Composing a Well-Typed Region

Chris Hawblitzel, Heng Huang[1], and Lea Wittie[2]

[1] Dartmouth College
[2] Bucknell University

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[11] simply ban aliasing, so that every object has only one pointer to it. Alias types[9] 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[10,12], 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[4,8].

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[12]) 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[12], 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	$n = 0 \mid succ(n)$
type variables	$\alpha = \alpha, \beta, \gamma, \delta, \epsilon, \rho, \chi, \ldots$
type variable env	$\Delta = \{\ldots, \alpha \mapsto \kappa, \ldots\}$

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

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: $1 = succ(0)$, $2 = succ(1)$, ...

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

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

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

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

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

Source language (extensions to target language), part 1

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

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

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

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

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

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

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.)

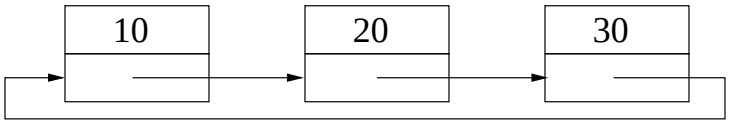
The source and target languages use an environment Θ to map recursive type names to recursive type definitions. For example, the environment $\Theta = \{\alpha_{List} \mapsto \langle \mathrm{int}, \alpha_{List} \rangle @ \rho\}$ defines a type name α_{List} for circular lists of integers residing in 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 [6].

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: $[\![\tau_r]\!] \to \mathit{lin}\langle [\![\tau_r]\!], [\![\tau_1]\!], [\![\tau_2]\!]\rangle$. Since the data supports both reading and writing, the pointer must also contain a *set* function of type $\mathit{lin}\langle [\![\tau_r]\!], [\![\tau_1]\!], [\![\tau_2]\!]\rangle \to$

$[\![\tau_r]\!]$. 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 the companion technical report[6] 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[9], 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 $\mathrm{Int}(5)$; 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 $\beta \mapsto \alpha_y$:

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

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)

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

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

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)

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

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

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

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

Source language (extensions to target language), part 2

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

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

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

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 $\overset{lin}{0}$:

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

Delayed types provide a way to implement incremental allocation inside regions. For example, rather than defining the type $[\![\varphi]\!] = \mathit{lin}\langle [\![\tau_{1,1}]\!], \ldots, [\![\tau_{3,2}]\!]\rangle$ all at once, a program makes six names $\alpha_{1,1} \ldots \alpha_{3,2}$ and defines $[\![\varphi]\!] = \mathit{lin}\langle \alpha_{1,1}, \ldots, \alpha_{3,2}\rangle$. 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 $\mathit{lin}\langle \alpha_{1,1}, \ldots, \alpha_{3,2}\rangle$ 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 $[\![\varphi]\!] = \mathit{lin}\langle \mathrm{Int}(0) \cup \alpha_{1,1}, \ldots, \mathrm{Int}(0) \cup \alpha_{3,2}\rangle$, 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.

The companion technical report[6] contains the complete compile-time translation of the source language (where the source language is a superset of the complete target language, parts 1-4), constructing the type $[\![\varphi]\!]$ shown above, and using the composition operator $\circ$ to build coercions that extract α_k values from deeper and deeper in the growing $[\![\varphi]\!]$ type. It proves the well-typedness of the translation and the type safety of the target language. As in section 4's translation, 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).

6 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 the 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: by extending the type system with type operators $(\lambda\alpha:\kappa.\tau)$ and integer case types $(\mathrm{case}\,\tau\,\mathrm{of}\,0.\tau\,\mathrm{or}\,succ(\alpha).\tau)$, it is possible to encode a "next-region" operator in the style of Fluet and Wang [4] (see [6] for details).

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: just build a linear tuple of regions, and use nonlinear coercions to extract regions from the tuple (see [6] for details). 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.

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[5] 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. 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.
5. 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.
6. Chris Hawblitzel, Heng Huang, and Lea Wittie. Composing a well-typed region. Technical Report TR2004-521, Dartmouth College, October 19, 2004.
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. Frederick Smith, David Walker, and Greg Morrisett. Alias types. In *In European Symposium on Programming*, 2000.
10. Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.
11. 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.
12. 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.