Composing a Well-Typed Region
(extended version)

Chris Hawblitzel, Heng Huang, and Lea Wittie

Dartmouth College Computer Science Technical Report TR2004-521
October 21, 2004

Abstract. Efficient low-level systems need more control over memory than safe high-level languages usually provide. In particular, safe languages usually prohibit explicit deallocation, in order to prevent dangling pointers. Regions provide one safe deallocation mechanism; indeed, many region calculi have appeared recently, each with its own set of operations and often complex rules. This paper encodes regions from lower-level typed primitives (linear memory, coercions, and delayed types), so that programmers can design their own region operations and rules.

1 Introduction

Efficient low-level systems need more control over memory than safe high-level languages usually provide. As a result, run-time systems are typically written in unsafe languages, such as C. The key challenge in safe low-level memory management is aliasing: if there are many pointers to an object, deallocating an object through one pointer leaves the other pointers dangling. One approach to solving this problem is to maintain control over aliasing. For example, linear types[13] simply ban aliasing, so that every object has only one pointer to it. Alias types[11] are more sophisticated, allowing a limited degree of aliasing, and tracking the aliasing to ensure that dangling pointers are never dereferenced. Another approach is to allow unlimited aliasing within a region of memory[12,14], but to limit the aliasing of the region itself. In this approach, programs deallocate entire regions at once, rather than deallocating individual objects within a region.

This paper encodes regions from controlled-memory primitives. From a theoretical perspective, the encoding provides a unified framework for controlled-aliasing and unlimited-aliasing approaches; from a practical perspective, the encoding lets programmers customize the design of regions for particular applications, such as typed garbage collection[5,8].

Section 2 of this paper encodes regions as linear tuples and pointers as functions. Each pointer consists of a *get* function that loads fields from the pointed-to heap object, and a *set* function that updates fields in the pointed-to heap object. Each function accepts a linear region as an argument and returns a new linear region as a result. The functions themselves are not linear, though — they are only conduits for linear data to pass through. Therefore, the program can freely copy the functions to form aliased, mutable data structures.

One problem with this encoding is the overhead of a run-time function call for each load and store. To eliminate this overhead, sections 3 and 4 of the paper replace run-time functions with *coercions* that manipulate *capabilities* for accessing memory, rather than actually performing loads and stores. A deeper problem with the encoding is allocation, since each new object in the region changes the type of the region. Section 5 proposes *delayed types*, which allow a region type to evolve without invalidating existing pointers. Section 6 argues that the low-level primitives (memory capabilities, coercions, and delayed types) can implement more than just simple regions: they can also implement forwarding pointers and aliased regions.

2 Regions as Tuples, Pointers as Functions

This section translates a source language containing explicit regions (roughly following Walker and Watkins[14]) into a target language that lacks built-in support for regions. The target language uses linear tuples to implement regions and nonlinear functions to implement region pointers. This implementation, while neither complete nor realistic by itself, demonstrates the intuition underlying the rest of the paper.

The target language supports two kinds of data: linear (lin) and nonlinear (non). Tuple types ($lin\langle\overrightarrow{\tau}\rangle$) are always linear, while function pointer types ($\tau_1 \to \tau_2$) and integer types (int) are always nonlinear. The expression "$lin\langle e_1, \ldots, e_n\rangle$" allocates a new linear tuple, and "let $\langle x_1, \ldots, x_n\rangle = e_1$ in e_2" deallocates a tuple e_1, binding variables $x_1 \ldots x_n$ to the contents of the tuple. We assume that integer and function pointer values fit in a single word, and therefore require no dynamic allocation (to make this practical, the type rules for functions require closed functions).

Linear data must be used exactly once, so that a tuple is never used after its deallocation. Following Walker and Watkins[14], the type rules enforce this single-use property with an environment splitting notation: writing $\Gamma = \Gamma_1, \Gamma_2$ indicates that Γ, Γ_1, and Γ_2 share the same nonlinear assumptions, but that each of Γ's linear assumptions appears in either Γ_1 or Γ_2, but not both. For convenience, we often write a combined context C, defined for the moment to be $C = \Delta; \Theta; \Gamma$. The notation $\overset{non}{C}$ denotes a context with no linear assumptions.

Target language syntax, typing rules, kinding rules (part 1)

linearity	$\phi = non \mid lin$
kinds	$\kappa = \phi$
types	$\tau = \text{int} \mid lin\langle\overrightarrow{\tau}\rangle \mid \tau_1 \to \tau_2 \mid \alpha$
expressions	$e = x \mid n \mid lin\langle\overrightarrow{e}\rangle \mid \text{let } \langle\overrightarrow{x}\rangle = e_1 \text{ in } e_2 \mid \lambda x{:}\tau.e \mid e_1\, e_2$
values	$v = n \mid lin\langle\overrightarrow{v}\rangle \mid \lambda x{:}\tau.e$
integers	$n = 0 \mid succ(n)$

$$
\begin{array}{rl}
\textit{type variables} & \alpha = \alpha, \beta, \gamma, \delta, \epsilon, \rho, \chi, \omega, \ldots \\
\textit{type variable env} & \Delta = \{\ldots, \alpha \mapsto \kappa, \ldots\} \\
\textit{recursive type env} & \Theta = \{\ldots, \alpha \mapsto \tau, \ldots\} \\
\textit{variable env} & \Gamma = \{\ldots, x \mapsto \tau, \ldots\} \\
 & \quad \textit{where } \overset{non(\Delta)}{\Gamma} = \{x \mapsto \tau \in \Gamma | \Delta \vdash \tau : non\} \\
\textit{combined env} & C = \Delta; \Theta; \Gamma \quad \textit{where } \overset{non}{C} = \Delta; \Theta; \overset{non(\Delta)}{\Gamma}
\end{array}
$$

Abbreviation: $(\text{let } x = e_1 \text{ in } e_2) = (\text{let } \langle x \rangle = lin\langle e_1 \rangle \text{ in } e_2)$
Abbreviation: $1 = succ(0)$, $2 = succ(1)$, ...

$$\Delta \vdash \text{int} : non \qquad \overset{non}{C} \vdash n : \text{int} \qquad \Delta, \alpha \mapsto \kappa \vdash \alpha : \kappa \qquad \overset{non}{C}, x \mapsto \tau \vdash x : \tau$$

$$\frac{\Delta \vdash \tau_1 : \phi_1 \qquad \ldots \qquad \Delta \vdash \tau_n : \phi_n}{\Delta \vdash lin\langle \tau_1, \ldots, \tau_n \rangle : lin} \qquad \frac{\Delta \vdash \tau_1 : \phi_1 \qquad \Delta \vdash \tau_2 : \phi_2}{\Delta \vdash \tau_1 \to \tau_2 : non}$$

$$\frac{\Delta; \Theta; \Gamma \vdash e : \tau \qquad \Theta \vdash \tau \equiv \tau'}{\Delta; \Theta; \Gamma \vdash e : \tau'} \qquad \frac{C_1 \vdash e_1 : \tau_1 \qquad \ldots \qquad C_k \vdash e_k : \tau_k}{C_1, \ldots, C_k \vdash lin\langle e_1, \ldots, e_k \rangle : lin\langle \tau_1, \ldots, \tau_k \rangle}$$

$$\frac{C_a \vdash e_a : lin\langle \tau_1, \ldots, \tau_k \rangle \qquad C_b, x_1 \mapsto \tau_1, \ldots, x_k \mapsto \tau_k \vdash e_b : \tau_b}{C_a, C_b \vdash \text{let } \langle x_1, \ldots, x_k \rangle = e_a \text{ in } e_b : \tau_b}$$

$$\frac{\Delta; \Theta; \{x \mapsto \tau_a\} \vdash e : \tau_b \qquad \Delta \vdash \tau_a : \phi}{\Delta; \Theta; \overset{non(\Delta)}{\Gamma} \vdash (\lambda x{:}\tau_a.e) : \tau_a \to \tau_b} \qquad \frac{C_f \vdash e_f : \tau_a \to \tau_b \qquad C_a \vdash e_a : \tau_a}{C_f, C_a \vdash e_f\, e_a : \tau_b}$$

Source language (extensions to target language), part 1

$$
\begin{array}{rl}
\textit{kinds} & \kappa = \ldots \mid \mathbf{rgn} \\
\textit{types} & \tau = \ldots \mid \text{Rgn}(\tau) \mid \langle \tau_1, \tau_2 \rangle @ \tau_{rgn} \\
\textit{expressions} & e = \ldots \mid \text{rgn}(\alpha) \mid \ell \\
 & \quad \mid \text{get}[e_{rgn}](e_{ptr}.n) \mid \text{set}[e_{rgn}](e_{ptr}.n \leftarrow e_{val}) \\
\textit{values} & v = \ldots \mid \text{rgn}(\alpha) \mid \ell \\
\textit{heaps} & H = \{\ldots, \ell \mapsto \langle v_1, v_2 \rangle @ \alpha, \ldots\} \\
\textit{heap type env} & \psi = \{\ldots, \ell \mapsto \langle \tau_1, \tau_2 \rangle @ \alpha, \ldots\} \\
\textit{live rgn env} & \Upsilon = \{\ldots, \alpha, \ldots\} \\
\textit{combined env} & C = \Delta; \Theta; \psi; \Upsilon; \Gamma \quad \textit{where } \overset{non}{C} = \Delta; \Theta; \psi; \{\}; \overset{non(\Delta)}{\Gamma}
\end{array}
$$

$$\frac{\Delta \vdash \tau : \mathbf{rgn}}{\Delta \vdash \text{Rgn}(\tau) : lin} \qquad \frac{\Delta \vdash \tau_1 : \phi_1 \qquad \Delta \vdash \tau_2 : \phi_2 \qquad \Delta \vdash \tau_{rgn} : \mathbf{rgn}}{\Delta \vdash \langle \tau_1, \tau_2 \rangle @ \tau_{rgn} : non}$$

$$\frac{\psi(\ell) = \langle \tau_1, \tau_2 \rangle @ \rho \qquad \rho \in \Upsilon \quad \Delta; \Theta; \psi; \{\}; \{\} \vdash v_1 : \tau_1 \quad \Delta; \Theta; \psi; \{\}; \{\} \vdash v_2 : \tau_2 \quad \Delta \vdash \tau_1 : non \qquad \Delta \vdash \tau_2 : non}{\Delta; \Theta; \psi; \Upsilon \vdash \ell \mapsto \langle v_1, v_2 \rangle @ \rho}$$

$$\frac{\forall \rho \in \Upsilon.(if\ \ell \mapsto \langle \tau_1, \tau_2 \rangle @ \rho \in \psi\ then\ \ell \mapsto \langle v_1, v_2 \rangle @ \rho \in H) \quad \forall \ell \in domain(H).(\Delta; \Theta; \psi; \Upsilon \vdash \ell \mapsto H(\ell))}{\Delta; \Theta; \psi; \Upsilon \vdash H}$$

$$\Delta, \alpha \mapsto \mathbf{rgn}; \Theta; \psi; \{\alpha\}; \overset{non(\Delta)}{\Gamma} \vdash \mathrm{rgn}(\alpha) : \mathrm{Rgn}(\alpha)$$

$$C, \ell \mapsto \tau \vdash \ell : \tau \qquad \frac{C_{rgn} \vdash e_{rgn} : \mathrm{Rgn}(\tau_{rgn}) \qquad C_{ptr} \vdash e_{ptr} : \langle \tau_1, \tau_2 \rangle @ \tau_{rgn}}{C_{rgn}, C_{ptr} \vdash \mathrm{get}[e_{rgn}](e_{ptr}.n) : lin\langle \mathrm{Rgn}(\tau_{rgn}), \tau_n \rangle}$$

$$\frac{C_{rgn} \vdash e_{rgn} : \mathrm{Rgn}(\tau_{rgn}) \qquad C_{ptr} \vdash e_{ptr} : \langle \tau_1, \tau_2 \rangle @ \tau_{rgn} \qquad C_{val} \vdash e_{val} : \tau_n}{C_{rgn}, C_{ptr}, C_{val} \vdash \mathrm{set}[e_{rgn}](e_{ptr}.n \leftarrow e_{val}) : \mathrm{Rgn}(\tau_{rgn})}$$

The source language extends the target language with regions of nonlinear, mutable pairs, along with pointers into the regions (labels ℓ, having type $\langle \tau_1, \tau_2 \rangle @ \tau_{rgn}$) and expressions to get and set the fields of the pairs. (We postpone expressions for pair allocation, region allocation, and region deallocation to section 5.)

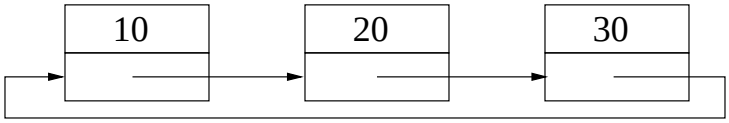
The source and target languages use an environment Θ to map recursive type names to recursive type definitions. For example, the environment $\Theta = \{\alpha_{List} \mapsto \langle \mathrm{int}, \alpha_{List} \rangle @ \rho\}$ defines a type name α_{List} for circular lists of integers residing in region ρ. Although the encodings in sections 2-4 work equally well with iso-recursive types (using explicit roll and unroll coercions), we use equi-recursive types here to set the stage for section 5's delayed types. If Θ contains the mapping $\alpha \mapsto \tau$, then α is considered equivalent to τ:

$$\Theta, \alpha \mapsto \tau \vdash \alpha \equiv \tau$$

The complete type equivalence rules (the rule shown above, plus structural rules, reflexivity, symmetry, and transitivity) are found in appendix C.

The source language defines a heap H that maps labels to pairs, where each pair is marked with its region. For example, here is a heap containing the circular list shown below, of type α_{List}:

$$H = \{\ell_1 \mapsto \langle 10, \ell_2 \rangle @ \rho, \ell_2 \mapsto \langle 20, \ell_3 \rangle @ \rho, \ell_3 \mapsto \langle 30, \ell_1 \rangle @ \rho\}$$



In general, a heap may contain pairs from any number of live regions. Let $\Upsilon = \{\rho_1, \ldots, \rho_m\}$ be the set of live region names, and let R_j contain the pairs allocated in region ρ_j, so that $H = R_1 \cup \ldots \cup R_m$:

$$R_j = \{\ell_{j,1} \mapsto \langle v_{j,1,1}, v_{j,1,2} \rangle @ \rho_j, \ell_{j,2} \mapsto \langle v_{j,2,1}, v_{j,2,2} \rangle @ \rho_j, \ldots\}$$

To type-check pointer expressions ℓ, define a heap type environment ψ mapping labels to pair types. Let φ_j contain the mappings for region R_j:

$$\varphi_j = \{\ell_{j,1} \mapsto \langle \tau_{j,1,1}, \tau_{j,1,2}\rangle @ \rho_j, \ell_{j,2} \mapsto \langle \tau_{j,2,1}, \tau_{j,2,2}\rangle @ \rho_j, \ldots\}$$

A program may legally hold pointers into deallocated regions, so the heap environment ψ may contain information about labels from dead regions as well as live regions; we'll write $\psi = \psi_{live} \cup \psi_{dead}$, where $\psi_{live} = \varphi_1 \cup \ldots \cup \varphi_m$. Even though a program can hold pointers into dead regions, it may not dereference these pointers. To prove that a region ρ is still live, a program must present a *capability* $\mathrm{rgn}(\rho)$ of type $\mathrm{Rgn}(\rho)$ whenever it dereferences a pointer into the region. This capability is linear; when a program deallocates a region, it relinquishes the capability, preventing it from dereferencing dangling pointers. To type-check the capabilities, Υ is treated linearly: $\Upsilon = \Upsilon_1, \Upsilon_2$ if each variable in Υ appears in either Υ_1 or Υ_2, but not both. Each get and set operation consumes the region capability and reproduces the capability, so that there is always exactly one capability for each region. For example, if x has type $\mathrm{Rgn}(\rho)$, then the expression $\mathrm{set}[x](\ell_1.1 \leftarrow 100)$ sets ℓ_1's first field to the value 100, and then returns x.

Throughout the paper, we use semantic brackets $[\![\ldots]\!]$ to indicate the source-to-target translation of a program's environments, types, and expressions. We start by translating regions into simple linear tuples:

$$\begin{aligned}[\![\mathrm{rgn}(\rho_j)]\!] = [\![R_j]\!] &= \mathit{lin}\langle [\![v_{j,1,1}]\!], [\![v_{j,1,2}]\!], [\![v_{j,2,1}]\!], [\![v_{j,2,2}]\!], [\![v_{j,3,1}]\!], [\![v_{j,3,2}]\!], \ldots\rangle \\ [\![\varphi_j]\!] &= \mathit{lin}\langle [\![\tau_{j,1,1}]\!], [\![\tau_{j,1,2}]\!], [\![\tau_{j,2,1}]\!], [\![\tau_{j,2,2}]\!], [\![\tau_{j,3,1}]\!], [\![\tau_{j,3,2}]\!], \ldots\rangle\end{aligned}$$

We define the region capability $[\![\mathrm{rgn}(\rho_j)]\!]$ to be the region itself, so that get and set operations can easily extract and update the values from the linear tuple. For this definition to be well formed, we require that the heap be well typed: $\Delta; \Theta; \psi; \Upsilon \vdash H$. The type rules for heap-allocated pairs allow only nonlinear values in the pairs, and nonlinear values cannot contain linear values, so we can prove that $[\![v_{j,1,1}]\!], [\![v_{j,1,2}]\!], \ldots$ do not contain the linear value $[\![\mathrm{rgn}(\rho_j)]\!]$ that we're trying to define. On the other hand, we cannot safely define $[\![\mathrm{Rgn}(\rho_j)]\!] = [\![\varphi_j]\!]$, since the types $\tau_{j,1,1}, \tau_{j,1,2}, \ldots$ could mention the type $\mathrm{Rgn}(\rho_j)$; consider, for example, $\tau_{j,1,1} = \mathrm{Rgn}(\rho_j) \to \mathrm{Rgn}(\rho_j)$. Therefore, we extend the recursive type environment $\Theta = \{\beta_1 \mapsto \tau_1, \beta_2 \mapsto \tau_2, \ldots\}$ with recursive types for $\rho_1 \ldots \rho_m$, and simply define $[\![\mathrm{Rgn}(\rho_j)]\!]$ to be ρ_j:

$$\begin{aligned}[\![\Theta]\!] &= \{\beta_1 \mapsto [\![\tau_1]\!], \beta_2 \mapsto [\![\tau_2]\!], \ldots\} \cup \{\rho_1 \mapsto [\![\varphi_1]\!], \rho_2 \mapsto [\![\varphi_2]\!], \ldots\} \\ [\![\mathrm{Rgn}(\tau)]\!] &= [\![\tau]\!]\end{aligned}$$

Except for pointer types, the translations of other data types are straightforward:

$$\begin{aligned}[\![\mathrm{int}]\!] &= \mathrm{int} \\ [\![\tau_1 \to \tau_2]\!] &= [\![\tau_1]\!] \to [\![\tau_2]\!] \\ [\![\mathit{lin}\langle \tau_1, \ldots, \tau_n\rangle]\!] &= \mathit{lin}\langle [\![\tau_1]\!], \ldots, [\![\tau_n]\!]\rangle \\ [\![\alpha]\!] &= \alpha \\ [\![\langle \tau_1, \tau_2\rangle @ \tau_r]\!] &= [\![\tau_r]\!] \to \mathit{lin}\langle [\![\tau_r]\!], [\![\tau_1]\!], [\![\tau_2]\!], \mathit{lin}\langle [\![\tau_r]\!], [\![\tau_1]\!], [\![\tau_2]\!]\rangle \to [\![\tau_r]\!]\rangle\end{aligned}$$

If data were read-only, a pointer type $[\![\langle \tau_1, \tau_2 \rangle @ \tau_r]\!]$ would consist of a *get* function that takes the heap, extracts the pointed-to data, and returns the data along with the heap: $[\![\tau_r]\!] \to lin\langle [\![\tau_r]\!], [\![\tau_1]\!], [\![\tau_2]\!] \rangle$. Since the data supports both reading and writing, the pointer must also contain a *set* function of type $lin\langle [\![\tau_r]\!], [\![\tau_1]\!], [\![\tau_2]\!] \rangle \to [\![\tau_r]\!]$. The read-write pointer type shown above combines the *get* and *set* functions by returning the *set* function from the *get* function:

$$
\begin{aligned}
&get_{j,k} = \lambda x{:}\rho_j.\text{let } \langle x_1, \ldots, x_{2n} \rangle = x \text{ in } lin\langle lin\langle x_1, \ldots, x_{2n} \rangle, x_{2k-1}, x_{2k}, set_{j,k} \rangle \\
&set_{j,k} = \lambda z{:}lin\langle \rho_j, [\![\tau_{j,k,1}]\!], [\![\tau_{j,k,2}]\!] \rangle.\text{let } \langle x, y_1, y_2 \rangle = z \text{ in} \\
&\qquad \text{let } \langle x_1, \ldots, x_{2n} \rangle = x \text{ in } lin\langle x_1, \ldots, x_{2k-2}, y_1, y_2, x_{2k+1}, \ldots, x_n \rangle \\
&[\![\ell_{j,k}]\!] = get_{j,k} \\
&[\![\text{get}[e_{rgn}](e_{ptr}.n)]\!] = \text{let } x_{rgn} = [\![e_{rgn}]\!] \text{ in let } x_{get} = [\![e_{ptr}]\!] \text{ in} \\
&\qquad \text{let } \langle x'_{rgn}, x_1, x_2, x_{set} \rangle = x_{get}\, x_{rgn} \text{ in } lin\langle x'_{rgn}, x_n \rangle \\
&[\![\text{set}[e_{rgn}](e_{ptr}.n \leftarrow e_{val})]\!] = \text{let } x_{rgn} = [\![e_{rgn}]\!] \text{ in let } x_{get} = [\![e_{ptr}]\!] \text{ in} \\
&\qquad \text{let } \langle x'_{rgn}, x_1, x_2, x_{set} \rangle = x_{get}\, x_{rgn} \text{ in let } x_n = [\![e_{val}]\!] \text{ in } x_{set}\, lin\langle x'_{rgn}, x_1, x_2 \rangle
\end{aligned}
$$

The translated get and set expressions simply call the *get* and *set* functions. Notice that the translated pointer type works for both live and dead regions. Suppose that $\ell_{j,k}$ is a "dangling pointer" into a dead region ρ_j. The translation $[\![\ell_{j,k}]\!]$ is a function that accepts ρ_j as an argument, and this function cannot be called if no value of type ρ_j exists; this is behavior we expect, since a dangling pointer cannot be dereferenced.

3 Linear Memory Capabilities

The previous section's encoding of pointer operations is not particularly efficient. The most outrageous inefficiency is the way the *get* and *set* functions treat a region — each read or write from a region deallocates and reallocates the entire region. Rather than using a linear tuple of region values, this section uses a linear tuple of *capabilities* that provide access to region values; the region values now reside in a global memory M, indexed by integer word addresses.

The only operations on M are load and store expressions. In particular, there are no built-in allocation and deallocation operations on M; allocation is built from load and store operations (for example, see appendix C for an encoding of a linear free list in the target language). The expression "store$[e_{mem}](e_{ptr} \leftarrow e_{val})$" places a value e_{val} into the memory at integer address e_{ptr}. The expression "load$[e_{mem}](e_{ptr})$" takes an integer address e_{ptr} and produces the value held in that address.

Values in memory may have different types at different times; in order to know the type τ_{val} of a value returned by a load operation, the program provides a capability e_{mem}, of type $\tau_{ptr} \mapsto \tau_{val}$, which is evidence that memory address τ_{ptr} currently holds a value of type τ_{val}. To ensure that loads do not use stale capabilities with out-of-date information, memory capabilities are linear, so that each memory operation consumes the current capability for a memory location

and produces a new capability for the memory location. This linearity is the key to safe deallocation; with the region encoding described in the next section, for example, a program can deallocate a region by simply extracting the capabilities from the region and storing different types of data in the memory previously occupied by the region.

Following alias types[11], we use singleton types to ensure that the correct capability accompanies an address: rather than giving e_{ptr} type int, we give it type $\mathrm{Int}(\tau_{ptr})$, a *singleton integer* type that contains only the integer τ_{ptr} mentioned in the capability type $\tau_{ptr} \mapsto \tau_{val}$. For example, the expression "5" has type Int(5); if x_{mem} has type $5 \mapsto \mathrm{float}$, then $\mathrm{load}[x_{mem}](5)$ is well-typed, but $\mathrm{load}[x_{mem}](6)$ is not. We also extend the type system with universal $(\forall\alpha:\kappa.\tau)$ and existential $(\exists\alpha:\kappa.\tau)$ polymorphism over any kind κ; for instance, the type $\exists\alpha{:}\mathbf{int}.lin\langle\mathrm{Int}(\alpha), \alpha \mapsto \mathrm{float}\rangle$ provides the singleton integer and capability needed to load a float from some address α. As a more detailed example, the following function swaps a value y_{val} of type α_y with a value in memory location x_{ptr} of type α_x, consuming a capability x_{mem} of type $\beta \mapsto \alpha_x$ and producing a capability x''_{mem} of type $\beta \mapsto \alpha_y$:

$$\begin{aligned}&\lambda z{:}lin\langle(\beta \mapsto \alpha_x), \mathrm{Int}(\beta), \alpha_y\rangle.\mathrm{let}\ \langle x_{mem}, x_{ptr}, y_{val}\rangle = z\ \mathrm{in}\\ &\quad \mathrm{let}\ \langle x'_{mem}, x_{val}\rangle = \mathrm{load}[x_{mem}](x_{ptr})\ \mathrm{in}\\ &\quad \mathrm{let}\ x''_{mem} = \mathrm{store}[x'_{mem}](x_{ptr} \leftarrow y_{val})\ \mathrm{in}\ lin\langle x''_{mem}, x_{val}\rangle\end{aligned}$$

As in section 2, the kind system includes kinds for linear and nonlinear data. The kinds $\overset{lin}{n}$ and $\overset{non}{n}$ not only describe the linearity of data, though, but also the size of data, measured in words. Values stored to memory M must have kind $\overset{non}{1}$. Capabilities of type $\tau_{ptr} \mapsto \tau_{val}$ are purely static entities used for type checking; they occupy no space at run-time and therefore have kind $\overset{lin}{0}$.

The environment Ψ maps memory addresses to the types of the values stored at the addresses. In contrast to the ψ of section 2, Ψ is linear, so that if $\Psi = \Psi_1, \Psi_2$, each assumption in Ψ appears in either Ψ_1 or Ψ_2, but not both. This linearity ensures that there is exactly one capability for each memory location at any given time in the program's execution. To represent a capability, the syntax for the abstract machine defines a special value, "fact", which has type $n \mapsto \tau_{val}$ in the environment $\Psi = \{n \mapsto \tau_{val}\}$.

Target language syntax, typing rules, kinding rules (part 2)

$$\begin{array}{rll}
\textit{kinds} & \kappa = \overset{\phi}{n} \mid \mathbf{int} & \\
\textit{types} & \tau = \ldots \mid \forall\alpha{:}\kappa.\tau \mid \exists\alpha{:}\kappa.\tau \mid non\langle\overrightarrow{\tau}\rangle & \\
 & \quad \mid \tau_1 \mapsto \tau_2 \mid 0 \mid succ(\tau) \mid \mathrm{Int}(\tau) & \\
\textit{expressions} & e = \ldots \mid non\langle\overrightarrow{e}\rangle \mid 0 \mid succ(e) \mid \mathrm{fact} \mid \lambda\alpha{:}\kappa.v \mid e\,\tau & \\
 & \quad \mid \mathrm{pack}[\tau_1, e]\,\mathrm{as}\,\exists\alpha{:}\kappa.\tau_2 \mid \mathrm{unpack}\,\alpha, x = e_1\,\mathrm{in}\,e_2 & \\
 & \quad \mid \mathrm{load}[e_{mem}](e_{ptr}) \mid \mathrm{store}[e_{mem}](e_{ptr} \leftarrow e_{data}) & \\
\textit{values} & v = \ldots \mid \lambda\alpha{:}\kappa.v \mid \mathrm{pack}[\tau_1, v]\,\mathrm{as}\,\exists\alpha{:}\kappa.\tau_2 &
\end{array}$$

$$\mid \; non\langle \overrightarrow{v} \rangle \mid 0 \mid succ(v) \mid \text{fact}$$

$$\textit{memory} \qquad M = \{\ldots, n \mapsto v, \ldots\}$$

$$\textit{memory type env} \qquad \Psi = \{\ldots, n \mapsto \tau, \ldots\}$$

$$\textit{combined env} \qquad C = \Delta; \Psi; \Theta; \Gamma \qquad \textit{where } \overset{non}{C} = \Delta; \{\}; \Theta; \overset{non(\Delta)}{\Gamma}$$

Abbreviation: $\text{int} = \exists \alpha : \mathbf{int}.\text{Int}(\alpha)$
Abbreviation: $\tau + 0 = \tau$, $\tau + 1 = succ(\tau)$, $\tau + 2 = succ(succ(\tau)), \ldots$
Abbreviation: $e + 0 = e$, $e + 1 = succ(e)$, $e + 2 = succ(succ(e)), \ldots$

$$\frac{\Delta \vdash \tau_1 : \overset{\phi_1}{n_1} \qquad \Delta \vdash \tau_2 : \overset{\phi_2}{n_2}}{\Delta \vdash \tau_1 \to \tau_2 : \overset{non}{1}} \qquad \frac{\Delta, \alpha \mapsto \kappa \vdash \tau : \overset{\phi}{n}}{\Delta \vdash \forall \alpha : \kappa . \tau : \overset{\phi}{n}} \qquad \frac{\Delta, \alpha \mapsto \kappa \vdash \tau : \overset{\phi}{n}}{\Delta \vdash \exists \alpha : \kappa . \tau : \overset{\phi}{n}}$$

$$\frac{\Delta \vdash \tau_1 : \overset{\phi_1}{n_1}, \ldots, \Delta \vdash \tau_k : \overset{\phi_k}{n_k} \qquad n = n_1 + \ldots + n_k}{\Delta \vdash \phi\langle \tau_1, \ldots, \tau_k \rangle : \overset{\phi}{n}} \qquad \frac{\Delta \vdash \tau : \mathbf{int}}{\Delta \vdash \text{Int}(\tau) : \overset{non}{1}}$$

$$\frac{\Delta \vdash \tau_1 : \mathbf{int} \qquad \Delta \vdash \tau_2 : \overset{non}{1}}{\Delta \vdash \tau_1 \mapsto \tau_2 : \overset{lin}{0}} \qquad \Delta \vdash 0 : \mathbf{int} \qquad \frac{\Delta \vdash \tau : \mathbf{int}}{\Delta \vdash succ(\tau) : \mathbf{int}}$$

$$\frac{C \vdash e : \text{Int}(\tau)}{C \vdash succ(e) : \text{Int}(succ(\tau))} \qquad \frac{C \vdash e_1 : \forall \alpha : \kappa . \tau_1 \qquad C \vdash \tau_2 : \kappa}{C \vdash e_1\, \tau_2 : [\alpha \leftarrow \tau_2]\tau_1}$$

$$\frac{C, \alpha \mapsto \kappa \vdash v : \tau}{C \vdash \lambda \alpha : \kappa . v : \forall \alpha : \kappa . \tau} \qquad \frac{C \vdash e : [\alpha \leftarrow \tau_1]\tau_2 \qquad C \vdash \tau_1 : \kappa \qquad C \vdash \exists \alpha : \kappa . \tau_2 : \overset{\phi}{n}}{C \vdash (\text{pack}[\tau_1, e] \text{ as } \exists \alpha : \kappa . \tau_2) : (\exists \alpha : \kappa . \tau_2)}$$

$$\frac{C_1 \vdash e_1 : (\exists \alpha : \kappa . \tau_1) \qquad C_2, \alpha \mapsto \kappa, x \mapsto \tau_1 \vdash e_2 : \tau_2 \qquad C_2 \vdash \tau_2 : \overset{\phi}{n}}{C_1, C_2 \vdash (\text{unpack}\, \alpha, x = e_1 \,\text{in}\, e_2) : \tau_2}$$

$$\frac{\begin{array}{ccc} C_1 \vdash e_1 : \tau_1 & \ldots & C_k \vdash e_k : \tau_k \\ C_1 \vdash \tau_1 : \overset{non}{n_1} & \ldots & C_k \vdash \tau_k : \overset{non}{n_k} \end{array}}{(C_1, \ldots, C_k) \vdash non\langle e_1, \ldots, e_k \rangle : non\langle \tau_1, \ldots, \tau_k \rangle}$$

$$\frac{C_a \vdash e_a : \phi\langle \tau_1, \ldots, \tau_k \rangle \qquad C_b, x_1 \mapsto \tau_1, \ldots, x_k \mapsto \tau_k \vdash e_b : \tau_b}{C_a, C_b \vdash \text{let } \langle x_1, \ldots, x_k \rangle = e_a \,\text{in}\, e_b : \tau_b}$$

$$\frac{C_{mem} \vdash e_{mem} : \tau_{addr} \mapsto \tau_{data} \qquad C_{ptr} \vdash e_{ptr} : \text{Int}(\tau_{addr})}{C_{mem}, C_{ptr} \vdash \text{load}[e_{mem}](e_{ptr}) : lin\langle (\tau_{addr} \mapsto \tau_{data}), \tau_{data} \rangle}$$

$$\frac{\begin{array}{c} C_{mem} \vdash e_{mem} : \tau_{addr} \mapsto \tau_{data} \\ C_{ptr} \vdash e_{ptr} : \text{Int}(\tau_{addr}) \\ C_{data} \vdash e_{data} : \tau'_{data} \qquad C_{data} \vdash \tau'_{data} : \overset{non}{1} \end{array}}{C_{mem}, C_{ptr}, C_{data} \vdash \text{store}[e_{mem}](e_{ptr} \leftarrow e_{data}) : \tau_{addr} \mapsto \tau'_{data}}$$

$$\overset{non}{C} \vdash 0 : \text{Int}(0) \qquad \overset{non}{C}, n \mapsto \tau \vdash \text{fact} : n \mapsto \tau$$

4 Coercions

Given section 3's support for memory capabilities, we can revisit section 2's translation of a region type φ_j. Rather than translating into a tuple-of-memory-values type $[\![\varphi_j]\!] = lin\langle [\![\tau_{j,1,1}]\!], [\![\tau_{j,1,2}]\!], \ldots\rangle$, as in section 2, we translate it into a tuple-of-memory-capabilities type. For each pair $\langle v_{j,k,1}, v_{j,k,2}\rangle$ in H, choose unique integer addresses $n_{j,k,1}$ and $n_{j,k,2}$ to hold the values $v_{j,k,1}$ and $v_{j,k,2}$, so that $n_{j,k,2} = n_{j,k,1} + 1$. The region type maps the pair addresses to the pair field types:

$$[\![\varphi_j]\!] = lin\langle n_{j,1,1} \mapsto [\![\tau_{j,1,1}]\!], n_{j,1,2} \mapsto [\![\tau_{j,1,2}]\!], n_{j,2,1} \mapsto [\![\tau_{j,2,1}]\!], n_{j,2,2} \mapsto [\![\tau_{j,2,2}]\!], \ldots\rangle$$

Pointer types $\langle \tau_1, \tau_2\rangle @ \tau_r$ are still translated into function types, but the functions no longer perform actual reads and writes to memory. Instead, each function merely retrieves capabilities $\gamma \mapsto [\![\tau_1]\!], \gamma + 1 \mapsto [\![\tau_2]\!]$, where γ is the address contained in the pointer, from the heap type τ_r. Since the capabilities are linear, a function cannot return both a capability from τ_r and all of τ_r, as this would require copying a linear value. Instead, the program calls one function to split τ_r into pieces (the capabilities $\gamma \mapsto [\![\tau_1]\!], \gamma + 1 \mapsto [\![\tau_2]\!]$ and β, which contains the rest of τ_r), and calls a second function to join the pieces back together to form τ_r:

$$\begin{aligned}[\![\langle \tau_1, \tau_2\rangle @ \tau_r]\!] = \exists \beta{:}\overset{lin}{0}\ .\exists \gamma{:}\mathbf{int}.non\langle \mathrm{Int}(\gamma), \tau_{split}, \tau_{join}\rangle \\ where\ \tau_{split} = [\![\tau_r]\!] \multimap lin\langle \beta, \gamma \mapsto [\![\tau_1]\!], \gamma + 1 \mapsto [\![\tau_2]\!]\rangle \\ where\ \tau_{join} = lin\langle \beta, \gamma \mapsto [\![\tau_1]\!], \gamma + 1 \mapsto [\![\tau_2]\!]\rangle \multimap [\![\tau_r]\!]\end{aligned}$$

Notice that the functions τ_{split} and τ_{join} exist solely to manipulate compile-time capabilities — their return values are of kind $\overset{lin}{0}$ and are thus erased before run-time. In fact, there's no reason to actually call the functions at run-time. To formalize this optimization, this section introduces *coercion expressions* "coerce(e)", which exist only to perform operations on capabilities, and are erased before run-time. In order to ensure that erasing coerce(e) causes no changes to the program's run-time behavior, e must satisfy three restrictions: it must have kind $\overset{\phi}{0}$, it must neither load nor store to memory, and, to ensure termination, it must not call any functions. The typing rules enforce these restrictions by extending the context C with a new environment Λ, defined to be $\Rightarrow$ for coercion expressions and $\multimap$ for ordinary expressions. For example, the rule for function calls becomes:

$$\frac{C_f, \multimap \vdash e_f : \tau_a \multimap \tau_b \qquad C_a, \multimap \vdash e_a : \tau_a}{(C_f, C_a), \multimap \vdash e_f\, e_a : \tau_b}$$

If $C = \Delta; \Psi; \Theta; \Gamma; \Lambda$, then we define $C, \Lambda' = \Delta; \Psi; \Theta; \Gamma; \Lambda'$, so that the expression $e_f\, e_a$ shown above type-checks only in a non-coercion environment. The rules for load and store are also modified to require a non-coercion environment: load's context must be $(C_{mem}, C_{ptr}), \multimap$, and store's context must be $(C_{mem}, C_{ptr}, C_{data}), \multimap$.

Since coercions do not execute at run-time in a real implementation, they can be thought of as a small logic language rather than a programming language. For example, a tuple of capabilities $lin\langle\tau_1, \tau_2\rangle$ corresponds to the logical conjunction of τ_1 and τ_2. The syntax and rules shown below extend the language with implication $\tau_1 \Rightarrow \tau_2$ and disjunction $\tau_1 \vee \tau_2$. For example, the expression

$\lambda x{:}lin\langle\tau_1, \tau_2\rangle \Rightarrow (\text{let } \langle x_1, x_2\rangle = x \text{ in } lin\langle x_2, x_1\rangle)$

has type $lin\langle\tau_1, \tau_2\rangle \Rightarrow lin\langle\tau_2, \tau_1\rangle$. We use function composition $e_1 \circ e_2$ rather than function application to build implications from other implications. Although banning function applications inside coercion expressions is crude, it is the simplest way to ensure coercion termination while still allowing coercions to manipulate recursive types. A simple inductive argument suffices to prove that if $(C, \Rightarrow) \vdash e : \tau$, then the evaluation of e halts in a finite number of steps.

Target language syntax, typing rules, kinding rules (part 3)

types	$\tau = \ldots \mid \tau_1 \Rightarrow \tau_2 \mid \tau_1 \vee \tau_2$
expressions	$e = \ldots \mid \text{coerce}(e) \mid \lambda x{:}\tau \Rightarrow e \mid e_1 \circ e_2$
	$\mid \text{disj}^n_{\tau_1 \vee \tau_2}(e) \mid \text{case } e_0 \text{ of } x_1.e_1 \text{ or } x_2.e_2$
values	$v = \ldots \mid \lambda x{:}\tau \Rightarrow e \mid v_1 \circ v_2 \mid \text{disj}^n_{\tau_1 \vee \tau_2}(v)$
coercion env	$\Lambda = \rightarrow \mid \Rightarrow$
combined env	$C = \Delta; \Psi; \Theta; \Gamma; \Lambda \quad$ *where* $\overset{non}{C} = \Delta; \{\}; \Theta; \overset{non(\Delta)}{\Gamma}; \Lambda$

Abbreviation: $\tau_1 \Leftrightarrow \tau_2 = non\langle\tau_1 \Rightarrow \tau_2, \tau_2 \Rightarrow \tau_1\rangle$

$$\frac{\Delta \vdash \tau_1 :\overset{\phi_1}{n_1} \qquad \Delta \vdash \tau_2 :\overset{\phi_2}{n_2}}{\Delta \vdash \tau_1 \Rightarrow \tau_2 :\overset{non}{0}} \qquad \frac{\Delta \vdash \tau_1 :\overset{\phi}{0} \qquad \Delta \vdash \tau_2 :\overset{\phi}{0}}{\Delta \vdash \tau_1 \vee \tau_2 :\overset{\phi}{0}}$$

$$\frac{C_f, \rightarrow\vdash e_f : \tau_a \Rightarrow \tau_b \qquad C_a, \rightarrow\vdash e_a : \tau_a}{(C_f, C_a), \rightarrow\vdash e_f\, e_a : \tau_b}$$

$$\frac{C_1 \vdash e_1 : \tau_b \Rightarrow \tau_c \qquad C_2 \vdash e_2 : \tau_a \Rightarrow \tau_b}{C_1, C_2 \vdash e_1 \circ e_2 : \tau_a \Rightarrow \tau_c} \qquad \frac{C, \Rightarrow\vdash e : \tau \qquad C \vdash \tau :\overset{\phi}{0}}{C, \rightarrow\vdash \text{coerce}(e) : \tau}$$

$$\frac{\overset{non}{C}, x \mapsto \tau_a, \Rightarrow\vdash e : \tau_b \qquad \overset{non}{C} \vdash \tau_b :\overset{\phi}{0}}{\overset{non}{C} \vdash (\lambda x{:}\tau_a \Rightarrow e) : \tau_a \Rightarrow \tau_b} \qquad \frac{C \vdash e : \tau_n \qquad C \vdash \tau_1 \vee \tau_2 :\overset{\phi}{0}}{C \vdash \text{disj}^n_{\tau_1 \vee \tau_2}(e) : \tau_1 \vee \tau_2}$$

$$\frac{C_a, \Rightarrow\vdash e_0 : \tau_1 \vee \tau_2 \qquad C_b, x_1 \mapsto \tau_1, \Rightarrow\vdash e_1 : \tau_b \qquad C_b, x_2 \mapsto \tau_2, \Rightarrow\vdash e_2 : \tau_b}{(C_a, C_b), \Rightarrow (\text{case } e_0 \text{ of } x_1.e_1 \text{ or } x_2.e_2) : \tau_b}$$

Using logical implications in place of run-time functions, the translated type for pointers becomes:

$$[\![\langle\tau_1, \tau_2\rangle @ \tau_r]\!] = \exists\beta{:}\overset{lin}{0}.\exists\gamma{:}\textbf{int}.non\langle \text{Int}(\gamma), [\![\tau_r]\!] \Leftrightarrow lin\langle\beta, \gamma \mapsto [\![\tau_1]\!], \gamma + 1 \mapsto [\![\tau_2]\!]\rangle\rangle$$

where $\tau_1 \Leftrightarrow \tau_2$ is an abbreviation for $non\langle \tau_1 \Rightarrow \tau_2, \tau_2 \Rightarrow \tau_1 \rangle$. Pointer get and set operations retrieve the memory capabilities from τ_r, perform a load or store, and then reconstitute τ_r. For example, a get operation becomes:

$$\begin{aligned}
&[\![\text{get}[e_{rgn}](e_{ptr}.n)]\!] = \text{let } x_{rgn} = [\![e_{rgn}]\!] \text{ in} \\
&\quad \text{unpack } \beta, x_{ptr} = [\![e_{ptr}]\!] \text{ in unpack } \gamma, x'_{ptr} = x_{ptr} \text{ in} \\
&\quad \text{let } \langle x_{addr}, x_f \rangle = x'_{ptr} \text{ in let } \langle x_{split}, x_{join} \rangle = x_f \text{ in} \\
&\quad \text{let } \langle x_\beta, x_1, x_2 \rangle = x_{split}\, x_{rgn} \text{ in} \\
&\quad \text{let } \langle x_n, y \rangle = \text{load}[x_n](x_{addr} + (n-1)) \text{ in } lin\langle x_{join}\, lin\langle x_\beta, x_1, x_2 \rangle, y \rangle
\end{aligned}$$

The only run-time operation in this code is the $\text{load}[x_n](x_{addr} + (n-1))$ expression; all the rest is compile-time unpacking and coercion.

5 Allocation

Consider a region with space for three pairs:

$$\varphi = \{\ell_1 \mapsto \langle \tau_{1,1}, \tau_{1,2} \rangle @ \rho, \ell_2 \mapsto \langle \tau_{2,1}, \tau_{2,2} \rangle @ \rho, \ell_3 \mapsto \langle \tau_{3,1}, \tau_{3,2} \rangle @ \rho\}$$

Section 2 encoded the region type as a tuple: $[\![\varphi]\!] = lin\langle [\![\tau_{1,1}]\!], \ldots, [\![\tau_{3,2}]\!] \rangle$. The only way to create such a tuple is to supply all the values of type $[\![\tau_{1,1}]\!], \ldots, [\![\tau_{3,2}]\!]$ at once. For large, long-lived regions, this approach is quite unrealistic — programs must be able to allocate region data incrementally. Very often, the types for ℓ_2 and ℓ_3 will not be known until after the allocation of ℓ_1. Unfortunately, the encodings of region ρ in sections 2-4 define a recursive type mapping $\rho \mapsto [\![\varphi]\!]$ once and for all, and there's no way to change this mapping to reflect new information about φ, nor is there any way to replace ρ with a new region name ρ' without invalidating all the pointers that already exist for ρ.

To accommodate the incremental determination of the types in regions, we introduce *delayed types*, which add new bindings $\alpha \mapsto \tau$ to the recursive type environment at run-time. The program first creates a name α for the binding, and later *commits* the name to a particular type τ. Upon committing α to τ, the program may substitute τ for α in any type it wants. To ensure that a program only commits α to a single type, the program obtains a linear capability of type $\text{Delay}(\alpha)$ when creating the name α, and relinquishes the capability when committing α. A linear environment Φ contains the currently uncommitted names.

Target language syntax, typing rules, kinding rules (part 4)

$$\begin{aligned}
\textit{types } \tau &= \ldots \mid \text{Delay}(\tau) \\
\textit{expressions } e &= \ldots \mid \text{delay}(\kappa) \mid \text{commit}[e_{delay}](e_{data} : (\alpha = \tau_\alpha \text{ in } \tau_{data})) \\
\textit{uncommitted env } \Phi &= \{\ldots, \alpha, \ldots\} \\
\textit{combined env } C &= \Delta; \Psi; \Phi; \Theta; \Gamma; \Lambda \qquad \textit{where } \overset{non}{C} = \Delta; \{\}; \{\}; \Theta; \overset{non(\Delta)}{\Gamma}; \Lambda
\end{aligned}$$

$$\frac{\Delta \vdash \tau : \kappa}{\Delta \vdash \text{Delay}(\kappa) : \overset{lin}{0}} \qquad \overset{non}{C} \vdash \text{delay}(\kappa) : \exists \alpha{:}\kappa.\text{Delay}(\alpha)$$

$$\frac{C_{delay} \vdash \tau_{delay} : \kappa \qquad C_{delay} \vdash \tau_\alpha : \kappa \qquad C_{delay} \vdash [\alpha \leftarrow \tau_\alpha]\tau_{data} :\overset{\phi}{n} \qquad C_{delay} \vdash e_{delay} : \text{Delay}(\tau_{delay}) \qquad C_{data} \vdash e_{data} : [\alpha \leftarrow \tau_{delay}]\tau_{data}}{C_{delay}, C_{data} \vdash \text{commit}[e_{delay}](e_{data} : (\alpha = \tau_\alpha \,\text{in}\, \tau_{data})) : [\alpha \leftarrow \tau_\alpha]\tau_{data}}$$

$$\Delta; \{\}; \{\alpha\}; \Theta; \overset{non(\Delta)}{\Gamma}; \Lambda \vdash \text{fact} : \text{Delay}(\alpha) \qquad \textit{where}\ \Delta(\alpha) = \kappa$$

Source language (extensions to target language), part 2

$$\begin{array}{ll} \textit{expressions} & e = \ldots \mid \text{alloc}[e_R]\,\langle e_1, e_2\rangle \mid \text{newrgn} \mid \text{freergn}(e_R) \\ \textit{combined env} & C = \Delta; \Psi; \Phi; \Theta; \psi; \Upsilon; \Gamma; \Lambda \end{array}$$

$$\frac{C_{rgn}, \rightarrow\!\vdash e_{rgn} : \text{Rgn}(\tau_{rgn}) \qquad C_1 \vdash \tau_1 : \overset{non}{1} \qquad C_1, \rightarrow\!\vdash e_1 : \tau_1 \qquad C_2 \vdash \tau_2 : \overset{non}{1} \qquad C_2, \rightarrow\!\vdash e_2 : \tau_2}{(C_{rgn}, C_1, C_2), \rightarrow\!\vdash \text{alloc}[e_{rgn}]\,\langle e_1, e_2\rangle : lin\langle \text{Rgn}(\tau_{rgn}), \langle \tau_1, \tau_2\rangle @ \tau_{rgn}\rangle}$$

$$\overset{non}{C}, \rightarrow\!\vdash \text{newrgn} : \exists \alpha\!:\!\mathbf{rgn}.\text{Rgn}(\alpha) \qquad \frac{C, \rightarrow\!\vdash e_{rgn} : \text{Rgn}(\tau_{rgn})}{C, \rightarrow\!\vdash \text{freergn}(e_{rgn}) : lin\langle\rangle}$$

For the sake of simple type-checking algorithms, we expect Θ and Φ to be empty at compile-time, so that our real compiler[7] need not implement them. (Note that Θ and Φ are still important to the theory, particularly for the proof of the language's safety.) Even with empty compile-time environments, a program can still use delayed types to describe and manipulate recursive types at compile-time. For example, here is an encoding of iso-recursive coercions $x_{roll(\alpha)}$ and $x_{unroll(\alpha)}$, of type $\tau \Rightarrow \alpha$ and $\alpha \Rightarrow \tau$ for any τ of kind $\overset{lin}{0}$:

$$\begin{array}{l} \text{unpack}\,\alpha, z = \text{delay}(\overset{lin}{0})\,\text{in let}\, x = non\langle \lambda y\!:\!\alpha \Rightarrow y, \lambda y\!:\!\alpha \Rightarrow y\rangle\,\text{in} \\ \text{let}\,\langle x_{roll(\alpha)}, x_{unroll(\alpha)}\rangle = \text{commit}[z](x : (\alpha' = \tau\,\text{in}\, non\langle \alpha' \Rightarrow \alpha, \alpha \Rightarrow \alpha'\rangle))\,\text{in}\ \ldots \end{array}$$

Delayed types provide a way to implement incremental allocation inside regions. For example, rather than defining the type $[\![\varphi]\!] = lin\langle [\![\tau_{1,1}]\!], \ldots, [\![\tau_{3,2}]\!]\rangle$ all at once, a program makes six names $\alpha_{1,1} \ldots \alpha_{3,2}$ and defines $[\![\varphi]\!] = lin\langle \alpha_{1,1}, \ldots, \alpha_{3,2}\rangle$. Then, as the program allocates pairs, it commits $\alpha_{1,1}$ to $\tau_{1,1}$, $\alpha_{1,2}$ to $\tau_{1,2}$ and so on. This strategy is still insufficient, though: until α is committed to some τ, there are no values of type α. Therefore, there's no way to create a tuple of type $lin\langle \alpha_{1,1}, \ldots, \alpha_{3,2}\rangle$ until $\alpha_{1,1} \ldots \alpha_{3,2}$ are all committed. One possible solution uses tagged union types $\tau_1 \cup \tau_2$ to define $[\![\varphi]\!] = lin\langle \text{Int}(0) \cup \alpha_{1,1}, \ldots, \text{Int}(0) \cup \alpha_{3,2}\rangle$, using "0" as a pre-commitment placeholder value for each field (much like Java's use of default values of "0", "false", and "null" to initialize static fields and array elements), but this introduces unnecessary run-time overhead. We can do better.

Rather than committing $\alpha_{k,n}$ to $\tau_{k,n}$, let's use three variables $\alpha_1, \alpha_2, \alpha_3$ and commit each α_k to $lin\langle n_{k,1} \mapsto \tau_{k,1}, n_{k,2} \mapsto \tau_{k,2}\rangle$, following the encoding from section 4. A naive definition of $[\![\varphi]\!]$ would be $lin\langle\alpha_1, \alpha_2, \alpha_3\rangle$, but this has the same problem described above: no values of types α_k can exist before α_k is committed, making it impossible to instantiate $[\![\varphi]\!]$. Taking inspiration from the aforementioned tagged unions, we replace α_k with a disjunction, which is much like a union, but without run-time tagging. More specifically, we write:

$$[\![\varphi]\!] = lin\langle \mathrm{Delay}(\delta_1) \vee \alpha_1, \mathrm{Delay}(\delta_2) \vee \alpha_2, \mathrm{Delay}(\delta_3) \vee \alpha_3\rangle$$

where $\delta_1, \delta_2, \delta_3$ are extra delayed names to assist the use of $\alpha_1, \alpha_2, \alpha_3$. Each time an α_k is committed to some $lin\langle n_{k,1} \mapsto \tau_{k,1}, n_{k,2} \mapsto \tau_{k,2}\rangle$, we'll commit δ_k to the type $non\langle\rangle$. Before δ_k's commitment, there is one capability with type $\mathrm{Delay}(\delta_k)$, which we use to form a value of type $\mathrm{Delay}(\delta_k) \vee \alpha_k$. After δ_k's commitment, there are no values of type of type $\mathrm{Delay}(\delta_k)$, since committing δ_k consumes the capability. Therefore, any post-commitment value of type $\mathrm{Delay}(\delta_k) \vee \alpha_k$ must contain an α_k value, not a $\mathrm{Delay}(\delta_k)$ value. In particular, if, starting with a variable x_{Delay} of type $\mathrm{Delay}(\delta_k)$, we commit δ_k to form a value x_δ of type δ_k:

$$\begin{array}{l}\quad \mathrm{let}\, x_f = \mathrm{commit}[x_{Delay}]((\lambda z\!:\!\delta_k \Rightarrow z) : (\delta'_k = non\langle\rangle\, \mathrm{in}\, \delta'_k \Rightarrow \delta_k))\, \mathrm{in} \\ \quad \mathrm{let}\, x_\delta = x_f\, non\langle\rangle\, \mathrm{in}\, \ldots\end{array}$$

then the following expression has type $(\mathrm{Delay}(\delta_k) \vee \alpha_k) \Rightarrow \alpha_k$:

$$\lambda x_{disjunct}\!:\!(\mathrm{Delay}(\delta_k) \vee \alpha_k) \Rightarrow (\mathrm{case}\, x_{disjunct}\, \mathrm{of}\, y_{Delay}.e_{Delay}\, \mathrm{or}\, y_\alpha.y_\alpha)$$

where $e_{Delay} = (\mathrm{let}\, \langle z\rangle = \mathrm{commit}[y_{Delay}](x_\delta : (\delta'_k = non\langle\alpha_k\rangle\, \mathrm{in}\, \delta'_k))\, \mathrm{in}\, z)$. Notice that both e_{Delay} and y_α have type α_k. The latter typing is trivial, while the former relies on a sneaky (but sound) proof by contradiction, exploiting the impossibility of simultaneously holding values of type $\mathrm{Delay}(\delta_k)$ and type δ_k: committing δ_k allows e_{Delay} to cast x_δ to any type of kind $\overset{non}{0}$, including the uninhabited linear-value-inside-a-nonlinear-tuple type $non\langle\alpha_k\rangle$. The same technique proves $lin\langle \mathrm{Delay}(\delta_k) \vee \alpha_k, \mathrm{Delay}(\alpha_k)\rangle \Rightarrow lin\langle \mathrm{Delay}(\delta_k), \mathrm{Delay}(\alpha_k)\rangle$, which is used to generate the variable x_{Delay} in the expression above.

The translated pointer type $[\![\langle\tau_1, \tau_2\rangle @ \tau_r]\!]$ remains the same as in section 4; the only difference is that to implement the $[\![\tau_r]\!] \Rightarrow lin\langle\beta, \gamma \mapsto [\![\tau_1]\!], \gamma+1 \mapsto [\![\tau_2]\!]\rangle$ implication, a pointer $[\![\ell_k]\!]$ uses the $(\mathrm{Delay}(\delta_k) \vee \alpha_k) \Rightarrow \alpha_k$ coercion to extract the $\alpha_k = lin\langle n_{k,1} \mapsto \tau_{k,1}, n_{k,2} \mapsto \tau_{k,2}\rangle$ value from the region $[\![\tau_r]\!]$, where $[\![\tau_r]\!] \equiv [\![\varphi]\!] = lin\langle \mathrm{Delay}(\delta_1) \vee \alpha_1, \ldots\rangle$.

The region described above contains only enough space for three pairs. Ideally, a region should be able to grow without bound (or, on a real computer, to grow until memory is exhausted). Therefore, we revise the region type $[\![\varphi]\!]$ to be:

$$\begin{array}{rl}[\![\varphi]\!] & = \mathrm{Delay}(\delta_1) \vee \chi_1 \\ & \equiv \mathrm{Delay}(\delta_1) \vee lin\langle\alpha_1, \mathrm{Delay}(\delta_2) \vee \chi_2\rangle \\ & \equiv \mathrm{Delay}(\delta_1) \vee lin\langle\alpha_1, \mathrm{Delay}(\delta_2) \vee lin\langle\alpha_2, \mathrm{Delay}(\delta_3) \vee \chi_3\rangle\rangle \\ & \ldots\end{array}$$

The type $[\![\varphi]\!]$ starts small and grows as $\chi_1, \chi_2, \chi_3, \chi_4, \ldots$ are committed. Each χ_k is committed to $lin\langle\alpha_k, \mathrm{Delay}(\delta_{k+1}) \vee \chi_{k+1}\rangle$, defining just a little more of

$[\![\varphi]\!]$, and using a newly allocated name χ_{k+1} to delay further definition. At any given time, the region contains only two uncommitted names, δ_k and χ_k, while $\delta_1 \ldots \delta_{k-1}$ and $\chi_1 \ldots \chi_{k-1}$ are already committed. The α_k variables no longer need to be delayed, and instead are defined immediately when χ_k is defined.

As $[\![\varphi]\!]$ grows, the program needs some way to reach the α_k and $\text{Delay}(\delta_k) \vee \chi_k$ values inside. The following coercion reaches k levels into $[\![\varphi]\!]$, shuffling everything in levels $1 \ldots k-1$ into a temporary structure of type ϵ_k:

$$[\![\varphi]\!] \Leftrightarrow lin\langle \epsilon_k, \text{Delay}(\beta_k) \vee \chi_k \rangle$$

where ϵ_k is defined as follows:

$$\begin{aligned} \epsilon_1 &= lin\langle\rangle \\ \epsilon_2 &= lin\langle lin\langle\rangle, \alpha_1\rangle \\ \epsilon_3 &= lin\langle lin\langle lin\langle\rangle, \alpha_1\rangle, \alpha_2\rangle \\ &\ldots \end{aligned}$$

When the region grows from size k to size $k+1$, the program uses the $[\![\varphi]\!] \Leftrightarrow lin\langle \epsilon_k, \text{Delay}(\beta_k) \vee \chi_k \rangle$ coercion to generate a new coercion $[\![\varphi]\!] \Leftrightarrow lin\langle \epsilon_{k+1}, \text{Delay}(\delta_{k+1}) \vee \chi_{k+1} \rangle$. First, as described above, the delayed definition $\delta_k = non\langle\rangle$ proves that $(\text{Delay}(\delta_k) \vee \chi_k) \Rightarrow \chi_k$. Next, χ_k is committed to $lin\langle \alpha_k, \text{Delay}(\delta_{k+1}) \vee \chi_{k+1} \rangle$. The rest follows from simple tuple reassociation:

$$\begin{aligned} [\![\varphi]\!] &\Leftrightarrow lin\langle \epsilon_k, \text{Delay}(\delta_k) \vee \chi_k \rangle \\ &\Leftrightarrow lin\langle \epsilon_k, \chi_k \rangle \\ &\Leftrightarrow lin\langle \epsilon_k, lin\langle \alpha_k, \text{Delay}(\delta_{k+1}) \vee \chi_{k+1} \rangle \rangle \\ &\Leftrightarrow lin\langle lin\langle \epsilon_k, \alpha_k \rangle, \text{Delay}(\delta_{k+1}) \vee \chi_{k+1} \rangle \\ &\Leftrightarrow lin\langle \epsilon_{k+1}, \text{Delay}(\delta_{k+1}) \vee \chi_{k+1} \rangle \end{aligned}$$

The composition of these steps yields $[\![\varphi]\!] \Leftrightarrow lin\langle \epsilon_{k+1}, \text{Delay}(\delta_{k+1}) \vee \chi_{k+1} \rangle$. The complete state of the region consists of $[\![\tau_r]\!] = [\![\varphi]\!]$ along with $\text{Delay}(\chi)$ and $[\![\varphi]\!] \Leftrightarrow lin\langle \epsilon, \text{Delay}(\delta) \vee \chi \rangle$ for the current χ, δ, and ϵ:

$$[\![\text{Rgn}(\tau_r)]\!] = \exists \chi{:}\overset{lin}{0}\ .\exists \delta{:}\overset{non}{0}\ .\exists \epsilon{:}\overset{lin}{0}\ .lin\langle [\![\tau_r]\!], \text{Delay}(\chi), [\![\tau_r]\!] \Leftrightarrow lin\langle \epsilon, \text{Delay}(\delta) \vee \chi \rangle \rangle$$

The program can use the coercion $[\![\varphi]\!] \Leftrightarrow lin\langle \epsilon_{k+1}, \text{Delay}(\delta_{k+1}) \vee \chi_{k+1} \rangle$ to retrieve the α_k from $\epsilon_{k+1} = lin\langle \epsilon_k, \alpha_k \rangle$. For example, if $\alpha_k = lin\langle n_k \mapsto \tau_{k,1}, n_k + 1 \mapsto \tau_{k,2} \rangle$, then it's easy to construct a coercion of type

$$[\![\varphi]\!] \Leftrightarrow lin\langle lin\langle \epsilon_k, \text{Delay}(\delta_{k+1}) \vee \chi_{k+1} \rangle, n_k \mapsto \tau_{k,1}, n_k + 1 \mapsto \tau_{k,2} \rangle$$

Packaging this coercion with a singleton integer for n_k forms the translated pointer type:

$$[\![\langle \tau_1, \tau_2 \rangle @ \tau_r]\!] = \exists \beta{:}\overset{lin}{0}\ .\exists \gamma{:}\mathbf{int}.non\langle \text{Int}(\gamma), [\![\tau_r]\!] \Leftrightarrow lin\langle \beta, \gamma \mapsto [\![\tau_1]\!], \gamma + 1 \mapsto [\![\tau_2]\!] \rangle \rangle$$

5.1 Free space management

Free space will consist of a linear linked list of free blocks, each having four words of memory holding arbitrary values:

$$\begin{aligned} \mathit{FreeBlock} = \exists\gamma{:}\mathbf{int}.\exists\beta_0{:}\overset{non}{1}.\exists\beta_1{:}\overset{non}{1}.\exists\beta_2{:}\overset{non}{1}.\exists\beta_3{:}\overset{non}{1}. \\ \mathit{lin}\langle \mathrm{Int}(\gamma), \gamma \mapsto \beta_0, \gamma+1 \mapsto \beta_1, \gamma+2 \mapsto \beta_2, \gamma+3 \mapsto \beta_3 \rangle \end{aligned}$$

An abstract recursive data type, *Allocator*, returns a free block on demand, along with another allocator. The *Allocator* data type is simply a closure-converted function, where the closure contains a nonlinear environment β and a linear environment γ:

$$\mathit{Allocator} = \exists\beta{:}\overset{non}{1}.\exists\gamma{:}\overset{lin}{0}.\mathit{lin}\langle\beta,\gamma,\mathit{lin}\langle\beta,\gamma\rangle \to \mathit{lin}\langle \mathit{Allocator}, \mathit{FreeBlock}\rangle\rangle$$

Unfortunately, memory is not infinite, and the allocator will run out of memory if the program tries to allocate more memory than is available. The best reaction to this would be to throw an exception, or at least halt the program gracefully, but the target language contains neither exception handling nor a "halt" expression. Rather than extend the target language, we simply encode a fake "halt" expression, implemented in appendix C as an infinite loop (and we promise that out-of-memory is the only place that the translation uses the "halt" expression).

Appendix C defines $\mathit{nilFree}$ and $\mathit{consFree}$ functions to construct an *Allocator*, implemented as a linked list, from linear memory capabilities. The evaluation of an $\mathrm{alloc}[e_R]\,\langle e_1, e_2\rangle$ expression transfers a free block from the current allocator to region e_R, and the evaluation of a $\mathrm{freergn}(e_R)$ expression returns all the blocks in e_R to the current allocator.

The $\mathrm{freergn}(e_R)$ expression needs to find all the blocks in region e_R. Therefore, we extend the region implementation with a closure $\omega, \mathit{lin}\langle\tau_{\mathit{free}},\epsilon,\omega\rangle \to \tau_{\mathit{free}}$, which transfers all the linear memory capabilities in ϵ back to the allocator. This closure uses its environment ω to hold a pointer to the first block in the region, which in turn holds a closure that points to the second block, and so on (these closures require two extra words of space in each block):

$$\begin{aligned} [\![\mathrm{Rgn}(\tau_r)]\!] = \exists\chi{:}\overset{lin}{0}.\exists\delta{:}\overset{non}{0}.\exists\epsilon{:}\overset{lin}{0}.\exists\omega{:}\overset{non}{1}. \\ \mathit{lin}\langle [\![\tau_r]\!], \mathrm{Delay}(\chi), [\![\tau_r]\!] \Leftrightarrow \mathit{lin}\langle \epsilon, \mathrm{Delay}(\delta)\vee\chi\rangle, \omega, \mathit{lin}\langle\tau_{\mathit{free}},\epsilon,\omega\rangle \to \tau_{\mathit{free}}\rangle \end{aligned}$$

The type τ_{free} contains the allocator, along with roll and unroll functions for the allocator. The translated program passes a variable x_{free} of type τ_{free} to and from each expression. Appendix C contains the complete source-to-target translation of expressions, types, and kinds.

6 Extensions

Sections 2-5 built a source-to-target translation for a particular source language. The target language is flexible enough to support other translations and other source language features, though. For example, the encoding of $[\![\mathrm{Rgn}(\tau)]\!]$ occupies two words, and the type τ_{free} occupies four words. It's possible to reduce the sizes of $[\![\mathrm{Rgn}(\tau)]\!]$ and τ_{free} to one word each, by storing their data in memory and using a singleton integer to point to the data, as in the following definition:

$$[\![\mathrm{Rgn}(\tau_r)]\!] = \exists\gamma{:}\mathbf{int}.lin\langle \mathrm{Int}(\gamma), RCap([\![\tau_r]\!],\gamma)\rangle$$

$$\begin{aligned} RCap([\![\tau_r]\!],\gamma) = \exists\chi{:}\overset{lin}{0}.\exists\delta{:}\overset{non}{0}.\exists\epsilon{:}\overset{lin}{0}.\exists\omega{:}\overset{non}{1}.lin\langle [\![\tau_r]\!], \mathrm{Delay}(\chi),\\ [\![\tau_r]\!] \Leftrightarrow lin\langle\epsilon, \mathrm{Delay}(\delta)\vee\chi\rangle, \gamma\mapsto\omega, \gamma+1\mapsto(lin\langle\tau_{free},\epsilon,\omega\rangle\to\tau_{free})\rangle \end{aligned}$$

This definition splits the region into a nonlinear, one-word handle of type $\mathrm{Int}(\gamma)$, and a linear capability $RCap([\![\tau_r]\!],\gamma)$ of kind $\overset{lin}{0}$. The alloc and freergn expressions require both the handle and the capability, but get and set expressions require only the capability.

The remainder of this section discusses two other features supported by the target language (or slightly extended versions of the target language): region aliasing and forwarding pointers. A separate paper[7] describes more dramatic extensions to the target language, integrating integer arithmetic and constraints[16] into the type system, in order to support stacks, arrays, and efficient memory layouts.

6.1 Region aliasing

The regions developed in sections 2-5 are linear, which limits their expressiveness. Crary *et al*[3] give an example of a function that expects two regions ρ_1 and ρ_2:

$$\lambda\rho_1{:}\mathbf{rgn}.\lambda\rho_2{:}\mathbf{rgn}.\lambda x{:}lin\langle\mathrm{Rgn}(\rho_1),\mathrm{Rgn}(\rho_2),\langle\mathrm{int},\mathrm{int}\rangle@\rho_1,\langle\mathrm{int},\mathrm{int}\rangle@\rho_2\rangle.\ldots.$$

A program cannot call this function with both ρ_1 and ρ_2 instantiated to the same region ρ, because this would require duplicating the capability linear $\mathrm{Rgn}(\rho)$. To overcome this limitation, Crary *et al* introduce a "calculus of capabilities" that distinguishes between linear assumptions $\{\tau^1\}$ and possibly aliased assumptions $\{\tau^+\}$, with an operator $\oplus$ to join assumptions together. The function above can use this notation to indicate that ρ_1 and ρ_2 are not necessarily distinct:

$$\lambda\rho_1{:}\mathbf{rgn}.\lambda\rho_2{:}\mathbf{rgn}.\lambda x{:}lin\langle\{\mathrm{Rgn}(\rho_1)^+\}\oplus\{\mathrm{Rgn}(\rho_2)^+\},\langle\mathrm{int},\mathrm{int}\rangle@\rho_1,\langle\mathrm{int},\mathrm{int}\rangle@\rho_2\rangle.\ldots.$$

Sections 4 described how coercions allow aliasing of linear memory capabilities inside a region; the same technique allows aliasing of any resource of kind $\overset{lin}{0}$, including the region capablities $RCap(\rho,\gamma)$ discussed above. To generalize the technique, we define an abbreviation $Alias[\tau_{pool}](\tau)$ that indicates a linear resource τ contained in a pool of linear resources τ_{pool} (notice that even though τ and τ_{pool} are linear, $Alias[\tau_{pool}](\tau)$ is nonlinear):

$$Alias[\tau_{pool}](\tau) = \exists\beta{:}\overset{lin}{0}.\tau_{pool} \Leftrightarrow lin\langle\tau, \beta\rangle$$

Using this idea, we describe an encoding of a subset of the capability calculus in our target language. Our encoding is limited, though, by its reliance on explicit coercions — in the capability calculus, for example, $\{\tau^+\} \oplus \{\tau^+\} \equiv \{\tau^+\}$, while our encoding can only establish a coercion, $\{\tau^+\} \oplus \{\tau^+\} \Leftrightarrow \{\tau^+\}$, not a type equivalence $\equiv$. Therefore, we believe our encoding to be less expressive than the full capability calculus. On the other hand, our target language supports many idioms, such as existential quantification of linear resources, that do not appear in the capability calculus, so it's difficult to compare the overall expressiveness of the two systems.

Here is a the definition of *capabilities* c in the capability calculus, where we assume that resources τ and variables ϵ have kind $\overset{lin}{0}$:

$$c = \{\} \mid \{\tau^1\} \mid \{\tau^+\} \mid \epsilon \mid c_1 \oplus c_2 \mid \overline{c}$$

Unfortunately, our encoding omits the stripping operator $\overline{c}$, although we can imitate some of the functionality of stripped variables $\overline{\epsilon}$ by using variables of kind $\overset{non}{0}$. The encoding of the remaining capabilities is as follows:

$$[\![c]\!] = \exists\alpha_{pool}{:}\overset{lin}{0}.lin\langle\alpha_{pool}, [\![c, \alpha_{pool}]\!]\rangle \; where \; \alpha \notin FV(c)$$

$$\begin{aligned}
[\![\{\}, \tau_{pool}]\!] &= lin\langle\rangle \\
[\![\{\tau^1\}, \tau_{pool}]\!] &= \tau \\
[\![\{\tau^+\}, \tau_{pool}]\!] &= Alias[\tau_{pool}](\tau) \\
[\![\epsilon, \tau_{pool}]\!] &= \epsilon \\
[\![c_1 \oplus c_2, \tau_{pool}]\!] &= lin\langle[\![c_1, \tau_{pool}]\!], [\![c_2, \tau_{pool}]\!]\rangle
\end{aligned}$$

Since $[\![c_1 \oplus c_2, \tau_{pool}]\!]$ is just a tuple, it is straightforward to construct coercions to reorder or reassociate capabilities:

$$[\![c_1 \oplus c_2, \tau_{pool}]\!] \Leftrightarrow [\![c_2 \oplus c_1, \tau_{pool}]\!]$$

$$[\![c_1 \oplus (c_2 \oplus c_3), \tau_{pool}]\!] \Leftrightarrow [\![(c_1 \oplus c_2) \oplus c_3, \tau_{pool}]\!]$$

Since $lin\langle[\![c, \tau_{pool}]\!], \tau\rangle = [\![c \oplus \{\tau^1\}, \tau_{pool}]\!]$, it is easy to introduce and remove linear resources in capabilities. Since $[\![\{\tau^+\}, \tau_{pool}]\!]$ is nonlinear, it is easy to duplicate and discard $\{\tau^+\}$ resources:

$$[\![c \oplus \{\tau^+\}, \tau_{pool}]\!] \Leftrightarrow [\![c \oplus (\{\tau^+\} \oplus \{\tau^+\}), \tau_{pool}]\!]$$

Push and pop The capability encoding described so far is much like the region encoding from sections 2-4: a program can create a capability all at once, but cannot build capabilities incrementally. This section describes a mechanism for pushing resources into τ_{pool} and popping them back out, in order to support a stack of regions. The solution is very different from section 5's use of delayed types to grow regions. First, the regions from section 5 grew but never shrank — there was no pop operation. Second, section 5 allowed the program to scatter a region's pointers throughout data structures and hide pointers in abstract data types. The push and pop operations presented here rely on the fact that all a capability's resources are kept in the capability itself, so that the operations can find each aliased resource $\{\tau_i^+\}$ and update the resource's type to reflect a new pool type. To allow this update, *Alias* must be polymorphic over all extensions γ to the current pool τ_{pool}:

$$Alias[\tau_{pool}](\tau) = \exists\beta{:}0^{lin}.\forall\gamma{:}0^{lin}.lin\langle\tau_{pool},\gamma\rangle \Leftrightarrow lin\langle\tau,\beta,\gamma\rangle$$

Furthermore, we require one extension to the target language, to make it possible to compose new polymorphic implications from old polymorphic implications:

Extensions to source and target languages for useful coercion polymorphism

$$\begin{array}{rl} \textit{expressions} & e = \ldots \mid \lambda\alpha{:}\kappa \Rightarrow e \\ \textit{values} & v = \ldots \mid \lambda\alpha{:}\kappa \Rightarrow e \end{array}$$

$$\frac{{}^{non}C, \alpha \mapsto \kappa, \Rightarrow \vdash e : \tau}{{}^{non}C \vdash (\lambda\alpha{:}\kappa \Rightarrow e) : \forall\alpha{:}\kappa.\tau} \qquad \frac{C, \Rightarrow \vdash e : \tau \qquad C \vdash \tau :^{\phi} 0}{C \vdash \text{coerce}(e) : \tau}$$

Suppose x has type $c \oplus \{\tau^1\}$, and a function f has type $c \oplus \{\tau^+\} \to c \oplus \{\tau^+\}$. Before passing x as an argument to f, the program must weaken x's type to $c \oplus \{\tau^+\}$. Furthermore, the program should be able to strengthen the return value from type $c \oplus \{\tau^+\}$ to type $c \oplus \{\tau^1\}$ when f returns. The capability calculus uses subtyping and bounded quantification to weaken and strengthen x's type, but our target language lacks these mechanisms. Instead, the program must explicitly coerce $[\![c \oplus \{\tau^1\}, \tau_{pool}]\!]$ to type $[\![c \oplus \{\tau^+\}, \tau'_{pool}]\!]$ by *pushing* the resource τ into the shared resource pool τ'_{pool}. After f returns, the program explicitly *pops* τ back out of the pool. The push coercion has type τ_{push}:

$$\tau_{push} = lin\langle\tau_{pool}, [\![c \oplus \{\tau^1\}, \tau_{pool}]\!]\rangle \Rightarrow lin\langle lin\langle\tau_{pool},\tau\rangle, [\![c \oplus \{\tau^+\}, lin\langle\tau_{pool},\tau\rangle]\!], \tau_{pop}\rangle$$

where

$$\tau_{pop} = lin\langle lin\langle\tau_{pool},\tau\rangle, [\![c \oplus \{\tau^+\}, lin\langle\tau_{pool},\tau\rangle]\!]\rangle \Rightarrow lin\langle\tau_{pool}, [\![c \oplus \{\tau^1\}, \tau_{pool}]\!]\rangle$$

To implement the coercion, we first reorder c to put the aliased resources last:

$$c = \epsilon_j \oplus \ldots \oplus \epsilon_1 \oplus \{\tau_n^1\} \oplus \ldots \oplus \{\tau_{k+1}^1\} \oplus \{\tau_k^+\} \oplus \ldots \oplus \{\tau_1^+\}$$

The push operation's implementation is complicated by the existing aliased resources in c; the type of each $\{\tau_i^+\}$ must change from $Alias[\tau_{pool}](\tau_i)$ to $Alias[lin\langle\tau_{pool},\tau\rangle](\tau_i)$. The coercion $\lambda x_i : Alias[\tau_{pool}](\tau_i) \Rightarrow e_i$ shown below has type $Alias[\tau_{pool}](\tau_i) \Rightarrow Alias[lin\langle\tau_{pool},\tau\rangle](\tau_i)$.

$$
\begin{array}{l}
\lambda x_i : Alias[\tau_{pool}](\tau_i) \Rightarrow e_i \\
\quad where\ e_i = \text{unpack}\,\beta, x_{all} = x_i \,\text{in} \\
\qquad\qquad \text{pack}[lin\langle\beta,\tau\rangle, \lambda\gamma' :^{lin}_{0} \Rightarrow e_i'] \,\text{as}\, Alias[lin\langle\tau_{pool},\tau\rangle](\tau_i) \\
\quad where\ e_i' = \text{let}\ \langle x_{to}, x_{from}\rangle = x_{all}\, lin\langle\gamma',\tau\rangle \,\text{in} \\
\qquad\qquad non\langle e_{i-post-to} \circ x_{to} \circ e_{i-pre-to}, e_{i-post-from} \circ x_{from} \circ e_{i-pre-from}\rangle \\
\quad where\ e_{i-pre-to} = \lambda x : lin\langle lin\langle\tau_{pool},\tau\rangle,\gamma'\rangle \Rightarrow \\
\qquad\qquad \text{let}\ \langle x'_{pool}, x'_{\gamma}\rangle = x \,\text{in let}\ \langle x_{pool}, x_{\tau}\rangle = x'_{pool} \,\text{in} \\
\qquad\qquad lin\langle x_{pool}, lin\langle x'_{\gamma}, x_{\tau}\rangle\rangle \\
\quad where\ e_{i-post-from} = \lambda x : lin\langle\tau_{pool}, lin\langle\gamma',\tau\rangle\rangle \Rightarrow \\
\qquad\qquad \text{let}\ \langle x_{pool}, x_{\gamma}\rangle = x \,\text{in let}\ \langle x'_{\gamma}, x_{\tau}\rangle = x_{\gamma} \,\text{in} \\
\qquad\qquad lin\langle lin\langle x_{pool}, x_{\tau}\rangle, x'_{\gamma}\rangle \\
\quad where\ e_{i-post-to} = \lambda x : lin\langle\tau_i, \beta, lin\langle\gamma',\tau\rangle\rangle \Rightarrow \\
\qquad\qquad \text{let}\ \langle x_i, x_{\beta}, x_{\gamma}\rangle = x \,\text{in let}\ \langle x'_{\gamma}, x_{\tau}\rangle = x_{\gamma} \,\text{in} \\
\qquad\qquad lin\langle x_i, lin\langle x_{\beta}, x_{\tau}\rangle, x'_{\gamma}\rangle \\
\quad where\ e_{i-pre-from} = \lambda x : lin\langle\tau_i, lin\langle\beta,\tau\rangle,\gamma'\rangle \Rightarrow \\
\qquad\qquad \text{let}\ \langle x_i, x'_{\beta}, x'_{\gamma}\rangle = x \,\text{in let}\ \langle x_{\beta}, x_{\tau}\rangle = x'_{\beta} \,\text{in} \\
\qquad\qquad lin\langle x_i, x_{\beta}, lin\langle x'_{\gamma}, x_{\tau}\rangle\rangle
\end{array}
$$

The following coercion, of type τ_{push}, uses the e_i expressions to build a new coercion $c \oplus \{\tau^+\}$ from the old coercion $c \oplus \{\tau^1\}$:

$$
\begin{array}{l}
e_{push} = \lambda x : lin\langle\tau_{pool}, [\![c \oplus \{\tau^1\}, \tau_{pool}]\!]\rangle \Rightarrow \\
\quad \text{let}\ \langle x_{pool}, x'\rangle = x \,\text{in} \\
\quad \text{let}\ \langle x_{c,0}, x_{\tau}\rangle = x' \,\text{in} \\
\quad \text{let}\ \langle x_{c,1}, x_1\rangle = x_{c,0} \,\text{in}\ \ldots \text{let}\ \langle x_{c,k}, x_k\rangle = x_{c,k-1} \,\text{in} \\
\quad \text{let}\ x'_{c,k} = x_{c,k} \,\text{in} \\
\quad \text{let}\ x'_{c,k-1} = lin\langle x'_{c,k}, e_k\rangle \,\text{in}\ \ldots \text{let}\ x'_{c,0} = lin\langle x'_{c,1}, e_1\rangle \,\text{in} \\
\quad \text{let}\ x'_{\tau} = \text{pack}[\tau_{pool}, \lambda\gamma :^{lin}_{0}\ .non\langle v_{to}, v_{from}\rangle] \,\text{as}\, Alias[lin\langle\tau_{pool},\tau\rangle](\tau) \,\text{in} \\
\quad lin\langle lin\langle x_{pool}, x_{\tau}\rangle, lin\langle x'_{c,0}, x'_{\tau}\rangle, e_{pop}\rangle \\
where\ v_{to} = \lambda x : lin\langle lin\langle\tau_{pool},\tau\rangle,\gamma\rangle \Rightarrow \\
\qquad\qquad \text{let}\ \langle x', x_{\gamma}\rangle = x \,\text{in let}\ \langle x_{pool}, x_{\tau}\rangle = x' \,\text{in}\ lin\langle x_{\tau}, x_{pool}, x_{\gamma}\rangle \\
where\ v_{from} = \lambda x : lin\langle\tau, \tau_{pool}, \gamma\rangle \Rightarrow \\
\qquad\qquad \text{let}\ \langle x_{\tau}, x_{pool}, x_{\gamma}\rangle = x \,\text{in}\ lin\langle lin\langle x_{pool}, x_{\tau}\rangle, x_{\gamma}\rangle \\
where\ e_{pop} = \lambda y : lin\langle lin\langle\tau_{pool},\tau\rangle, [\![c \oplus \{\tau^+\}, lin\langle\tau_{pool},\tau\rangle]\!]\rangle \Rightarrow \\
\quad \text{let}\ \langle y', y_{c,0}\rangle = y \,\text{in} \\
\quad \text{let}\ \langle y_{pool}, y_{\tau}\rangle = y' \,\text{in} \\
\quad \text{let}\ \langle y_{c,1}, y_1\rangle = y_{c,0} \,\text{in}\ \ldots \text{let}\ \langle y_{c,k}, y_k\rangle = y_{c,k-1} \,\text{in} \\
\quad \text{let}\ y'_{c,k} = y_{c,k} \,\text{in} \\
\quad \text{let}\ y'_{c,k-1} = lin\langle y'_{c,k}, x_k\rangle \,\text{in}\ \ldots \text{let}\ y'_{c,0} = lin\langle y'_{c,1}, x_1\rangle \,\text{in} \\
\quad lin\langle y_{pool}, lin\langle y'_{c,0}, y_{\tau}\rangle\rangle
\end{array}
$$

The type τ_{pop} requires a value of type $lin\langle\tau_{pool}, \tau\rangle$ to recover the original capability c, which raises the issue of τ_{pool}'s scope. Should τ_{pool} be hidden by existential quantification, as suggested by the definition $[\![c]\!] = \exists\alpha{:}\overset{lin}{0}\ .lin\langle\alpha, [\![c, \alpha]\!]\rangle$, or should the program's functions use universal quantification? For example, the function f of type $c \oplus \{\tau^+\} \to c \oplus \{\tau^+\}$ could be implemented either way:

$$\exists\alpha{:}\overset{lin}{0}\ .lin\langle\alpha, [\![c \oplus \{\tau^+\}, \alpha]\!]\rangle \to \exists\alpha{:}\overset{lin}{0}\ .lin\langle\alpha, [\![c \oplus \{\tau^+\}, \alpha]\!]\rangle$$

$$\forall\alpha{:}\overset{lin}{0}\ .lin\langle\alpha, [\![c \oplus \{\tau^+\}, \alpha]\!]\rangle \to lin\langle\alpha, [\![c \oplus \{\tau^+\}, \alpha]\!]\rangle$$

Since τ_{pop} relies on a specific τ_{pool}, only the universally quantified f allows the caller to pop τ from the pool after f returns. Therefore, we expect typical functions to have the form $\forall\alpha{:}\overset{lin}{0}\ .lin\langle\alpha, [\![c, \alpha]\!], \ldots\rangle \to lin\langle\alpha, [\![c', \alpha]\!], \ldots\rangle$, where α is the same in the argument type's capability and the return type's capability. This allows c' to add and remove linear resources, and to reorder, duplicate, and discard nonlinear resources, but it does not allow c' to push aliased resources onto c. For example, $c = \epsilon \oplus \{\tau^1\}$ and $c' = \epsilon \oplus \{\tau^+\}$ would require a different α for c and c'. It's not clear whether this is a significant limitation; in this example, the type c' is not very useful anyway, because it is too weak to allow the caller to pop τ from c', and too strong to allow the called function to pop τ, which leaves τ stranded in c' forever. Moreover, it's still possible to support functions of the form $\forall\alpha{:}\overset{lin}{0}\ .lin\langle\alpha, [\![c, \alpha]\!], \ldots\rangle \to lin\langle\tau', [\![c', \tau']\!], \ldots\rangle$ where $\alpha \neq \tau'$, but the caller and callee must pass the appropriate push/pop coercions along with the capabilities to ensure that every pushed capability gets popped, even if the push happens in a different function than the pop.

6.2 Forwarding pointers

Regions provide a safe deallocation mechanism, but exploiting this deallocation requires sophisticated program analysis or programmer assistance. Otherwise, garbage may accumulate in long-lived regions, causing a program to run out of memory. Wang and Appel[15] observed that a program can copy live data from one region ρ_1 into a second region ρ_2 and then deallocate ρ_1, effectively constructing a copying garbage collector written entirely with type-safe language features. Such a *typed* (or *type-preserving*) collector offers the generality of traditional garbage collection without requiring the collector to be trusted. There's one troublesome detail: in order to avoid copying the same object multiple times, the collector establishes forwarding pointers from objects in ρ_1 to objects in ρ_2. In subsequent collections, ρ_2 holds pointers to ρ_3, and ρ_3 holds pointers to ρ_4, and so on. It's difficult to write the first region's object types without knowing all the region names $\rho_1, \rho_2, \rho_3, \ldots$ in advance (not easy, since it's hard to say how many collections will occur before starting the program). Several solutions have been proposed [15][9][8], but each new proposal changes the typing rules for regions, often in *ad hoc* and complex ways.

We argue that with only standard, general-purpose extensions, the target language can implement a "next-region" operator in the style of Fluet and Wang [5]. Consider a simple recursive linked list type for region ρ:

$$List(\rho) = \langle \text{int}, List(\rho) \rangle @ \rho$$

Imagine a type operator $NextRgn$ with the property that $NextRgn(\rho_k) = \rho_{k+1}$. Then the following list type accommodates forwarding pointers from ρ_k to ρ_{k+1}:

$$List(\rho) = \langle (\text{int} \cup List(NextRgn(\rho))), List(\rho) \rangle @ \rho$$

For simplicity, we use a tagged union type $\tau_1 \cup \tau_2$ to allow the first word of the object to hold an int before the collector copies the object, and a forwarding pointer $List(NextRgn(\rho))$ after the copy, although this causes some unnecessary run-time overhead to deal with the tagging.

To support type operators like $NextRgn$ at all, the type system requires extensions for abstraction ($\lambda\alpha : \kappa_1.\tau$, of kind $\kappa_1 \rightarrow \kappa_2$) and type application ($\tau_1\ \tau_2$). Standard rules for kinding and type equality are shown below. In addition, the extensions include a case operator[4] for integer types, defined so that $(\text{case } 0 \text{ of } \tau_{zero} \text{ or } \tau_{succ}) \equiv \tau_{zero}$ and $(\text{case } succ(\tau) \text{ of } \tau_{zero} \text{ or } \tau_{succ}) \equiv (\tau_{succ}\ \tau)$.

Extensions to source and target languages for higher-order types

$$\begin{array}{ll} \textit{kinds} & \kappa = \ldots \mid \kappa_1 \rightarrow \kappa_2 \\ \textit{types} & \tau = \ldots \mid \lambda\alpha{:}\kappa.\tau \mid \tau_1\ \tau_2 \mid \text{case } \tau_N \text{ of } \tau_{zero} \text{ or } \tau_{succ} \end{array}$$

$$\frac{\Delta, \alpha \mapsto \kappa_a \vdash \tau : \kappa_b}{\Delta \vdash (\lambda\alpha{:}\kappa_a.\tau) : \kappa_a \rightarrow \kappa_b} \qquad \frac{\Delta \vdash \tau_f : \kappa_a \rightarrow \kappa_b \qquad \Delta \vdash \tau_a : \kappa_a}{\Delta \vdash (\tau_f\ \tau_a) : \kappa_b}$$

$$\frac{\Delta \vdash \tau_N : \textbf{int} \qquad \Delta \vdash \tau_{zero} : \kappa \qquad \Delta \vdash \tau_{succ} : \textbf{int} \rightarrow \kappa}{\Delta \vdash \text{case } \tau_N \text{ of } \tau_{zero} \text{ or } \tau_{succ} : \kappa}$$

$$\Theta \vdash (\lambda\alpha{:}\kappa.\tau_b)\ \tau_a \equiv [\alpha \leftarrow \tau_a]\tau_b$$

$$\Theta \vdash \lambda\alpha{:}\kappa.\tau\ \alpha \equiv \tau \quad \textit{where}\ \alpha \notin FV(\tau)$$

$$\Theta \vdash \text{case } 0 \text{ of } \tau_{zero} \text{ or } \tau_{succ} \equiv \tau_{zero}$$

$$\Theta \vdash \text{case } succ(\tau) \text{ of } \tau_{zero} \text{ or } \tau_{succ} \equiv \tau_{succ}\ \tau$$

$$\frac{\Theta \vdash \tau \equiv \tau' \qquad \alpha \notin FV(\Theta)}{\Theta \vdash \lambda\alpha{:}\kappa.\tau \equiv \lambda\alpha{:}\kappa.\tau'} \qquad \frac{\Theta \vdash \tau_1 \equiv \tau_1' \qquad \Theta \vdash \tau_2 \equiv \tau_2'}{\Theta \vdash \tau_1\ \tau_2 \equiv \tau_1'\ \tau_2'}$$

$$\frac{\Theta \vdash \tau_1 \equiv \tau_1' \qquad \Theta \vdash \tau_2 \equiv \tau_2' \qquad \Theta \vdash \tau_3 \equiv \tau_3'}{\Theta \vdash \text{case } \tau_1 \text{ of } \tau_2 \text{ or } \tau_3 \equiv \text{case } \tau_1' \text{ of } \tau_2' \text{ or } \tau_3'}$$

There are two obstacles to implementing the $NextRgn$ type operator. First, the type operator's range is an infinite set of regions; since the program can only allocate one region at a time, it will not be able to define the entire infinite $NextRgn$ function at once. Second, the type system lacks a conditional type that distinguishes between regions; there's no direct way to say "if the argument to $NextRgn$ is ρ_3, return ρ_4." Delayed types solve the first problem, so that the program can define a finite subset of the function and postpone further definitions until later. The integer case type shown above solves the second problem.

The integer case type applies to integers, not regions, but there's a trick that connects regions to integers. Let ρ have kind $\mathbf{int} \to \mathbf{rgn}$, and let $\rho_n = (\rho\, n)$. Then define

$$NextRgn(\rho) = \lambda\beta{:}\mathbf{int}.(\rho\, succ(\beta))$$

so that $(NextRgn(\rho)\, n) = (\rho\, succ(n))$. The encoding of the $List$ data structure remains as shown above, except that $@(\rho\, 0)$ replaces $@\rho$:

$$List(\rho) = \langle(\mathrm{int} \cup List(NextRgn(\rho))), List(\rho)\rangle @(\rho\, 0)$$

To hide the $(\rho\, 0)$, define the following type abbreviations:

$$\langle\tau_1, \tau_2\rangle \hat{@} \tau_r = \langle\tau_1, \tau_2\rangle @(\tau_r\, 0)$$

$$\widehat{\mathrm{Rgn}}(\tau) = \mathrm{Rgn}(\tau\, 0)$$

$$\widehat{\mathbf{rgn}} = \mathbf{int} \to \mathbf{rgn}$$

The definition of the type operator ρ uses "case" to distinguish between different integer arguments, and delayed types χ_k to grow as more ρ_k appear:

$$\begin{aligned}
\rho &= \chi_0 \\
&\equiv \lambda\beta{:}\mathbf{int}.\mathrm{case}\,\beta\,\mathrm{of}\,\rho_0\,\mathrm{or}\,\chi_1 \\
&\equiv \lambda\beta{:}\mathbf{int}.\mathrm{case}\,\beta\,\mathrm{of}\,\rho_0\,\mathrm{or}\,\lambda\beta'{:}\mathbf{int}.\mathrm{case}\,\beta'\,\mathrm{of}\,\rho_1\,\mathrm{or}\,\chi_2 \\
&\equiv \lambda\beta{:}\mathbf{int}.\mathrm{case}\,\beta\,\mathrm{of}\,\rho_0\,\mathrm{or}\,\lambda\beta'{:}\mathbf{int}.\mathrm{case}\,\beta'\,\mathrm{of}\,\rho_1\,\mathrm{or}\,\lambda\beta''{:}\mathbf{int}.\mathrm{case}\,\beta''\,\mathrm{of}\,\rho_2\,\mathrm{or}\,\chi_3 \\
&\ldots
\end{aligned}$$

where the program commits each delayed type χ_k to the type

$$\chi_k = \lambda\beta{:}\mathbf{int}.\mathrm{case}\,\beta\,\mathrm{of}\,\rho_k\,\mathrm{or}\,\chi_{k+1}$$

Notice that $\chi_0 \equiv \rho$, $\chi_1 \equiv NextRgn(\rho)$, $\chi_2 \equiv NextRgn(NextRgn(\rho))$, and so on. The function below, of type

$$\forall\rho{:}\widehat{\mathbf{rgn}}.\mathrm{Delay}(\rho) \to lin\langle\widehat{\mathrm{Rgn}}(\rho), \mathrm{Delay}(NextRgn(\rho))\rangle$$

exploits the relationship between χ and ρ to generate each "next" region on demand, given a delayed type capability for the current region:

$$
\begin{array}{l}
\lambda\rho{:}\widehat{\mathbf{rgn}}.\lambda x_{delay}:\mathrm{Delay}(\rho). \\
\quad \mathrm{unpack}\,\alpha_{rgn}, x_{rgn} = \mathrm{newrgn}\,\mathrm{in} \\
\quad \mathrm{unpack}\,\rho_{next}, x_{delayNext} = \mathrm{delay}(\widehat{\mathbf{rgn}})\,\mathrm{in} \\
\quad \mathrm{let}\,y = \lambda z{:}lin\langle\widehat{\mathrm{Rgn}}(\rho), \mathrm{Delay}(NextRgn(\rho))\rangle.z\,\mathrm{in} \\
\quad \mathrm{let}\,y' = \mathrm{commit}[x_{delay}](y : (\delta = \tau_\delta\,\mathrm{in}\,\tau_{y'}))\,\mathrm{in} \\
\quad y'\,lin\langle x_{rgn}, x_{delayNext}\rangle
\end{array}
$$

where:

$$
\begin{array}{l}
\tau_\delta = \lambda\beta{:}\mathbf{int}.\mathrm{case}\,\beta\,\mathrm{of}\,\alpha_{rgn}\,\mathrm{or}\,\rho_{next} \\
\tau_{y'} = lin\langle\widehat{\mathrm{Rgn}}(\delta), \mathrm{Delay}(NextRgn(\delta))\rangle \to lin\langle\widehat{\mathrm{Rgn}}(\rho), \mathrm{Delay}(NextRgn(\rho))\rangle
\end{array}
$$

The code is well-typed because:

$$
\begin{array}{r}
\{\} \vdash \widehat{\mathrm{Rgn}}(\tau_\delta) \equiv \mathrm{Rgn}(\alpha_{rgn}) \\
\{\} \vdash \mathrm{Delay}(NextRgn(\tau_\delta)) \equiv \mathrm{Delay}(\rho_{next})
\end{array}
$$

7 Safety and termination

This section summarizes the proofs of safety of the target language (as defined in appendix A), as well as the termination of the target language's coercions.

Since delayed types are the target language's novel feature, we point out the key aspects of the proofs relating to delayed types:

- Progress relies on Θ being a function, not a relation: each $\alpha \in \mathit{domain}(\Theta)$ maps to only one type. If this weren't the case, we could have $\Theta = \{\alpha \mapsto \mathrm{int}, \alpha \mapsto (\mathrm{int} \to \mathrm{int})\}$, so that $\Theta \vdash \mathrm{int} \equiv \mathrm{int} \to \mathrm{int}$, which would destroy the canonical forms lemma.
- Preservation relies on $\Theta(\alpha)$ never changing once it has been established; the program can extend Θ with new mappings, but it cannot remove or alter old mappings, as this could potentially invalidate existing typings. Therefore, it's crucial that a "commit" expression for a variable α only be well-typed when $\alpha \notin \mathit{domain}(\Theta)$. The rule for $\vdash C$ ensures this by requiring $\Phi \cap \mathit{domain}(\Theta) = \{\}$, and the linearity of Φ ensures that α disappears from Φ when it appears in Θ.

7.1 Typing lemmas

Lemma 1. *(Typing inversion) If $C \vdash e : \tau$, then one of the typing rules other than $\frac{C \vdash e:\tau' \quad C \vdash \tau' \equiv \tau}{C \vdash e:\tau}$ concludes that $C \vdash e : \tau'$ where $C \vdash \tau \equiv \tau'$. For example:*

– If $C \vdash succ(e) : \tau$, then $\frac{C \vdash e : \mathrm{Int}(\tau_0)}{C \vdash succ(e) : \mathrm{Int}(succ(\tau_0))}$, where $C \vdash \tau \equiv \mathrm{Int}(succ(\tau_0))$.

Proof. By induction on $C \vdash e : \tau$. The induction strips away $\frac{C \vdash e : \tau' \qquad C \vdash \tau' \equiv \tau}{C \vdash e : \tau}$ rules from $C \vdash e : \tau$ until it finds a rule different from $\frac{C \vdash e : \tau' \qquad C \vdash \tau' \equiv \tau}{C \vdash e : \tau}$.

Lemma 2. *(Type weakening) If $\Delta \vdash \tau : \kappa$ and $\Delta' = \Delta, \overrightarrow{\alpha \mapsto \kappa_\alpha}$, then $\Delta' \vdash \tau : \kappa$.*

Proof. By induction on $\Delta \vdash \tau : \kappa$.

Lemma 3. *(Type equivalence weakening) If $\Theta \vdash \tau_1 \equiv \tau_2$, and $\Theta' = \Theta, \overrightarrow{\alpha \mapsto \tau_\alpha}$, then $\Theta' \vdash \tau_1 \equiv \tau_2$.*

Proof. By induction on $\Theta \vdash \tau_1 \equiv \tau_2$.

Lemma 4. *(Expression weakening) If $\Delta; \Psi; \Phi; \Theta; \Gamma; \Lambda \vdash e : \tau$, and $\Delta' = \Delta, \overrightarrow{\alpha \mapsto \kappa}$ and $\Theta' = \Theta, \overrightarrow{\beta \mapsto \tau_\beta}$ and $\Gamma' = \Gamma, \overrightarrow{x \mapsto \tau_x}$, where $\Delta \vdash \tau_{xk} :^{non}_{n_k}$, then $\Delta'; \Psi; \Phi; \Theta'; \Gamma'; \Lambda \vdash e : \tau$.*

Proof. By induction on $\Delta; \Psi; \Phi; \Theta; \Gamma; \Lambda \vdash e : \tau$.

Lemma 5. *(Type equivalence inversion) If $\Theta \vdash \tau \equiv \tau'$ where τ and τ' are not type variables, then:*

– If $\tau = \forall \alpha : \kappa . \tau_1$ and $\alpha \notin FV(\Theta)$, then $\tau' = \forall \alpha : \kappa . \tau_1'$ and $\Theta \vdash \tau_1 \equiv \tau_1'$
– If $\tau = \exists \alpha : \kappa . \tau_1$ and $\alpha \notin FV(\Theta)$, then $\tau' = \exists \alpha : \kappa . \tau_1'$ and $\Theta \vdash \tau_1 \equiv \tau_1'$
– If $\tau = \tau_1 \to \tau_2$, then $\tau' = \tau_1' \to \tau_2'$ and $\Theta \vdash \tau_1 \equiv \tau_1'$ and $\Theta \vdash \tau_2 \equiv \tau_2'$
– If $\tau = \tau_1 \Rightarrow \tau_2$, then $\tau' = \tau_1' \Rightarrow \tau_2'$ and $\Theta \vdash \tau_1 \equiv \tau_1'$ and $\Theta \vdash \tau_2 \equiv \tau_2'$
– If $\tau = \phi\langle \tau_1, \ldots, \tau_k \rangle$, then $\tau' = \phi\langle \tau_1', \ldots, \tau_k' \rangle$ and $\Theta \vdash \tau_1 \equiv \tau_1' \ \ldots \ \Theta \vdash \tau_k \equiv \tau_k'$
– If $\tau = \tau_1 \vee \tau_2$ then $\tau' = \tau_1' \vee \tau_2'$ and $\Theta \vdash \tau_1 \equiv \tau_1'$ and $\Theta \vdash \tau_2 \equiv \tau_2'$
– If $\tau = \tau_1 \mapsto \tau_2$, then $\tau' = \tau_1' \mapsto \tau_2'$ and $\Theta \vdash \tau_1 \equiv \tau_1'$ and $\Theta \vdash \tau_2 \equiv \tau_2'$
– If $\tau = succ(\tau_1)$, then $\tau' = succ(\tau_1')$ and $\Theta \vdash \tau_1 \equiv \tau_1'$
– If $\tau = \mathrm{Int}(\tau_1)$, then $\tau' = \mathrm{Int}(\tau_1')$ and $\Theta \vdash \tau_1 \equiv \tau_1'$
– If $\tau = \mathrm{Delay}(\tau_1)$, then $\tau' = \mathrm{Delay}(\tau_1')$ and $\Theta \vdash \tau_1 \equiv \tau_1'$

Proof. First, rephrase the type equivalence rules $\Theta \vdash \tau \equiv \tau'$ as reduction rules $\Theta \vdash \tau \Rrightarrow \tau'$, simply by taking all the rules except symmetry and transitivity, and replacing $\equiv$ with $\Rrightarrow$. For example:

$$\Theta \vdash \tau \Rrightarrow \tau$$

$$\Theta, \alpha \mapsto \tau \vdash \alpha \Rrightarrow \tau$$

$$\frac{\Theta \vdash \tau \Rrightarrow \tau' \qquad \alpha \notin FV(\Theta)}{\Theta \vdash \forall \alpha : \kappa . \tau \Rrightarrow \forall \alpha : \kappa . \tau'}$$

$$\frac{\Theta \vdash \tau_1 \Rrightarrow \tau_1' \qquad \Theta \vdash \tau_2 \Rrightarrow \tau_2'}{\Theta \vdash \tau_1 \to \tau_2 \Rrightarrow \tau_1' \to \tau_2'}$$

Also write an alternate ("single-step") version of the lemma with $\Rrightarrow$, saying that if $\Theta \vdash \tau \Rrightarrow \tau'$ where τ and τ' are not type variables, then:

– If $\tau = \forall\alpha\!:\!\kappa.\tau_1$ and $\alpha \notin FV(\Theta)$, then $\tau' = \forall\alpha\!:\!\kappa.\tau_1'$ and $\Theta \vdash \tau_1 \Rrightarrow \tau_1'$
– If $\tau = \tau_1 \to \tau_2$, then $\tau' = \tau_1' \to \tau_2'$ and $\Theta \vdash \tau_1 \Rrightarrow \tau_1'$ and $\Theta \vdash \tau_2 \Rrightarrow \tau_2'$
– ...

The proof for the single-step version of the lemma follows by simple case analysis on the derivation of $\Theta \vdash \tau \Rrightarrow \tau'$.

Now prove, by a standard tiling argument (see Pierce[10], for example), that if $\Theta \vdash \tau \equiv \tau'$, then there is some τ'' so that $\Theta \vdash \tau \overset{*}{\Rrightarrow} \tau''$ and $\Theta \vdash \tau' \overset{*}{\Rrightarrow} \tau''$, where $\overset{*}{\Rrightarrow}$ indicates zero or more reductions using $\Rrightarrow$. Repeated applications of the single-step lemma to $\Theta \vdash \tau \overset{*}{\Rrightarrow} \tau''$ and $\Theta \vdash \tau' \overset{*}{\Rrightarrow} \tau''$ show that:

– If $\tau = \forall\alpha\!:\!\kappa.\tau_1$ and $\alpha \notin FV(\Theta)$, then $\tau' = \forall\alpha\!:\!\kappa.\tau_1'$ and $\Theta \vdash \tau_1 \overset{*}{\Rrightarrow} \tau_1''$ and $\Theta \vdash \tau_1' \overset{*}{\Rrightarrow} \tau_1''$
– If $\tau = \tau_1 \to \tau_2$, then $\tau' = \tau_1' \to \tau_2'$ and $\Theta \vdash \tau_1 \overset{*}{\Rrightarrow} \tau_1''$ and $\Theta \vdash \tau_2 \overset{*}{\Rrightarrow} \tau_2''$ and $\Theta \vdash \tau_1\,' \overset{*}{\Rrightarrow} \tau_1''$ and $\Theta \vdash \tau_2' \overset{*}{\Rrightarrow} \tau_2''$
– ...

The main lemma follows by observing that if $\Theta \vdash \tau \overset{*}{\Rrightarrow} \tau'$, then $\Theta \vdash \tau \equiv \tau'$ (an easy induction).

Lemma 6. *(Type equivalence kinds) If $\Theta \vdash \tau \equiv \tau'$ and $\Delta \vdash \tau : \kappa$ and $\forall\alpha \mapsto \tau \in \Theta.(\Delta \vdash \tau : \Delta(\alpha))$, then $\Delta \vdash \tau' : \kappa$.*

Proof. By induction on $\Theta \vdash \tau \equiv \tau'$.

Lemma 7. *(Environment typing) For each typing rule with conclusion $C \vdash e : \tau$, if we know that $\vdash C$, then for each premise in the rule of the form $C' \vdash e' : \tau'$, we know $\vdash C'$.*

Proof. Proved simultaneously with expression kinds lemma below. Note that if $C = C', C''$, then the proof is trivial, because C, C', C'' all share the same Δ and Θ, and all assumptions in C' also appear in C.

Lemma 8. *(Expression kinds) If $C \vdash e : \tau$, where $C = \Delta;\Psi;\Phi;\Theta;\Gamma;\Lambda$ and $\vdash C$, then $\Delta \vdash \tau :\overset{\phi}{n}$.*

Proof. By induction on $C \vdash e : \tau$.

Case 1. $\dfrac{\Delta;\{\};\{\};\Theta;\{x \mapsto \tau_a\};\to \vdash e:\tau_b \qquad \Delta \vdash \tau_a :\overset{\phi}{n}}{\Delta;\{\};\{\};\Theta;\overset{non}{\Gamma};\Lambda \vdash (\lambda x{:}\tau_a.e):\tau_a \to \tau_b}$. The $\Delta \vdash \tau_a : \kappa$ condition ensures that $\Delta \vdash \{x \mapsto \tau_a\}$. Induction proves that $\Delta \vdash \tau_b :\overset{\phi'}{n'}$, so that $\Delta \vdash \tau_a \to \tau_b : \overset{non}{1}$.

Case 2. $\dfrac{C_a \vdash e_a : \phi\langle\tau_1,\ldots,\tau_k\rangle \qquad C_b, x_1 \mapsto \tau_1,\ldots,x_k \mapsto \tau_k \vdash e_b : \tau_b}{C_a, C_b \vdash \text{let } \langle x_1,\ldots,x_k\rangle = e_a \text{ in } e_b : \tau_b}$. By induction, $\Delta \vdash \phi\langle\tau_1,\ldots,\tau_k\rangle :\overset{\phi_a}{n_a}$; inversion on this shows that $\Delta \vdash \tau_1 :\overset{\phi_1}{n_1} \ldots \Delta \vdash \tau_k :\overset{\phi_k}{n_k}$, so that $\vdash C_b, x_1 \mapsto \tau_1,\ldots,x_k \mapsto \tau_k$. Induction shows that $\Delta \vdash \tau_b$.

Case 3. $\frac{C_1 \vdash e_1 : (\exists \alpha \kappa . \tau_1) \quad C_2, \alpha \mapsto \kappa, x \mapsto \tau_1 \vdash e_2 : \tau_2 \quad C_2 \vdash \tau_2 :^{\phi} n}{C_1, C_2 \vdash (\mathsf{unpack}\, \alpha, x = e_1 \,\mathsf{in}\, e_2) : \tau_2}$. To show $\vdash C_2, \alpha \mapsto \kappa, x \mapsto \tau_1$, use induction to show $\Delta \vdash (\exists \alpha : \kappa . \tau_1) :^{\phi'} n'$ and invert this to show $\Delta, \alpha \mapsto \kappa \vdash \tau_1 :^{\phi'} n'$.

Case 4. $\frac{C \vdash e : \tau' \quad C \vdash \tau' \equiv \tau}{C \vdash e : \tau}$. By induction, $\Delta \vdash \tau' :^{\phi'} n'$. By the type equivalence kinds lemma, $\Delta \vdash \tau :^{\phi'} n'$.

Lemma 9. *(Nonlinear value) If $C \vdash v :^{non} n$, then there is some C' so that $C =^{non} C'$.*

Proof. By induction on $C \vdash v :^{non} n$. Notice that $v = (\lambda x : \tau . e)$ and $v = (\lambda x : \tau \Rightarrow e)$ and $v = (\lambda \alpha : \kappa \Rightarrow e)$ are base cases of the induction, since they only type-check in nonlinear environments. Also notice that the typing rule for $v = non\langle v_1, \ldots, v_k \rangle$, requires nonlinear $v_1, \ldots, v_k$, which allows induction into the $v_1, \ldots, v_k$.

Lemma 10. *(Coercion environment change) If $C, \Lambda \vdash v : \tau$, then $C, \Lambda' \vdash v : \tau$*

Proof. By induction on $C, \Lambda \vdash v : \tau$. Notice that $v = (\lambda x : \tau . e)$ and $v = (\lambda x : \tau \Rightarrow e)$ and $v = (\lambda \alpha : \kappa \Rightarrow e)$ are base cases of the induction.

7.2 Preservation

Lemma 11. *(Type substitution) If $\Delta, \alpha_1 \mapsto \kappa_1, \ldots, \alpha_n \mapsto \kappa_n \vdash \tau : \kappa$ and $\Delta \vdash \tau_1 : \kappa_1 \ldots \Delta \vdash \tau_n : \kappa_n$, then $\Delta \vdash [\alpha_1 \leftarrow \tau_1, \ldots \alpha_n \leftarrow \tau_n]\tau : \kappa$.*

Proof. By induction on the derivation of $\Delta, \alpha_1 \mapsto \kappa_1, \ldots, \alpha_n \mapsto \kappa_n \vdash \tau : \kappa$. Let $[\sigma] = [\alpha_1 \leftarrow \tau_1, \ldots \alpha_n \leftarrow \tau_n]$. Sample cases:

Case 1. $\Delta, \alpha_1 \mapsto \kappa_1, \ldots, \alpha_n \mapsto \kappa_n \vdash \alpha_k : \kappa_k$. In this case, $[\sigma]\tau = \tau_k$, and we know $\Delta \vdash \tau_k : \kappa_k$.

Case 2. $\frac{\Delta, \alpha_1 \mapsto \kappa_1, \ldots, \alpha_n \mapsto \kappa_n, \alpha \mapsto \kappa \vdash \tau :^{\phi} n}{\Delta, \alpha_1 \mapsto \kappa_1, \ldots, \alpha_n \mapsto \kappa_n \vdash \forall \alpha \kappa . \tau :^{\phi} n}$. By weakening, $\Delta, \alpha \mapsto \kappa \vdash \tau_1 : \kappa_1 \ldots \Delta, \alpha \mapsto \kappa \vdash \tau_n : \kappa_n$ By induction (using environment $\Delta, \alpha \mapsto \kappa$), $\Delta, \alpha \mapsto \kappa \vdash [\sigma]\tau :^{\phi} n$. The typing of $[\sigma]\forall \alpha : \kappa . \tau = \forall \alpha : \kappa . [\sigma]\tau$ follows immediately.

Lemma 12. *(Type equivalence substitution) If $\Theta \vdash \tau \equiv \tau'$ and $\alpha_1 \notin domain(\Theta), \ldots, \alpha_n \notin domain(\Theta)$ and $[\sigma] = [\alpha_1 \leftarrow \tau_1, \ldots \alpha_n \leftarrow \tau_n]$, then $[\sigma]\Theta \vdash [\sigma]\tau \equiv [\sigma]\tau'$.*

Proof. By induction on $\Theta \vdash \tau \equiv \tau'$. Sample cases:

Case 1. $\Theta, \alpha \mapsto \tau \vdash \alpha \equiv \tau$. Since no α_k in $[\sigma]$ appear in $domain(\Theta, \alpha \mapsto \tau)$, we know $[\sigma]\alpha = \alpha$, and can conclude $([\sigma]\Theta), \alpha \mapsto ([\sigma]\tau) \vdash \alpha \equiv ([\sigma]\tau)$.

Case 2. $\frac{\Theta \vdash \tau \equiv \tau' \qquad \alpha \notin FV(\Theta)}{\Theta \vdash \forall \alpha \kappa.\tau \equiv \forall \alpha \kappa.\tau'}$. Alpha-rename $\forall \alpha : \kappa.\tau'$ so that $\alpha \notin \{\alpha_1, \ldots, \alpha_n\}$ and $\alpha \notin FV([\sigma]\Theta)$. By induction, $[\sigma]\Theta \vdash [\sigma]\tau \equiv [\sigma]\tau'$. Therefore, $[\sigma]\Theta \vdash \forall \alpha : \kappa.[\sigma]\tau \equiv \forall \alpha : \kappa.[\sigma]\tau'$, so $[\sigma]\Theta \vdash [\sigma]\forall \alpha : \kappa.\tau \equiv [\sigma]\forall \alpha : \kappa.\tau'$.

Lemma 13. *(Expression-type substitution) Let $C = \Delta; \Psi; \Phi; \Theta; \Gamma; \Lambda$. Define the substitution $[\sigma] = [\overrightarrow{\alpha \leftarrow \tau_\alpha}]$. Suppose the following conditions hold:*

- no α_k appears in $\Delta; \Psi; \Phi; \Theta$ (note: to satisfy this condition, other lemmas typically use alpha-renaming to rename the α_k before invoking substitution)
- $C, \overrightarrow{\alpha \mapsto \kappa_\alpha} \vdash e : \tau$
- for each $\alpha_k \leftarrow \tau_{\alpha k}$ in $\overrightarrow{\alpha \leftarrow \tau_\alpha}$, $\Delta \vdash \tau_{\alpha k} : \kappa_{\alpha k}$.

Then we can conclude that $[\sigma]C \vdash [\sigma]e : [\sigma]\tau$.

Proof. By induction on $C, \overrightarrow{\alpha \mapsto \kappa_\alpha} \vdash e : \tau$. Sample cases:

Case 1. $C, \overrightarrow{\alpha \mapsto \kappa_\alpha} \vdash x : \tau$, where $C = \Delta; \{\}; \{\}; \Theta; \overset{non}{\Gamma_0}, \{x \mapsto \tau\}; \Lambda$. Here, $[\sigma]C = \Delta; \{\}; \{\}; \Theta; [\sigma] \overset{non}{\Gamma_0}, \{x \mapsto [\sigma]\tau\}; \Lambda$ and $[\sigma]e = x$.

Case 2. $\frac{C_1 \vdash e_1 : (\exists \beta \kappa.\tau_1) \qquad C_2, \beta \mapsto \kappa, x \mapsto \tau_1 \vdash e_2 : \tau_2 \qquad C_2 \vdash \tau_2 : \overset{\phi}{n}}{C_1, C_2 \vdash (\text{unpack}\, \beta, x = e_1 \,\text{in}\, e_2) : \tau_2}$, where β is alpha-renamed so that $\beta \notin \{\alpha_1, \ldots, \alpha_n\}$. By induction, $[\sigma]C_1 \vdash [\sigma]e_1 : [\sigma](\exists \beta : \kappa.\tau_1)$ (note that $[\sigma](\exists \beta : \kappa.\tau_1) = (\exists \beta : \kappa.[\sigma]\tau_1)$) and $([\sigma]C_2), \beta \mapsto \kappa, x \mapsto [\sigma]\tau_1 \vdash [\sigma]e_2 : [\sigma]\tau_2$. By type substitution, $[\sigma]C_2 \vdash [\sigma]\tau_2 : \overset{\phi}{n}$. Since C_1, C_2 is well-formed, each $x_k \mapsto \tau_{xk}$ in C_1 and C_2 have some $\Delta \vdash \tau_{xk} : \overset{\phi_{xk}}{n_{xk}}$, which by type substitution implies $\Delta \vdash [\sigma]\tau_{xk} : \overset{\phi_{xk}}{n_{xk}}$, so that $[\sigma]C = ([\sigma]C_1), ([\sigma]C_2)$.

Lemma 14. *(Expression-value substitution) Let $C = \Delta; \Psi; \Phi; \Theta; \Gamma; \Lambda$. Define the substitution $[\sigma] = [\overrightarrow{x \leftarrow v_x}, \overrightarrow{y \leftarrow v_y}, \overrightarrow{z \leftarrow v_z}]$. Suppose the following conditions hold:*

- $\vdash C'$, where $C' = C, \overrightarrow{C_x}, \overrightarrow{C_y}, \overrightarrow{C_z} = \Delta; \Psi'; \Phi'; \Theta; \Gamma'; \Lambda$
- for each z_k in $\overrightarrow{z}$, $z_k \notin domain(\Gamma)$.
- $C, \overrightarrow{x \mapsto \tau_x}, \overrightarrow{y \mapsto \tau_y} \vdash e : \tau$
- for each $x_k \leftarrow v_{xk}$ in $\overrightarrow{x \leftarrow v_x}$, $C_{xk} \vdash v_{xk} : \tau_{xk}$ and $\Delta \vdash \tau_{xk} : \overset{non}{n_{xk}}$.
- for each $y_k \leftarrow v_{yk}$ in $\overrightarrow{y \leftarrow v_y}$, $C_{yk} \vdash v_{yk} : \tau_{yk}$ and $\Delta \vdash \tau_{yk} : \overset{lin}{n_{yk}}$.
- for each $z_k \leftarrow v_{zk}$ in $\overrightarrow{z \leftarrow v_z}$, $C_{zk} \vdash v_{zk} : \tau_{zk}$ and $\Delta \vdash \tau_{zk} : \overset{lin}{n_{zk}}$.

Then we can conclude that $C, \overrightarrow{C_y} \vdash [\sigma]e : \tau$.

Proof. By induction on $C, \overrightarrow{x \mapsto \tau_x}, \overrightarrow{y \mapsto \tau_y} \vdash e : \tau$. Sample cases:

Case 1. $C, \overrightarrow{x \mapsto \tau_x}, \overrightarrow{y \mapsto \tau_y} \vdash x_k : \tau_{xk}$, where $C, \overrightarrow{x \mapsto \tau_x}, \overrightarrow{y \mapsto \tau_y} = \Delta; \{\}; \{\}; \Theta; \overset{non}{\Gamma_0}, \{x_k \mapsto \tau_{xk}\}; \Lambda$. Since $\overset{non}{\Gamma_0}$ contains no linear assumptions, $\overrightarrow{y}$ must be empty, so $C, \overrightarrow{C_y} = C$. We know that $[\sigma]x_k = v_{xk}$ and $C_{xk} \vdash v_{xk} : \tau_{xk}$. By the nonlinear environment lemma, C_{xk} is nonlinear. Since $C' = C, \overrightarrow{C_x}, \overrightarrow{C_y}, \overrightarrow{C_z}$ and C and C_{xk} are nonlinear, $C = C_{xk}$, so that $C \vdash v_{xk} : \tau_{xk}$.

Case 2. $\dfrac{C_1, \overrightarrow{x \mapsto \tau_x}, \overrightarrow{y' \mapsto \tau'_y} \vdash e_1 : (\exists \beta\kappa.\tau_1) \qquad C_2, \overrightarrow{x \mapsto \tau_x}, \overrightarrow{y'' \mapsto \tau''_y}, \beta \mapsto \kappa, x \mapsto \tau_1 \vdash e_2 : \tau_2 \qquad C_2 \vdash \tau_2 :^{\phi} n}{C_1, C_2, \overrightarrow{x \mapsto \tau_x}, \overrightarrow{y \mapsto \tau_y} \vdash (\text{unpack}\,\beta, x = e_1\,\text{in}\,e_2) : \tau_2}$,
where x is alpha-renamed so that it does not appear in $\{\overrightarrow{x}, \overrightarrow{y}, \overrightarrow{z}\}$, and β is alpha-renamed so that it does not appear in $[\sigma]$, and $\Delta \vdash \{\overrightarrow{y \mapsto \tau_y}\} = \{\overrightarrow{y' \mapsto \tau'_y}\}, \{\overrightarrow{y'' \mapsto \tau''_y}\}$. By induction (putting $\overrightarrow{y'' \mapsto \tau''_y}$ in the z variables), $C_1 \vdash [\sigma]e_1 : (\exists \beta : \kappa.\tau_1)$. By induction (putting $\overrightarrow{y' \mapsto \tau'_y}$ into the z variables), $C_2, \beta \mapsto \kappa, x \mapsto \tau_1 \vdash [\sigma]e_2 : \tau_2$.

Theorem 1. *(Preservation) If* $\Delta; \Psi_{spare}, \Psi_e; \Phi_{spare}, \Phi_e; \Theta; \{\}; \Lambda \vdash (M, e : \tau)$ *and* $\Delta; \Psi_e; \Phi_e; \Theta; \{\}; \Lambda \vdash e : \tau$ *and* $(M, e) \longrightarrow (M', e')$*, then there is some* $\Delta', \Psi'_e, \Phi'_e, \Theta'$ *so that* $\Delta \subseteq \Delta'$ *and* $\Theta \subseteq \Theta'$ *and* $\Delta'; \Psi_{spare}, \Psi'_e; \Phi_{spare}, \Phi'_e; \Theta'; \{\}; \Lambda \vdash (M', e' : \tau)$ *and* $\Delta'; \Psi'_e; \Phi'_e; \Theta'; \{\}; \Lambda \vdash e' : \tau$.

Proof. By induction on the derivation of $C \vdash e : \tau$, where $C = \Delta; \Psi_e; \Phi_e; \Theta; \{\}; \Lambda$. Hold Δ and Θ and $\Psi_{Me} = \Psi_{spare}, \Psi_e$ and $\Phi_{Me} = \Phi_{spare}, \Phi_e$ fixed throughout the induction as Ψ_{spare} and Ψ_e and Φ_{spare} and Φ_e and Λ vary. Sample cases:

Case 1. $e = e_f\, e_a$ and $e' = e'_f\, e_a$, where:

$$\frac{C_f, \rightarrow \vdash e_f : \tau_a \Rightarrow \tau_b \qquad C_a, \rightarrow \vdash e_a : \tau_a}{(C_f, C_a), \rightarrow \vdash e_f\, e_a : \tau_b}$$

where $C_a = \Delta; \Psi_a; \Phi_a; \Theta; \{\}; \Lambda$ and $C_f = \Delta; \Psi_f; \Phi_f; \Theta; \{\}; \Lambda$. Note that $\Psi_{Me} = (\Psi_{spare}, \Psi_a), \Psi_f$ and $\Phi_{Me} = (\Phi_{spare}, \Phi_a), \Phi_f$.

By induction, $C'_f, \rightarrow \vdash e'_f : \tau_a \Rightarrow \tau_b$, where $C'_f = \Delta'; \Psi'_f; \Phi'_f; \Theta'; \{\}; \Lambda$ and $\Delta \subseteq \Delta'$ and $\Theta \subseteq \Theta'$ and $\Psi'_{Me} = (\Psi_{spare}, \Psi_a), \Psi'_f$ and $\Phi'_{Me} = (\Phi_{spare}, \Phi_a), \Phi'_f$.

Let $C'_a = \Delta'; \Psi_a; \Phi_a; \Theta'; \{\}; \Lambda$. By expression weakening on Δ and Θ, $C'_a, \rightarrow \vdash e_a : \tau_a$. Choose $\Psi'_e = \Psi'_f, \Psi_a$ and $\Phi'_e = \Phi'_f, \Phi_a$ (we know that these are well formed, because $\Psi'_{Me} = (\Psi_{spare}, \Psi_a), \Psi'_f$ and $\Phi'_{Me} = (\Phi_{spare}, \Phi_a), \Phi'_f$ are well-formed). We can conclude:

$$\frac{C'_f, \rightarrow \vdash e'_f : \tau_a \Rightarrow \tau_b \qquad C'_a, \rightarrow \vdash e_a : \tau_a}{(C'_f, C'_a), \rightarrow \vdash e'_f\, e_a : \tau_b}$$

Induction also tells us that $\Delta'; (\Psi_{spare}, \Psi_a), \Psi'_f; (\Phi_{spare}, \Phi_a), \Phi'_f; \Theta'; \{\}; \Lambda \vdash (M', e'_f : \tau_a \Rightarrow \tau_b)$, which implies that $(\Phi_{spare}, \Phi'_e) \cap domain(\Theta') = \{\}$ and $\forall n \in domain(\Psi').(\Delta'; \{\}; \{\}; \Theta'; \{\}; \rightarrow \vdash M'(n) : \Psi'(n))$, where $\Psi' = \Psi_{spare}, \Psi'_e$. From this, we conclude that $\Delta'; \Psi_{spare}, \Psi'_e; \Phi_{spare}, \Phi'_e; \Theta'; \{\}; \Lambda \vdash (M', e' : \tau)$.

Case 2. $e = (\lambda x{:}\tau_x \Rightarrow e_f)\, v_a$ and $e' = \text{coerce}([x \leftarrow v_a]e_f)$, where

$$\frac{C_f, \rightarrow \vdash (\lambda x{:}\tau_x \Rightarrow e_f) : \tau_a \Rightarrow \tau_b \qquad C_a, \rightarrow \vdash v_a : \tau_a}{(C_f, C_a), \rightarrow \vdash (\lambda x{:}\tau_a \Rightarrow e_f)\, v_a : \tau_b}$$

where $C_a = \Delta; \Psi_a; \Phi_a; \Theta; \{\}; \Lambda$. By inversion, $C_f = \Delta; \{\}; \{\}; \Theta; \{\}; \Lambda$ and:

$$\frac{C_f, x \mapsto \tau_x, \Rightarrow\vdash e_f : \tau_f \qquad C_f \vdash \tau_f :\overset{\phi}{0}}{C_f \vdash (\lambda x{:}\tau_x \Rightarrow e_f) : \tau_a \Rightarrow \tau_b}$$

where $\Theta \vdash \tau_x \Rightarrow \tau_f \equiv \tau_a \Rightarrow \tau_b$, which we use to show $C_a, \rightarrow\vdash v_a : \tau_x$. By the coercion-environment-change lemma, $C_a, \Rightarrow\vdash v_a : \tau_x$. By expression substitution:

$$(C_f, C_a), \Rightarrow\vdash [x \leftarrow v_a]e_f : \tau_f$$

By the typing rule for coerce:

$$\frac{(C_f, C_a), \Rightarrow\vdash [x \leftarrow v_a]e_f : \tau_f \qquad (C_f, C_a) \vdash \tau_f :\overset{\phi}{0}}{C_f, C_a \vdash \text{coerce}([x \leftarrow v_a]e_f) : \tau_f}$$

Choose $\Theta' = \Theta$ and $\Phi'_e = \Phi_e$ and $\Psi'_e = \Psi_e$ and $\Delta' = \Delta$. Since $M' = M$, the well-typing of (M', e) follows immediately.

Case 3. $e = \text{delay}(\kappa)$ and $e' = \text{pack}[\beta, \text{fact}]\,\text{as}\,\exists\alpha{:}\kappa.\text{Delay}(\alpha)$, where β is fresh and:

$$C \vdash \text{delay}(\kappa) : \exists\alpha{:}\kappa.\text{Delay}(\alpha)$$

where $C = \Delta; \{\}; \{\}; \Theta; \{\}; \Lambda$. Choose $\Theta' = \Theta$ and $\Phi'_e = \{\beta\}$ and $\Psi'_e = \{\}$ and $\Delta' = (\Delta, \beta \mapsto \kappa)$. The type variable β is fresh, so that $\beta \notin \mathit{domain}(\Theta)$ and $(\Phi_{spare}, \Phi'_e) \cap \mathit{domain}(\Theta') = \{\}$. Then we can conclude:

$$\frac{\Delta'; \{\}; \{\beta\}; \Theta; \{\}; \Lambda \vdash \text{fact} : \text{Delay}(\beta) \qquad C' \vdash \beta : \kappa \qquad C' \vdash \exists\alpha{:}\kappa.\text{Delay}(\alpha) :\overset{lin}{0}}{\Delta'; \{\}; \{\beta\}; \Theta; \{\}; \Lambda \vdash (\text{pack}[\beta, \text{fact}]\,\text{as}\,\exists\alpha{:}\kappa.\text{Delay}(\alpha)) : (\exists\alpha{:}\kappa.\text{Delay}(\alpha))}$$

$M' = M$, so that weakening on Δ establishes the typings of each $M'(n)$, so that $\Delta'; \Psi_{spare}, \Psi'_e; \Phi_{spare}, \Phi'_e; \Theta'; \{\}; \Lambda \vdash (M', e' : \tau)$.

Case 4. $e = \text{commit}[\text{fact}](v : (\alpha = \tau_\alpha\,\text{in}\,\tau_{data}))$ and $e' = v$, where:

$$\frac{C_{delay} \vdash \text{fact} : \text{Delay}(\tau_{delay}) \qquad C_{data} \vdash v : [\alpha \leftarrow \tau_{delay}]\tau_{data}}{C_{delay}, C_{data} \vdash \text{commit}[\text{fact}](v : (\alpha = \tau_\alpha\,\text{in}\,\tau_{data})) : [\alpha \leftarrow \tau_\alpha]\tau_{data}}$$

where $C_{data} = \Delta; \Psi_{data}; \Phi_{data}; \Theta; \{\}; \Lambda$. By inversion, $C_{delay} = \Delta; \{\}; \{\beta\}; \Theta; \{\}; \Lambda$ and $\Theta \vdash \tau_{delay} \equiv \beta$. Choose $\Delta' = \Delta$, $\Phi'_e = \Phi_{data}$,$\Psi'_e = \Psi_{data}$, and $\Theta' = (\Theta, \beta \mapsto \tau_\alpha)$. By weakening, $\Theta' \vdash \tau_{delay} \equiv \beta$. A simple induction on τ_{data} proves that $\Theta' \vdash [\alpha \leftarrow \tau_{delay}]\tau_{data} \equiv [\alpha \leftarrow \tau_\alpha]\tau_{data}$. With this, we can conclude:

$$\frac{\Delta'; \Psi'_e; \Phi'_e; \Theta'; \{\}; \Lambda \vdash v : [\alpha \leftarrow \tau_{delay}]\tau_{data} \qquad \Theta' \vdash [\alpha \leftarrow \tau_{delay}]\tau_{data} \equiv [\alpha \leftarrow \tau_\alpha]\tau_{data}}{\Delta'; \Psi'_e; \Phi'_e; \Theta'; \{\}; \Lambda \vdash v : [\alpha \leftarrow \tau_\alpha]\tau_{data}}$$

$M' = M$, so that weakening on Θ establishes the typings of each $M'(n)$, so that $\Delta'; \Psi_{spare}, \Psi'_e; \Phi'; \Theta'; \{\}; \Lambda \vdash (M', e' : \tau)$.

Case 5. $e = \text{coerce}(e_1)$ and $e' = \text{coerce}(e_1')$, where:

$$\frac{C, \Rightarrow \vdash e_1 : \tau \qquad C \vdash \tau \overset{\phi}{:} 0}{C \vdash \text{coerce}(e_1) : \tau}$$

By induction, $C', \Rightarrow \vdash e_1' : \tau$, where $C' = \Delta'; \Psi_e'; \Phi_e'; \Theta'; \{\}; \Lambda$ and $\Delta \subseteq \Delta'$ and $\Theta \subseteq \Theta'$. By type weakening, $C' \vdash \tau \overset{\phi}{:} 0$, so we can conclude:

$$\frac{C', \Rightarrow \vdash e_1' : \tau \qquad C' \vdash \tau \overset{\phi}{:} 0}{C' \vdash \text{coerce}(e_1') : \tau}$$

Induction also tells us that $\Delta'; \Psi_{spare}, \Psi_e'; \Phi_{spare}, \Phi_e'; \Theta'; \{\}; \Lambda \vdash (M', e_1' : \tau)$, which implies that $\Phi' \cap domain(\Theta') = \{\}$ and $\forall n \in domain(\Psi').(\Delta'; \{\}; \{\}; \Theta'; \{\}; \rightarrow \vdash M'(n) : \Psi'(n))$, where $\Psi = \Psi_{spare}, \Psi_e'$. From this, we conclude that

$$\Delta'; \Psi_{spare}, \Psi_e'; \Phi_{spare}, \Phi_e'; \Theta'; \{\}; \Lambda \vdash (M', \text{coerce}(e_1') : \tau)$$

.

Case 6. $e = \text{coerce}(v)$ and $e' = v$, where:

$$\frac{C, \Rightarrow \vdash v : \tau \qquad C \vdash \tau \overset{\phi}{:} 0}{C \vdash \text{coerce}(v) : \tau}$$

Choose $\Delta' = \Delta$ and $\Phi_e' = \Phi_e$ and $\Psi_e' = \Psi_e$ and $\Theta' = \Theta$, so that $C' = C$. By the coercion-environment-change lemma on $C, \Rightarrow \vdash v : \tau$, we know that $C \vdash v : \tau$, so that $C' \vdash v : \tau$. Since $M' = M$, the typing of $(M', v : \tau)$ follows immediately.

Case 7. $e = \text{store[fact]}(n \leftarrow v_{data})$ and $e' = \text{fact}$ and $M' = [n \leftarrow v_{data}]M$, where:

$$\frac{\begin{array}{c} C_m, \rightarrow \vdash \text{fact} : \tau_{addr} \mapsto \tau_{data} \\ C_p, \rightarrow \vdash n : \text{Int}(\tau_{addr}) \\ C_d, \rightarrow \vdash v_{data} : \tau_{data}' \qquad C_d \vdash \tau_{data}' \overset{non}{:} 1 \end{array}}{(C_m, C_p, C_d), \rightarrow \vdash \text{store[fact]}(n \leftarrow v_{data}) : \tau_{addr} \mapsto \tau_{data}'}$$

By inversion, $C_m = \Delta; \{n_{ma} \mapsto \tau_{md}\}; \{\}; \Theta; \{\}; \Lambda$ and $\Theta \vdash \tau_{addr} \mapsto \tau_{data} \equiv n_{ma} \mapsto \tau_{md}$. Also by inversion, $C_p = \Delta; \{\}; \{\}; \Theta; \{\}; \Lambda$ and $\Theta \vdash \text{Int}(n) \equiv \text{Int}(\tau_{addr})$. By the non-linear value lemma, $C_d = \Delta; \{\}; \{\}; \Theta; \{\}; \Lambda$. By type equality inversion, $\Theta \vdash n_{ma} \equiv \tau_{addr} \equiv n$ and $n_{ma} = n$.

Choose $\Psi_e' = \{n \mapsto \tau_{data}'\}$ and $\Phi_e' = \Phi_e = \{\}$ and $\Theta' = \Theta$ and $\Delta' = \Delta$. Then we conclude:

$$\Delta'; \{n \mapsto \tau_{data}'\}; \{\}; \Theta'; \{\}; \rightarrow \vdash \text{fact} : n \mapsto \tau_{data}'$$

For all $k \neq n$, $M'(k) = M(k)$, so that $M'(k)$ remains well-typed with type $\Psi'_{Me}(k) = \Psi_{Me}(k)$, where $\Psi'_{Me} = \Psi_{spare}, \Psi'_e$. We know that $M'(n) = v_{data}$, and we use v_{data}'s typing to type $M'(n)$:

$$C_d, \rightarrow \vdash M'(n) : \Psi'(n)$$

The well-typing of (M, e) follows immediately.

7.3 Progress

Lemma 15. *(Canonical Forms)*

1. If $C \vdash v : (\forall \alpha{:}\kappa.\tau)$, then $v = \lambda\alpha{:}\kappa.v'$ or $v = \lambda\alpha{:}\kappa \Rightarrow e$
2. If $C \vdash v : (\exists \alpha{:}\kappa.\tau)$, then $v = \text{pack}[\tau', v']\,\text{as}\,\exists\alpha{:}\kappa.\tau$
3. If $C \vdash v : \tau_1 \rightarrow \tau_2$, then $v = \lambda x{:}\tau_1.e$
4. If $C \vdash v : \tau_1 \Rightarrow \tau_2$, then $v = \lambda x{:}\tau_1 \Rightarrow e$ or $v = v_1 \circ v_2$
5. If $C \vdash v : \phi\langle\tau_1, \ldots, \tau_k\rangle$, then $v = \phi\langle v_1, \ldots, v_k\rangle$
6. If $C \vdash v : \tau_1 \vee \tau_2$, then $v = \text{disj}^n_{\tau_1 \vee \tau_2}(v')$
7. If $C \vdash v : \tau_1 \mapsto \tau_2$, then $v = \text{fact}$
8. If $C \vdash v : \text{Int}(\tau)$, then $v = 0$ or $v = succ(v')$
9. If $C \vdash v : \text{Delay}(\tau)$, then $v = \text{fact}$

Proof. For each type τ above, proof by induction on $C \vdash v : \tau$. For example, for $C \vdash v : \tau_1 \rightarrow \tau_2$ there are two possible cases:

Case 1. $\frac{C \vdash v : \tau' \qquad C \vdash \tau' \equiv \tau}{C \vdash v : \tau}$, where $\tau = \tau_1 \rightarrow \tau_2$. By type equivalence inversion, $\tau' = \tau'_1 \rightarrow \tau'_2$, and we can use induction.

Case 2. $\frac{\Delta;\{\};\{\};\Theta;\{x \mapsto \tau_a\};\rightarrow \vdash e : \tau_b}{\Delta;\{\};\{\};\Theta;\overset{non}{\Gamma};\Lambda \vdash (\lambda x{:}\tau_a.e) : \tau_a \rightarrow \tau_b}$. In this case, $v = (\lambda x{:}\tau_a.e)$.

Theorem 2. *(Progress) If e is not a value and $\Delta;\Psi_{Me};\Phi_{Me};\Theta;\{\};\Lambda \vdash (M, e : \tau)$, then there is some (M', e') so that $(M, e) \longrightarrow (M', e')$.*

Proof. Observe that by the rule for typing (M, e), we know that $\Psi_{Me} = \Psi_{spare}, \Psi$ and $\Phi_{Me} = \Phi_{spare}, \Phi$ and $C \vdash e : \tau$, where $C = \Delta;\Psi;\Phi;\Theta;\{\};\Lambda$. Now prove that (M, e) steps, by induction on the derivation of $C \vdash e : \tau$ (holding M and Ψ_{Me} and Φ_{Me} fixed throughout the induction as C, e, and τ vary). Sample cases:

Case 1. $e = x$, where $\overset{non}{C'}, x \mapsto \tau \vdash x : \tau$

Impossible: our assumptions require that $\Gamma = \{\}$, so C cannot contain $x \mapsto \tau$.

Case 2. $e = v_f\, v_a$, where:

$$\frac{C_f, \rightarrow \vdash v_f : \tau_a \Rightarrow \tau_b \qquad C_a, \rightarrow \vdash v_a : \tau_a}{(C_f, C_a), \rightarrow \vdash v_f\, v_a : \tau_b}$$

By canonical forms, either $v_f = \lambda x{:}\tau_a \Rightarrow e_b$ or $v_f = v_{f1} \circ v_{f2}$. In either case, $v_f\, v_a$ steps.

Case 3. $e = \text{load}[v_{mem}](v_{ptr})$, where:

$$\frac{C_{mem}, \rightarrow \vdash v_{mem} : \tau_{addr} \mapsto \tau_{data} \qquad C_{ptr}, \rightarrow \vdash v_{ptr} : \text{Int}(\tau_{addr})}{(C_{mem}, C_{ptr}), \rightarrow \vdash \text{load}[v_{mem}](v_{ptr}) : lin\langle(\tau_{addr} \mapsto \tau_{data}), \tau_{data}\rangle}$$

By canonical forms, $v_{mem} = \text{fact}$ and $v_{ptr} = n$. By inversion on the typing of v_{ptr}, we know $\Theta \vdash n \equiv \tau_{addr}$. By inversion on the typing of v_{mem}, we know $C_{mem} = C'_{mem}, n' \mapsto \tau'_{data}$ where $\Theta \vdash \tau_{addr} \equiv n'$. By transitivity, $\Theta \vdash n \equiv n'$. By type equivalence inversion, $n = n'$. Therefore, C_{mem} contains the mapping $n \mapsto \tau'_{data}$, and therefore $(C_{mem}, C_{ptr}), \rightarrow = \Delta; \Psi; \Phi; \Theta; \{\}; \Lambda$ contains the mapping $n \mapsto \tau'_{data}$, so $n \in domain(\Psi_{Me}) = domain(\Psi_{spare}, \Psi)$. By the rule for $\Delta; \Psi_{Me}; \Phi_{Me}; \Theta; \{\}; \Lambda \vdash (M, e : \tau)$, we know that $n \in domain(M)$, so that:

$$(M, \text{load}[\text{fact}](n)) \longrightarrow (M, lin\langle\text{fact}, M(n)\rangle)$$

.

7.4 Termination

Define the *evaluation size* of an expression e to be $S(\Pi, e)$, where $\Pi = \{\ldots, x \mapsto n, \ldots\}$ puts an upper bound on the evaluation size of any expression substituted for $x \in domain(\Pi)$.

$$\begin{aligned}
S(\Pi, \lambda\alpha\!:\!\kappa.v) &= S(\Pi, v) \\
S(\Pi, \lambda\alpha\!:\!\kappa \Rightarrow e) &= S(\Pi, e) \\
S(\Pi, e\,\tau) &= 1 + S(\Pi, e) \\
S(\Pi, \text{pack}[\tau_1, e] \text{ as } \exists\alpha\!:\!\kappa.\tau_2) &= S(\Pi, e) \\
S(\Pi, \text{unpack}\,\alpha, x = e_1 \text{ in } e_2) &= 1 + S(\Pi, e_1) + S((\Pi, x \mapsto S(\Pi, e_1)), e_2) \\
S(\Pi, x) &= \Pi(x) \\
S(\Pi, \lambda x\!:\!\tau.e) &= 0 \\
S(\Pi, \lambda x\!:\!\tau \Rightarrow e) &= 0 \\
S(\Pi, e_1 \circ e_2) &= S(\Pi, e_1) + S(\Pi, e_2) \\
S(\Pi, \text{coerce}(e)) &= 1 + S(\Pi, e) \\
S(\Pi, \phi\langle e_1, \ldots, e_k\rangle) &= S(\Pi, e_1) + \ldots + S(\Pi, e_k) \\
S(\Pi, \text{let } \langle\overrightarrow{x}\rangle = e_1 \text{ in } e_2) &= 1 + S(\Pi, e_1) + S((\Pi, \overrightarrow{x \mapsto S(\Pi, e_1)}), e_2) \\
S(\Pi, \text{disj}^n_{\tau_1 \vee \tau_2}(e)) &= S(\Pi, e) \\
S(\Pi, \text{case } e_0 \text{ of } x_1.e_1 \text{ or } x_2.e_2) &= 1 + S(\Pi, e_0) + S((\Pi, x_1 \mapsto S(\Pi, e_0)), e_1) \\
&\quad + S((\Pi, x_2 \mapsto S(\Pi, e_0)), e_2) \\
S(\Pi, 0) &= 0 \\
S(\Pi, succ(e)) &= S(\Pi, e) \\
S(\Pi, \text{delay}(\kappa)) &= 1 \\
S(\Pi, \text{commit}[e_{delay}](e_{data} : (\alpha = \tau_\alpha \text{ in } \tau_{data}))) &= 1 + S(\Pi, e_{delay}) + S(\Pi, e_{data}) \\
S(\Pi, \text{fact}) &= 0
\end{aligned}$$

Let $S(\Pi, e_1\, e_2)$, $S(\Pi, \text{load}[e_{mem}](e_{ptr}))$, and $S(\Pi, \text{store}[e_{mem}](e_{ptr} \leftarrow e_{data}))$ be undefined.

Lemma 16. *(weakening) If $S(\Pi, e) = n$ and $\Pi' = \Pi, x_1 \mapsto n_1, \ldots, x_k \mapsto n_k$, then $S(\Pi, e) = S(\Pi', e)$.*

Proof. By induction on e.

Lemma 17. *(monotonicity) If $S(\Pi, e) = n$ and $\forall x \in domain(\Pi).(\Pi(x) \leq \Pi'(x))$, then $S(\Pi, e) \leq S(\Pi', e)$.*

Proof. By induction on e.

Lemma 18. *(substitution) If $S(\Pi', e) = n$, and $\Pi' = \Pi, x_1 \mapsto n_1, \ldots, x_k \mapsto n_k$ and $S(\Pi, v_1) \leq n_1, \ldots, S(\Pi, v_k) \leq n_k$, then $S(\Pi, [\overrightarrow{\alpha \leftarrow \tau}, \overrightarrow{x \leftarrow v}]e) \leq n$.*

Proof. By induction on e. Let $[\sigma] = [\overrightarrow{\alpha \leftarrow \tau}, \overrightarrow{x \leftarrow v}]$. Sample cases:

Case 1. x_j, where $\overrightarrow{x \leftarrow v} = \ldots, x_j \leftarrow v_j, \ldots$

$S(\Pi, [\sigma]x_j]) = S(\Pi, v_j) \leq n_j = \Pi'(x_j) = S(\Pi', x_j)$

Case 2. $\text{let } \langle \overrightarrow{y} \rangle = e_a \text{ in } e_b$:

Let $\Pi_y = \Pi, \overrightarrow{y \mapsto S(\Pi, [\sigma]e_a)}$ and $\Pi'_y = \Pi', \overrightarrow{y \mapsto S(\Pi, [\sigma]e_a)}$. Note that by weakening, $S(\Pi_y, v_j) = S(\Pi, v_j)$ for all $1 \leq j \leq k$, so that $S(\Pi_y, v_j) \leq n_j$; the induction on e_b below relies on this weakening.

$$
\begin{aligned}
& S(\Pi, [\sigma](\text{let } \langle \overrightarrow{y} \rangle = e_a \text{ in } e_b)) \\
&= S(\Pi, \text{let } \langle \overrightarrow{y} \rangle = [\sigma]e_a \text{ in } [\sigma]e_b)) \\
&= 1 + S(\Pi, [\sigma]e_a) + S(\Pi_y, [\sigma]e_b) \\
(\textit{by induction}) &\leq 1 + S(\Pi', e_a) + S(\Pi'_y, e_b) \\
&= 1 + S(\Pi', e_a) + S((\Pi', \overrightarrow{y \mapsto S(\Pi, [\sigma]e_a)}), e_b) \\
(\textit{by monotonicity}) &\leq 1 + S(\Pi', e_a) + S((\Pi', \overrightarrow{y \mapsto S(\Pi', e_a)}), e_b) \\
&= S(\Pi', \text{let } \langle \overrightarrow{y} \rangle = e_a \text{ in } e_b)
\end{aligned}
$$

Theorem 3. *(Termination) If $\Delta; \Psi; \Phi; \Theta; \{\}; \Rightarrow \vdash e : \tau$, then $S(\{\}, e)$ is defined, and the evaluation of (M, e) halts in no more than $S(\{\}, e)$ steps.*

Proof. Follows from the two lemmas below.

Lemma 19. *If $\Delta; \Psi; \Phi; \Theta; \Gamma; \Rightarrow \vdash e : \tau$ and $domain(\Gamma) = domain(\Pi)$, then $S(e)$ is defined.*

Proof. By induction on e's typing derivation.

Lemma 20. *If $S(\{\}, e) = n$ and $(M, e) \longrightarrow (M', e')$, then $S(\{\}, e') = n'$ where $n > n'$.*

Proof. By induction on the derivation of $(M, e) \longrightarrow (M', e')$. Sample cases:

Case 1. $(M, \text{case disj}^1_{\tau_1 \vee \tau_2}(v) \text{ of } x_1.e_1 \text{ or } x_2.e_2) \longrightarrow (M, [x_1 \leftarrow v]e_1)$. Use the substitution lemma:

$$\begin{aligned}
n &= S(\{\}, \text{case disj}^1_{\tau_1 \vee \tau_2}(v) \text{ of } x_1.e_1 \text{ or } x_2.e_2) \\
&= 1 + S(\{\}, \text{disj}^1_{\tau_1 \vee \tau_2}(v)) + S(\{x_1 \mapsto S(\{\}, \text{disj}^1_{\tau_1 \vee \tau_2}(v))\}, e_1) \\
&\qquad + S(\{x_2 \mapsto S(\{\}, \text{disj}^1_{\tau_1 \vee \tau_2}(v))\}, e_2) \\
&\geq 1 + S(\{\}, \text{disj}^1_{\tau_1 \vee \tau_2}(v)) + S([x_1 \leftarrow v]e_1) + S([x_2 \leftarrow v]e_2) \\
&> S(\{\}, [x_1 \leftarrow v]e_1)
\end{aligned}$$

Case 2. $(M, (v_1 \circ v_2)\, v_3) \longrightarrow (M, v_1\, (v_2\, v_3))$

Impossible: $S(\{\}, (v_1 \circ v_2)\, v_3)$ is not defined.

Case 3. $\frac{(M, e_1) \longrightarrow (M', e_1')}{(M, e_1 \circ e_2) \longrightarrow (M', e_1' \circ e_2)}$

$$\begin{aligned}
S(\{\}, e_1 \circ e_2) &= S(\{\}, e_1) + S(\{\}, e_2) \\
&> S(\{\}, e_1') + S(\{\}, e_2) \qquad (\textit{by induction}) \\
&= S(\{\}, e_1' \circ e_2)
\end{aligned}$$

8 Conclusions and Related Work

This paper has described an encoding of regions as linear tuples of memory capabilities, where coercions extract the capabilities from the regions, and delayed types extend a region's type as the region grows. Although the region encoding is elaborate, the primitives that make up the encoding are small, orthogonal, and general-purpose. For example, the original inspiration for delayed types was not for building regions or recursive types, but for encoding forwarding pointers, as described in section 6.2. As another example of the primitives' generality, the combination of linear tuples of capabilities and nonlinear coercions not only allows aliasing of data objects inside regions, but also allows aliasing of the regions themselves, as described in section 6.1. Although the aliasing of regions appears less expressive than in the capability calculus of Crary *et al*[3], it comes for free with our target language, rather than requiring special extensions to the target language.

The encodings of region types and region operations are efficient: pointers are only one word in size, and after compile-time coercions are stripped away, a get operation compiles down to a single load operation, and a set operation compiles down to a single store operation. The main inefficiency is the use of linked lists to track the pairs allocated to each region, so that the freergn(e) expression can find the pairs and return them to a global free list. It would be more efficient to

allocate pairs from larger blocks of memory (see [7] for extensions to the type system that remove this inefficiency).

Monnier[8] describes the construction of a typed garbage collector using a hybrid of regions, alias types, and a logic language. Regions are built into the type system, rather than being derived data types. The key challenge in any typed garbage collector is typing forwarding pointers; Monnier uses the alias-type aspect of the language to delay the introduction of forwarding pointer types until needed.

Linear TAL [2] shares our goal of building a memory management system from low-level linear memory primitives. Their approach uses copying to transform nonlinear data into linear data, eliminating all aliasing, which simplifies the type system but does not handle cyclic, mutable data.

Many logic languages have been used to prove properties about memory management. Foundational PCC[6] uses a logic language external to the programming language, and can use induction over typing judgments to prove a heap extension or region extension lemma. By contrast, our approach proves the well-typedness of region extension from within the program, using delayed types. Perhaps closer to our approach is separation logic, which was used to prove the correctness of a garbage collector for a heap with unlimited aliasing[1]. It would be interesting to see what analogue region extension or delayed types have in a separation logic treatment of low-level unlimited-aliasing allocation.

References

1. L. Birkedal, N. Torp-Smith, and J. Reynolds. Local reasoning about a copying garbage collector. In *Symposium on Principles of programming languages*, 2004.
2. James Cheney and Greg Morrisett. A linearly typed assembly language. Technical report, Department of Computer Science, Cornell University.
3. Karl Crary, David Walker, and Greg Morrisett. Typed memory management in a calculus of capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 262–275. ACM Press, 1999.
4. Karl Crary and Stephanie Weirich. Flexible type analysis. In *International Conference on Functional Programming*, pages 233–248, 1999.
5. Matthew Fluet and Daniel Wang. Implementation and performance evaluation of a safe runtime system in cyclone. In *Workshop on Semantics, Program Analysis, and Computing Environments For Memory Management*, 2004.
6. N. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof carrying-code. In *Proc. Seventeenth Annual IEEE Symposium on Logic In Computer Science (LICS'02)*, 2002.
7. Chris Hawblitzel, Edward Wei, Heng Huang, Eric Krupski, and Lea Wittie. Low-level linear memory management. In *Workshop on Semantics, Program Analysis, and Computing Environments For Memory Management*, 2004.
8. Stefan Monnier. Typed regions. In *Workshop on Semantics, Program Analysis, and Computing Environments For Memory Management*, 2004.
9. Stefan Monnier, Bratin Saha, and Zhong Shao. Principled scavenging. In *Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation*, pages 81–91. ACM Press, 2001.

10. Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
11. Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *In European Symposium on Programming*, 2000.
12. Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
13. P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, 1990.
14. David Walker and Kevin Watkins. On regions and linear types (extended abstract). In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 181–192. ACM Press, 2001.
15. Daniel C. Wang and Andrew W. Appel. Type-preserving garbage collectors. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 166–178. ACM Press, 2001.
16. Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN '98 conference on Programming language design and implementation*, pages 249–257. ACM Press, 1998.

Appendix A: Syntax, typing, and evaluation rules

This section contains the complete target language syntax and rules. It includes the $\lambda\alpha{:}\kappa \Rightarrow e$ extension and relaxed coerce(e) typing rule from section 6.1.1.

linearity: $\phi = non \mid lin$

integers: $n = 0 \mid succ(n)$

type variables: $\alpha = \alpha, \beta, \gamma, \delta, \epsilon, \rho, \chi, \omega, \ldots$

kinds: $\kappa = \overset{\phi}{n} \mid \mathbf{int}$

types

$$\begin{aligned}\tau = {} & \alpha \mid \forall\alpha{:}\kappa.\tau \mid \exists\alpha{:}\kappa.\tau \mid \tau_1 \to \tau_2 \mid \tau_1 \Rightarrow \tau_2 \\ & \mid \phi\langle\overrightarrow{\tau}\rangle \mid \tau_1 \vee \tau_2 \mid \tau_1 \mapsto \tau_2 \mid 0 \mid succ(\tau) \mid \mathrm{Int}(\tau) \mid \mathrm{Delay}(\tau)\end{aligned}$$

expressions

$$\begin{aligned}e = {} & \lambda\alpha{:}\kappa.v \mid \lambda\alpha{:}\kappa \Rightarrow e \mid e\,\tau \mid \mathrm{pack}[\tau_1, e]\,\mathrm{as}\,\exists\alpha{:}\kappa.\tau_2 \mid \mathrm{unpack}\,\alpha, x = e_1\,\mathrm{in}\,e_2 \\ & \mid x \mid \lambda x{:}\tau.e \mid \lambda x{:}\tau \Rightarrow e \mid e_1\,e_2 \mid e_1 \circ e_2 \mid \mathrm{coerce}(e) \\ & \mid \phi\langle\overrightarrow{e}\rangle \mid \mathrm{let}\,\langle\overrightarrow{x}\rangle = e_1\,\mathrm{in}\,e_2 \mid \mathrm{disj}^n_{\tau_1\vee\tau_2}(e) \mid \mathrm{case}\,e_0\,\mathrm{of}\,x_1.e_1\,\mathrm{or}\,x_2.e_2 \\ & \mid \mathrm{load}[e_{mem}](e_{ptr}) \mid \mathrm{store}[e_{mem}](e_{ptr} \leftarrow e_{data}) \\ & \mid 0 \mid succ(e) \mid \mathrm{delay}(\kappa) \mid \mathrm{commit}[e_{delay}](e_{data} : (\alpha = \tau_\alpha\,\mathrm{in}\,\tau_{data})) \mid \mathrm{fact}\end{aligned}$$

values

$$\begin{aligned}v = {} & \lambda\alpha{:}\kappa.v \mid \lambda\alpha{:}\kappa \Rightarrow e \mid \mathrm{pack}[\tau_1, v]\,\mathrm{as}\,\exists\alpha{:}\kappa.\tau_2 \mid \lambda x{:}\tau.e \mid \lambda x{:}\tau \Rightarrow e \mid v_1 \circ v_2 \\ & \mid \phi\langle\overrightarrow{v}\rangle \mid \mathrm{disj}^n_{\tau_1\vee\tau_2}(v) \mid 0 \mid succ(v) \mid \mathrm{fact}\end{aligned}$$

abbreviations

$$(\mathrm{let}\,x = e_1\,\mathrm{in}\,e_2) = (\mathrm{let}\,\langle x\rangle = lin\langle e_1\rangle\,\mathrm{in}\,e_2)$$

$$\mathrm{int} = \exists\alpha{:}\mathbf{int}.\mathrm{Int}(\alpha)$$

$$\tau_1 \Leftrightarrow \tau_2 = non\langle\tau_1 \Rightarrow \tau_2, \tau_2 \Rightarrow \tau_1\rangle$$

$$1 = succ(0),\ 2 = succ(1),\ \ldots$$

$$\tau + 0 = \tau,\ \tau + 1 = succ(\tau),\ \tau + 2 = succ(succ(\tau)), \ldots$$

$$e + 0 = e,\ e + 1 = succ(e),\ e + 2 = succ(succ(e)), \ldots$$

$$(\mathrm{unpack}\,\alpha_1, \alpha_2 \ldots, \alpha_n, x = e_1\,\mathrm{in}\,e_2) = (\mathrm{unpack}\,\alpha_1, x = e_1\,\mathrm{in}\,\mathrm{unpack}\,\alpha_2, \ldots, \alpha_n, x = x\,\mathrm{in}\,e_2)$$

$$(\mathrm{unpack}\,\alpha_1 \ldots, \alpha_n, \langle x_1, \ldots, x_m\rangle = e_1\,\mathrm{in}\,e_2) = (\mathrm{unpack}\,\alpha_1, \ldots, \alpha_n, x = e_1\,\mathrm{in}\,\mathrm{let}\,\langle x_1, \ldots, x_m\rangle = x\,\mathrm{in}\,e_2)$$

$$\begin{aligned}&(\mathrm{pack}[\tau_1, \tau_2, \ldots, \tau_n, e]\,\mathrm{as}\,\exists\alpha_1{:}\kappa_1.\exists\alpha_2{:}\kappa_2.\ldots\exists\alpha_n{:}\kappa_n.\tau) = \\ &\quad \mathrm{pack}[\tau_1, \mathrm{pack}[\tau_2, \ldots, \tau_n, e]\,\mathrm{as}\,\exists\alpha_2{:}\kappa_2.\ldots\exists\alpha_n{:}\kappa_n.[\alpha_1 \leftarrow \tau_1]\tau]\,\mathrm{as}\,\exists\alpha_1{:}\kappa_1.\exists\alpha_2{:}\kappa_2.\ldots\exists\alpha_n{:}\kappa_n.\tau\end{aligned}$$

environments

$$
\begin{aligned}
\textit{memory } M &= \{\ldots, n \mapsto v, \ldots\} \\
\textit{type variable env } \Delta &= \{\ldots, \alpha \mapsto \kappa, \ldots\} \\
\textit{memory type env } \Psi &= \{\ldots, n \mapsto \tau, \ldots\} \\
\textit{uncommitted env } \Phi &= \{\ldots, \alpha, \ldots\} \\
\textit{recursive type env } \Theta &= \{\ldots, \alpha \mapsto \tau, \ldots\} \\
\textit{variable env } \Gamma &= \{\ldots, x \mapsto \tau, \ldots\} \qquad \textit{where } \overset{\phi(\Delta)}{\Gamma} = \{x \mapsto \tau \in \Gamma | \Delta \vdash \tau :\overset{\phi}{n}\} \\
\textit{coercion env } \Lambda &= \rightarrow \mid \Rightarrow \\
\textit{combined env } C &= \Delta; \Psi; \Phi; \Theta; \Gamma; \Lambda \qquad \textit{where } \overset{non}{C} = \Delta; \{\}; \{\}; \Theta; \overset{non(\Delta)}{\Gamma}; \Lambda
\end{aligned}
$$

environment splitting and extension

$\Psi = \Psi_1, \Psi_2$ iff $\Psi = \Psi_1 \cup \Psi_2$ and $domain(\Psi_1) \cap domain(\Psi_2) = \{\}$.

$\Phi = \Phi_1, \Phi_2$ iff $\Phi = \Phi_1 \cup \Phi_2$ and $\Phi_1 \cap \Phi_2 = \{\}$.

$\Delta \vdash \Gamma = \Gamma_1, \Gamma_2$ iff

$\Gamma = \overset{non(\Delta)}{\Gamma} \cup \overset{lin(\Delta)}{\Gamma}$ and $\overset{non(\Delta)}{\Gamma} = \overset{non(\Delta)}{\Gamma_1} = \overset{non(\Delta)}{\Gamma_2}$ and $\overset{lin(\Delta)}{\Gamma} = \overset{lin(\Delta)}{\Gamma_1} \cup \overset{lin(\Delta)}{\Gamma_2}$

and $domain(\overset{lin(\Delta)}{\Gamma_1}) \cap domain(\overset{lin(\Delta)}{\Gamma_2}) = \{\}$.

$C = C_1, C_2$ iff $\Psi = \Psi_1, \Psi_2$ and $\Phi = \Phi_1, \Phi_2$ and $\Delta \vdash \Gamma = \Gamma_1, \Gamma_2$ and:

$C = (\Delta; \Psi; \Phi; \Theta; \Gamma; \Lambda)$

$C_1 = (\Delta; \Psi_1; \Phi_1; \Theta; \Gamma_1; \Lambda)$

$C_2 = (\Delta; \Psi_2; \Phi_2; \Theta; \Gamma_2; \Lambda)$

We sometimes write $C = (C_1, \ldots, C_n)$. This means that if $n \geq 1$, then there is some C' so that $C' = (C_1, \ldots, C_{n-1})$ and $C = C', C_n$, and if $n = 0$, then $C = \overset{non}{C}$.

If $n \notin domain(M)$, then $M, n \mapsto v = M \cup \{n \mapsto v\}$.

If $\alpha \notin domain(\Delta)$, then $\Delta, \alpha \mapsto \kappa = \Delta \cup \{\alpha \mapsto \kappa\}$.

If $n \notin domain(\Psi)$, then $\Psi, n \mapsto \tau = \Psi \cup \{n \mapsto \tau\}$.

If $\alpha \notin \Phi$, then $\Phi, \alpha = \Phi \cup \{\alpha\}$.

If $\alpha \notin domain(\Theta)$, then $\Theta, \alpha \mapsto \tau = \Theta \cup \{\alpha \mapsto \tau\}$.

If $x \notin domain(\Gamma)$, then $\Gamma, x \mapsto \tau = \Gamma \cup \{x \mapsto \tau\}$.

$(\Delta; \Psi; \Phi; \Theta; \Gamma; \Lambda), \alpha \mapsto \kappa = (\Delta, \alpha \mapsto \kappa; \Psi; \Phi; \Theta; \Gamma; \Lambda)$

$(\Delta; \Psi; \Phi; \Theta; \Gamma; \Lambda), n \mapsto \tau = (\Delta; \Psi, n \mapsto \tau; \Phi; \Theta; \Gamma; \Lambda)$

$(\Delta; \Psi; \Phi; \Theta; \Gamma; \Lambda), \alpha = (\Delta; \Psi; \Phi, \alpha; \Theta; \Gamma; \Lambda)$

$(\Delta; \Psi; \Phi; \Theta; \Gamma; \Lambda), \alpha \mapsto \tau = (\Delta; \Psi; \Phi; \Theta, \alpha \mapsto \tau; \Gamma; \Lambda)$

$(\Delta; \Psi; \Phi; \Theta; \Gamma; \Lambda), x \mapsto \tau = (\Delta; \Psi; \Phi; \Theta; \Gamma, x \mapsto \tau; \Lambda)$

$(\Delta; \Psi; \Phi; \Theta; \Gamma; \Lambda), \Lambda' = (\Delta; \Psi; \Phi; \Theta; \Gamma; \Lambda')$

judgments

$\Delta \vdash \tau : \kappa$

$\Theta \vdash \tau_1 \equiv \tau_2$

$\vdash C$

$C \vdash e : \tau$

$C \vdash (M, e : \tau)$

$(M, e) \rightarrow (M', e')$

We often write $(\Delta; \Psi; \Phi; \Theta; \Gamma; \Lambda) \vdash \tau : \kappa$ as an abbreviation for $\Delta \vdash \tau : \kappa$.

We often write $(\Delta; \Psi; \Phi; \Theta; \Gamma; \Lambda) \vdash \tau_1 \equiv \tau_2$ as an abbreviation for $\Theta \vdash \tau_1 \equiv \tau_2$.

kinding rules

$$\Delta, \alpha \mapsto \kappa \vdash \alpha : \kappa \qquad \frac{\Delta, \alpha \mapsto \kappa \vdash \tau :^{\phi} n}{\Delta \vdash \forall \alpha{:}\kappa.\tau :^{\phi} n} \qquad \frac{\Delta, \alpha \mapsto \kappa \vdash \tau :^{\phi} n}{\Delta \vdash \exists \alpha{:}\kappa.\tau :^{\phi} n}$$

$$\frac{\Delta \vdash \tau_1 :^{\phi_1} n_1 \qquad \Delta \vdash \tau_2 :^{\phi_2} n_2}{\Delta \vdash \tau_1 \to \tau_2 :^{non} 1} \qquad \frac{\Delta \vdash \tau_1 :^{\phi_1} n_1 \qquad \Delta \vdash \tau_2 :^{\phi_2} n_2}{\Delta \vdash \tau_1 \Rightarrow \tau_2 :^{non} 0}$$

$$\frac{\Delta \vdash \tau_1 :^{\phi_1} n_1, \ldots, \Delta \vdash \tau_k :^{\phi_k} n_k \qquad n = n_1 + \ldots + n_k}{\Delta \vdash \phi\langle \tau_1, \ldots, \tau_k \rangle :^{\phi} n} \qquad \frac{\Delta \vdash \tau_1 :^{\phi} 0 \qquad \Delta \vdash \tau_2 :^{\phi} 0}{\Delta \vdash \tau_1 \vee \tau_2 :^{\phi} 0}$$

$$\Delta \vdash 0 : \mathbf{int} \qquad \frac{\Delta \vdash \tau : \mathbf{int}}{\Delta \vdash succ(\tau) : \mathbf{int}} \qquad \frac{\Delta \vdash \tau : \mathbf{int}}{\Delta \vdash \mathrm{Int}(\tau) :^{non} 1}$$

$$\frac{\Delta \vdash \tau_1 : \mathbf{int} \qquad \Delta \vdash \tau_2 :^{non} 1}{\Delta \vdash \tau_1 \mapsto \tau_2 :^{lin} 0} \qquad \frac{\Delta \vdash \tau : \kappa}{\Delta \vdash \mathrm{Delay}(\kappa) :^{lin} 0}$$

type equivalence rules

Define $FV(\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_k \mapsto \tau_k\}) = \{\alpha_1, \ldots, \alpha_k\} \cup FV(\tau_1) \cup \ldots \cup FV(\tau_k)$

$$\Theta \vdash \tau \equiv \tau \qquad \frac{\Theta \vdash \tau_1 \equiv \tau_2}{\Theta \vdash \tau_2 \equiv \tau_1} \qquad \frac{\Theta \vdash \tau_1 \equiv \tau_2 \qquad \Theta \vdash \tau_2 \equiv \tau_3}{\Theta \vdash \tau_1 \equiv \tau_3}$$

$$\Theta, \alpha \mapsto \tau \vdash \alpha \equiv \tau \qquad \frac{\Theta \vdash \tau \equiv \tau' \qquad \alpha \notin FV(\Theta)}{\Theta \vdash \forall \alpha{:}\kappa.\tau \equiv \forall \alpha{:}\kappa.\tau'} \qquad \frac{\Theta \vdash \tau \equiv \tau' \qquad \alpha \notin FV(\Theta)}{\Theta \vdash \exists \alpha{:}\kappa.\tau \equiv \exists \alpha{:}\kappa.\tau'}$$

$$\frac{\Theta \vdash \tau_1 \equiv \tau_1' \qquad \Theta \vdash \tau_2 \equiv \tau_2'}{\Theta \vdash \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'} \qquad \frac{\Theta \vdash \tau_1 \equiv \tau_1' \qquad \Theta \vdash \tau_2 \equiv \tau_2'}{\Theta \vdash \tau_1 \Rightarrow \tau_2 \equiv \tau_1' \Rightarrow \tau_2'}$$

$$\frac{\Theta \vdash \tau_1 \equiv \tau_1' \qquad \ldots \qquad \Theta \vdash \tau_k \equiv \tau_k'}{\Theta \vdash \phi\langle \tau_1, \ldots, \tau_k \rangle \equiv \phi\langle \tau_1', \ldots, \tau_k' \rangle} \qquad \frac{\Theta \vdash \tau_1 \equiv \tau_1' \qquad \Theta \vdash \tau_2 \equiv \tau_2'}{\Theta \vdash \tau_1 \vee \tau_2 \equiv \tau_1' \vee \tau_2'}$$

$$\frac{\Theta \vdash \tau_1 \equiv \tau_1' \qquad \Theta \vdash \tau_2 \equiv \tau_2'}{\Theta \vdash \tau_1 \mapsto \tau_2 \equiv \tau_1' \mapsto \tau_2'} \qquad \frac{\Theta \vdash \tau \equiv \tau'}{\Theta \vdash succ(\tau) \equiv succ(\tau')}$$

$$\frac{\Theta \vdash \tau \equiv \tau'}{\Theta \vdash \mathrm{Int}(\tau) \equiv \mathrm{Int}(\tau')} \qquad \frac{\Theta \vdash \tau \equiv \tau'}{\Theta \vdash \mathrm{Delay}(\tau) \equiv \mathrm{Delay}(\tau')}$$

typing rules

$$\frac{\begin{array}{c}\forall n \mapsto \tau \in \Psi.(\Delta \vdash \tau :^{non} 1) \\ \Phi \cap domain(\Theta) = \{\} \qquad \Phi \subseteq domain(\Delta) \\ \forall \alpha \mapsto \tau \in \Theta.(\Delta \vdash \tau : \Delta(\alpha)) \\ \forall x_k \mapsto \tau_k \in \Gamma.(\Delta \vdash \tau_k :^{\phi_k} n_k)\end{array}}{\vdash \Delta; \Psi; \Phi; \Theta; \Gamma; \Lambda}$$

$$\frac{\begin{array}{c}\Psi = \Psi_{spare}, \Psi_e \qquad \Phi = \Phi_{spare}, \Phi_e \\ \vdash \Delta; \Psi; \Phi; \Theta; \Gamma; \Lambda \\ \forall n \in domain(\Psi).(\Delta; \{\}; \{\}; \Theta; \{\}; \to\!\vdash M(n) : \Psi(n)) \\ \Delta; \Psi_e; \Phi_e; \Theta; \Gamma; \Lambda \vdash e : \tau\end{array}}{\Delta; \Psi; \Phi; \Theta; \Gamma; \Lambda \vdash (M, e : \tau)}$$

$$\frac{C \vdash e : \tau \qquad C \vdash \tau \equiv \tau'}{C \vdash e : \tau'} \qquad \frac{C, \alpha \mapsto \kappa \vdash v : \tau}{C \vdash \lambda\alpha{:}\kappa.v : \forall\alpha{:}\kappa.\tau} \qquad \frac{\overset{non}{C}, \alpha \mapsto \kappa, \Rightarrow\vdash e : \tau}{\overset{non}{C} \vdash (\lambda\alpha{:}\kappa \Rightarrow e) : \forall\alpha{:}\kappa.\tau}$$

$$\frac{C \vdash e_1 : \forall\alpha{:}\kappa.\tau_1 \qquad C \vdash \tau_2 : \kappa}{C \vdash e_1\ \tau_2 : [\alpha \leftarrow \tau_2]\tau_1} \qquad \frac{C \vdash e : [\alpha \leftarrow \tau_1]\tau_2 \qquad C \vdash \tau_1 : \kappa \qquad C \vdash \exists\alpha{:}\kappa.\tau_2 :\overset{\phi}{n}}{C \vdash (\text{pack}[\tau_1, e]\,\text{as}\,\exists\alpha{:}\kappa.\tau_2) : (\exists\alpha{:}\kappa.\tau_2)}$$

$$\frac{C_1 \vdash e_1 : (\exists\alpha{:}\kappa.\tau_1) \qquad C_2, \alpha \mapsto \kappa, x \mapsto \tau_1 \vdash e_2 : \tau_2 \qquad C_2 \vdash \tau_2 :\overset{\phi}{n}}{C_1, C_2 \vdash (\text{unpack}\,\alpha, x = e_1\,\text{in}\,e_2) : \tau_2} \qquad \overset{non}{C}, x \mapsto \tau \vdash x : \tau$$

$$\frac{\Delta; \{\}; \{\}; \Theta; \{x \mapsto \tau_a\}; \rightarrow\vdash e : \tau_b \qquad \Delta \vdash \tau_a :\overset{\phi}{n}}{\Delta; \{\}; \{\}; \Theta; \overset{non}{\Gamma}; \Lambda \vdash (\lambda x{:}\tau_a.e) : \tau_a \rightarrow \tau_b} \qquad \frac{\overset{non}{C}, x \mapsto \tau_a, \Rightarrow\vdash e : \tau_b \qquad \overset{non}{C} \vdash \tau_b :\overset{\phi}{0}}{\overset{non}{C} \vdash (\lambda x{:}\tau_a \Rightarrow e) : \tau_a \Rightarrow \tau_b}$$

$$\frac{C_f, \rightarrow\vdash e_f : \tau_a \rightarrow \tau_b \qquad C_a, \rightarrow\vdash e_a : \tau_a}{(C_f, C_a), \rightarrow\vdash e_f\ e_a : \tau_b} \qquad \frac{C_f, \rightarrow\vdash e_f : \tau_a \Rightarrow \tau_b \qquad C_a, \rightarrow\vdash e_a : \tau_a}{(C_f, C_a), \rightarrow\vdash e_f\ e_a : \tau_b}$$

$$\frac{C_1 \vdash e_1 : \tau_b \Rightarrow \tau_c \qquad C_2 \vdash e_2 : \tau_a \Rightarrow \tau_b}{C_1, C_2 \vdash e_1 \circ e_2 : \tau_a \Rightarrow \tau_c} \qquad \frac{C, \Rightarrow\vdash e : \tau \qquad C \vdash \tau :\overset{\phi}{0}}{C \vdash \text{coerce}(e) : \tau}$$

$$\frac{C_1 \vdash e_1 : \tau_1 \qquad \ldots \qquad C_k \vdash e_k : \tau_k}{C_1, \ldots, C_k \vdash lin\langle e_1, \ldots, e_k\rangle : lin\langle \tau_1, \ldots, \tau_k\rangle} \qquad \frac{\begin{array}{cccc} C_1 \vdash e_1 : \tau_1 & \ldots & C_k \vdash e_k : \tau_k \\ C_1 \vdash \tau_1 :\overset{non}{n_1} & \ldots & C_k \vdash \tau_k :\overset{non}{n_k} \end{array}}{(C_1, \ldots, C_k) \vdash non\langle e_1, \ldots, e_k\rangle : non\langle \tau_1, \ldots, \tau_k\rangle}$$

$$\frac{C_a \vdash e_a : \phi\langle \tau_1, \ldots, \tau_k\rangle \qquad C_b, x_1 \mapsto \tau_1, \ldots, x_k \mapsto \tau_k \vdash e_b : \tau_b}{C_a, C_b \vdash \text{let}\,\langle x_1, \ldots, x_k\rangle = e_a\,\text{in}\,e_b : \tau_b} \qquad \frac{C \vdash e : \tau_n \qquad C \vdash \tau_1 \vee \tau_2 :\overset{\phi}{0}}{C \vdash \text{disj}^n_{\tau_1 \vee \tau_2}(e) : \tau_1 \vee \tau_2}$$

$$\frac{C_a, \Rightarrow\vdash e_0 : \tau_1 \vee \tau_2 \qquad C_b, x_1 \mapsto \tau_1, \Rightarrow\vdash e_1 : \tau_b \qquad C_b, x_2 \mapsto \tau_2, \Rightarrow\vdash e_2 : \tau_b}{(C_a, C_b), \Rightarrow (\text{case}\,e_0\,\text{of}\,x_1.e_1\,\text{or}\,x_2.e_2) : \tau_b}$$

$$\frac{C_{mem}, \rightarrow\vdash e_{mem} : \tau_{addr} \mapsto \tau_{data} \qquad C_{ptr}, \rightarrow\vdash e_{ptr} : \text{Int}(\tau_{addr})}{(C_{mem}, C_{ptr}), \rightarrow\vdash \text{load}[e_{mem}](e_{ptr}) : lin\langle(\tau_{addr} \mapsto \tau_{data}), \tau_{data}\rangle}$$

$$\frac{\begin{array}{cc} C_{mem}, \rightarrow\vdash e_{mem} : \tau_{addr} \mapsto \tau_{data} & C_{data} \vdash \tau'_{data} :\overset{non}{1} \\ C_{ptr}, \rightarrow\vdash e_{ptr} : \text{Int}(\tau_{addr}) & C_{data}, \rightarrow\vdash e_{data} : \tau'_{data} \end{array}}{(C_{mem}, C_{ptr}, C_{data}), \rightarrow\vdash \text{store}[e_{mem}](e_{ptr} \leftarrow e_{data}) : \tau_{addr} \mapsto \tau'_{data}}$$

$$\overset{non}{C} \vdash 0 : \text{Int}(0) \qquad \frac{C \vdash e : \text{Int}(\tau)}{C \vdash succ(e) : \text{Int}(succ(\tau))} \qquad \overset{non}{C} \vdash \text{delay}(\kappa) : \exists\alpha{:}\kappa.\text{Delay}(\alpha)$$

$$\frac{\begin{array}{cc} C_{delay} \vdash \tau_{delay} : \kappa \qquad C_{delay} \vdash \tau_\alpha : \kappa \qquad C_{delay} \vdash [\alpha \leftarrow \tau_\alpha]\tau_{data} :\overset{\phi}{n} & \\ C_{delay} \vdash e_{delay} : \text{Delay}(\tau_{delay}) & C_{data} \vdash e_{data} : [\alpha \leftarrow \tau_{delay}]\tau_{data} \end{array}}{C_{delay}, C_{data} \vdash \text{commit}[e_{delay}](e_{data} : (\alpha = \tau_\alpha\,\text{in}\,\tau_{data})) : [\alpha \leftarrow \tau_\alpha]\tau_{data}}$$

$$\overset{non}{C}, n \mapsto \tau \vdash \text{fact} : n \mapsto \tau$$

$$\Delta; \{\}; \{\alpha\}; \Theta; \overset{non(\Delta)}{\Gamma}; \Lambda \vdash \text{fact} : \text{Delay}(\alpha) \qquad where\ \Delta(\alpha) = \kappa$$

evaluation rules

$$\begin{aligned} E[e'] = {} & e'\,\tau \mid \text{pack}[\tau_1, e'] \text{ as } \exists\alpha{:}\kappa.\tau_2 \mid \text{unpack}\,\alpha, x = e' \text{ in } e_2 \\ \mid {} & e'\,e_2 \mid v_1\,e' \mid e' \circ e_2 \mid v_1 \circ e' \mid \text{coerce}(e') \\ \mid {} & \phi\langle v_1, \ldots, v_j, e', e_k, \ldots, e_n\rangle \mid \text{let } \langle\overrightarrow{x}\rangle = e' \text{ in } e_2 \mid \text{disj}^n_{\tau_1\vee\tau_2}(e') \mid \text{case } e' \text{ of } x_1.e_1 \text{ or } x_2.e_2 \\ \mid {} & \text{load}[e'](e_{ptr}) \mid \text{load}[v_{mem}](e') \\ \mid {} & \text{store}[e'](e_{ptr} \leftarrow e_{data}) \mid \text{store}[v_{mem}](e' \leftarrow e_{data}) \mid \text{store}[v_{mem}](v_{ptr} \leftarrow e') \\ \mid {} & succ(e') \mid \text{commit}[e'](e_{data} : (\alpha = \tau_\alpha \text{ in } \tau_{data})) \mid \text{commit}[v_{delay}](e' : (\alpha = \tau_\alpha \text{ in } \tau_{data})) \end{aligned}$$

$$\frac{e \longrightarrow e'}{(M, e) \longrightarrow (M, e')} \qquad \frac{(M, e) \longrightarrow (M', e')}{(M, E[e]) \longrightarrow (M', E[e'])}$$

$$(M, \text{load}[\text{fact}](n)) \longrightarrow (M, lin\langle \text{fact}, M(n)\rangle)$$

$$((M, n \mapsto v), \text{store}[\text{fact}](n \leftarrow v)) \longrightarrow ((M, n \mapsto v'), \text{fact})$$

$(\lambda\alpha{:}\kappa.v)\,\tau \longrightarrow [\alpha \leftarrow \tau]v$
$(\lambda\alpha{:}\kappa \Rightarrow e)\,\tau \longrightarrow \text{coerce}([\alpha \leftarrow \tau]e)$
$(\lambda x{:}\tau.e)\,v \longrightarrow [x \leftarrow v]e$
$(\lambda x{:}\tau \Rightarrow e)\,v \longrightarrow \text{coerce}([x \leftarrow v]e)$
$(v_1 \circ v_2)\,v_3 \longrightarrow v_1\,(v_2\,v_3)$
$\text{coerce}(v) \longrightarrow v$
$\text{unpack}\,\alpha, x = (\text{pack}[\tau, v] \text{ as } \exists\alpha'{:}\kappa.\tau') \text{ in } e \longrightarrow [\alpha \leftarrow \tau, x \leftarrow v]e$
$\text{let } \langle x_1, \ldots, x_k\rangle = \phi\langle v_1, \ldots, v_k\rangle \text{ in } e \longrightarrow [x_1 \leftarrow v_1, \ldots, x_k \leftarrow v_k]e$
$\text{case disj}^n_{\tau_1\vee\tau_2}(v) \text{ of } x_1.e_1 \text{ or } x_2.e_2 \longrightarrow [x_n \leftarrow v]e_n$
$\text{delay}(\kappa) \longrightarrow (\text{pack}[\beta, \text{fact}] \text{ as } \exists\alpha{:}\kappa.\text{Delay}(\alpha))$ where β is fresh
$\text{commit}[\text{fact}](v : (\alpha = \tau_\alpha \text{ in } \tau_{data})) \longrightarrow v$

Appendix B: Source language extensions

This section describes the source language extensions to the target language.

$$\begin{aligned} \textit{kinds} \quad & \kappa = \ldots \mid \mathbf{rgn} \\ \textit{types} \quad & \tau = \ldots \mid \text{Rgn}(\tau) \mid \langle\tau_1, \tau_2\rangle@\tau_{rgn} \\ \textit{expressions} \quad & e = \ldots \mid \text{rgn}(\alpha) \mid \ell \\ & \quad \mid \text{get}[e_{rgn}](e_{ptr}.n) \mid \text{set}[e_{rgn}](e_{ptr}.n \leftarrow e_{val}) \\ & \quad \mid \text{alloc}[e_R]\,\langle e_1, e_2\rangle \mid \text{newrgn} \mid \text{freergn}(e_R) \\ \textit{values} \quad & v = \ldots \mid \text{rgn}(\alpha) \mid \ell \\ \textit{heaps} \quad & H = \{\ldots, \ell \mapsto \langle v_1, v_2\rangle@\alpha, \ldots\} \\ \textit{heap type env} \quad & \psi = \{\ldots, \ell \mapsto \langle\tau_1, \tau_2\rangle@\alpha, \ldots\} \\ \textit{live rgn env} \quad & \Upsilon = \{\ldots, \alpha, \ldots\} \\ \textit{combined env} \quad & C = \Delta; \Psi; \Phi; \Theta; \psi; \Upsilon; \Gamma; \Lambda \qquad \textit{where } \overset{non}{C} = \Delta; \{\}; \{\}; \Theta; \psi; \{\}; \overset{non(\Delta)}{\Gamma}; \Lambda \end{aligned}$$

kinding rules

$$\frac{\Delta \vdash \tau : \mathbf{rgn}}{\Delta \vdash \text{Rgn}(\tau) : \overset{lin}{2}} \qquad \frac{\Delta \vdash \tau_1 : \phi_1 \qquad \Delta \vdash \tau_2 : \phi_2 \qquad \Delta \vdash \tau_{rgn} : \mathbf{rgn}}{\Delta \vdash \langle\tau_1, \tau_2\rangle@\tau_{rgn} : \overset{non}{1}}$$

type equivalence rules

$$\frac{\Theta \vdash \tau \equiv \tau'}{\Theta \vdash \mathrm{Rgn}(\tau) \equiv \mathrm{Rgn}(\tau')} \qquad \frac{\Theta \vdash \tau_1 \equiv \tau_1' \qquad \Theta \vdash \tau_2 \equiv \tau_2' \qquad \Theta \vdash \tau_3 \equiv \tau_3'}{\Theta \vdash \langle \tau_1, \tau_2 \rangle @ \tau_3 \equiv \langle \tau_1', \tau_2' \rangle @ \tau_3'}$$

typing rules

$$\frac{\begin{array}{c}\forall n \mapsto \tau \in \Psi.(\Delta \vdash \tau : \overset{non}{1}) \\ \Phi \cap domain(\Theta) = \{\} \qquad \Phi \subseteq domain(\Delta) \\ \forall \alpha \mapsto \tau \in \Theta.(\Delta \vdash \tau : \Delta(\alpha)) \\ \forall x_k \mapsto \tau_k \in \Gamma.(\Delta \vdash \tau_k : \overset{\phi_k}{n_k}) \\ \forall \ell \mapsto \langle \tau_1, \tau_2 \rangle @ \rho \in \psi.(\Delta \vdash \rho : \mathbf{rgn})\end{array}}{\vdash \Delta; \Psi; \Phi; \Theta; \psi; \Upsilon; \Gamma; \Lambda} \qquad \frac{\begin{array}{c}\Psi = \Psi_{spare}, \Psi_e \qquad \Phi = \Phi_{spare}, \Phi_e \qquad \Upsilon = \Upsilon_{spare}, \Upsilon_e \\ \vdash \Delta; \Psi; \Phi; \Theta; \psi; \Upsilon; \Gamma; \Lambda \\ \forall n \in domain(\Psi).(\Delta; \{\}; \{\}; \Theta; \psi; \{\}; \{\}; \rightarrow\vdash M(n) : \Psi(n)) \\ \Delta; \Psi_e; \Phi_e; \Theta; \psi; \Upsilon_e; \Gamma; \Lambda \vdash e : \tau \\ \Delta; \Theta; \psi; \Upsilon \vdash H\end{array}}{\Delta; \Psi; \Phi; \Theta; \psi; \Upsilon; \Gamma; \Lambda \vdash (H, M, e : \tau)}$$

$$\frac{\begin{array}{c}\psi(\ell) = \langle \tau_1, \tau_2 \rangle @ \rho \qquad \rho \in \Upsilon \\ \Delta; \{\}; \{\}; \Theta; \psi; \{\}; \{\}; \rightarrow\vdash v_1 : \tau_1 \\ \Delta; \{\}; \{\}; \Theta; \psi; \{\}; \{\}; \rightarrow\vdash v_2 : \tau_2 \\ \Delta \vdash \tau_1 : non \qquad \Delta \vdash \tau_2 : non\end{array}}{\Delta; \Theta; \psi; \Upsilon \vdash \ell \mapsto \langle v_1, v_2 \rangle @ \rho} \qquad \frac{\begin{array}{c}\forall \rho \in \Upsilon.(if\ \ell \mapsto \langle \tau_1, \tau_2 \rangle @ \rho \in \psi \\ \qquad then\ \ell \mapsto \langle v_1, v_2 \rangle @ \rho \in H) \\ \forall \ell \in domain(H).(\Delta; \Theta; \psi; \Upsilon \vdash \ell \mapsto H(\ell))\end{array}}{\Delta; \Theta; \psi; \Upsilon \vdash H}$$

$$\Delta, \alpha \mapsto \mathbf{rgn}; \{\}; \{\}\Theta; \psi; \{\alpha\}; \overset{non(\Delta)}{\Gamma}; \Lambda \vdash \mathrm{rgn}(\alpha) : \mathrm{Rgn}(\alpha) \qquad C, \ell \mapsto \tau \vdash \ell : \tau$$

$$\frac{C_{rgn}, \rightarrow\vdash e_{rgn} : \mathrm{Rgn}(\tau_{rgn}) \qquad C_{ptr}, \rightarrow\vdash e_{ptr} : \langle \tau_1, \tau_2 \rangle @ \tau_{rgn}}{(C_{rgn}, C_{ptr}), \rightarrow\vdash \mathrm{get}[e_{rgn}](e_{ptr}.n) : lin\langle \mathrm{Rgn}(\tau_{rgn}), \tau_n \rangle}$$

$$\frac{C_{rgn}, \rightarrow\vdash e_{rgn} : \mathrm{Rgn}(\tau_{rgn}) \qquad C_{ptr}, \rightarrow\vdash e_{ptr} : \langle \tau_1, \tau_2 \rangle @ \tau_{rgn} \qquad C_{val}, \rightarrow\vdash e_{val} : \tau_n}{(C_{rgn}, C_{ptr}, C_{val}), \rightarrow\vdash \mathrm{set}[e_{rgn}](e_{ptr}.n \leftarrow e_{val}) : \mathrm{Rgn}(\tau_{rgn})}$$

$$\frac{C_{rgn}, \rightarrow\vdash e_{rgn} : \mathrm{Rgn}(\tau_{rgn}) \qquad \begin{array}{c}C_1 \vdash \tau_1 : \overset{non}{1} \qquad C_2 \vdash \tau_2 : \overset{non}{1} \\ C_1, \rightarrow\vdash e_1 : \tau_1 \qquad C_2, \rightarrow\vdash e_2 : \tau_2\end{array}}{(C_{rgn}, C_1, C_2), \rightarrow\vdash \mathrm{alloc}[e_{rgn}]\,\langle e_1, e_2 \rangle : lin\langle \mathrm{Rgn}(\tau_{rgn}), \langle \tau_1, \tau_2 \rangle @ \tau_{rgn} \rangle}$$

$$\overset{non}{C}, \rightarrow\vdash \mathrm{newrgn} : \exists \alpha{:}\mathbf{rgn}.\mathrm{Rgn}(\alpha) \qquad \frac{C, \rightarrow\vdash e_{rgn} : \mathrm{Rgn}(\tau_{rgn})}{C, \rightarrow\vdash \mathrm{freergn}(e_{rgn}) : lin\langle\rangle}$$

evaluation rules

$$\begin{array}{rl} E[e'] = \ldots & \mid \mathrm{get}[e'](e_{ptr}.n) \mid \mathrm{get}[v_{rgn}](e'.n) \\ & \mid \mathrm{set}[e'](e_{ptr}.n \leftarrow e_{val}) \mid \mathrm{set}[v_{rgn}](e'.n \leftarrow e_{val}) \mid \mathrm{set}[v_{rgn}](v_{ptr}.n \leftarrow e') \\ & \mid \mathrm{alloc}[e']\,\langle e_1, e_2 \rangle \mid \mathrm{alloc}[v_R]\,\langle e', e_2 \rangle \mid \mathrm{alloc}[v_R]\,\langle v_1, e' \rangle \mid \mathrm{freergn}(e') \end{array}$$

$$\frac{e \longrightarrow e'}{(H, M, e) \longrightarrow (H, M, e')} \qquad \frac{(H, M, e) \longrightarrow (H', M', e')}{(H, M, E[e]) \longrightarrow (H', M', E[e'])}$$

$(H, M, \mathrm{load}[\mathrm{fact}](n)) \longrightarrow (H, M, lin\langle \mathrm{fact}, M(n) \rangle)$
$(H, (M, n \mapsto v), \mathrm{store}[\mathrm{fact}](n \leftarrow v')) \longrightarrow (H, (M, n \mapsto v'), \mathrm{fact})$
$((H, \ell \mapsto \langle v_1, v_2 \rangle @ \rho), M, \mathrm{get}[\mathrm{rgn}(\rho)](\ell.n)) \longrightarrow ((H, \ell \mapsto \langle v_1, v_2 \rangle @ \rho), M, lin\langle \mathrm{rgn}(\rho), v_n \rangle)$
$((H, \ell \mapsto \langle v_1, v_2 \rangle @ \rho), M, \mathrm{set}[\mathrm{rgn}(\rho)](\ell.1 \leftarrow v')) \longrightarrow ((H, \ell \mapsto \langle v', v_2 \rangle @ \rho), M, \mathrm{rgn}(\rho))$
$((H, \ell \mapsto \langle v_1, v_2 \rangle @ \rho), M, \mathrm{set}[\mathrm{rgn}(\rho)](\ell.2 \leftarrow v')) \longrightarrow ((H, \ell \mapsto \langle v_1, v' \rangle @ \rho), M, \mathrm{rgn}(\rho))$
$(H, M, \mathrm{alloc}[\mathrm{rgn}(\rho)]\,\langle v_1, v_2 \rangle) \longrightarrow ((H, \ell \mapsto \langle v_1, v_2 \rangle @ \rho), M, lin\langle \mathrm{rgn}(\alpha), \ell \rangle) \qquad where\ \ell\ is\ fresh$
$(H, M, \mathrm{newrgn}) \longrightarrow (H, M, \mathrm{pack}[\rho, \mathrm{rgn}(\rho)]\ \mathrm{as}\ \exists \alpha{:}\mathbf{rgn}.\mathrm{rgn}(\alpha)) \qquad where\ \rho\ is\ fresh$
$(H, M, \mathrm{freergn}(\mathrm{rgn}(\rho))) \longrightarrow (\{(\ell \mapsto \langle v_1, v_2 \rangle @ \alpha) \in H | \alpha \neq \rho\}, M, lin\langle\rangle)$

Appendix C: Source-to-target translation

This section describes a compile-time (heap $H = \{\}$) translation of a source language expression into a target language expression, and summarizes a proof of the well-typedness of the translation. The translation does not rely on the $\lambda\alpha{:}\kappa \Rightarrow e$ expression from section 6.1.1.

Suppose that $C = \Delta; \Psi; \Phi; \Theta; \{\}; \{\}; \Gamma; \Lambda$ and $C \vdash e : \tau$, where the variables α_{Alloc} and x_{free} appear nowhere in e and C. Define the translated expression $[\![e]\!]$ by annotating the derivation of $C \vdash e : \tau$ as $C \vdash e : \tau \rightsquigarrow [\![e]\!]$. For brevity's sake, we'll often omit the premises of the type judgments.

The translation $C, \rightarrow\vdash e : \tau \rightsquigarrow [\![e]\!]$ passes a free list from expression to expression, so that $[\![C, \rightarrow]\!], x_{free} \mapsto \tau_{free} \vdash [\![e]\!] : lin\langle \tau_{free}, [\![\tau]\!] \rangle$. The translation $C, \Rightarrow\vdash e : \tau \rightsquigarrow [\![e]\!]$ passes no free list, so that $[\![C, \Rightarrow]\!] \vdash [\![e]\!] : lin\langle [\![\tau]\!] \rangle$. To aid the passing of the free list, define the abbrevation:

$$(\text{letfree}\, x_1, \ldots, x_k = e_1, \ldots, e_k \,\text{in}\, e) = (\text{let}\, \langle x_{free}, x_1 \rangle = e_1 \,\text{in}\, \ldots \text{let}\, \langle x_{free}, x_k \rangle = e_k \,\text{in}\, e)$$

where $x_1, \ldots, x_k$ are fresh variables.

Here are the translations of the form $C, \rightarrow\vdash e : \tau \rightsquigarrow [\![e]\!]$:

$$\frac{C_f, \rightarrow\vdash e_f : \tau_a \Rightarrow \tau_b \rightsquigarrow [\![e_f]\!] \qquad C_a, \rightarrow\vdash e_a : \tau_a \rightsquigarrow [\![e_a]\!]}{(C_f, C_a), \rightarrow\vdash e_f\, e_a : \tau_b \rightsquigarrow \text{letfree}\, x_f, x_a = [\![e_f]\!], [\![e_a]\!] \,\text{in}\, lin\langle x_{free}, x_f\, x_a \rangle}$$

$$\frac{C_f, \rightarrow\vdash e_f : \tau_a \rightarrow \tau_b \rightsquigarrow [\![e_f]\!] \qquad C_a, \rightarrow\vdash e_a : \tau_a \rightsquigarrow [\![e_a]\!]}{(C_f, C_a), \rightarrow\vdash e_f\, e_a : \tau_b \rightsquigarrow \text{letfree}\, x_f, x_a = [\![e_f]\!], [\![e_a]\!] \,\text{in}\, x_f\, lin\langle x_{free}, x_a \rangle}$$

$$\frac{C, \alpha \mapsto \kappa, \rightarrow\vdash v : \tau \overset{value}{\rightsquigarrow} [\![value\, v]\!]}{C, \rightarrow\vdash \lambda\alpha{:}\kappa.v : \forall\alpha{:}\kappa.\tau \rightsquigarrow lin\langle x_{free}, \lambda\alpha{:}[\![\kappa]\!].[\![value\, v]\!] \rangle}$$

$C, \rightarrow\vdash (e\,\tau) : \tau' \rightsquigarrow \text{letfree}\, x = [\![e]\!] \,\text{in}\, lin\langle x_{free}, x\,[\![\tau]\!] \rangle$
$C, \rightarrow\vdash (\text{pack}[\tau_1, e] \,\text{as}\, \exists\alpha{:}\kappa.\tau_2) : \tau \rightsquigarrow \text{letfree}\, x = [\![e]\!] \,\text{in}\, lin\langle x_{free}, \text{pack}[[\![\tau_1]\!], x] \,\text{as}\, \exists\alpha{:}[\![\kappa]\!].[\![\tau_2]\!] \rangle$
$C, \rightarrow\vdash (\text{unpack}\,\alpha, x = e_1 \,\text{in}\, e_2) : \tau \rightsquigarrow \text{letfree}\, x_1 = [\![e_1]\!] \,\text{in}\, \text{unpack}\,\alpha, x = x_1 \,\text{in}\, [\![e_2]\!]$
$C, \rightarrow\vdash (x) : \tau \rightsquigarrow lin\langle x_{free}, x \rangle$
$C, \rightarrow\vdash (\lambda x{:}\tau.e) : \tau' \rightsquigarrow lin\langle x_{free}, \lambda x{:}lin\langle \tau_{free}, [\![\tau]\!] \rangle.\text{let}\, \langle x_{free}, x \rangle = x \,\text{in}\, [\![e]\!] \rangle$
$C, \rightarrow\vdash (\lambda x{:}\tau \Rightarrow e) : \tau' \rightsquigarrow lin\langle x_{free}, \lambda x{:}[\![\tau]\!] \Rightarrow [\![e]\!] \rangle$
$C, \rightarrow\vdash (\lambda\alpha{:}\kappa \Rightarrow e) : \tau' \rightsquigarrow lin\langle x_{free}, \lambda\alpha{:}[\![\kappa]\!] \Rightarrow [\![e]\!] \rangle$
$C, \rightarrow\vdash (e_1 \circ e_2) : \tau \rightsquigarrow \text{letfree}\, x_1, x_2 = [\![e_1]\!], [\![e_2]\!] \,\text{in}\, lin\langle x_{free}, x_1 \circ x_2 \rangle$
$C, \rightarrow\vdash (\text{coerce}(e)) : \tau \rightsquigarrow lin\langle x_{free}, \text{coerce}([\![e]\!]) \rangle$
$C, \rightarrow\vdash (\phi\langle e_1, \ldots, e_k \rangle) : \tau \rightsquigarrow \text{letfree}\, x_1, \ldots, x_n = [\![e_1]\!], \ldots, [\![e_k]\!] \,\text{in}\, lin\langle x_{free}, \phi\langle x_1, \ldots, x_k \rangle \rangle$
$C, \rightarrow\vdash (\text{let}\, \langle x_1, \ldots, x_k \rangle = e_1 \,\text{in}\, e_2) : \tau \rightsquigarrow \text{letfree}\, x = [\![e_1]\!] \,\text{in}\, \text{let}\, \langle x_1, \ldots, x_k \rangle = x \,\text{in}\, [\![e_2]\!]$
$C, \rightarrow\vdash (\text{disj}^n_{\tau_1 \vee \tau_2}(e)) : \tau \rightsquigarrow \text{letfree}\, x = [\![e]\!] \,\text{in}\, lin\langle x_{free}, \text{disj}^n_{[\![\tau_1 \vee \tau_2]\!]}(x) \rangle$
$C, \rightarrow\vdash (\text{load}[e_{mem}](e_{ptr})) : \tau \rightsquigarrow \text{letfree}\, x_{mem}, x_{ptr} = [\![e_{mem}]\!], [\![e_{ptr}]\!] \,\text{in}\, lin\langle x_{free}, \text{load}[x_{mem}](x_{ptr}) \rangle$
$C, \rightarrow\vdash (\text{store}[e_{mem}](e_{ptr} \leftarrow e_{data})) : \tau \rightsquigarrow$
$\qquad \text{letfree}\, x_{mem}, x_{ptr}, x_{data} = [\![e_{mem}]\!], [\![e_{ptr}]\!], [\![e_{data}]\!] \,\text{in}\, lin\langle x_{free}, \text{store}[x_{mem}](x_{ptr} \leftarrow x_{data}) \rangle$
$C, \rightarrow\vdash (0) : \tau \rightsquigarrow lin\langle x_{free}, 0 \rangle$
$C, \rightarrow\vdash (succ(e)) : \tau \rightsquigarrow \text{letfree}\, x = [\![e]\!] \,\text{in}\, lin\langle x_{free}, succ(x) \rangle$
$C, \rightarrow\vdash (\text{delay}(\kappa)) : \tau \rightsquigarrow lin\langle x_{free}, \text{delay}([\![\kappa]\!]) \rangle$
$C, \rightarrow\vdash (\text{commit}[e_{delay}](e_{data} : (\alpha = \tau_\alpha \,\text{in}\, \tau_{data}))) : \tau \rightsquigarrow$
$\qquad \text{letfree}\, x_{delay}, x_{data} = [\![e_{delay}]\!], [\![e_{data}]\!] \,\text{in}\, lin\langle x_{free}, \text{commit}[x_{delay}](x_{data} : (\alpha = [\![\tau_\alpha]\!] \,\text{in}\, [\![\tau_{data}]\!])) \rangle$
$C, \rightarrow\vdash (\text{fact}) : \tau \rightsquigarrow lin\langle x_{free}, \text{fact} \rangle$
$C, \rightarrow\vdash (\text{alloc}[e_R]\,\langle e_1, e_2 \rangle) : lin\langle \text{Rgn}(\tau_{rgn}), \langle \tau_1, \tau_2 \rangle @ \tau_{rgn} \rangle \rightsquigarrow$
$\qquad \text{letfree}\, x_R, x_1, x_2 = [\![e_R]\!], [\![e_1]\!], [\![e_2]\!] \,\text{in}\, alloc[[\![\tau_{rgn}]\!], [\![\tau_1]\!], [\![\tau_2]\!]](x_{free}, x_R, x_1, x_2)$
$C, \rightarrow\vdash (\text{newrgn}) : \tau \rightsquigarrow newrgn(x_{free})$

$$\frac{C, \rightarrow\vdash e_R : \text{Rgn}(\tau_R) \rightsquigarrow [\![e_R]\!]}{C, \rightarrow\vdash \text{freergn}(e_R) : lin\langle\rangle \rightsquigarrow \text{letfree}\, x_R = [\![e_R]\!] \,\text{in}\, lin\langle freergn[[\![\tau_R]\!]](x_{free}, x_R), lin\langle\rangle \rangle}$$

$$\frac{C_{rgn}, \rightarrow\vdash e_{rgn} : \mathrm{Rgn}(\tau_{rgn}) \rightsquigarrow [\![e_{rgn}]\!] \qquad C_{ptr}, \rightarrow\vdash e_{ptr} : \langle \tau_1, \tau_2 \rangle @ \tau_{rgn} \rightsquigarrow [\![e_{ptr}]\!]}{\begin{array}{c}(C_{rgn}, C_{ptr}), \rightarrow\vdash \mathrm{get}[e_{rgn}](e_{ptr}.n) : lin\langle \mathrm{Rgn}(\tau_{rgn}), \tau_n \rangle \rightsquigarrow \\ \mathrm{letfree}\, x_{rgn}, x_{ptr} = [\![e_{rgn}]\!], [\![e_{ptr}]\!] \,\mathrm{in}\, lin\langle x_{free}, get_n[[\![\tau_{rgn}]\!], [\![\tau_1]\!], [\![\tau_2]\!]](x_{rgn}, x_{ptr}) \rangle\end{array}}$$

$$\frac{\begin{array}{c}C_{rgn}, \rightarrow\vdash e_{rgn} : \mathrm{Rgn}(\tau_{rgn}) \rightsquigarrow [\![e_{rgn}]\!] \\ C_{ptr}, \rightarrow\vdash e_{ptr} : \langle \tau_1, \tau_2 \rangle @ \tau_{rgn} \rightsquigarrow [\![e_{ptr}]\!] \\ C_{val}, \rightarrow\vdash e_{val} : \tau_n \rightsquigarrow [\![e_{val}]\!]\end{array}}{\begin{array}{c}(C_{rgn}, C_{ptr}, C_{val}), \rightarrow\vdash \mathrm{set}[e_{rgn}](e_{ptr}.n \leftarrow e_{val}) : \mathrm{Rgn}(\tau_{rgn}) \rightsquigarrow \\ \mathrm{letfree}\, x_{rgn}, x_{ptr}, x_{val} = [\![e_{rgn}]\!], [\![e_{ptr}]\!], [\![e_{val}]\!] \,\mathrm{in}\, lin\langle x_{free}, set_n[[\![\tau_{rgn}]\!], [\![\tau_1]\!], [\![\tau_2]\!]](x_{rgn}, x_{ptr}, x_{val}) \rangle\end{array}}$$

Here are the translations of the form $C, \Rightarrow\vdash e : \tau \rightsquigarrow [\![e]\!]$:

$C, \Rightarrow\vdash (\mathrm{case}\, e_0 \,\mathrm{of}\, x_1.e_1 \,\mathrm{or}\, x_2.e_2) : \tau \rightsquigarrow \mathrm{case}\, [\![e_0]\!] \,\mathrm{of}\, x_1.[\![e_1]\!] \,\mathrm{or}\, x_2.[\![e_2]\!]$
$C, \Rightarrow\vdash (\lambda\alpha{:}\kappa.v) : \tau \rightsquigarrow \lambda\alpha{:}[\![\kappa]\!].[\![value\, v]\!]$
$C, \Rightarrow\vdash (e\, \tau) : \tau' \rightsquigarrow [\![e]\!]\, [\![\tau]\!]$
$C, \Rightarrow\vdash (\mathrm{pack}[\tau_1, e] \,\mathrm{as}\, \exists\alpha{:}\kappa.\tau_2) : \tau \rightsquigarrow \mathrm{pack}[[\![\tau_1]\!], [\![e]\!]] \,\mathrm{as}\, \exists\alpha{:}[\![\kappa]\!].[\![\tau_2]\!]$
$C, \Rightarrow\vdash (\mathrm{unpack}\, \alpha, x = e_1 \,\mathrm{in}\, e_2) : \tau \rightsquigarrow \mathrm{unpack}\, \alpha, x = [\![e_1]\!] \,\mathrm{in}\, [\![e_2]\!]$
$C, \Rightarrow\vdash (x) : \tau \rightsquigarrow x$
$C, \Rightarrow\vdash (\lambda x{:}\tau.e) : \tau' \rightsquigarrow \lambda x{:}lin\langle \tau_{free}, [\![\tau]\!] \rangle.\mathrm{let}\, \langle x_{free}, x \rangle = x \,\mathrm{in}\, [\![e]\!]$
$C, \Rightarrow\vdash (\lambda x{:}\tau \Rightarrow e) : \tau' \rightsquigarrow \lambda x{:}[\![\tau]\!] \Rightarrow [\![e]\!]$
$C, \Rightarrow\vdash (\lambda\alpha{:}\kappa \Rightarrow e) : \tau' \rightsquigarrow \lambda x{:}[\![\kappa]\!] \Rightarrow [\![e]\!]$
$C, \Rightarrow\vdash (e_1 \circ e_2) : \tau \rightsquigarrow [\![e_1]\!] \circ [\![e_2]\!]$
$C, \Rightarrow\vdash (\mathrm{coerce}(e)) : \tau \rightsquigarrow \mathrm{coerce}([\![e]\!])$
$C, \Rightarrow\vdash (\phi\langle e_1, \ldots, e_k \rangle) : \tau \rightsquigarrow \phi\langle [\![e_1]\!], \ldots, [\![e_k]\!] \rangle$
$C, \Rightarrow\vdash (\mathrm{let}\, \langle x_1, \ldots, x_k \rangle = e_1 \,\mathrm{in}\, e_2) : \tau \rightsquigarrow \mathrm{let}\, \langle x_1, \ldots, x_k \rangle = [\![e_1]\!] \,\mathrm{in}\, [\![e_2]\!]$
$C, \Rightarrow\vdash (\mathrm{disj}^n_{\tau_1 \vee \tau_2}(e)) : \tau \rightsquigarrow \mathrm{disj}^n_{[\![\tau_1 \vee \tau_2]\!]}([\![e]\!])$
$C, \Rightarrow\vdash (0) : \tau \rightsquigarrow 0$
$C, \Rightarrow\vdash (succ(e)) : \tau \rightsquigarrow succ([\![e]\!])$
$C, \Rightarrow\vdash (\mathrm{delay}(\kappa)) : \tau \rightsquigarrow \mathrm{delay}([\![\kappa]\!])$
$C, \Rightarrow\vdash (\mathrm{commit}[e_{delay}](e_{data} : (\alpha = \tau_\alpha \,\mathrm{in}\, \tau_{data}))) : \tau \rightsquigarrow \mathrm{commit}[[\![e_{delay}]\!]]([\![e_{data}]\!] : (\alpha = [\![\tau_\alpha]\!] \,\mathrm{in}\, [\![\tau_{data}]\!]))$
$C, \Rightarrow\vdash (\mathrm{fact}) : \tau \rightsquigarrow \mathrm{fact}$

The translation of the $\equiv$ rule is independent of the environment Λ:

$$\frac{C \vdash e : \tau \rightsquigarrow [\![e]\!] \qquad C \vdash \tau \equiv \tau'}{C \vdash e : \tau' \rightsquigarrow [\![e]\!]}$$

Finally, to help translate the v inside $(\lambda\alpha{:}\kappa.v)$, we define a special translation $C \vdash v : \tau \overset{value}{\rightsquigarrow} [\![value\, v]\!]$ for values:

$C \vdash (\lambda\alpha{:}\kappa.v) : \tau \overset{value}{\rightsquigarrow} \lambda\alpha{:}[\![\kappa]\!].[\![value\, v]\!]$
$C \vdash (\lambda\alpha{:}\kappa \Rightarrow e) : \tau \overset{value}{\rightsquigarrow} \lambda\alpha{:}[\![\kappa]\!] \Rightarrow [\![e]\!]$
$C \vdash (\mathrm{pack}[\tau_1, v] \,\mathrm{as}\, \exists\alpha{:}\kappa.\tau_2) : \tau \overset{value}{\rightsquigarrow} \mathrm{pack}[[\![\tau_1]\!], [\![value\, v]\!]] \,\mathrm{as}\, \exists\alpha{:}[\![\kappa]\!].[\![\tau_2]\!]$
$C \vdash (\lambda x{:}\tau.e) : \tau' \overset{value}{\rightsquigarrow} \lambda x{:}lin\langle \tau_{free}, [\![\tau]\!] \rangle.\mathrm{let}\, \langle x_{free}, x \rangle = x \,\mathrm{in}\, [\![e]\!]$
$C \vdash (\lambda x{:}\tau \Rightarrow e) : \tau' \overset{value}{\rightsquigarrow} \lambda x{:}[\![\tau]\!] \Rightarrow [\![e]\!]$
$C \vdash (v_1 \circ v_2) : \tau \overset{value}{\rightsquigarrow} [\![value\, v_1]\!] \circ [\![value\, v_2]\!]$
$C \vdash (\phi\langle v_1, \ldots, v_n \rangle) : \tau \overset{value}{\rightsquigarrow} \phi\langle [\![value\, v_1]\!], \ldots, [\![value\, v_n]\!] \rangle$
$C \vdash (\mathrm{disj}^n_{\tau_1 \vee \tau_2}(v)) : \tau \overset{value}{\rightsquigarrow} \mathrm{disj}^n_{[\![\tau_1 \vee \tau_2]\!]}([\![value\, v]\!])$
$C \vdash (0) : \tau \overset{value}{\rightsquigarrow} 0$
$C \vdash (succ(v)) : \tau \overset{value}{\rightsquigarrow} succ([\![value\, v]\!])$
$C \vdash (\mathrm{fact}) : \tau \overset{value}{\rightsquigarrow} \mathrm{fact}$

$$\frac{C \vdash v : \tau \overset{value}{\rightsquigarrow} [\![value\, v]\!] \qquad C \vdash \tau \equiv \tau'}{C \vdash v : \tau' \overset{value}{\rightsquigarrow} [\![value\, v]\!]}$$

Type translation

$[\![\alpha]\!] = \alpha$
$[\![\forall\alpha\!:\!\kappa.\tau]\!] = \forall\alpha\!:\![\![\kappa]\!].[\![\tau]\!]$
$[\![\exists\alpha\!:\!\kappa.\tau]\!] = \exists\alpha\!:\![\![\kappa]\!].[\![\tau]\!]$
$[\![\tau_1 \to \tau_2]\!] = lin\langle\tau_{free}, [\![\tau_1]\!]\rangle \to lin\langle\tau_{free}, [\![\tau_2]\!]\rangle$
$[\![\tau_1 \Rightarrow \tau_2]\!] = [\![\tau_1]\!] \Rightarrow [\![\tau_2]\!]$
$[\![\phi\langle\tau_1, \ldots, \tau_k\rangle]\!] = \phi\langle[\![\tau_1]\!], \ldots, [\![\tau_k]\!]\rangle$
$[\![\tau_1 \vee \tau_2]\!] = [\![\tau_1]\!] \vee [\![\tau_2]\!]$
$[\![\tau_1 \mapsto \tau_2]\!] = [\![\tau_1]\!] \mapsto [\![\tau_2]\!]$
$[\![0]\!] = 0$
$[\![succ(\tau)]\!] = succ([\![\tau]\!])$
$[\![\mathrm{Int}(\tau)]\!] = \mathrm{Int}([\![\tau]\!])$
$[\![\mathrm{Delay}(\tau)]\!] = \mathrm{Delay}([\![\tau]\!])$
$[\![\mathrm{Rgn}(\tau)]\!] = Rgn([\![\tau]\!])$
$[\![\langle\tau_1, \tau_2\rangle@\tau_{rgn}]\!] = Ptr([\![\tau_1]\!], [\![\tau_2]\!], [\![\tau_{rgn}]\!])$

$FreeBlock = \exists\gamma\!:\!\mathbf{int}.\exists\beta_0\!:\!\overset{non}{1}.\exists\beta_1\!:\!\overset{non}{1}.\exists\beta_2\!:\!\overset{non}{1}.\exists\beta_3\!:\!\overset{non}{1}.lin\langle\mathrm{Int}(\gamma), \gamma \mapsto \beta_0, \gamma+1 \mapsto \beta_1, \gamma+2 \mapsto \beta_2, \gamma+3 \mapsto \beta_3\rangle$

$Allocator = \exists\beta\!:\!\overset{non}{1}.\exists\gamma\!:\!\overset{lin}{0}.lin\langle\beta, \gamma, lin\langle\beta, \gamma\rangle \to lin\langle\alpha_{Alloc}, FreeBlock\rangle\rangle$

$\tau_{free} = lin\langle\alpha_{Alloc}, Allocator \to \alpha_{Alloc}, \alpha_{Alloc} \to Allocator\rangle$

$Rgn(\tau_R) = \exists\chi\!:\!\overset{lin}{0}.\exists\delta\!:\!\overset{non}{0}.\exists\epsilon\!:\!\overset{lin}{0}.\exists\omega\!:\!\overset{non}{1}.lin\langle\tau_R, \mathrm{Delay}(\chi), \tau_R \Leftrightarrow lin\langle\epsilon, \mathrm{Delay}(\delta) \vee \chi\rangle, \omega, lin\langle\tau_{free}, \epsilon, \omega\rangle \to \tau_{free}\rangle$

$PtrCap(\tau_R, \tau_M) = \exists\alpha\!:\!\overset{lin}{0}.\tau_R \Leftrightarrow lin\langle\alpha, \tau_M\rangle$

$Ptr(\tau_R, \tau_1, \tau_2) = \exists\gamma\!:\!\mathbf{int}.\exists\epsilon\!:\!\overset{lin}{0}.\exists\omega\!:\!\overset{non}{1}.non\langle\mathrm{Int}(\gamma), PtrCap(\tau_R, MTyp(\gamma, \tau_1, \tau_2, \epsilon, \omega))\rangle$

$MTyp(\tau_N, \tau_1, \tau_2, \tau_\epsilon, \tau_\omega) = lin\langle\tau_N \mapsto (lin\langle\tau_{free}, \tau_\epsilon, \tau_\omega\rangle \to \tau_{free}), (\tau_N+1) \mapsto \tau_\omega, (\tau_N+2) \mapsto \tau_1, (\tau_N+3) \mapsto \tau_2\rangle$

Kind translation

$[\![\overset{\phi}{n}]\!] = \overset{\phi}{n}$
$[\![\mathbf{int}]\!] = \mathbf{int}$
$[\![\mathbf{rgn}]\!] = \overset{lin}{0}$

Environment translation

$$[\![\Delta;\Psi;\Phi;\Theta;\{\};\{\};\Gamma;\Lambda]\!] = [\![\Delta]\!];[\![\Psi]\!];\Phi;[\![\Theta]\!];[\![\Gamma]\!];\Lambda$$
$$[\![\{\ldots, \alpha \mapsto \kappa, \ldots\}]\!] = \{\ldots, \alpha \mapsto [\![\kappa]\!], \ldots\}, \alpha_{Alloc} \mapsto \overset{lin}{2}$$
$$[\![\{\ldots, n \mapsto \tau, \ldots\}]\!] = \{\ldots, n \mapsto [\![\tau]\!], \ldots\}$$
$$[\![\{\ldots, \alpha \mapsto \tau, \ldots\}]\!] = \{\ldots, \alpha \mapsto [\![\tau]\!], \ldots\}$$
$$[\![\{\ldots, x \mapsto \tau, \ldots\}]\!] = \{\ldots, x \mapsto [\![\tau]\!], \ldots\}$$

nilFree

$\overset{non}{C} \vdash nilFree : \exists \alpha_{Alloc} \overset{lin}{:} \overset{lin}{2} .lin\langle \alpha_{Alloc}, Allocator \to \alpha_{Alloc}, \alpha_{Alloc} \to Allocator \rangle$

$nilFree =$

$\quad \text{unpack}\, \alpha_{Alloc}, x_{Delay} = \text{Delay}(\overset{lin}{2})\, \text{in}$

$\quad \text{let}\, x_{pair} = non\langle \lambda x{:}\alpha_{Alloc}.x, \lambda x{:}\alpha_{Alloc}.x \rangle\, \text{in}$

$\quad \text{let}\, \langle x_{roll}, x_{unroll} \rangle = \text{commit}[x_{Delay}](x_{pair} : (\alpha' = Allocator\, \text{in}\, non\langle \alpha' \to \alpha_{Alloc}, \alpha_{Alloc} \to \alpha' \rangle))\, \text{in}$

$\quad \text{let}\, x_f = \lambda x{:}lin\langle \text{Int}(0), lin\langle\rangle \rangle .halt[\overset{lin}{3}]\, lin\langle \alpha_{Alloc}, FreeBlock \rangle\, non\langle\rangle\, \text{in}$

$\quad \text{let}\, x_A = \text{pack}[\text{Int}(0), lin\langle\rangle, lin\langle 0, lin\langle\rangle, x_f \rangle]\, \text{as}\, Allocator\, \text{in}$

$\quad \text{pack}[\alpha_{Alloc}, lin\langle x_{roll}\, x_A, x_{roll}, x_{unroll} \rangle]\, \text{as}\, \exists \alpha_{Alloc} \overset{lin}{:} \overset{lin}{2} .\tau_{free}$

consFree

$C' \vdash consFree[\tau_N, \tau_0, \tau_1, \tau_2, \tau_3](x_{free}, x_N, x_0, x_1, x_2, x_3) : \tau_{free}$

$where\, C' = \overset{non}{C}, \to, \alpha_{Alloc} \mapsto \overset{lin}{2}, x_{free} \mapsto \tau_{free}, x_N \mapsto \text{Int}(\tau_N), x_0 \mapsto (\tau_N + 0 \mapsto \tau_0), \ldots, x_3 \mapsto (\tau_N + 3 \mapsto \tau_3)$

$consFree[\tau_N, \tau_0, \tau_1, \tau_2, \tau_3](x_{free}, x_N, x_0, x_1, x_2, x_3) =$

$\quad \text{let}\, \langle x_A, x_{roll}, x_{unroll} \rangle = x_{free}\, \text{in}$

$\quad \text{unpack}\, \beta, \gamma, \langle x_\beta, x_\gamma, x_f \rangle = x_{unroll}\, x_A\, \text{in}$

$\quad \text{let}\, x_0 = \text{store}[x_0](x_N \leftarrow x_f)\, \text{in}$

$\quad \text{let}\, x_1 = \text{store}[x_1](x_N + 1 \leftarrow x_\beta)\, \text{in}$

$\quad \text{let}\, x_2 = \text{store}[x_2](x_N + 2 \leftarrow x_{roll})\, \text{in}$

$\quad \text{let}\, x_A = lin\langle x_N, lin\langle x_\gamma, x_0, x_1, x_2, x_3 \rangle, consFun \rangle\, \text{in}$

$\quad \text{let}\, x_A = \text{pack}[\text{Int}(\tau_N), \tau_{cons-\gamma}, x_A]\, \text{as}\, Allocator\, \text{in}$

$\quad lin\langle x_{roll}\, x_A, x_{roll}, x_{unroll} \rangle$

$\tau_{cons-\gamma} = lin\langle \gamma, \tau_N \mapsto \tau_{cons-0}, \tau_N + 1 \mapsto \beta, \tau_N + 2 \mapsto \tau_{cons-2}, \tau_N + 3 \mapsto \tau_3 \rangle$

$\tau_{cons-0} = lin\langle \beta, \gamma \rangle \to lin\langle \alpha_{Alloc}, FreeBlock \rangle$

$\tau_{cons-2} = Allocator \to \alpha_{Alloc}$

$consFun = \lambda x{:}lin\langle \text{Int}(\tau_N), \tau_{cons-\gamma} \rangle.$

$\quad \text{let}\, \langle x_N, x' \rangle = x\, \text{in}$

$\quad \text{let}\, \langle x_\gamma, x_0, x_1, x_2, x_3 \rangle = x'\, \text{in}$

$\quad \text{let}\, \langle x_0, x_f \rangle = \text{load}[x_0](x_N)\, \text{in}$

$\quad \text{let}\, \langle x_1, x_\beta \rangle = \text{load}[x_1](x_N + 1)\, \text{in}$

$\quad \text{let}\, x_{block} = lin\langle x_N, x_0, x_1, x_2, x_3 \rangle\, \text{in}$

$\quad \text{let}\, x_{block} = \text{pack}[\tau_N, \tau_{cons-0}, \beta, \tau_{cons-2}, \tau_3, x_{block}]\, \text{as}\, FreeBlock\, \text{in}$

$\quad \text{let}\, x_A = \text{pack}[\beta, \gamma, lin\langle x_\beta, x_\gamma, x_f \rangle]\, \text{as}\, Allocator\, \text{in}$

$\quad lin\langle x_{roll}\, x_A, x_{block} \rangle$

halt If the program runs out of memory, it goes into an infinite loop. The target language lacks direct support for infinite loops, but the following closure uses a recursive type α_{inf} to encode an infinite loop:

$\{\alpha_{inf} \mapsto \overset{non}{2}\} \vdash Inf[\overset{\phi}{n}] : \overset{non}{2}$

$\overset{non}{C}, \to, \alpha_{inf} \mapsto \overset{non}{2} \vdash inf[\overset{\phi}{n}] : (\forall \gamma\, \overset{\phi}{n} .non\langle \alpha_{inf} \to Inf[\overset{\phi}{n}], \alpha_{inf} \rangle \to \gamma)$

$\overset{non}{C}, \to \vdash halt[\overset{\phi}{n}] : \forall \gamma{:}[\overset{\phi}{n}].non\langle\rangle \to \gamma$

$Inf[\overset{\phi}{n}] = \exists \beta{:}\overset{non}{1} .non\langle \beta, \forall \gamma\, \overset{\phi}{n} .non\langle \beta, \alpha_{inf} \rangle \to \gamma \rangle$

$inf[\overset{\phi}{n}] = \lambda \gamma\, \overset{\phi}{n} .\lambda x{:}non\langle \alpha_{inf} \to Inf[\overset{\phi}{n}], \alpha_{inf} \rangle.$

$\quad \text{let}\, \langle x_{unroll}, x_\alpha \rangle = x\, \text{in}\, \text{unpack}\, \beta, \langle z_\beta, z_f \rangle = x_{unroll}\, x_\alpha\, \text{in}\, (z_f\, \gamma)\, non\langle z_\beta, x_\alpha \rangle$

$halt[\overset{\phi}{n}] = \lambda \gamma\, \overset{\phi}{n} .\lambda x_{empty}{:}non\langle\rangle.$

$\quad \text{unpack}\, \alpha_{inf}, x_{delay} = \text{delay}(\overset{non}{1})\, \text{in}$

$\quad \text{let}\, x_{pair} = non\langle (\lambda x{:}\alpha_{inf}.x), (\lambda x{:}\alpha_{inf}.x) \rangle\, \text{in}$

$\quad \text{let}\, \langle x_{roll}, x_{unroll} \rangle = \text{commit}[x_{delay}](x_{pair} : (\alpha = Inf\, \text{in}\, non\langle \alpha \to \alpha_{inf}, \alpha_{inf} \to \alpha \rangle))\, \text{in}$

$\quad \text{let}\, x_{inf} = \text{pack}[\alpha_{inf} \to Inf, non\langle x_{unroll}, inf[\overset{\phi}{n}] \rangle]\, \text{as}\, Inf[\overset{\phi}{n}]\, \text{in}\, (inf[\overset{\phi}{n}]\, \gamma)\, non\langle x_{unroll}, (x_{roll}\, x_{inf}) \rangle$

get

$$\overset{non}{C}, \rightarrow, \alpha_{Alloc} \mapsto \overset{lin}{2}, x_R \mapsto Rgn(\tau_R), x_{ptr} \mapsto Ptr(\tau_R, \tau_1, \tau_2) \vdash get_n[\tau_R, \tau_1, \tau_2](x_R, x_{ptr}) : lin\langle Rgn(\tau_R), \tau_n \rangle$$

$get_n[\tau_R, \tau_1, \tau_2](x_R, x_{ptr}) =$

$\quad \text{unpack}\, \chi, \delta, \epsilon, \omega, \langle x_r, x_{Delay-\chi}, x_f, x_\omega, x_{done} \rangle = x_R \,\text{in}$

$\quad \text{unpack}\, \gamma, \epsilon', \omega', \langle x_N, x_{cap} \rangle = x_{ptr} \,\text{in}$

$\quad \text{unpack}\, \alpha, \langle x_{to}, x_{from} \rangle = x_{cap} \,\text{in}$

$\quad \text{let}\, \langle x_\alpha, x_{mem} \rangle = x_{to}\, x_r \,\text{in}$

$\quad \text{let}\, \langle x_{mem-0}, x_{mem-1}, x_{mem-2}, x_{mem-3} \rangle = x_{mem} \,\text{in}$

$\quad \text{let}\, \langle x_{mem-(n+1)}, x_n \rangle = \text{load}[x_{mem-(n+1)}](x_N + (n+1)) \,\text{in}$

$\quad \text{let}\, x_r = x_{from}\, lin\langle x_\alpha, lin\langle x_{mem-0}, x_{mem-1}, x_{mem-2}, x_{mem-3} \rangle \rangle \,\text{in}$

$\quad \text{let}\, x_R = lin\langle x_r, x_{Delay-\chi}, x_f, x_\omega, x_{done} \rangle \,\text{in}$

$\quad \text{let}\, x_R = \text{pack}[\chi, \delta, \epsilon, \omega, x_R] \,\text{as}\, Rgn(\tau_R) \,\text{in}$

$\quad lin\langle x_R, x_n \rangle$

set

$$\overset{non}{C}, \rightarrow, \alpha_{Alloc} \mapsto \overset{lin}{2}, x_R \mapsto Rgn(\tau_R), x_{ptr} \mapsto Ptr(\tau_R, \tau_1, \tau_2), x_{val} \mapsto \tau_n \vdash set_n[\tau_R, \tau_1, \tau_2](x_R, x_{ptr}, x_{val}) : Rgn(\tau_R)$$

$set_n[\tau_R, \tau_1, \tau_2](x_R, x_{ptr}, x_{val}) =$

$\quad \text{unpack}\, \chi, \delta, \epsilon, \omega, \langle x_r, x_{Delay-\chi}, x_f, x_\omega, x_{done} \rangle = x_R \,\text{in}$

$\quad \text{unpack}\, \gamma, \epsilon', \omega', \langle x_N, x_{cap} \rangle = x_{ptr} \,\text{in}$

$\quad \text{unpack}\, \alpha, \langle x_{to}, x_{from} \rangle = x_{cap} \,\text{in}$

$\quad \text{let}\, \langle x_\alpha, x_{mem} \rangle = x_{to}\, x_r \,\text{in}$

$\quad \text{let}\, \langle x_{mem-0}, x_{mem-1}, x_{mem-2}, x_{mem-3} \rangle = x_{mem} \,\text{in}$

$\quad \text{let}\, \langle x_{mem-(n+1)} \rangle = \text{store}[x_{mem-(n+1)}](x_N + (n+1) \leftarrow x_{val}) \,\text{in}$

$\quad \text{let}\, x_r = x_{from}\, lin\langle x_\alpha, lin\langle x_{mem-0}, x_{mem-1}, x_{mem-2}, x_{mem-3} \rangle \rangle \,\text{in}$

$\quad \text{let}\, x_R = lin\langle x_r, x_{Delay-\chi}, x_f, x_\omega, x_{done} \rangle \,\text{in}$

$\quad \text{let}\, x_R = \text{pack}[\chi, \delta, \epsilon, \omega, x_R] \,\text{as}\, Rgn(\tau_R) \,\text{in}$

$\quad x_R$

newrgn

$$\overset{non}{C}, \rightarrow, \alpha_{Alloc} \mapsto \overset{lin}{2}, x_{free} \mapsto \tau_{free} \vdash newrgn(x_{free}) : \exists \rho{:}\overset{lin}{0}\ .Rgn(\rho)$$

$newrgn(x_{free}) =$

$\quad \text{unpack}\, \chi, x_{Delay-\chi} = \text{delay}(\overset{lin}{0}) \,\text{in}$

$\quad \text{unpack}\, \delta, x_{Delay-\delta} = \text{delay}(\overset{non}{0}) \,\text{in}$

$\quad \text{let}\, x_R = lin\langle \text{disj}^1_{(\text{Delay}(\delta) \vee \chi)}(x_{Delay-\delta}), x_{Delay-\chi}, non\langle newRgnTo, newRgnFrom \rangle, 0, newRgnDone \rangle \,\text{in}$

$\quad \text{pack}[\text{Delay}(\delta) \vee \chi, \chi, \delta, lin\langle\rangle, \text{Int}(0), x_R] \,\text{as}\, \exists \rho{:}\overset{lin}{0}\ .Rgn(\rho)$

$newRgnTo = \lambda x{:}\text{Delay}(\delta) \vee \chi \Rightarrow lin\langle lin\langle\rangle, x \rangle$

$newRgnFrom = \lambda x{:}lin\langle lin\langle\rangle, \text{Delay}(\delta) \vee \chi \rangle \Rightarrow$

$\quad \text{let}\, \langle y, z \rangle = x \,\text{in let}\, \langle\rangle = y \,\text{in}\, z$

$newRgnDone = \lambda x{:}lin\langle \tau_{free}, lin\langle\rangle, \text{Int}(0) \rangle.$

$\quad \text{let}\, \langle x_{free}, x_\epsilon, x_\omega \rangle = x \,\text{in let}\, \langle\rangle = x_\epsilon \,\text{in}\, x_{free}$

freergn

$\overset{non}{C}, \to, \alpha_{Alloc} \mapsto \overset{lin}{2}, x_{free} \mapsto \tau_{free}, x_R \mapsto Rgn(\tau_R) \vdash freergn[\tau_R](x_{free}, x_R) : \tau_{free}$

$freergn[\tau_R](x_{free}, x_R) =$

$\quad \text{unpack}\, \chi, \delta, \epsilon, \omega, \langle x_r, x_{Delay-\chi}, x_f, x_\omega, x_{done} \rangle = x_R \,\text{in}$

$\quad \text{let}\, \langle x_{f-to}, x_{f-from} \rangle = x_f \,\text{in}$

$\quad \text{let}\, \langle x_\epsilon, x_u \rangle = x_{f-to}\, x_r \,\text{in}$

$\quad \text{let}\, \langle\rangle = \text{coerce}(freergnCo) \,\text{in}$

$\quad x_{done}\, lin\langle x_{free}, x_\epsilon, x_\omega \rangle$

$freergnCo = \text{case}\, x_u \,\text{of}\, x_{Delay-\delta}.freergnCo1 \,\text{or}\, x_\chi.freergnCo2$

$freergnCo1 =$

$\quad \text{let}\, \langle\rangle = \text{commit}[x_{Delay-\delta}](non\langle\rangle : (\delta = non\langle\rangle \,\text{in}\, non\langle\rangle)) \,\text{in}$

$\quad \text{commit}[x_{Delay-\chi}](lin\langle\rangle : (\delta = lin\langle\rangle \,\text{in}\, lin\langle\rangle))$

$freergnCo2 = \text{commit}[x_{Delay-\chi}](x_\chi : (\chi = lin\langle\rangle \,\text{in}\, \chi))$

alloc

$C' \vdash alloc[\tau_R, \tau_1, \tau_2](x_{free}, x_R, x_1, x_2) : lin\langle \tau_{free}, lin\langle Rgn(\tau_R), Ptr(\tau_R, \tau_1, \tau_2) \rangle \rangle$

$where\, C' = \overset{non}{C}, \to, \alpha_{Alloc} \mapsto \overset{lin}{2}, x_{free} \mapsto \tau_{free}, x_R \mapsto Rgn(\tau_R), x_1 \mapsto \tau_1, x_2 \mapsto \tau_2$

$alloc[\tau_R, \tau_1, \tau_2](x_{free}, x_R, x_1, x_2) =$

$\quad allocBlock$

$\quad initBlock$

$\quad addBlockToRegion$

$allocBlock =$

$\quad \text{let}\, \langle x_A, x_{roll}, x_{unroll} \rangle = x_{free} \,\text{in}$

$\quad \text{unpack}\, \beta_A, \gamma_A, \langle x_{A\beta}, x_{A\gamma}, x_{Afun} \rangle = x_{unroll}\, x_A \,\text{in}$

$\quad \text{let}\, \langle x_A, x_{FreeBlock} \rangle = x_{Afun}\, lin\langle x_{A\beta}, x_{A\gamma} \rangle \,\text{in}$

$initBlock =$

$\quad \text{unpack}\, \alpha_N, \beta_0, \beta_1, \beta_2, \beta_3, \langle x_N, x_{mem-0}, x_{mem-1}, x_{mem-2}, x_{mem-3} \rangle = x_{FreeBlock} \,\text{in}$

$\quad \text{let}\, x_{mem-2} = \text{store}[x_{mem-2}](x_N + 2 \leftarrow x_1) \,\text{in}$

$\quad \text{let}\, x_{mem-3} = \text{store}[x_{mem-3}](x_N + 3 \leftarrow x_2) \,\text{in}$

$addBlockToRegion =$

$\quad \text{unpack}\, \chi, \delta, \epsilon, \omega, \langle x_r, x_{Delay-\chi}, x_f, x_\omega, x_{done} \rangle = x_R \,\text{in}$

$\quad \text{let}\, \langle x_{f-to}, x_{f-from} \rangle = x_f \,\text{in}$

$\quad \text{let}\, \langle x_\epsilon, x_u \rangle = x_{f-to}\, x_r \,\text{in}$

$\quad \text{let}\, \langle x_{Delay-\chi}, x_{Delay-\delta} \rangle = \text{coerce}(getDelayDelta) \,\text{in}$

$\quad \text{unpack}\, \chi', x'_{Delay-\chi} = \text{delay}(\overset{lin}{0}) \,\text{in}$

$\quad \text{unpack}\, \delta', x'_{Delay-\delta} = \text{delay}(\overset{non}{0}) \,\text{in}$

$\quad \text{let}\, x_f = \text{commit}[x_{Delay-\delta}](x_f : (\delta = non\langle\rangle \,\text{in}\, \tau_R \Leftrightarrow lin\langle \epsilon, \text{Delay}(\delta) \vee \chi \rangle)) \,\text{in}$

$\quad \text{let}\, x_f = \text{commit}[x_{Delay-\chi}](x_f : (\chi = lin\langle MTyp(\alpha_N, \tau_1, \tau_2, \epsilon, \omega), \text{Delay}(\delta') \vee \chi' \rangle \,\text{in}\, \tau_R \Leftrightarrow lin\langle \epsilon, \text{Delay}(non\langle\rangle) \vee \chi \rangle)) \,\text{in}$

$\quad \text{let}\, \langle x_{f-to}, x_{f-from} \rangle = x_f \,\text{in}$

$\quad \text{let}\, x'_f = non\langle to_b \circ to_a \circ x_{f-to}, x_{f-from} \circ from_a \circ from_b \rangle \,\text{in}$

$\quad \text{let}\, x_{ptrcap} = non\langle to_c \circ to_a \circ x_{f-to}, x_{f-from} \circ from_a \circ from_c \rangle \,\text{in}$

$\quad \text{let}\, x_{ptrcap} = \text{pack}[lin\langle \epsilon, \text{Delay}(\delta') \vee \chi' \rangle, x_{ptrcap}] \,\text{as}\, PtrCap(\tau_R, MTyp(\alpha_N, \tau_1, \tau_2, \epsilon, \omega)) \,\text{in}$

$\quad \text{let}\, x_{ptr} = \text{pack}[\alpha_N, \epsilon, \omega, non\langle x_N, x_{ptrcap} \rangle] \,\text{as}\, Ptr(\tau_R, \tau_1, \tau_2) \,\text{in}$

$\quad \text{let}\, x_{mem-0} = \text{store}[x_{mem-0}](x_N \leftarrow x_{done}) \,\text{in}$

$\quad \text{let}\, x_{mem-1} = \text{store}[x_{mem-1}](x_N + 1 \leftarrow x_\omega) \,\text{in}$

$\quad \text{let}\, x'_\epsilon = lin\langle x_\epsilon, lin\langle x_{mem-0}, x_{mem-1}, x_{mem-2}, x_{mem-3} \rangle \rangle \,\text{in}$

$\quad \text{let}\, x'_u = \text{disj}^1_{\text{Delay}(\delta') \vee \chi'}(x'_{Delay-\delta}) \,\text{in}$

$\quad \text{let}\, x'_r = (x_{f-from} \circ from_a \circ from_b)\, lin\langle x'_\epsilon, x'_u \rangle \,\text{in}$

$\quad \text{let}\, x'_R = lin\langle x'_r, x'_{Delay-\chi}, x'_f, x_N, deallocBlock \rangle \,\text{in}$

$\quad \text{let}\, x'_R = \text{pack}[\chi', \delta', lin\langle \epsilon, MTyp(\alpha_N, \tau_1, \tau_2, \epsilon, \omega) \rangle, \text{Int}(\alpha_N), x'_R] \,\text{as}\, Rgn(\tau_R) \,\text{in}$

$\quad lin\langle lin\langle x_A, x_{roll}, x_{unroll} \rangle, lin\langle x'_R, x_{ptr} \rangle \rangle$

$$
\begin{array}{l}
getDelayDelta = \\
\quad \mathrm{case}\, x_u\, \mathrm{of}\, x_{Delay-\delta}.lin\langle x_{Delay-\chi}, x_{Delay-\delta}\rangle\, \mathrm{or}\, x_\chi. \\
\quad\quad \mathrm{commit}[x_{Delay-\chi}](x_\chi : (\chi = lin\langle \mathrm{Delay}(\chi), \mathrm{Delay}(\delta)\rangle\, \mathrm{in}\, \chi)) \\
\tau_\chi = lin\langle MTyp(\alpha_N, \tau_1, \tau_2, \epsilon, \omega), \mathrm{Delay}(\delta') \vee \chi'\rangle \\
\tau_\delta = non\langle\rangle \\
to-a = \lambda x_{\epsilon u} : lin\langle \epsilon, \mathrm{Delay}(\tau_\delta) \vee \tau_\chi\rangle \Rightarrow \\
\quad \mathrm{let}\, \langle x_\epsilon, x_u\rangle = x_{\epsilon u}\, \mathrm{in} \\
\quad \mathrm{case}\, x_u\, \mathrm{of}\, x_{Delay-\delta}.\mathrm{let}\, \langle x_\chi\rangle = \mathrm{commit}[x_{Delay-\delta}](non\langle\rangle : (\delta = non\langle\tau_\chi\rangle\, \mathrm{in}\, \delta))\, \mathrm{in}\, lin\langle x_\epsilon, x_\chi\rangle \\
\quad \mathrm{or}\, x_\chi.lin\langle x_\epsilon, x_\chi\rangle \\
from-a = \lambda x_{\epsilon\chi} : lin\langle \epsilon, \tau_\chi\rangle \Rightarrow \\
\quad \mathrm{let}\, \langle x_\epsilon, x_\chi\rangle = x_{\epsilon\chi}\, \mathrm{in}\, lin\langle x_\epsilon, \mathrm{disj}^2_{\mathrm{Delay}(\tau_\delta)\vee\tau_\chi}(x_\chi)\rangle \\
to-b = \lambda x_{\epsilon mu} : lin\langle \epsilon, lin\langle MTyp(\alpha_N, \tau_1, \tau_2, \epsilon, \omega), \mathrm{Delay}(\delta') \vee \chi'\rangle\rangle \Rightarrow \\
\quad \mathrm{let}\, \langle x_\epsilon, x_{mu}\rangle = x_{\epsilon mu}\, \mathrm{in}\, \mathrm{let}\, \langle x_m, x_u\rangle = x_{mu}\, \mathrm{in}\, lin\langle lin\langle x_\epsilon, x_m\rangle, x_u\rangle \\
from-b = \lambda x_{\epsilon mu} : lin\langle lin\langle \epsilon, MTyp(\alpha_N, \tau_1, \tau_2, \epsilon, \omega)\rangle, \mathrm{Delay}(\delta') \vee \chi'\rangle \Rightarrow \\
\quad \mathrm{let}\, \langle x_{\epsilon m}, x_u\rangle = x_{\epsilon mu}\, \mathrm{in}\, \mathrm{let}\, \langle x_\epsilon, x_m\rangle = x_{\epsilon m}\, \mathrm{in}\, lin\langle x_\epsilon, lin\langle x_m, x_u\rangle\rangle \\
to-c = \lambda x_{\epsilon mu} : lin\langle \epsilon, lin\langle MTyp(\alpha_N, \tau_1, \tau_2, \epsilon, \omega), \mathrm{Delay}(\delta') \vee \chi'\rangle\rangle \Rightarrow \\
\quad \mathrm{let}\, \langle x_\epsilon, x_{mu}\rangle = x_{\epsilon mu}\, \mathrm{in}\, \mathrm{let}\, \langle x_m, x_u\rangle = x_{mu}\, \mathrm{in}\, lin\langle lin\langle x_\epsilon, x_u\rangle, x_m\rangle \\
from-c = \lambda x_{\epsilon um} : lin\langle lin\langle \epsilon, \mathrm{Delay}(\delta') \vee \chi'\rangle, MTyp(\alpha_N, \tau_1, \tau_2, \epsilon, \omega)\rangle \Rightarrow \\
\quad \mathrm{let}\, \langle x_{\epsilon u}, x_m\rangle = x_{\epsilon mu}\, \mathrm{in}\, \mathrm{let}\, \langle x_\epsilon, x_u\rangle = x_{\epsilon u}\, \mathrm{in}\, lin\langle x_\epsilon, lin\langle x_m, x_u\rangle\rangle \\
deallocBlock = \lambda x_{free\epsilon mN} : lin\langle \tau_{free}, lin\langle \epsilon, MTyp(\alpha_N, \tau_1, \tau_2, \epsilon, \omega)\rangle, \mathrm{Int}(\alpha_N)\rangle. \\
\quad \mathrm{let}\, \langle x_{free}, x_{\epsilon m}, x_N\rangle = x_{free\epsilon mN}\, \mathrm{in} \\
\quad \mathrm{let}\, \langle x_A, x_{roll}, x_{unroll}\rangle = x_{free}\, \mathrm{in} \\
\quad \mathrm{let}\, \langle x_\epsilon, x_m\rangle = x_{\epsilon m}\, \mathrm{in} \\
\quad \mathrm{let}\, \langle x_{mem-0}, x_{mem-1}, x_{mem-2}, x_{mem-3}\rangle = x_m\, \mathrm{in} \\
\quad \mathrm{let}\, \langle x_{mem-0}, x_{free-tail}\rangle = \mathrm{load}[x_{mem-0}](x_N)\, \mathrm{in} \\
\quad \mathrm{let}\, \langle x_{mem-1}, x_\omega\rangle = \mathrm{load}[x_{mem-1}](x_N + 1)\, \mathrm{in} \\
\quad \mathrm{unpack}\, \beta, \gamma, \langle x_{A\beta}, x_{A\gamma}, x_{Af}\rangle = x_{unroll}\, x_A\, \mathrm{in} \\
\quad \mathrm{let}\, x_{mem-0} = \mathrm{store}[x_{mem-0}](x_N \leftarrow x_{Af})\, \mathrm{in} \\
\quad \mathrm{let}\, x_{mem-1} = \mathrm{store}[x_{mem-1}](x_N + 1 \leftarrow x_{A\beta})\, \mathrm{in} \\
\quad \mathrm{let}\, x_{mem-2} = \mathrm{store}[x_{mem-2}](x_N + 2 \leftarrow x_{roll})\, \mathrm{in} \\
\quad \mathrm{let}\, x'_A = lin\langle x_N, lin\langle x_{A\gamma}, x_{mem-0}, x_{mem-1}, x_{mem-2}, x_{mem-3}\rangle, allocatorCode\rangle\, \mathrm{in} \\
\quad \mathrm{let}\, x'_A = \mathrm{pack}[\mathrm{Int}(\alpha_N), \tau_{deallocpack}, x'_A]\, \mathrm{as}\, Allocator\, \mathrm{in} \\
\quad x_{free-tail}\, lin\langle lin\langle x_{roll}\, x'_A, x_{roll}, x_{unroll}\rangle, x_\epsilon, x_\omega\rangle \\
\tau_{deallocpack} = lin\langle \gamma, \alpha_N \mapsto (lin\langle \beta, \gamma\rangle \rightarrow lin\langle \alpha_{Alloc}, FreeBlock\rangle), \\
\qquad\qquad\qquad\qquad \alpha_N + 1 \mapsto \beta, \alpha_N + 2 \mapsto (Allocator \rightarrow \alpha_{Alloc}), \alpha_N + 3 \mapsto \tau_2\rangle \\
allocatorCode = \lambda x_{nm} : lin\langle \mathrm{Int}(\alpha_N), \tau_{deallocpack}\rangle. \\
\quad \mathrm{let}\, \langle x_N, x_m\rangle = x_{nm}\, \mathrm{in} \\
\quad \mathrm{let}\, \langle x_{A\gamma}, x_{mem-0}, x_{mem-1}, x_{mem-2}, x_{mem-3}\rangle = x_m\, \mathrm{in} \\
\quad \mathrm{let}\, \langle x_{mem-0}, x_{Af}\rangle = \mathrm{load}[x_{mem-0}](x_N)\, \mathrm{in} \\
\quad \mathrm{let}\, \langle x_{mem-1}, x_{A\beta}\rangle = \mathrm{load}[x_{mem-1}](x_N + 1)\, \mathrm{in} \\
\quad \mathrm{let}\, \langle x_{mem-2}, x_{roll}\rangle = \mathrm{load}[x_{mem-2}](x_N + 2)\, \mathrm{in} \\
\quad \mathrm{let}\, x_A = lin\langle x_{A\beta}, x_{A\gamma}, x_{Af}\rangle\, \mathrm{in} \\
\quad \mathrm{let}\, x_A = \mathrm{pack}[\beta, \gamma, x_A]\, \mathrm{as}\, Allocator\, \mathrm{in} \\
\quad \mathrm{let}\, x_{freeBlock} = \mathrm{pack}[\alpha_N, lin\langle \beta, \gamma\rangle \rightarrow lin\langle \alpha_{Alloc}, FreeBlock\rangle, \beta, (Allocator \rightarrow \alpha_{Alloc}), \tau_2, \\
\qquad\qquad\qquad\qquad lin\langle x_N, x_{mem-0}, x_{mem-1}, x_{mem-2}, x_{mem-3}\rangle]\, \mathrm{as}\, FreeBlock\, \mathrm{in} \\
\quad lin\langle x_{roll}\, x_A, x_{freeBlock}\rangle
\end{array}
$$

Translation well-kindedness

Lemma 21. *If $\Delta \vdash \tau : \kappa$ in the source language, then $[\![\Delta]\!] \vdash [\![\tau]\!] : [\![\kappa]\!]$ in the target language.*

Proof. By induction on the derivation of $\Delta \vdash \tau : \kappa$. Sample cases:

Case 1. $\Delta \vdash \mathrm{Rgn}(\tau) : \overset{lin}{2}$ and $\Delta \vdash \tau : \mathbf{rgn}$ and $[\![\mathrm{Rgn}(\tau)]\!] = Rgn([\![\tau]\!])$. By induction on τ, $[\![\Delta]\!] \vdash [\![\tau]\!] : [\![\mathbf{rgn}]\!]$, where $[\![\mathbf{rgn}]\!] = \overset{lin}{0}$. This implies that $[\![\Delta]\!] \vdash Rgn([\![\tau]\!]) : \overset{lin}{2}$.

Case 2. $\Delta \vdash \tau_1 \to \tau_2 : \overset{non}{1}$ and $[\![\tau_1 \to \tau_2]\!] = lin\langle \tau_{free}, [\![\tau_1]\!] \rangle \to lin\langle \tau_{free}, [\![\tau_2]\!] \rangle$. Because $\alpha_{Alloc} \mapsto \overset{lin}{2} \in [\![\Delta]\!]$, we can conclude $[\![\Delta]\!] \vdash \tau_{free} : \overset{lin}{4}$. Using induction on τ_1 and τ_2, we conclude $[\![\Delta]\!] \vdash lin\langle \tau_{free}, [\![\tau_1]\!] \rangle \to lin\langle \tau_{free}, [\![\tau_2]\!] \rangle : \overset{non}{1}$.

Translation well-typedness

Lemma 22. *All of the following hold:*

- If $C, \to \vdash e : \tau \rightsquigarrow [\![e]\!]$ in the source language, then $[\![C, \to]\!], x_{free} \mapsto \tau_{free} \vdash [\![e]\!] : lin\langle \tau_{free}, [\![\tau]\!] \rangle$ in the target language.
- If $C, \Rightarrow \vdash e : \tau \rightsquigarrow [\![e]\!]$ in the source language, then $[\![C, \Rightarrow]\!] \vdash [\![e]\!] : [\![\tau]\!]$ in the target language.
- If $e = v$ and $C \vdash v : \tau \overset{value}{\rightsquigarrow} [\![value\, v]\!]$ in the source language, then $[\![C]\!] \vdash [\![v]\!] : [\![\tau]\!]$ in the target language.

Proof. By induction on e's typing derivation. Sample cases:

Case 1.

$$\frac{\Delta; \{\}; \{\}; \Theta; \{\}; \{\}; \{x \mapsto \tau_a\}; \to \vdash e : \tau_b \rightsquigarrow [\![e]\!] \qquad \Delta \vdash \tau_a : \overset{\phi}{n}}{\Delta; \{\}; \{\}; \Theta; \{\}; \{\}; \overset{non}{\Gamma}; \Lambda \vdash (\lambda x{:}\tau_a.e) : \tau_a \to \tau_b \overset{value}{\rightsquigarrow} \lambda x{:}lin\langle \tau_{free}, [\![\tau_a]\!] \rangle.\mathrm{let}\ \langle x_{free}, x \rangle = x\ \mathrm{in}\ [\![e]\!]}$$

By induction, $[\![\Delta]\!]; \{\}; \{\}; [\![\Theta]\!]; \{x \mapsto [\![\tau_a]\!]\}, x_{free} \mapsto \tau_{free}; \to \vdash [\![e]\!] : lin\langle \tau_{free}, [\![\tau_b]\!] \rangle$. From this, we can conclude that

$$[\![\Delta]\!]; \{\}; \{\}; [\![\Theta]\!]; \{x \mapsto lin\langle \tau_{free}, [\![\tau_a]\!] \rangle\}; \to \vdash \mathrm{let}\ \langle x_{free}, x \rangle = x\ \mathrm{in}\ [\![e]\!] : lin\langle \tau_{free}, [\![\tau_b]\!] \rangle$$

By translation well-kindedness, $[\![\Delta]\!] \vdash [\![\tau_a]\!] : \overset{\phi}{n}$, and so $[\![\Delta]\!] \vdash lin\langle \tau_{free}, [\![\tau_a]\!] \rangle : \overset{\phi}{n'}$, where $n' = n + 4$. From this, we can conclude:

$$[\![\Delta]\!]; \{\}; \{\}; [\![\Theta]\!]; [\![\overset{non}{\Gamma}]\!]; \Lambda \vdash \lambda x{:}lin\langle \tau_{free}, [\![\tau_a]\!] \rangle.\mathrm{let}\ \langle x_{free}, x \rangle = x\ \mathrm{in}\ [\![e]\!] : [\![\tau_a \to \tau_b]\!]$$

Case 2.

$$\frac{C, \to \vdash e_R : \mathrm{Rgn}(\tau_R) \rightsquigarrow [\![e_R]\!]}{C, \to \vdash \mathrm{freergn}(e_R) : lin\langle\rangle \rightsquigarrow \mathrm{letfree}\, x_R = [\![e_R]\!]\, \mathrm{in}\, lin\langle freergn[[\![\tau_R]\!]](x_{free}, x_R), lin\langle\rangle\rangle}$$

By induction, $[\![C, \to]\!], x_{free} \mapsto \tau_{free} \vdash [\![e_R]\!] : lin\langle \tau_{free}, Rgn([\![\tau_R]\!]) \rangle$. Type-checking $freergn[\tau_R](x_{free}, x_R)$ yields:

$$[\![\overset{non}{C}]\!], \to, x_{free} \mapsto \tau_{free}, x_R \mapsto Rgn([\![\tau_R]\!]) \vdash freergn[[\![\tau_R]\!]](x_{free}, x_R) : \tau_{free}$$

$$[\![\overset{non}{C}]\!], \to, x_{free} \mapsto \tau_{free}, x_R \mapsto Rgn([\![\tau_R]\!]) \vdash lin\langle freergn[[\![\tau_R]\!]](x_{free}, x_R), lin\langle\rangle\rangle : lin\langle \tau_{free}, lin\langle\rangle\rangle$$

From this, we can conclude:

$$[\![\overset{non}{C}]\!], \to, x_{free} \mapsto \tau_{free} \vdash [\![\mathrm{freergn}(e_R)]\!] : lin\langle \tau_{free}, lin\langle\rangle\rangle$$

where $[\![\mathrm{freergn}(e_R)]\!] = \mathrm{let}\ \langle x_{free}, x_R \rangle = [\![e_R]\!]\, \mathrm{in}\, lin\langle freergn[[\![\tau_R]\!]](x_{free}, x_R), lin\langle\rangle\rangle$