Vdash^q \mathcal{C} :: \Psi', (c_{\mathsf{L}}^k : B_{\mathsf{L}})$*, then* $\Psi, (c_{\mathsf{L}}^{n-m} : A_{\mathsf{L}}) \Vdash^q \mathcal{C} :: \Psi', (c_{\mathsf{L}}^{k-m} : B_{\mathsf{L}})$ *for any* $n - m, k - m \geq 0$*.*

Proof. We proceed by induction on the judgment $\Psi, (c_{\mathsf{L}}^n : A_{\mathsf{L}}) \Vdash^q \mathcal{C} :: \Psi', (c_{\mathsf{L}}^k : B_{\mathsf{L}})$.

- If $\mathcal{C} = \cdot$, then $\Psi, (c_{\mathsf{L}}^n : A_{\mathsf{L}}) = \Psi', (c_{\mathsf{L}}^k : B_{\mathsf{L}})$ and we apply C:Empty to get $\Psi, (c_{\mathsf{L}}^{n-m} : A_{\mathsf{L}}) \Vdash^0 \mathcal{C} :: \Psi', (c_{\mathsf{L}}^{k-m} : B_{\mathsf{L}})$.
- If $\mathcal{C} = \mathsf{cell}(d, W, n')$ or $\mathcal{C} = \mathsf{thread}(d, p, [\eta], P), \mathsf{cell}(d, -, n')$, then we know $\Psi, (c_{\mathsf{L}}^n : A_{\mathsf{L}}) = \Psi_1 + \Psi_2$ for some Ψ_1, Ψ_2, with $\Psi', (c_{\mathsf{L}}^k : B_{\mathsf{L}}) = \Psi_2, (d^{n'} : D)$. Then, $\Psi, (c_{\mathsf{L}}^n : A_{\mathsf{L}}) = \Psi, ((c_{\mathsf{L}}^k : B_{\mathsf{L}}) + (c_{\mathsf{L}}^{n-k} : C_{\mathsf{L}}))$, so $\Psi_1 = \Psi_1', (c_{\mathsf{L}}^{n-k} : C_{\mathsf{L}})$ while $\Psi_2 = \Psi_2', (c_{\mathsf{L}}^k : B_{\mathsf{L}})$ (then, $\Psi' = \Psi_2', (d^{n'} : D)$).

  Next, we can say that $\Psi, (c_{\mathsf{L}}^{n-m} : A_{\mathsf{L}}) = \Psi, ((c_{\mathsf{L}}^{k-m} : B_{\mathsf{L}}) + (c_{\mathsf{L}}^{n-k} : C_{\mathsf{L}}))$, and thus we have $\Psi_1'' = \Psi_1', (c_{\mathsf{L}}^{n-k} : C_{\mathsf{L}})$ while $\Psi_2'' = \Psi_2', (c_{\mathsf{L}}^{k-m} : B_{\mathsf{L}})$.

Then, we can apply C:Val, C:Cont, or C:Thread (depending on $\mathcal{C}$) to get that $\Psi, (c_{\mathsf{L}}^{n-m} : A_{\mathsf{L}}) \Vdash^{q} \mathcal{C} :: \Psi_2'', (d^{n'} : D)$. Notably, $\Psi_2'', (d^{n'} : D) = \Psi_2', (c_{\mathsf{L}}^{k-m} : B_{\mathsf{L}}), (d^{n'} : D) = \Psi', (c_{\mathsf{L}}^{k-m} : B_{\mathsf{L}})$. Thus, we have the desired conclusion.

- If $\mathcal{C} = \mathcal{C}_1, \mathcal{C}_2$, we have $\Psi, (c_{\mathsf{L}}^{n} : A_{\mathsf{L}}) \Vdash^{q_1} \mathcal{C}_1 :: \Psi_1$ and $\Psi_1 \Vdash^{q_2} \mathcal{C}_2 :: \Psi', (c_{\mathsf{L}}^{k} : B_{\mathsf{L}})$. Obviously, since c_{L} remains in the context after $\mathcal{C}_1, \mathcal{C}_2$, we must have $\Psi_1 = \Psi_1', (c_{\mathsf{L}}^{n'} : C_{\mathsf{L}})$ for some n'. By the inductive hypothesis, we have that $\Psi, (c_{\mathsf{L}}^{n-x} : A_{\mathsf{L}}) \Vdash^{q_1} \mathcal{C}_1 :: \Psi_1', (c_{\mathsf{L}}^{n'-x} : C_{\mathsf{L}})$. Then, by the inductive hypothesis on $\mathcal{C}_2$, we have $\Psi_1', (c_{\mathsf{L}}^{n'-x} : C_{\mathsf{L}}) \Vdash^{q_2} \mathcal{C}_2 :: \Psi', (c_{\mathsf{L}}^{k-x} : B_{\mathsf{L}})$. Finally, we apply C:Join to get $\Psi, (c_{\mathsf{L}}^{n-x} : A_{\mathsf{L}}) \Vdash^{q_1+q_2} \mathcal{C}_1, \mathcal{C}_2 :: \Psi', (c_{\mathsf{L}}^{k-x} : A_{\mathsf{L}})$.

□

Theorem 10 (Preservation). *If $\Psi \Vdash^{q+w} \mathcal{C} :: \Psi'$, and $\mathcal{C} \longmapsto^{w} \mathcal{C}'$ for some configuration $\mathcal{C}'$, then $\Psi \Vdash^{q} \mathcal{C}' :: \Psi'$.*

Proof. We proceed by induction on $\mathcal{C} \longmapsto^{w} \mathcal{C}'$, using inversions on configuration and process typing.

$\oplus R^0$ We consider the step

$$\mathcal{C}_1, \mathsf{thread}(c_m, 0, [\eta, d_m/y, c_m/x], x_m.i(y)), \mathsf{cell}(c_m, -, n), \mathcal{C}_2 \longmapsto \mathcal{C}_1, \mathsf{cell}(c_m, i(d_m), n), \mathcal{C}_2$$

First, by a series of inversions on C:Join, we get that

(i) $\Psi \Vdash^{q_1} \mathcal{C}_1 :: \Psi_1$

(ii) $\Psi_1 \Vdash^{q_2} \mathsf{thread}(c_m, 0, [\eta, d_m/y, c_m/x], x_m.i(y)), \mathsf{cell}(c_m, -, n) :: \Psi_2$

(iii) $\Psi_2 \Vdash^{q_3} \mathcal{C}_2 :: \Psi'$

where $q_1 + q_2 + q_3 = q + w = q$ (since $w = 0$). By inversion on (ii), we get

$$\frac{\Psi_3 \vdash \eta, d_m/y : \Gamma \qquad \Gamma \vdash^{0} x_m.i(y) :: (x_m : \oplus_m\{l : A_m^l\}_{l \in L}) \qquad \Psi_1 = \Psi_3 + \Psi_2'}{\Psi_1 \Vdash^{0} \mathsf{thread}(c_m, 0, [\eta, d_m/y, c_m/x], x_m.i(y)), \mathsf{cell}(c_m, -, n) :: \Psi_2', (c_m^n : \oplus_m\{l : A_m^l\}_{l \in L})}\ \text{C:Thread}$$

noting that $q_2 = 0$ and $\Psi_2 = \Psi_2', (c_m^n : \oplus_m\{l : A_m^l\}_{l \in L})$.

Using the second premise with inversion on $\oplus$R^0, we get that $\Gamma = \Gamma_W + (y : A_m^i)$.

By Lemma 7, we have that the remainder of Ψ_3 and η apart from one reference to d_m is weakenable. Then, by E2, we know that $\Psi_3 = (\Psi_{3,W} + (d_m^{\sigma} : A_m^i))$ for some weakenable $\Psi_{3,W}$.

We apply C:Val and Val-$\oplus$R^0 to get

$$\dfrac{\dfrac{i \in L}{\Psi_{3,W} + (d_m^\sigma : B_m^i) \vdash^q i(d_m) : \oplus_m\{l : B_m^l\}_{l \in L}}\ \textsc{Val-}\oplus\mathrm{R}^0 \qquad \Psi_1 = (\Psi_{3,W} + (d_m^\sigma : A_m^i)) + \Psi_2'}{\Psi_1 \Vdash^q \mathsf{cell}(c_m, i(d_m), n) :: \Psi_2', (c_m^n : \oplus_m\{l : A_m^l\}_{l \in L})}\ \textsc{C:Val}$$

Clearly, the addresses used and provided in this judgment are the same as those in (ii). Then, by C:Join, we have $\Psi \Vdash^q \mathcal{C}_1, \mathsf{cell}(c_m, i(d_m), n), \mathcal{C}_2 :: \Psi'$, as desired.

$\otimes R^0$ We consider the step

$$\mathcal{C}_1, \mathsf{thread}(c_m, 0, [\eta, d_m/y, e_m/w, c_m/x], x_m.\langle y, w\rangle), \mathsf{cell}(c_m, -, n), \mathcal{C}_2 \longmapsto \mathcal{C}_1, \mathsf{cell}(c_m, \langle d_m, e_m\rangle, n), \mathcal{C}_2$$

First, by a series of inversions on C:Join, we get that

(i) $\Psi \Vdash^{q_1} \mathcal{C}_1 :: \Psi_1$

(ii) $\Psi_1 \Vdash^{q_2} \mathsf{thread}(c_m, 0, [\eta, d_m/y, e_m/w, c_m/x], x_m.\langle y, w\rangle), \mathsf{cell}(c_m, -, n) :: \Psi_2$

(iii) $\Psi_2 \Vdash^{q_3} \mathcal{C}_2 :: \Psi'$

where $q_1 + q_2 + q_3 = q + w = q$ (since $w = 0$). By inversion on (ii), we get

$$\dfrac{\Psi_3 \vdash \eta, d_m/y, e_m/w : \Gamma \qquad \Gamma \vdash^0 x_m.\langle y, w\rangle :: (x_m : A_m \otimes_m B_m) \qquad \Psi_1 = \Psi_3 + \Psi_2'}{\Psi_1 \Vdash^0 \mathsf{thread}(c_m, 0, [\eta, d_m/y, e_m/w, c_m/x], x_m.\langle y, w\rangle), \mathsf{cell}(c_m, -, n) :: \Psi_2', (c_m^n : \oplus_m\{l : A_m^l\}_{l \in L})}\ \textsc{C:Thread}$$

noting that $q_2 = 0$ and $\Psi_2 = \Psi_2', (c_m^n : A_m \otimes_m B_m)$.

Using the second premise with inversion on $\otimes$R^0, we get that $\Gamma = \Gamma_W + (y : A_m) + (w : B_m)$.

By Lemma 7, we have that the remainder of Ψ_3 and η apart from references to d_m and e_m (which, notably, might be the same cell) is weakenable. Then by E2, $\Psi_3 = \Psi_{3,W} + (d_m^\sigma : A_m) + (e_m^{\sigma'} : B_m)$.
We apply C:Val and Val-$\otimes$R^0 to get

$$\dfrac{\dfrac{}{\Psi_{3,W} + (d_m^\sigma : A_m) + (e_m^{\sigma'} : B_m) \vdash^q \langle d_m, e_m\rangle : A_m \otimes_m B_m}\ \textsc{Val-}\otimes\mathrm{R}^0 \qquad \Psi_1 = (\Psi_{3,W} + (d_m^\sigma : A_m) + (e_m^{\sigma'} : B_m)) + \Psi_2'}{\Psi_1 \Vdash^q \mathbf{cell}(c_m, \langle d_m, e_m\rangle, n) :: \Psi_2', (c_m^n : A_m \otimes_m B_m)}\ \textsc{C:Val}$$

Clearly, the addresses used and provided in this judgment are the same as those in (ii). Then, by C:Join, we have $\Psi \Vdash^{q} \mathcal{C}_1, \mathsf{cell}(c_m, \langle d_m, e_m \rangle, n), \mathcal{C}_2 :: \Psi'$, as desired.

$\mathbf{1}R^0, \downarrow R^0, \triangleright R^0$ These cases follow similarly to those above.

$\multimap R, \& R, \uparrow R, \triangleleft R$ We consider the step

$$\mathcal{C}_1, \mathsf{thread}(c_m, q, [\eta, c_m/x_m], \mathsf{case}\ x_m\ \{K\}), \mathsf{cell}(c_m, -, 1), \mathcal{C}_2 \longmapsto \mathcal{C}_1, \mathsf{cell}(c_m, [\eta]\{q\}K, \sigma), \mathcal{C}_2$$

First, by a series of inversions on C:Join, we get that

(i) $\Psi \Vdash^{q_1} \mathcal{C}_1 :: \Psi_1$

(ii) $\Psi_1 \Vdash^{q_2} \mathsf{thread}(c_m, 0, [\eta, c_m/x_m], \mathsf{case}\ x_m\ \{K\}), \mathsf{cell}(c_m, -, \sigma) :: \Psi_2$

(iii) $\Psi_2 \Vdash^{q_3} \mathcal{C}_2 :: \Psi'$

where $q_1 + q_2 + q_3 = q + w = q'$ (since $w = 0$). By inversion on (ii), we get

$$\frac{\Psi_3 \vdash \eta : \Gamma \qquad \Gamma \vdash^{q_2} \mathsf{case}\ x_m\ \{K\} :: (x_m : A_m) \qquad \Psi_1 = \Psi_3 + \Psi_2'}{\Psi_1 \Vdash^{q_2} \mathsf{thread}(c_m, q_2, [\eta, c_m/x_m], \mathsf{case}\ x_m\ \{K\}), \mathsf{cell}(c_m, -, \sigma) :: \Psi_2', (c_m^{\sigma} : A_m)}\ \text{C:Thread}$$

noting that $q_2 = q$ and $\Psi_2 = \Psi_2', (c_m^{\sigma} : A_m)$.

Now, we can apply C:Cont to get

$$\frac{\Psi_3 \vdash \eta : \Gamma \qquad \Gamma \vdash^{q} \mathsf{case}\ x_m\ \{K\} :: (x_m : A_m) \qquad \Psi_1 = \Psi_3 + \Psi_2'}{\Psi_1 \Vdash^{q} \mathsf{cell}(c_m, [\eta]\{q\}K, \sigma) :: \Psi_2', (c_m^{\sigma} : A_m)}\ \text{C:Cont}$$

Clearly, the addresses used and provided in this judgment are the same as those in (ii), with the same cost of $q_2 = q$. Then, by C:Join, we have $\Psi \Vdash^{q'} \mathcal{C}_1, \mathsf{cell}(c_m, [\eta]\{q\}K, \sigma), \mathcal{C}_2 :: \Psi'$, as desired.

$\oplus L$ First, we consider the multilinear case. We consider the step

$$\begin{aligned} &\mathcal{C}_1, \mathsf{cell}(a, D, k), \mathcal{C}_2, \mathsf{cell}(c_m, i(a), n), \mathcal{C}_3, \\ &\mathsf{thread}(d, q, [\eta, c_m/x_m, d/z], \mathsf{case}\ x_m\ \{l(y) \Rightarrow P_l\}_{l \in L}), \mathsf{cell}(d, -, n''), \mathcal{C}_4 \\ \longmapsto\ &\mathcal{C}_1, \mathsf{cell}(a, D, k \oplus 1), \mathcal{C}_2, \mathsf{cell}(c_m, i(a), n-1), \mathcal{C}_3, \mathsf{thread}(d, q, [\eta, a/y, d/z], P_i), \mathsf{cell}(d, -, n''), \mathcal{C}_4 \end{aligned}$$

$\mathcal{C}$ has the following typing, by inversions on C:Join:

(i) $\Psi \Vdash^{w_1} \mathcal{C}_1 :: \Psi_1$

(ii) $\Psi_1 \Vdash^{w_2} \mathsf{cell}(a, D, k) :: \Psi_2$

(iii) $\Psi_2 \Vdash^{w_3} \mathcal{C}_2 :: \Psi_3$

(iv) $\Psi_3 \Vdash^{w_4} \mathsf{cell}(c_m, i(a), n) :: \Psi_4$

(v) $\Psi_4 \Vdash^{w_5} \mathcal{C}_3 :: \Psi_5$

(vi) $\Psi_5 \Vdash^{w_6} \mathsf{thread}(d, q, [\eta, c_m/x_m, d/z], \mathsf{case}\ x_m\ \{l(y) \Rightarrow P\}_{l \in L}), \mathsf{cell}(d, -, n'') :: \Psi_6$

(vii) $\Psi_6 \Vdash^{w_7} \mathcal{C}_4 :: \Psi'$

To show that $\mathcal{C}_1, \mathsf{cell}(a, D, k \oplus 1), \mathcal{C}_2, \mathsf{cell}(c_m, i(a), n-1), \mathcal{C}_3, \mathsf{thread}(d, q, [\eta, a/y, d/z], P_i), \mathsf{cell}(d, -, n''), \mathcal{C}_4$ has the same typing, we must show

(a) $\Psi \Vdash^{w_1} \mathcal{C}_1 :: \Psi_1$

(b) $\Psi_1 \Vdash^{w'_2} \mathsf{cell}(a, D, k \oplus 1) :: \Psi'_2$

(c) $\Psi'_2 \Vdash^{w'_3} \mathcal{C}_2 :: \Psi'_3$

(d) $\Psi'_3 \Vdash^{w'_4} \mathsf{cell}(c_m, i(a), n-1) :: \Psi'_4$

(e) $\Psi'_4 \Vdash^{w'_5} \mathcal{C}_3 :: \Psi'_5$

(f) $\Psi'_5 \Vdash^{w'_6} \mathsf{thread}(d, q, [\eta, a/y, d/z], P_i), \mathsf{cell}(d, -, n'') :: \Psi'_6$

(g) $\Psi'_6 \Vdash^{w'_7} \mathcal{C}_4 :: \Psi'$

where $w_1 + w'_2 + w'_3 + w'_4 + w'_5 + w'_6 + w'_7 = w_1 + w_2 + w_3 + w_4 + w_5 + w_6 + w_7$.

First, we examine the initial configuration, providing names for each type seen by the configuration. When we allocate cell a, we get $\Psi_2 = \Psi''_2, (a^k : A)$ for some A.

Then, $\mathcal{C}_2$ can use up some of a; this occurs through splitting contexts using +. Thus, after typing $\mathcal{C}_2$, we have $\Psi_3 = \Psi''_3, (a^{k_1} : A_1)$, where $A = (A_1 + B_1)$ and $(a^{k-k_1} : B_1)$ was used up by $\mathcal{C}_2$.

Next, we allocate c_m by Val-$\oplus$R^0, referring to a. Thus, one reference to a is used up, leaving $\Psi_4 = \Psi''_4, (a^{k_1-1} : A_2), (c^n_m : \oplus_m \{l : X^l_m\}_{l \in L})$, where $A_1 = (A_2 + B_2)$. Notably, $X^i = B_2$.

We then use up some more of a and some of c in $\mathcal{C}_3$. We are left with $(a^{k_2} : A_3)$ where $A_2 = (A_3 + B_3)$ and $\mathcal{C}_3$ used up $(a^{k_1-1-k_2} : B_3)$. We also have $(c^{n_1}_m : \oplus_m \{l : {X_1}^l_m\}_{l \in L})$, where $X^l = (X^l_1 + Y^l_1)$ (by the $+$ relation on sum types), and $\mathcal{C}_3$ used up $(c^{n-n_1}_m : \oplus_m \{l : {Y_1}^l_m\}_{l \in L})$.

Finally, we consider the running thread that reads from c_m. Some of a may be used up in typing P_l (i.e. in the environment), while the rest remains in the resulting configuration. Specifically, after typing this thread, we are left with $(a^{k_3} : A_4)$ for $A_3 = (A_4 + B_4)$ and $(a^{k_2 - k_3} : B_4)$ was used up by P_l. When it comes to c_m, we have one additional place a reference is used: in the case expression. Overall, we use some of c_m in typing this thread and leave the rest in the resulting configuration. After typing the thread, we are left with $(c^{n_2} : \oplus_m \{l : X_{2m}^l\}_{l \in L})$, where $X_1^l = (X_2^l + Y_2^l)$, and $(c_m^{n_1 - n_2} : \oplus_m \{l : Y_{2m}^l\}_{l \in L})$ is used by the thread. Within these references, one is used for the variable x_m, so we have $(x : \oplus_m \{l : Y_{3m}^l\}_{l \in L})$ where $Y_2^l = (Y_3^l + Z_3^l)$. The remainder of the cell (i.e., $(c_m^{n_1 - n_2 - 1} : \oplus_m \{l : Z_{3m}^l\}_{l \in L})$) is used in P_l.
In addition, by inversion on (vi) with C:Thread and $\oplus$L, we know that

$$\dfrac{\Psi_{5a} \vdash \eta, c_m/x_m : \Gamma \qquad \dfrac{\dfrac{\Gamma, (y : A) \vdash^w 6 \ P_i :: (z : D)}{\Gamma \vdash^w 6 \ \mathbf{case}\ x_m\ \{l(y) \Rightarrow P\}_{l \in L} :: (z : D)} \oplus \mathrm{L} \qquad \Psi_5 = \Psi_{5a} + \Psi_{5b}}{\Psi_5 \Vdash^w 6 \ \mathbf{thread}(d, q, [\eta, c_m/x_m, d/z], \mathbf{case}\ x_m\ \{l(y) \Rightarrow P\}_{l \in L}), \mathbf{cell}(d, -, n'') :: \Psi_{5b}, (d^{n''} : D)} \ \text{C:Thread}$$

where $\Psi_6 = \Psi_{5b}, (d^{n''} : D)$. Notably, $\Gamma = \Gamma', (x_m : \oplus_m \{l : Y_{3m}^l\}_{l \in L})$.

Now, we turn our attention to the resulting configuration. Obviously, the first requirement is satisfied trivially. Next, when a is allocated, it has the same type as before (by C:Val or C:Thread, depending on whether the cell has been written), with one additional reference. Specifically, we get $\Psi_2' = \Psi_2'', (a^{k \oplus 1} : A)$. Then, in $\mathcal{C}_2$, we again use up the same amount of a as in the original configuration. Then, by Lemma 8, we have $\Psi_3' = \Psi_3'', (a^{k_1 \oplus 1} : A_1)$.

Next, when we allocate c_m, we note that a might appear to have a different type than it did originally; this is because it must share its type with an extra reference, in the case that it is linear. Specifically, we type c_m as $(c_m^{n-1} : \oplus_m \{l : X'^l_m\}_{l \in L})$ where $X'^i = Y_1^i + X_2^i + Z_3^i$. Then we are left with $(a^{k_1 \ominus 1} : A_2) + (a^1 : Y_3^i)$. We can do this because since $X^i = B_2$, we know that $B_2 = (Y_1^i + X_2^i + Z_3^i + Y_3^i)$, where Y_1^i was used by $\mathcal{C}_3$, X_2^i remains in the resulting configuration fed to $\mathcal{C}_4$, and Z_3^i is used for P_l.

Then, in $\mathcal{C}_3$, some of $(a^{A_2} : k_1 - 1)$ is used up, as in the initial configuration. This leaves us with $(a^{k_2} : A_3) + (a^1 : Y_3^i)$, by Lemma 8. Similarly, some amount of c_m is also used up. Given that we begin with $(c_m^{n-1} : \oplus_m \{l : X'^l_m\}_{l \in L})$ and use up $(c_m^{n-n_1} : \oplus_m \{l : Y_{1m}^l\}_{l \in L})$, we are left with $(c_m^{n_1 - 1} : \oplus_m \{l : X''^l_m\}_{l \in L})$ where $X''^l = (X_2^l + Z_3^l)$.

Finally, we consider the thread running P_i. We use $(a^1 : Y_3^i)$ in the environment when we substitute it for y, so $(y : Y_3^i)$. Otherwise, the same amount of $(a^{k_2} : A_3)$ is used up by P_i as in the initial configuration, so we are left with the same result in terms of a. Similarly, we use some of

$(c_m^{n_1-1} : \oplus_m\{l : {X''}^l_m\}_{l\in L})$ in P_i and leave the rest in the resulting configuration, exactly as before. Then, we can obviously type P_i as desired, by the premises of the original thread's typing. Thus, we end up with the same $\Psi'_6 = \Psi_6$. Then, by (vii), (g) obviously holds and gives us the desired conclusion.

Then, by C:Join, we have preservation in this case.

$\triangleright L$ We consider the step

$$\begin{array}{l} \quad \mathcal{C}_1, \mathsf{cell}(a, D, k), \mathcal{C}_2, \mathsf{cell}(c_m, \mathsf{pot}\ q\ (a), n), \mathcal{C}_3, \\ \quad \mathsf{thread}(d, q', [\eta, c_m/x_m, d/z], \mathsf{case}\ x_m\ \{\mathsf{pot}\ q''\ (y) \Rightarrow P\}), \mathsf{cell}(d, -, n''), \mathcal{C}_4 \\ \longmapsto \mathcal{C}_1, \mathsf{cell}(a, D, k \oplus 1), \mathcal{C}_2, \mathsf{cell}(c_m, \mathsf{pot}\ (q - q'')\ (a), n-1), \mathcal{C}_3, \\ \quad \mathsf{thread}(d, q' + q'', [\eta, a/y, d/z], P_i), \mathsf{cell}(d, -, n''), \mathcal{C}_4 \end{array}$$

$\mathcal{C}$ has the following typing, by inversions on C:Join:

(i) $\Psi \Vdash^{w_1} \mathcal{C}_1 :: \Psi_1$

(ii) $\Psi_1 \Vdash^{w_2} \mathsf{cell}(a, D, k) :: \Psi_2$

(iii) $\Psi_2 \Vdash^{w_3} \mathcal{C}_2 :: \Psi_3$

(iv) $\Psi_3 \Vdash^{w_4} \mathsf{cell}(c_m, \mathsf{pot}\ q\ (a), n) :: \Psi_4$

(v) $\Psi_4 \Vdash^{w_5} \mathcal{C}_3 :: \Psi_5$

(vi) $\Psi_5 \Vdash^{w_6} \mathsf{thread}(d, q, [\eta, c_m/x_m, d/z], \mathsf{case}\ x_m\ \{\mathsf{pot}\ q''\ (y) \Rightarrow P\}), \mathsf{cell}(d, -, n'') :: \Psi_6$

(vii) $\Psi_6 \Vdash^{w_7} \mathcal{C}_4 :: \Psi'$

To show that

$$\begin{array}{l} \mathcal{C}_1, \mathsf{cell}(a, D, k \oplus 1), \mathcal{C}_2, \mathsf{cell}(c_m, \mathsf{pot}\ (q - q'')\ (a), n-1), \\ \mathcal{C}_3, \mathsf{thread}(d, q + q'', [\eta, a/y, d/z], P), \mathsf{cell}(d, -, n''), \mathcal{C}_4 \end{array}$$

has the same typing, we must show

(a) $\Psi \Vdash^{w_1} \mathcal{C}_1 :: \Psi_1$

(b) $\Psi_1 \Vdash^{w'_2} \mathsf{cell}(a, D, k \oplus 1) :: \Psi'_2$

(c) $\Psi'_2 \Vdash^{w'_3} \mathcal{C}_2 :: \Psi'_3$

(d) $\Psi'_3 \Vdash^{w'_4} \mathsf{cell}(c_m, \mathsf{pot}\ (q - q'')\ (a), n-1) :: \Psi'_4$

(e) $\Psi_4' \Vdash^{w_5'} \mathcal{C}_3 :: \Psi_5'$

(f) $\Psi_5' \Vdash^{w_6'} \mathsf{thread}(d, q+q'', [\eta, a/y, d/z], P), \mathsf{cell}(d, -, n'') :: \Psi_6'$

(g) $\Psi_6' \Vdash^{w_7'} \mathcal{C}_4 :: \Psi'$

where $w_1 + w_2' + w_3' + w_4' + w_5' + w_6' + w_7' = w_1 + w_2 + w_3 + w_4 + w_5 + w_6 + w_7$.

First, we examine the initial configuration, providing names for each type seen by the configuration. When we allocate cell a, we get $\Psi_2 = \Psi_2'', (a^k : A)$ for some A.

Then, $\mathcal{C}_2$ can use up some of a; this occurs through splitting contexts using +. Thus, after typing $\mathcal{C}_2$, we have $\Psi_3 = \Psi_3'', (a^{k_1} : A_1)$, where $A = (A_1 + B_1)$ and $(a^{k-k_1} : B_1)$ was used up by $\mathcal{C}_2$.

Next, we allocate c_m by Val-$\triangleright$R^0, referring to a. This costs q potential, meaning that $q = w_4$. Thus, one reference to a is used up, leaving $\Psi_4 = \Psi_4'', (a^{k_1-1} : A_2), (c_m^n : \triangleright^q X_{\mathsf{L}})$, where $A_1 = (A_2 + B_2)$. Notably, $X = B_2$.

We then use up some more of a and some of c in $\mathcal{C}_3$. We are left with $(a^{k_2} : A_3)$ where $A_2 = (A_3 + B_3)$ and $\mathcal{C}_3$ used up $(a^{k_1-1-k_2} : B_3)$. We also have $(c_m^{n_1} : \triangleright^{q_1} X_{1_{\mathsf{L}}})$, where $X = (X_1 + Y_1)$ and $q = q_1 + q_2$ (by the + relation on sum types), and $\mathcal{C}_3$ used up $(c_m^{n-n_1} : \triangleright^{q_2} Y_{1_{\mathsf{L}}})$.

Finally, we consider the running thread that reads from c_m. Some of a may be used up in typing P (i.e. in the environment), while the rest remains in the resulting configuration. Specifically, after typing this thread, we are left with $(a^{k_3} : A_4)$ for $A_3 = (A_4 + B_4)$ and $(a^{k_2-k_3} : B_4)$ was used up by P_l. When it comes to c_m, we have one additional place a reference is used: in the case expression. Overall, we use some of c_m in typing this thread and leave the rest in the resulting configuration. After typing the thread, we are left with $(c^{n_2} : \triangleright^{q_3} X_{2_{\mathsf{L}}})$, where $X_1 = (X_2 + Y_2)$, $q_1 = q_3 + q_4$, and $(c_m^{n_1-n_2} : \triangleright^{q_4} Y_{2_{\mathsf{L}}})$ is used by the thread. Within these references, one is used for the variable x_m, so we have $(x : \triangleright^{q_5} Y_{3_{\mathsf{L}}})$ where $Y_2 = (Y_3 + Z_3)$ and $q_4 = q_5 + q_6$. The remainder of the cell (i.e., $(c_m^{n_1-n_2-1} : \triangleright^{q_6} Z_{3_{\mathsf{L}}})$) is used in P.
In addition, by inversion on (vi) with C:Thread and $\triangleright L$, we know that

$$\dfrac{\dfrac{\Psi_{5a} \vdash \eta, c_m/x_m : \Gamma \qquad \dfrac{\Gamma, (y : A) \vdash^{w_6+q''} P :: (z : D)}{\Gamma \vdash^{w_6} \mathbf{case}\ x_m\ \{\mathbf{pot}\ q''\ (y) \Rightarrow P\} :: (z : D)} \triangleright \mathrm{L} \qquad \Psi_5 = \Psi_{5a} + \Psi_{5b}}{\Psi_5 \Vdash^{w_6} \mathbf{thread}(d, q, [\eta, c_m/x_m, d/z], \mathbf{case}\ x_m\ \{\mathbf{pot}\ q''\ (y) \Rightarrow P\}), \mathbf{cell}(d, -, n'') :: \Psi_{5b}, (d^{n''} : D)}\ \text{C:Thread}$$

where $\Psi_6 = \Psi_{5b}, (d^{n''} : D)$. Notably, $\Gamma = \Gamma', (x_m : \triangleright^{q''} Y_{3_{\mathsf{L}}})$, so $q'' = q_5$.

Now, we turn our attention to the resulting configuration. Obviously, the first requirement is satisfied trivially. Next, when a is allocated, it has the same type as before (by C:Val or C:Thread, depending on

whether the cell has been written), with one additional reference. Specifically, we get $\Psi_2' = \Psi_2'', (a^{k \oplus 1} : A)$. Then, in $\mathcal{C}_2$, we again use up the same amount of a as in the original configuration. Then, by Lemma 8, we have $\Psi_3' = \Psi_3'', (a^{k_1 \oplus 1} : A_1)$.

Next, when we allocate c_m, we note that a might appear to have a different type than it did originally; this is because it must share its type with an extra reference, in the case that it is linear. In addition, c_m now carries $q - q''$ potential rather than q potential. Specifically, we type c_m as $(c_m^{n-1} : \triangleright^{(q_2+q_3+q_6)} X'_{\mathsf{L}})$ where $X' = Y_1 + X_2 + Z_3$. Then we are left with $(a^{k_1 \ominus 1} : A_2) + (a^1 : Y_3)$. We can do this because since $X = B_2$, we know that $B_2 = (Y_1 + X_2 + Z_3 + Y_3)$, where Y_1 and q_2 potential was used by $\mathcal{C}_3$, X_2 and q_3 potential remains in the resulting configuration fed to $\mathcal{C}_4$, and Z_3 and q_6 potential is used for P_l. This allocation now only costs $q_2 + q_3 + q_6$ potential, rather than a full $q = q_3 + q_5 + q_6 + q_2$, leaving us an additional q_5 to allocate elsewhere.

Then, in $\mathcal{C}_3$, some of $(a^{A_2} : k_1 - 1)$ is used up, as in the initial configuration. This leaves us with $(a^{k_2} : A_3) + (a^1 : Y_3)$, by Lemma 8. Similarly, some amount of c_m is also used up. Given that we begin with $(c_m^{n-1} : \triangleright^{(q_2+q_3+q_6)} X'_{\mathsf{L}})$ and use up $(c_m^{n-n_1} : \triangleright^{q_2} Y_{1\mathsf{L}})$, we are left with $(c_m^{n_1-1} : \triangleright^{(q_3+q_6)} X''_{\mathsf{L}})$ where $X'' = (X_2 + Z_3)$.

Finally, we consider the thread running P. We use $(a^1 : Y_3)$ in the environment when we substitute it for y, so $(y : Y_3)$. Otherwise, the same amount of $(a^{k_2} : A_3)$ is used up by P as in the initial configuration, so we are left with the same result in terms of a. Similarly, we use some of $(c_m^{n_1-1} : \triangleright^{(q_3+q_6)} X''_{\mathsf{L}})$ in P and leave the rest in the resulting configuration, exactly as before. Then, we can obviously type P as desired, by the premises of the original thread's typing. Notably, we need $w_6 + q_5$ to type this thread, but we have an extra q_5 potential accessible to us because we did not use it in typing c_m. Thus, we end up with the same $\Psi_6' = \Psi_6$. Then, by (vii), (g) obviously holds and gives us the desired conclusion.

Then, by C:Join and the fact that the potential used remains constant, we have preservation in this case.

$\otimes L$, $\mathbf{1}L$, $\downarrow L$ These cases follow similarly to those above.

$\oplus L2$ Next, we consider the case where c_m is not multilinear. We consider the step

$$\begin{aligned} &\mathcal{C}_1, \mathsf{cell}(c_m, i(c'), \sigma), \mathcal{C}_2, \\ &\mathsf{thread}(d_k, q, [(\eta + c_m/x_m), d_k/z], \mathsf{case}\ x_m\ \{(l(y) \Rightarrow P_l)_{l \in L}\}), \mathsf{cell}(d_k, -, n), \mathcal{C}_3 \\ \longmapsto\ &\mathcal{C}_1, [\mathsf{cell}(c_m, i(c'), \sigma)], \mathcal{C}_2, \mathsf{thread}(d_k, q, [\eta, c'/y, d_k/z], P_i), \mathsf{cell}(d_k, -, n), \mathcal{C}_3 \end{aligned}$$

$\mathcal{C}$ has the following typing, by inversions on C:Join:

(i) $\Psi \Vdash^{w_1} \mathcal{C}_1 :: \Psi_1$

(ii) $\Psi_1 \Vdash^{w_2} \mathsf{cell}(c_m, i(c'), \sigma) :: \Psi_2$

(iii) $\Psi_2 \Vdash^{w_3} \mathcal{C}_2 :: \Psi_3$

(iv) $\Psi_3 \Vdash^{w_4} \mathsf{thread}(d_k, q, [(\eta + c_m/x_m), d_k/z], \mathsf{case}\ x_m\ \{(l(y) \Rightarrow P_l)_{l \in L}\}), \mathsf{cell}(d_k, -, n) :: \Psi_4$

(v) $\Psi_4 \Vdash^{w_5} \mathcal{C}_3 :: \Psi'$

To show that $\mathcal{C}_1, [\mathsf{cell}(c_m, i(c'), \sigma)], \mathcal{C}_2, \mathsf{thread}(d_k, q, [\eta, c'/y, d_k/z], P_i), \mathsf{cell}(d_k, -, n), \mathcal{C}_3$ has the same typing, we case on whether c_m is linear or unrestricted.

Unrestricted In the unrestricted case, we note the following:

(a) $\Psi \Vdash^{w_1} \mathcal{C}_1 :: \Psi_1$, directly by (i)

(b) $\Psi_1 \Vdash^{w_2} \mathsf{cell}(c_m, i(c'), \omega) :: \Psi_2$, directly by (ii)

(c) $\Psi_2 \Vdash^{w_3} \mathcal{C}_2 :: \Psi_3$, directly by (iii)

(d) By inversion on (iv) we have

$$\dfrac{\Psi_{3a} \vdash \eta + c_m/x_m : \Gamma \qquad \dfrac{\dfrac{\Gamma, (y : A) \vdash^{w_4} P_i :: (z : D)}{\Gamma \vdash^{w_4} \mathbf{case}\ x_m\ \{l(y) \Rightarrow P\}_{l \in L} :: (z : D)} \oplus\mathrm{L} \qquad \Psi_3 = \Psi_{3a} + \Psi_{3b}}{\Psi_3 \Vdash^{w_4} \mathbf{thread}(d, q, [(\eta + c_m/x_m), d_k/z], \mathbf{case}\ x_m\ \{l(y) \Rightarrow P\}_{l \in L}), \mathbf{cell}(d, -, n) :: \Psi_{3b}, (d_k^n : D)}\ \text{C:Thread}$$

In addition, since c_m is unrestricted, c' must also be an unrestricted address, so it must persist into Ψ_3 and thus into Ψ_{3a} and Γ. Then, we see

$$\dfrac{\Psi_{3a} \vdash (\eta + c_m/x_m), c'/y : \Gamma, (y : A) \qquad \Gamma, (y : A) \vdash^{w_4} P_i :: (z : D) \qquad \Psi_3 = \Psi_{3a} + \Psi_{3b}}{\Psi_3 \Vdash^{w_4} \mathbf{thread}(d, q, [(\eta + c_m/x_m), d_k/z], \mathbf{case}\ x_m\ \{l(y) \Rightarrow P\}_{l \in L}), \mathbf{cell}(d, -, n) :: \Psi_{3b}, (d_k^n : D)}\ \text{C:Thread}$$

so we conclude $\Psi_3 \Vdash^{w_4} \mathsf{thread}(d_k, q, [\eta, c_m/x_m, c'/y, d_k/z], P_i), \mathsf{cell}(d_k, -, n) :: \Psi_4$.

(e) $\Psi_4 \Vdash^{w_5} \mathcal{C}_3 :: \Psi'$, directly by (v)

Then, by C:Join, we have the desired conclusion.

Linear In the linear case, we note the following:

(a) $\Psi \Vdash^{w_1} \mathcal{C}_1 :: \Psi_1$, directly by (i)

(b) By C:Val and (ii), we have

$$\dfrac{\Psi_1' \vdash^0 i(c) : A_m \qquad \Psi_1 = \Psi_1' + \Psi_1''}{\Psi_1 \Vdash^0 \mathsf{cell}(c_m, i(c'), 1) :: \Psi_1'', (c_m^1 : A_m)}\ \text{C:Val}$$

noting that w_2 must be 0 and $\Psi_2 = \Psi_1'', (c_m^1 : A_m)$. Because c_m is linear, we know that c' must also be linear. Then, $\Psi_1' = \Psi_{1,W}', (c'^1 : A_m^i)$. This means that $\Psi_1 = \Psi_1'' + (\Psi_{1,W}', (c'^1 : A_m^i))$. Since $\Psi_{1,W}'$ is weakenable, it persists into Ψ_1'' and Ψ_2, so $\Psi_1 = \Psi_1'' + (c'^1 : A_m^i)$.

Because c_m has a reference count of 1 and was used in the original configuration in the running thread, it must not be used in $\mathcal{C}_2$. Then, given $\Psi_3 = \Psi_3', (c_m^1 : A_m)$, we get $\Psi_2 \Vdash^{w_2+w_3} \mathcal{C}_2 :: \Psi_3' + (c'^1 : A_m^i)$ (noting that $w_2 = 0$).

(c) By inversion on (iv) we have

$$\frac{\Psi_{3a}, (c_m^1 : A_m) \vdash \eta, c_m/x_m : \Gamma \qquad \dfrac{\Gamma, (y : A) \vdash^{w_4} P_i :: (z : D)}{\Gamma \vdash^{w_4} \mathbf{case}\ x_m\ \{l(y) \Rightarrow P\}_{l \in L} :: (z : D)} \oplus \mathrm{L} \qquad \Psi_3' = \Psi_{3a}, (c_m^1 : A_m) + \Psi_{3b}}{\Psi_3', (c_m^1 : A_m) \Vdash^{w_4} \mathbf{thread}(d, q, [\eta, c_m/x_m, d_k/z], \mathbf{case}\ x_m\ \{l(y) \Rightarrow P\}_{l \in L}), \mathbf{cell}(d, -, n) :: \Psi_{3b}, (d_k^n : D)}\ \text{C:Thread}$$

Then, we see

$$\frac{\Psi_{3a} + (c'^1 : A_m^i) \vdash \eta, c'/y : \Gamma, (y : A) \qquad \Gamma, (y : A) \vdash^{w_4} P_i :: (z : D) \qquad \Psi_3' + (c'^1 : A_m^i) = (\Psi_{3a} + (c'^1 : A_m^i)) + \Psi_{3b}}{\Psi_3' + (c'^1 : A_m^i) \Vdash^{w_4} \mathbf{thread}(d, q, [\eta, d_k/z], \mathbf{case}\ x_m\ \{l(y) \Rightarrow P\}_{l \in L}), \mathbf{cell}(d, -, n) :: \Psi_{3b}, (d_k^n : D)}\ \text{C:Thread}$$

so we conclude $\Psi_3' + (c'^1 : A_m^i) \Vdash^{w_4} \mathsf{thread}(d_k, q, [\eta, c'/y, d_k/z], P_i), \mathsf{cell}(d_k, -, n) :: \Psi_4$.

(d) $\Psi_4 \Vdash^{w_5} \mathcal{C}_3 :: \Psi'$, directly by (v)

Then, by C:Join, we have the desired conclusion.

$\otimes L2$, $\mathbf{1}L2$, $\downarrow L2$ These cases follow similarly to those above.

$\&L^0$ We consider the step

$$\begin{aligned} &\mathcal{C}_1, \mathsf{cell}(c_m, [\eta]\{p\}(l(y) \Rightarrow P_l)_{l \in L}, \sigma), \mathcal{C}_2, \\ &\mathsf{thread}(d_k, q, [(\eta' + c_m/x_m), d_k/z], x_m.i(z)), \mathsf{cell}(d_k, -, n), \mathcal{C}_3 \\ \longmapsto\ &\mathcal{C}_1, [\mathsf{cell}(c_m, [\eta]\{p\}(l(y) \Rightarrow P_l)_{l \in L}, \sigma)], \mathcal{C}_2, \mathsf{thread}(d_k, q + p, [\eta' + \eta, d_k/y], P_i), \mathsf{cell}(d_k, -, n), \mathcal{C}_3 \end{aligned}$$

$\mathcal{C}$ has the following typing, by inversions on C:Join:

(i) $\Psi \Vdash^{w_1} \mathcal{C}_1 :: \Psi_1$

(ii) $\Psi_1 \Vdash^{w_2} \mathsf{cell}(c_m, [\eta]\{p\}(l(y) \Rightarrow P_l)_{l \in L}, \sigma) :: \Psi_2$

(iii) $\Psi_2 \Vdash^{w_3} \mathcal{C}_2 :: \Psi_3$

(iv) $\Psi_3 \Vdash^{w_4} \mathsf{thread}(d_k, q, [(\eta' + c_m/x_m), d_k/z], x_m.i(z)), \mathsf{cell}(d_k, -, n) :: \Psi_4$

(v) $\Psi_4 \Vdash^{w_5} \mathcal{C}_3 :: \Psi'$

It suffices to show:

(a) $\Psi \Vdash^{w_1} \mathcal{C}_1 :: \Psi_1$

(b) $\Psi_1 \Vdash^{w_3} \mathcal{C}_2 :: \Psi_3'$

(c) $\Psi_3' \Vdash^{w_4'} \mathsf{thread}(d_k, q+p, [\eta'+\eta, d_k/z], P_i), \mathsf{cell}(d_k, -, n) :: \Psi_4'$

(d) $\Psi_4' \Vdash^{w_5} \mathcal{C}_3 :: \Psi'$

where $w_4' = w_2 + w_4$.

Now, we consider two cases, based on whether c_m is linear or shared:

Linear Obviously, (a) follows directly from (i). By (ii), we have

$$\frac{\Psi_{1a} \vdash \eta : \Gamma \qquad \dfrac{\Gamma \vdash^{w_2} P_l :: (y : B_m^l) \text{ (for all } l)}{\Gamma \vdash^{w_2} \mathbf{case}\ x\ \{(l(y) \Rightarrow P_l)_{l \in L}\} :: (x : \&_m\{l : B_m^l\}_{l\in L})}\ \&\mathrm{R} \qquad \Psi_1 = \Psi_{1a} + \Psi_2''}{\Psi_1 \Vdash^{w_2} \mathbf{cell}(c_m, [\eta]\{w_2\}(l(y) \Rightarrow P_l)_{l\in L}, 1) :: \Psi_2'', (c_m^1 : \&_m\{l : B_m^l\}_{l\in L})}\ \text{C:Cont}$$

where $w_2 = p$.

Then, since $\Psi_1 = \Psi_{1a} + \Psi_2''$ and $\mathcal{C}_2$ does not use c_m, we have that $\Psi_1 \Vdash^{w_3} \mathcal{C}_2 :: \Psi_3' + \Psi_{1a}$, by Lemma 8, where $\Psi_3 = \Psi_3', (c_m^1 : \&_m\{l : B_m^l\}_{l\in L})$.
We also have by (iv) that

$$\frac{\Psi_{3a} \vdash \eta', c_m/x_m : \Gamma_W, (x : \&_m\{l : B_m^l\}_{l\in L}) \qquad \dfrac{i \in L}{\Gamma_W, (x : \&_m\{l : B_m^l\}_{l\in L}) \vdash^{w_4} x_m.i(z) :: (z : A_k)} \qquad \Psi_3 = \Psi_{3a} + \Psi_{3b}}{\Psi_3 \Vdash^{w_4} \mathbf{thread}(d_k, q, [\eta', c_m/x_m, d_k/z], x_m.i(z)), \mathbf{cell}(d_k, -, n) :: \Psi_{3b}, (d_k^\eta : A_k)}$$

noting that $q = w_4 = 0$. Now, by inversion on E2, we have that Ψ_{3a} must be some $\Psi_{3a}', (c_m^1 : \&_m\{l : B_m^l\}_{l\in L})$. Thus, $\Psi_3' = \Psi_{3a}' + \Psi_{3b}$.

Next, we consider (c):

$$\frac{\begin{array}{c}\Psi_{3a}' + \Psi_{1a} \vdash \eta + \eta' : \Gamma_W, \Gamma \\ \Gamma_W, \Gamma \vdash^p P_i :: (y : A_k) \qquad \Psi_3' + \Psi_{1a} = \Psi_{3a}' + \Psi_{3b} + \Psi_{1a}\end{array}}{\Psi_3' + \Psi_{1a} \Vdash^p \mathsf{thread}(d_k, p, [\eta+\eta', d_k/y], P_i), \mathsf{cell}(d_k, -, n) :: \Psi_{3b}, (d_k^\eta : A_k)}\ \text{C:Thread}$$

noting that $q + p = p$.

Now, we can conclude that the desired typing holds, since (d) follows from (v). Also, we have $w_2 + w_4 = w_4' = p$ since $w_4 = 0$ and $w_2 = p$.

Shared Obviously, (a), (b), and (c) follow directly from (i), (ii), and (iii). By (ii), we have

$$\frac{\Psi_{1a} \vdash \eta : \Gamma \qquad \dfrac{\Gamma \vdash^{w_2} P_l :: (y : B_m^l) \text{ (for all } l)}{\Gamma \vdash^{w_2} \mathbf{case}\ x\ \{(l(y) \Rightarrow P_l)_{l \in L}\} :: (x : \&_m\{l : B_m^l\}_{l\in L})}\ \&\mathrm{R} \qquad \Psi_1 = \Psi_{1a} + \Psi_2''}{\Psi_1 \Vdash^{w_2} \mathbf{cell}(c_m, [\eta]\{w_2\}(l(y) \Rightarrow P_l)_{l\in L}, \omega) :: \Psi_2'', (c_m^\omega : \&_m\{l : B_m^l\}_{l\in L})}\ \text{C:Cont}$$

where $w_2 = p = 0$ because c_m is unrestricted.
We also have by (iv) that

$$\frac{\Psi_{3a} \vdash \eta' : \Gamma_W, (x : \&_m\{l : B_m^l\}_{l \in L}) \qquad \dfrac{}{\Gamma_W, (x : \&_m\{l : B_m^l\}_{l \in L}) \vdash^{w_4} x.m.i(z) :: (z : B_m^i)}\ i \in L \qquad \Psi_3 = \Psi_{3a} + \Psi_{3b}}{\Psi_3 \Vdash^{w_4} \mathbf{thread}(d_k, q, [(\eta' + c_m/x_m), d_k/z], x.m.i(z)), \mathbf{cell}(d_k, -, n) :: \Psi_{3b}, (d_k^n : B_m^i)}$$

noting that $q = w_4 = 0$ and that $c_m/x_m \in \eta'$.

Now, we must show

$$\frac{\Psi'_{3a} \vdash \eta' + \eta : \Gamma_W, \Gamma \qquad \Gamma_W, \Gamma \vdash^0 P_i :: (y : B_m^i) \qquad \Psi_3 = \Psi'_{3a} + \Psi_{3b}}{\Psi_3 \Vdash^0 \mathsf{thread}(d_k, q, [(\eta' + \eta + c_m/x_m), d_k/y], P_i), \mathsf{cell}(d_k, -, n) :: \Psi_{3b}, (d_k^n : B_m^i)}\ \textsc{C:Thread}$$

The first premise holds since Ψ'_{3a} can be expressed as $\Psi_{3a} + \Psi_{1a}$ (since Ψ_{1a} is unrestricted). The second premise holds by the premise of (ii), and the third premise holds by the fact that $\Psi_3 = \Psi_{3a} + \Psi_{3b}$. Thus, we can conclude that (e) follows from (v), and that the potential is the same as in the original configuration, so $\mathcal{C}'$ has the desired typing.

$\multimap L^0$, $\uparrow L^0$, $\triangleleft L^0$ Similarly to above.

Cut We examine the step

$$\begin{array}{ll} & \mathcal{C}_1, \mathsf{thread}(c_m, w_1 + w_2, [\eta_1 + \eta_2, c_m/y], x \leftarrow^{w_1} P\ ;\ Q), \mathsf{cell}(c_m, -, n), \mathcal{C}_2 \\ \longmapsto & \mathcal{C}_1, \mathsf{thread}(a, w_1, [\eta_1, a/x], P), \mathsf{cell}(a, -, 1), \\ & \mathsf{thread}(c_m, w_2, [\eta_2, a/x, c_m/y], Q), \mathsf{cell}(c_m, -, n), \mathcal{C}_2\ (a\ \mathsf{fresh}) \end{array}$$

Now, by C:Join on the original configuration, we have

(i) $\Psi \Vdash^{q_1} \mathcal{C}_1 :: \Psi_1$

(ii) $\Psi_1 \Vdash^{q_2} \mathsf{thread}(c_m, w_1 + w_2, [\eta_1 + \eta_2, c_m/y], x \leftarrow^{w_1} P\ ;\ Q), \mathsf{cell}(c_m, -, n) :: \Psi_2$

(iii) $\Psi_2 \Vdash^{q_3} \mathcal{C}_2 :: \Psi'$

By (ii) we have

$$\frac{\Psi_{1a} \vdash \eta_1 + \eta_2 : \Gamma \qquad \Gamma \vdash^{w_1+w_2} x \leftarrow^{w_1} P\ ;\ Q :: (y : A_m) \qquad \Psi_1 = \Psi_{1a} + \Psi_{1b}}{\Psi_1 \Vdash^{w_1+w_2} \mathsf{thread}(c_m, w_1 + w_2, [\eta_1 + \eta_2, c_m/y], x \leftarrow^{w_1} P\ ;\ Q), \mathsf{cell}(c_m, -, n) :: \Psi_{1b}, (c_m^n : A_m)}\ \textsc{C:Thread}$$

where $\Psi_2 = \Psi_{1b}, (c_m^n : A_m)$. Then, by inversion on the second premise, we have

$$\frac{m \leq k \qquad \Gamma_1 \vdash^{w_1} P :: (x : B_k) \qquad \Gamma_2, (x : B_k) \vdash^{w_2} Q :: (y : A_m) \qquad \Gamma = \Gamma_1 + \Gamma_2}{\Gamma \vdash^{w_1 + w_2} x \leftarrow^{w_1} P \,;\, Q :: (y : A_m)} \textsc{Cut}$$

Given this, we will show:

(a) $\Psi \Vdash^{q_1} \mathcal{C}_1 :: \Psi_1$

(b) $\Psi_1 \Vdash^{q_2} \mathsf{thread}(a, w_1, [\eta_1, a/x], P), \mathsf{cell}(a, -, 1), \mathsf{thread}(c_m, w_2, [\eta_2, a/x, c_m/y], Q), \mathsf{cell}(c_m, -, n) :: \Psi_2$

(c) $\Psi_2 \Vdash^{q_3} \mathcal{C}_2 :: \Psi'$

Obviously, (a) holds by (i), and (c) holds by (iii). It remains to show (b).

First, let $\Psi_{1a} = \Psi_a + \Psi_b$ where $\Psi_a \vdash \eta_1 : \Gamma_1, \Psi_b \vdash \eta_2 : \Gamma_2$; we know this is possible because $\Psi_{1a} \vdash \eta_1 + \eta_2 : \Gamma_1 + \Gamma_2$. Then, note that we have

$$\frac{\Psi_a \vdash \eta_1 : \Gamma_1 \qquad \Gamma_1 \vdash^{w_1} P :: (x : B_k) \qquad \Psi_1 = (\Psi_a + \Psi_b) + \Psi_{1b}}{\Psi_1 \Vdash^{w_1} \mathsf{thread}(a, w_1, [\eta_1, a/x], P), \mathsf{cell}(a, -, 1) :: \Psi_{1b} + (\Psi_b, (a^1 : B_k))} \textsc{C:Thread}$$

We also have

$$\frac{\begin{array}{c}\Psi_b, (a^1 : B_k) \vdash \eta_2, a/x : \Gamma_2, (x : B_k) \\ \Gamma_2, (x : B_k) \vdash^{w_2} Q :: (y : A_m) \qquad \Psi_{1b} + (\Psi_b, (a^1 : B_k)) = \Psi_{1b} + (\Psi_b, (a^1 : B_k))\end{array}}{\Psi_{1b} + (\Psi_b, (a^1 : B_k)) \Vdash^{w_2} \mathsf{thread}(c_m, w_2, [\eta_2, a/x, c_m/y], Q), \mathsf{cell}(c_m, -, n) :: \Psi_{1b}, (c_m^n : A_m)} \textsc{C:Thread}$$

Finally, putting these together, we have

$$\frac{\begin{array}{c}\Psi_1 \Vdash^{w_1} \mathsf{thread}(a, w_1, [\eta_1, a/x], P), \mathsf{cell}(a, -, 1) :: \Psi_1' \\ \Psi_1' \Vdash^{w_2} \mathsf{thread}(c_m, w_2, [\eta_2, a/x, c_m/y], Q), \mathsf{cell}(c_m, -, n) :: \Psi_2\end{array}}{\Psi_1 \Vdash^{w_1 + w_2} \mathsf{thread}(a, w_1, [\eta_1, a/x], P), \mathsf{cell}(a, -, 1), \mathsf{thread}(c_m, w_2, [\eta_2, a/x, c_m/y], Q), \mathsf{cell}(c_m, -, n) :: \Psi_2} \textsc{C:Join}$$

Thus, we can conclude that (b) holds.

Then, by C:Join, the resulting configuration is well-typed.

Call We consider the step

$$\begin{array}{l}\mathcal{C}_1, \mathsf{thread}(c_m, q, [\eta, c_m/z], z \leftarrow p[\zeta]), \mathsf{cell}(c_m, -, n), \mathcal{C}_2 \\ \longmapsto \mathcal{C}_1, \mathsf{thread}(c_m, q, [\eta \circ \zeta, c_m/x], P), \mathsf{cell}(c_m, -, n), \mathcal{C}_2 \text{ (given } P = x \leftarrow p\ \Delta \in \Sigma)\end{array}$$

Now, by C:Join on the original configuration, we have

(i) $\Psi \Vdash^{q_1} \mathcal{C}_1 :: \Psi_1$

(ii) $\Psi_1 \Vdash^{q_2} \mathsf{thread}(c_m, q, [\eta, c_m/z], z \leftarrow p[\zeta]), \mathsf{cell}(c_m, -, n) :: \Psi_2$

(iii) $\Psi_2 \Vdash^{q_3} \mathcal{C}_2 :: \Psi'$

By inversion on (ii), we have

$$\frac{\Psi_{1a} \vdash \eta : \Gamma \qquad \Gamma \vdash^q z \leftarrow p[\zeta] :: (z : A_m) \qquad \Psi_1 = \Psi_{1a} + \Psi_{1b}}{\Psi_1 \Vdash^q \mathsf{thread}(c_m, q, [\eta, c_m/z], z \leftarrow p[\zeta]), \mathsf{cell}(c_m, -, n) :: \Psi_{1b}, (c_m^n : A_m)} \text{ C:Thread}$$

where $\Psi_2 = \Psi_{1b}, (c_m^n : A_m)$. Then, by inversion on the second premise, we have

$$\frac{\Gamma \vdash \zeta : \Delta \qquad \Delta \vdash^q P :: (x : A_m) \in \Sigma}{\Gamma \vdash^q z \leftarrow p[\zeta] :: (z : A_m)} \text{ Call}$$

Notably, since everything in Σ has been typechecked, we have $\Delta \vdash^q P :: (x : A_m)$ well-typed.

Then, we see that

$$\frac{\Psi_{1a} \vdash \eta : \Gamma \qquad \Gamma \vdash \zeta : \Delta}{\Psi_{1a} \vdash \eta \circ \zeta : \Delta} \text{ Clos-Comp}$$

so

$$\frac{\Psi_{1a} \vdash \eta \circ \zeta : \Delta \qquad \Delta \vdash^q P :: (x : A_m) \qquad \Psi_1 = \Psi_{1a} + \Psi_{1b}}{\Psi_1 \Vdash^q \mathsf{thread}(c_m, q, [\eta \circ \zeta, c_m/x], P), \mathsf{cell}(c_m, -, n) :: \Psi_{1b}, (c_m^n : A_m)} \text{ C:Thread}$$

Then, we can conclude that

(a) $\Psi \Vdash^{q_1} \mathcal{C}_1 :: \Psi_1$ by (i)

(b) $\Psi_1 \Vdash^{q_2} \mathsf{thread}(c_m, q, [\eta \circ \zeta, c_m/x], P), \mathsf{cell}(c_m, -, n) :: \Psi_{1b}, (c_m^n : A_m)$ by (ii) and the above

(c) $\Psi_2 \Vdash^{q_3} \mathcal{C}_2 :: \Psi'$ by (iii)

Finally, by C:Join, the desired typing holds.

IdK We consider the step

$$\begin{array}{l} \mathcal{C}_1, \mathsf{cell}(c_m, [\eta']\{p\}K, \sigma), \mathcal{C}_2, \\ \quad \mathsf{thread}(d_m, 0, [(\eta + c_m/x_m), d_m/y_m], y_m \leftarrow x_m), \mathsf{cell}(d_m, -, n), \mathcal{C}_3 \\ \longmapsto \mathcal{C}_1, [\mathsf{cell}(c_m, [\eta']\{p\}K, \sigma)], \mathcal{C}_2, \mathsf{cell}(d_m, [\eta' + \eta]\{p\}K, \sigma), \mathcal{C}_3 \end{array}$$

$\mathcal{C}$ has the following typing, by inversions on C:Join:

(i) $\Psi \Vdash^{w_1} \mathcal{C}_1 :: \Psi_1$

(ii) $\Psi_1 \Vdash^{w_2} \mathsf{cell}(c_m, [\eta']\{p\}K, \sigma) :: \Psi_2$

(iii) $\Psi_2 \Vdash^{w_3} \mathcal{C}_2 :: \Psi_3$

(iv) $\Psi_3 \Vdash^{w_4} \mathsf{thread}(d_m, 0, [(\eta + c_m/x_m), d_m/y_m], y_m \leftarrow x_m), \mathsf{cell}(d_m, -, 1) :: \Psi_4$

(v) $\Psi_4 \Vdash^{w_5} \mathcal{C}_3 :: \Psi'$

By (ii), we have

$$\frac{\Psi_{1a} \vdash \eta' : \Gamma \qquad \Gamma \vdash^p \mathsf{case}\ x_m\ \{K\} :: (x_m : A_m) \qquad \Psi_1 = \Psi_{1a} + \Psi_2''}{\Psi_1 \Vdash^p \mathsf{cell}(c_m, [\eta']\{p\}K, \sigma) :: \Psi_2'', (c_m^\sigma : A_m)}\ \text{C:Cont}$$

We also have by (iii) that $\Psi_2'', (c_m^\sigma : A_m) \Vdash^{w_3} \mathcal{C}_2 :: \Psi_3$.

Then, note that by (iv), we have

$$\frac{\Psi_{3a} \vdash \eta + c_m/x_m : \Gamma_W, (x : A_m) \qquad \overline{\Gamma_W, (x : A_m) \vdash^0 y_m \leftarrow x_m :: (y_m : A_m)} \qquad \Psi_3 = \Psi_{3a} + \Psi_{3b}}{\Psi_3 \Vdash^0 \mathsf{thread}(d_m, 0, [(\eta + c_m/x_m), d_m/y_m], y_m \leftarrow x_m), \mathsf{cell}(d_m, -, \sigma) :: \Psi_{3b}, (d_m^\sigma : A_m)}$$

Notably, $w_4 = 0$ and $w_2 = p$.

We case on whether c_m is unrestricted or linear.

Linear Here, it suffices to show

(a) $\Psi \Vdash^{w_1} \mathcal{C}_1 :: \Psi_1$

(b) $\Psi_1 \Vdash^{w_3} \mathcal{C}_2 :: \Psi_3'$

(c) $\Psi_3' \Vdash^{w_2 + w_4} \mathsf{cell}(d_m, [\eta' + \eta]\{p\}K, 1) :: \Psi_4$

(d) $\Psi_4 \Vdash^{w_5} \mathcal{C}_3 :: \Psi'$

Obviously, (a) holds directly by (i).

Also, Ψ_{3a} must contain $(c_m^1 : A_m)$ in order to typecheck $\eta, c_m/x_m$. We write Ψ_{3a} as $\Psi'_{3a}, (c_m^1 : A_m)$, so $\Psi_3 = \Psi''_3, (c_m^1 : A_m)$. Then, by Lemma 8 and Lemma 9, we have $\Psi''_2 \Vdash^{w_3} \mathcal{C}_2 :: \Psi''_3$, and thus $\Psi_1 \Vdash^{w_3} \mathcal{C}_2 :: \Psi''_3 + \Psi_{1a}$, so (b) holds.

Next, we see that (c) holds:

$$\frac{\Psi_{1a} + \Psi'_{3a} \vdash \eta' + \eta : \Gamma + \Gamma_W \qquad \Gamma + \Gamma_W \vdash^p \mathsf{case}\ x_m\ \{K\} :: (x : A_m) \qquad \Psi'_3 = \Psi_{1a} + (\Psi'_{3a} + \Psi_{3b})}{\Psi'_3 \Vdash^p \mathsf{cell}(d_m, [\eta' + \eta]\{p\}K, 1) :: \Psi_{3b}, (d_m^1 : A_m)}\ \text{C:Cont}$$

Finally, we conclude that (d) holds by (v), and our desired typing holds.

Unrestricted We must show

(a) $\Psi \Vdash^{w_1} \mathcal{C}_1 :: \Psi_1$

(b) $\Psi_1 \Vdash^{w_2} \mathsf{cell}(c_m, [\eta']\{p\}K, \omega) :: \Psi_2$

(c) $\Psi_2 \Vdash^{w_3} \mathcal{C}_2 :: \Psi_3$

(d) $\Psi_3 \Vdash^{w_4} \mathsf{cell}(d_m, [\eta' + \eta]\{p\}K, \omega) :: \Psi_4$

(e) $\Psi_4 \Vdash^{w_5} \mathcal{C}_3 :: \Psi'$

Obviously, (a), (b), and (c) hold by (i), (ii), and (iii), respectively.

First, note that since c_m is unrestricted, all of Ψ_{1a} must also be unrestricted. Then, $\Psi_{1a} \subseteq \Psi_2$, where $\Psi_2 = \Psi_{1a} + (\Psi''_2, (c_m^\omega : A_m))$. This same argument holds for Ψ_3, so $\Psi_3 = \Psi_3 + \Psi_{1a} = (\Psi_{3a} + \Psi_{3b}) + \Psi_{1a}$. Also, notably, $p = 0$. In addition, $c_m/x_m \in \eta$.

Then, we see that (d) holds:

$$\frac{\Psi_{1a} + \Psi_{3a} \vdash \eta' + \eta : \Gamma + \Gamma'_W \qquad \Gamma + \Gamma'_W \vdash^p \mathsf{case}\ x_m\ \{K\} :: (x : A_m) \qquad \Psi_3 = \Psi_{1a} + (\Psi_{3a} + \Psi_{3b})}{\Psi_3 \Vdash^p \mathsf{cell}(d_m, [\eta' + \eta]\{p\}K, \omega) :: \Psi_{3b}, (d_m^\omega : A_m)}\ \text{C:Cont}$$

Given this, we can conclude that (e) holds by (v) and thus the desired typing holds.

Work We consider the step

$$\mathcal{C}_1, \mathsf{thread}(d_m, w, [\eta], \mathsf{work}\ \{r\}\ ;\ P), \mathsf{cell}(d_m, -, n), \mathcal{C}_2 \longmapsto^r \mathcal{C}_1, \mathsf{thread}(d_m, w - r, [\eta], P), \mathsf{cell}(d_m, -, n), \mathcal{C}_2$$

It suffices to show that if $\Psi_1 \Vdash^q \mathsf{thread}(d_m, w, [\eta], \mathsf{work}\ \{r\}\ ;\ P), \mathsf{cell}(d_m, -, n) :: \Psi_2$ then $\Psi_1 \Vdash^{q-r} \mathsf{thread}(d_m, w - r, [\eta], P), \mathsf{cell}(d_m, -, n) :: \Psi_2$, because the typing of $\mathcal{C}_1$ and $\mathcal{C}_2$ will remain the same and

thus by C:Join we will have the desired conclusion.

Now, by inversion on $\Psi_1 \Vdash^q \mathsf{thread}(d_m, w, [\eta], \mathsf{work}\ \{r\}\ ;\ P), \mathsf{cell}(d_m, -, n) :: \Psi_2$, we have that $q = w$ and that for some Ψ_1', Ψ_1'' where $\Psi_1 = \Psi_1' + \Psi_1''$, the following hold:

- $\Psi_1' \vdash \eta' : \Gamma$
- $\Gamma \vdash^w \mathsf{work}\ \{r\}\ ;\ P :: (x : A_m)$

where $\eta = \eta', d_m/x$ and $\Psi_2 = \Psi_1'', (d_m^1 : A_m)$. By inversion on $\Gamma \vdash^w \mathsf{work}\ \{r\}\ ;\ P :: (x : A_m)$, we have $\Gamma \vdash^{w-r} P :: (x : A_m)$.

Putting this together with $\Psi_1 = \Psi_1' + \Psi_1''$ and $\Psi_1' \vdash \eta' : \Gamma$, we get $\Psi_1 \Vdash^{q-r} \mathsf{thread}(d_m, w-r, [\eta], P), \mathsf{cell}(d_m, -, n) :: \Psi_2$, by C:Thread. Thus, the desired conclusion holds.

Alias We consider the following step:

$$\begin{array}{ll} & \mathcal{C}_1, \mathsf{cell}(c, V, n), \mathcal{C}_2, \mathsf{thread}(d, q, [\eta, c/x], \mathsf{alias}\ x : Q_1\ \mathsf{as}\ y : Q_2, z : Q_3\ ;\ P), \mathsf{cell}(d, -, n'), \mathcal{C}_3 \\ \longmapsto & \mathcal{C}_1, \mathsf{cell}(c, V, n+1), \mathcal{C}_2, \mathsf{thread}(d, q, [\eta, c/y, c/z], P), \mathsf{cell}(d, -, n'), \mathcal{C}_3 \end{array}$$

$\mathcal{C}$ has the following typing, by inversions on C:Join:

(i) $\Psi \Vdash^{w_1} \mathcal{C}_1 :: \Psi_1$

(ii) $\Psi_1 \Vdash^{w_2} \mathsf{cell}(c, V, n) :: \Psi_2$

(iii) $\Psi_2 \Vdash^{w_3} \mathcal{C}_2 :: \Psi_3$

(iv) $\Psi_3 \Vdash^{w_4} \mathsf{thread}(d, q, [\eta, c/x, d/w], \mathsf{alias}\ x : Q_1\ \mathsf{as}\ y : Q_2, z : Q_3\ ;\ P), \mathsf{cell}(d, -, n') :: \Psi_4$

(v) $\Psi_4 \Vdash^{w_5} \mathcal{C}_3 :: \Psi_5$

It suffices to show that

(a) $\Psi \Vdash^{w_1} \mathcal{C}_1 :: \Psi_1$

(b) $\Psi_1 \Vdash^{w_2} \mathsf{cell}(c, V, n+1) :: \Psi_2'$

(c) $\Psi_2' \Vdash^{w_3} \mathcal{C}_2 :: \Psi_3'$

(d) $\Psi_3' \Vdash^{w_4} \mathsf{thread}(d, q, [\eta, c/y, c/z, d/w], P), \mathsf{cell}(d, -, n') :: \Psi_4$

(e) $\Psi_4 \Vdash^{w_5} \mathcal{C}_3 :: \Psi_5$

Obviously, (a) holds by (i). Then, (b) holds by C:Val and (ii), with $\Psi_2' = \Psi_2 + (c^1 : A)$. Then, by Lemma 8, (c) holds with $\Psi_3' = \Psi_3 + (c^1 : A)$.

Next, we apply inversion on (iv):

$$\frac{\Psi_{3a} \vdash \eta', c/x : \Gamma \qquad \Gamma \vdash^{w_4} \mathsf{alias}\ x : Q_1\ \mathsf{as}\ y : Q_2, z : Q_3\ ;\ P :: (w : B) \qquad \Psi_3 = \Psi_{3a} + \Psi_{3b}}{\Psi_3 \Vdash^{w_4} \mathsf{thread}(d, q, [\eta, c/x, d/w], \mathsf{alias}\ x : Q_1\ \mathsf{as}\ y : Q_2, z : Q_3\ ;\ P), \mathsf{cell}(d, -, n') :: \Psi_{3b}, (d^{n'} : B)}\ \text{C:Thread}$$

Notably, $q = w_4$, $\Psi_4 = \Psi_3''$, $(d^1 : B)$, and $\eta = \eta', d/w$.

By the second premise, we have $\Gamma = \Gamma', (x : Q_1)$, $Q_1 = (Q_2 + Q_3)$, and $\Gamma', (y : Q_2), (z : Q_3) \vdash^{w_4} P :: (w : B)$. Then $Q_1 = A$ and $\Psi_{3a} = \Psi_{3a}' + (c^1 : A)$.

Since $A = Q_1$ is linear, we have that $\Psi_3' = \Psi_{3a}' + (c^2 : A) + \Psi_{3b}$.

Thus, we have the following:

$$\frac{\begin{array}{c}\Psi_{3a}' + (c^2 : A) \vdash \eta', c/y_{Q_2}, c/z_{Q_3} : \Gamma', (y : Q_2), (z : Q_3) \\ \Gamma', (y : Q_2), (z : Q_3) \vdash^{w_4} P :: (w : B) \qquad \Psi_3 = \Psi_{3a} + \Psi_{3b}\end{array}}{\Psi_3' \Vdash^{w_4} \mathsf{thread}(d, q, [\eta', c/y, c/z, d/w], P), \mathsf{cell}(d, -, n') :: \Psi_{3b}, (d^{n'} : B)}\ \text{C:Thread}$$

Finally, by (v), we have (e), and thus by C:Join, the resulting configuration has the same type as the original.

Drop We consider the step

$$\begin{array}{l}\quad \mathcal{C}_1, \mathsf{cell}(c, V, n), \mathcal{C}_2, \mathsf{thread}(d, q, [\eta, c/x], \mathsf{drop}\ x : Q\ ;\ P), \mathsf{cell}(d, -, n'), \mathcal{C}_3 \\ \longmapsto \mathcal{C}_1, \mathsf{cell}(c, V, n-1), \mathcal{C}_2, \mathsf{thread}(d, q, [\eta], P), \mathsf{cell}(d, -, n'), \mathcal{C}_3\ (\text{if } n > 0)\ (\text{undefined if } n \leq 0)\end{array}$$

$\mathcal{C}$ has the following typing, by inversions on C:Join:

(i) $\Psi \Vdash^{w_1} \mathcal{C}_1 :: \Psi_1$

(ii) $\Psi_1 \Vdash^{w_2} \mathsf{cell}(c, V, n) :: \Psi_2$

(iii) $\Psi_2 \Vdash^{w_3} \mathcal{C}_2 :: \Psi_3$

(iv) $\Psi_3 \Vdash^{w_4} \mathsf{thread}(d, q, [\eta, c/x], \mathsf{drop}\ x : Q\ ;\ P), \mathsf{cell}(d, -, n') :: \Psi_4$

(v) $\Psi_4 \Vdash^{w_5} \mathcal{C}_3 :: \Psi_5$

It suffices to show that

(a) $\Psi \Vdash^{w_1} \mathcal{C}_1 :: \Psi_1$

(b) $\Psi_1 \Vdash^{w_2} \mathsf{cell}(c, V, n-1) :: \Psi_2'$

(c) $\Psi_2' \Vdash^{w_3} \mathcal{C}_2 :: \Psi_3'$

(d) $\Psi_3' \Vdash^{w_4} \mathsf{thread}(d, q, [\eta], P), \mathsf{cell}(d, -, n') :: \Psi_4$

(e) $\Psi_4 \Vdash^{w_5} \mathcal{C}_3 :: \Psi_5$

Obviously, (a) holds by (i). Then, by C:Val, $\Psi_2 = \Psi_2'', (c^n : A)$ for some Ψ_2'', so $\Psi_2' = \Psi_2'', (c^{n-1} : A)$. Then, by Lemma 9, if $\Psi_3 = \Psi_3'', (c^k : A)$, we have (c) with $\Psi_3' = \Psi_3'', (c^{k-1} : A)$ (noting that $k \geq 1$ because it is used in (iv)), where since A has no potential it cannot be split.

Next, by the typing of (iv), we have

$$\frac{\begin{array}{c}\Psi_{3a} + (c^1 : A) \vdash \eta', c/x : \Gamma, (x : A) \\ \Gamma, (x : A) \vdash^{w_4} \mathsf{drop}\ x : Q\ ;\ P :: (z : B) \qquad \Psi_3 = \Psi_{3a} + (c^1 : A) + \Psi_{3b}\end{array}}{\Psi_3 \Vdash^{w_4} \mathsf{thread}(d, q, [\eta', d/z, c/x], \mathsf{drop}\ x : Q\ ;\ P), \mathsf{cell}(d, -, n') :: \Psi_{3b}, (d^{n'} : B)}\ \textsc{C:Thread}$$

where we know that $(x : A)$ because A has no potential and $A = Q$.

Then, we get $\Gamma \vdash^{w_4} P :: (z : B)$ by inversion on the typing of the drop term. So we have $\Psi_{3a} \vdash \eta' : \Gamma$, $\Gamma \vdash^{w_4} P :: (z : B)$, and $\Psi_3' = \Psi_{3a} + \Psi_{3b}$, so we can type $\Psi_3' \Vdash^{w_4} \mathsf{thread}(d, q, [\eta], P), \mathsf{cell}(d, -, n') :: \Psi_4$.

Finally, (e) holds by (v), and we have the desired conclusion.

□