Linear Regions Are All You Need

Matthew Fluet[1], Greg Morrisett[2], and Amal Ahmed[2]

[1] Cornell University
Ithaca, NY
fluet@cs.cornell.edu
[2] Harvard University
Cambridge, MA
greg@eecs.harvard.edu, amal@eecs.harvard.edu

Abstract. The type-and-effects system of the Tofte-Talpin region calculus makes it possible to safely reclaim objects without a garbage collector. However, it requires that regions have last-in-first-out (LIFO) lifetimes following the block structure of the language. We introduce λ^{rgnUL}, a core calculus that is powerful enough to encode Tofte-Talpin-like languages, and that eliminates the LIFO restriction. The target language has an extremely simple, substructural type system. To prove the power of the language, we sketch how Tofte-Talpin-style regions, as well as the first-class dynamic regions and unique pointers of the Cyclone programming language can be encoded in λ^{rgnUL}.

1 Introduction

Most type-safe languages rely upon a garbage collector to reclaim storage safely. But there are domains, such as device drivers and embedded systems, where today's garbage collection algorithms result in unacceptable space or latency overheads. In these settings, programmers have been forced to use languages, like C, where memory management can be tailored to the application, but where the lack of type-safety has lead to numerous bugs. To address these concerns, we have been developing Cyclone [1], a type-safe dialect of C that is intended to give programmers as much control over memory management as possible while retaining strong, static typing.

The initial design of Cyclone was based upon the region type system of Tofte and Talpin [2]. Data are allocated within lexically-scoped *regions* and all of the objects in a region are deallocated at the end of the region's scope. Unfortunately, the last-in-first-out (LIFO) lifetimes of lexically-scoped regions place severe restrictions on when data can be effectively reclaimed, and we found many programs that resulted in (unbounded) leaks when compared to a garbage collected implementation.

To address these concerns, we added a number of new features to Cyclone, including *dynamic regions* and *unique pointers* that provide more control over memory management. Dynamic regions are not restricted to LIFO lifetimes and can be treated as first-class objects. They are particularly well suited for iterative

computations, CPS-based computations, and event-based servers where lexical regions do not suffice. Unique pointers are essentially lightweight, dynamic regions that hold exactly one object. To ensure soundness, both dynamic regions and unique pointers depend upon a notion of *linear capabilities* which must be carefully threaded through a program. To alleviate this tedium, Cyclone provides convenient mechanisms to temporarily "open" a dynamic region or unique pointer and treat it as if it were in a freshly allocated, lexically-scoped region.

The efficacy of these new memory management features was detailed in previous papers [3, 4], where we analyzed a range of applications, including a streaming media server, a space-conscious web server, and a Scheme runtime system with a copying garbage collector. And while the soundness of Cyclone's lexical regions and type-and-effects system has been established [5, 6], a model that justifies the soundness of the new features has eluded our grasp, due to sheer complexity.

Therefore, the goal of this work is to provide a simple model where we can easily *encode* the key features of Cyclone in a uniform target language for which type soundness may be easily established. The first step of our encoding was detailed in a previous paper [6], where we gave a translation from a type-and-effects, region-based language to a monadic variant of System F called $\mathsf{F}^{\mathsf{RGN}}$. This calculus is summarized in Section 2. The meat of this paper picks up where this translation left off by further translating $\mathsf{F}^{\mathsf{RGN}}$ to a substructural polymorphic lambda calculus where the internals of the indexed monad are exposed (Section 3). The target language and translation are extremely simple, yielding a relatively straightforward proof of soundness for lexically scoped regions. Then, in Section 5, we sketch how the features in the target language allow us to encode Cyclone's dynamic regions and unique pointers, as well as their interactions with lexically-scoped regions. Throughout, it is our intention that the target calculus serve as a compiler intermediate language and as vehicle for formal reasoning, not as a high-level programming language.

2 Source Calculus: $\mathsf{F}^{\mathsf{RGN}}$

Launchbury and Peyton Jones introduced the ST monad to encapsulate stateful computations within the pure functional language Haskell [7]. Three key insights give rise to a safe and efficient implementation of stateful computations. First, a stateful computation is represented as a *store transformer*, a description of commands to be applied to an initial store to yield a final store. Second, the store can not be duplicated, because the state type is opaque and all primitive store transformers use the store in a single-threaded manner; hence, a stateful computation can update the store in place. Third, parametric polymorphism can be used to safely encapsulate and run a stateful computation.

All of these insights can be carried over to the region case, where we interpret stores as stacks of regions. We introduce the types and operations associated with the rgn monad:

$$\tau \ ::= \ldots \mid \mathsf{rgn}\, s\, \tau \mid \mathsf{ref}\, s\, \tau \mid \mathsf{hnd}\, s \mid \mathsf{pf}\, (s_1 \leq s_2)$$

$$
\begin{aligned}
\texttt{return} &: \forall \varsigma.\, \forall \alpha.\, \alpha \to \mathsf{rgn}\, \varsigma\, \alpha \\
\texttt{then} &: \forall \varsigma.\, \forall \alpha, \beta.\, \mathsf{rgn}\, \varsigma\, \alpha \to (\alpha \to \mathsf{rgn}\, \varsigma\, \beta) \to \mathsf{rgn}\, \varsigma\, \beta \\
\texttt{new} &: \forall \varsigma.\, \forall \alpha.\, \mathsf{hnd}\, \varsigma \to \alpha \to \mathsf{rgn}\, \varsigma\, (\mathsf{ref}\, \varsigma\, \alpha) \\
\texttt{read} &: \forall \varsigma.\, \forall \alpha.\, \mathsf{ref}\, \varsigma\, \alpha \to \mathsf{rgn}\, \varsigma\, \alpha \\
\texttt{write} &: \forall \varsigma.\, \forall \alpha.\, \mathsf{ref}\, \varsigma\, \alpha \to \alpha \to \mathsf{rgn}\, \varsigma\, \mathbf{1} \\
\texttt{runRgn} &: \forall \alpha.\, (\forall \varsigma.\, \mathsf{rgn}\, \varsigma\, \alpha) \to \alpha \\
\texttt{letRgn} &: \forall \varsigma_1.\, \forall \alpha.\, (\forall \varsigma_2.\, \mathsf{pf}\, (\varsigma_1 \leq \varsigma_2) \to \mathsf{hnd}\, \varsigma_2 \to \mathsf{rgn}\, \varsigma_2\, \alpha) \to \mathsf{rgn}\, \varsigma_1\, \alpha \\
\texttt{coerceRgn} &: \forall \varsigma_1, \varsigma_2.\, \forall \alpha.\, \mathsf{pf}\, (\varsigma_1 \leq \varsigma_2) \to \mathsf{rgn}\, \varsigma_1\, \alpha \to \mathsf{rgn}\, \varsigma_2\, \alpha \\
\texttt{reflSub} &: \forall \varsigma.\, \mathsf{pf}\, (\varsigma \leq \varsigma) \\
\texttt{transSub} &: \forall \varsigma_1, \varsigma_2, \varsigma_3.\, \mathsf{pf}\, (\varsigma_1 \leq \varsigma_2) \to \mathsf{pf}\, (\varsigma_2 \leq \varsigma_3) \to \mathsf{pf}\, (\varsigma_1 \leq \varsigma_3)
\end{aligned}
$$

The type $\mathsf{rgn}\, s\, \tau$ is the type of computations which transform a stack indexed by s and deliver a value of type τ. The type $\mathsf{ref}\, s\, \tau$ is the type of mutable references allocated in the region at the top of the stack indexed by s and containing a value of type τ. The type $\mathsf{hnd}\, s$ is the type of handles for the region at the top of the stack indexed by s; we require a handle to allocate a reference in a region, but do not require a handle to read or write a reference.

The operations `return` and `then` are the *unit* and *bind* operations of the rgn monad, the former lifting a value to a computation and the latter sequencing computations. The next three operations are primitive stack transformers. `new` takes a region handle and an initial value and yields a stack transformer, which, when applied to a stack of regions, allocates and initializes a fresh reference in the appropriate region, and delivers the reference and the augmented stack. Similarly, `read` and `write` yield computations that respectively query and update the mappings of references to values in the current stack of regions. Note that all of these operations require the stack index ς of rgn and ref to be equal.

Finally, the operation `runRgn` encapsulates a stateful computation. To do so, it takes a stack transformer as its argument, applies it to an initial empty stack of regions, and returns the result while discarding the final stack (which should be empty). Note that to apply `runRgn`, we instantiate α with the type of the result to be returned, and then supply a stack transformer, *which is polymorphic in the stack index* ς. The effect of this universal quantification is that the stack transformer makes no assumptions about the initial stack (e.g., the existence of pre-allocated regions or references). Furthermore, the instantiation of the type variable α occurs outside the scope of the stack variable ς; this prevents the stack transformer from delivering a value whose type mentions ς. Thus, references or computations depending on the final stack cannot escape beyond the encapsulation of `runRgn`.

However, the above does not suffice to encode region-based languages. The difficulty is that, in a region-based language, it is critical to allocate variables in and read variables from an outer (older) region while in the scope of an inner (younger) region. To accommodate this essential idiom, we include a powerful `letRgn` operation that is similar to `runRgn` in the sense that it encapsulates a stateful computation. Operationally, `letRgn` transforms a stack by (1) creating a new region on the top of the stack, (2) applying a stack transformer to the augmented stack to yield a transformed stack, (3) destroying the region on the

$$
\begin{array}{lrl}
\textit{Kinds} & \kappa & ::= \mathsf{STACK} \mid \star \\
\\
\textit{Type-level Variables} & \varepsilon, \varsigma, \alpha & ::= \textit{TVars} \\
\textit{Stack Indices} & s & ::= \varsigma \\
\textit{Types} & \tau & ::= \alpha \mid \tau_1 \to \tau_2 \mid \mathbf{1} \mid \tau_1 \times \tau_2 \mid \forall \varepsilon{:}\kappa.\, \tau \\
 & & \quad \mathsf{rgn}\, s\, \tau \mid \mathsf{ref}\, s\, \tau \mid \mathsf{hnd}\, s \mid \mathsf{pf}\, (s_1 \leq s_2) \\
\textit{Type-level Terms} & \epsilon & ::= s \mid \tau \\
\textit{Type-level Contexts} & \Delta & ::= \bullet \mid \Delta, \varepsilon{:}\kappa \\
\\
\mathsf{rgn}\ \textit{Monad Operations} & \textit{ops} & ::= \mathtt{runRgn} \mid \mathtt{coerceRgn} \mid \mathtt{transSub} \mid \\
 & & \quad \mathtt{return} \mid \mathtt{then} \mid \mathtt{letRgn} \mid \mathtt{new} \mid \mathtt{read} \mid \mathtt{write} \\
\textit{Expressions} & e & ::= \textit{ops} \mid x \mid \lambda x{:}\tau.\, e \mid e_1\, e_2 \mid \langle\rangle \mid \mathtt{let}\ \langle\rangle = e_1\ \mathtt{in}\ e_2 \mid \\
 & & \quad \langle e_1, e_2 \rangle \mid \mathtt{let}\ \langle x_1, x_2 \rangle = e_1\ \mathtt{in}\ e_2 \mid \Lambda \varepsilon{:}\kappa.\, e \mid e\,[\epsilon] \\
\textit{Expression-level Contexts} & \Gamma & ::= \bullet \mid \Gamma, x{:}\tau
\end{array}
$$

Fig. 1. Syntax of $\mathsf{F}^{\mathsf{RGN}}$

top of the transformed stack and yielding the bottom of the transformed stack. Ignoring for the moment the argument of type $\mathsf{pf}\,(\varsigma_1 \leq \varsigma_2)$, we see that we may apply exactly the same reasoning as applied to `runRgn`: the computation makes no assumptions about the newly augmented stack ς_2, nor can the newly augmented stack ς_2 be leaked through the return value.

What, then, is the role of the $\mathsf{pf}\,(\varsigma_1 \leq \varsigma_2)$? The answer lies in the fact that the stack index ς_2 does not denote an arbitrary stack; rather, it should denote a stack that is related to ς_1 by the addition of a newly created region (i.e., $\varsigma_2 \equiv r{::}\varsigma_1$). In fact, we may consider ς_1 to be a subtype of ς_2, since every region in the stack ς_1 is also in the stack ς_2; values of type $\mathsf{pf}\,(s_1 \leq s_2)$ are witnesses of this relationship. The operation `coerceRgn` applies a subtyping witness to a stack transformer for the substack to yield a stack transformer for the superstack; intuitively, the operation is sound as a stack transformer may simply ignore extra regions. The operations `reflSub` and `transSub` are combinators witnessing the reflexivity and transitivity of the subtyping relation.

Figure 1 gives the complete syntax for $\mathsf{F}^{\mathsf{RGN}}$, which is a natural extension of System F. We introduce a simple kind system to support abstraction over both types and stack indices. (In the text, we often omit kind annotations, using the convention that ς stands for a type-level variable of STACK kind, and α of $\star$.)

We adopt the standard type system for System F; the only typing judgement of interest is $\Delta; \Gamma \vdash e : \tau$ meaning that expression e has type τ, where Δ records the free type-level variables and their kinds and Γ records the free expression-level variables and their types. The types for the rgn monad operations are as given in the text above.

Our previous work [6] gave an operational semantics for $\mathsf{F}^{\mathsf{RGN}}$ and proved the type soundness of $\mathsf{F}^{\mathsf{RGN}}$. However, the operational semantics of $\mathsf{F}^{\mathsf{RGN}}$ is somewhat cumbersome, due to the intertwining of contexts for pure evaluation and monadic evaluation. Hence, in the present setting, we will *define* the operational behavior of $\mathsf{F}^{\mathsf{RGN}}$ by its translation into the target language of Section 3.

3 Target Calculus: λ^{rgnUL}

In another line of work [8], we introduced λ^{URAL}, a core substructural polymorphic λ-calculus, and then extended it to $\lambda^{\mathsf{refURAL}}$ by adding a rich collection of mutable references. Providing four sorts of substructural qualifiers (unrestricted, relevant, affine, and linear) allowed us to encode and study the interactions of different forms of uniqueness that appear in other high-level programming languages. Notable features of $\lambda^{\mathsf{refURAL}}$ include: deallocation of references; strong (type-varying) updates; and storage of unique objects in shared references.

Here, we augment $\lambda^{\mathsf{refURAL}}$ by adding region primitives, and also simplify the language by removing features, such as the relevant and affine qualifiers, that do not play a part in the translation. We call the resulting language λ^{rgnUL}.

In contrast to the `letRgn` operation of $\mathsf{F}^{\mathsf{RGN}}$, which encapsulates the creation and destruction of a region, the primitives of λ^{rgnUL} include `newrgn` and `freergn` for separately creating and destroying a region. All access to a region (for allocating, reading, and writing references) is mediated by a linear *capability* that is produced by `newrgn` and consumed by `freergn`.

As noted above, λ^{rgnUL} is a *substructural* polymorphic λ-calculus. A *substructural* type system provides the core mechanisms necessary to restrict the number and order of uses of data and operations. In our calculus, types and variables are qualified as unrestricted (U) or linear (L). Essentially, unrestricted variables are allowed to be used an arbitrary number of times, while linear variables are allowed to be used exactly once.

Figure 2 gives the syntax for λ^{rgnUL}, excluding intermediate terms that would appear in an operational semantics. Many of the types and expressions are based on a traditional polymorphic λ-calculus.

We structure our types τ as a qualifier q applied to a pre-type $\overline{\tau}$, yielding the two sorts of types noted above. The qualifier of a type dictates the number of uses of variables of the type, while the pre-type dictates the introduction and elimination forms. The pre-types $\mathbf{1}_{\otimes}$, $\tau_1 \otimes \cdots \otimes \tau_n$, and $\tau_1 \multimap \tau_2$ correspond to the unit, product, and function types of the polymorphic λ-calculus. Quantification over qualifiers, region names, pre-types, and types is provided by the pre-types $\forall\varepsilon{:}\kappa.\,\tau$ and $\exists\varepsilon{:}\kappa.\,\tau$. (In the text, we often omit kind annotations, using the convention that ξ stands for a type-level variable of QUAL kind, ϱ of RGN, $\overline{\alpha}$ of $\overline{\star}$, and α of $\star$.)

The pre-types $\mathsf{ref}\, r\, \tau$ and $\mathsf{hnd}\, r$ are similar to the corresponding types in $\mathsf{F}^{\mathsf{RGN}}$; the former is the type of mutable references allocated in the region r and the latter is the type of handles for the region r. The pre-type $\mathsf{cap}\, r$ is the type of capabilities for accessing the region named r. We shall shortly see how linear capabilities effectively mediate access to a region.

Space precludes us from giving a detailed description of the type system for λ^{rgnUL}; the major features are entirely standard for a substructural setting [9, 8]. First, in order to ensure the correct relationship between a data structure and its components, we extend the lattice ordering on constant qualifiers to arbitrary qualifiers ($\Delta \vdash q \preceq q'$), types ($\Delta \vdash \tau \preceq q'$), and contexts ($\Delta \vdash \Gamma \preceq q'$). Second, we introduce a judgement $\Delta \vdash \Gamma_1 \boxdot \Gamma_2 \rightsquigarrow \Gamma$ that splits the assumptions in Γ

$$
\begin{array}{lll}
\textit{Kinds} & & \kappa ::= \mathsf{QUAL} \mid \mathsf{RGN} \mid \overline{\star} \mid \star \\
\\
\textit{Type-level Variables} & \varepsilon, \xi, \varrho, \overline{\alpha}, \alpha & ::= \textit{TVars} \\
\textit{Constant Qualifiers} & & \mathtt{q} \in \textit{Quals} = \{\mathsf{U}, \mathsf{L}\} \qquad \mathsf{U} \sqsubseteq \mathsf{L} \\
\textit{Qualifiers} & & q ::= \xi \mid \mathtt{q} \\
\textit{Constant Region Names} & & \mathtt{r} \in \textit{RNames} \\
\textit{Region Names} & & r ::= \varrho \mid \mathtt{r} \\
\textit{PreTypes} & & \overline{\tau} ::= \overline{\alpha} \mid \tau_1 \multimap \tau_2 \mid \mathbf{1}_{\otimes} \mid \\
 & & \qquad \tau_1 \otimes \cdots \otimes \tau_n \mid \forall \varepsilon{:}\kappa.\, \tau \mid \exists \varepsilon{:}\kappa.\, \tau \mid \\
 & & \qquad \mathsf{ref}\, r\, \tau \mid \mathsf{hnd}\, r \mid \mathsf{cap}\, r \\
\textit{Types} & & \tau ::= \alpha \mid {}^{q}\overline{\tau} \\
\textit{Type-level Terms} & & \epsilon ::= q \mid r \mid \overline{\tau} \mid \tau \\
\textit{Type-level Contexts} & & \Delta ::= \bullet \mid \Delta, \varepsilon{:}\kappa \\
\\
\textit{Region Primitives} & \textit{prims} & ::= \mathtt{newrgn} \mid \mathtt{freergn} \mid \mathtt{new} \mid \mathtt{read} \mid \mathtt{write} \\
\textit{Expressions} & e & ::= \textit{prims} \mid x \mid {}^{q}\lambda x{:}\tau.\, e \mid e_1\, e_2 \mid {}^{q}\langle\rangle \mid \mathtt{let}\ \langle\rangle = e_1\ \mathtt{in}\ e_2 \mid \\
 & & \qquad {}^{q}\langle e_1, \ldots, e_n\rangle \mid \mathtt{let}\ \langle x_1, \ldots, x_n\rangle = e_1\ \mathtt{in}\ e_2 \mid \\
 & & \qquad {}^{q}\Lambda \varepsilon{:}\kappa.\, e \mid e\,[\epsilon] \mid {}^{q}\mathtt{pack}(\epsilon{:}\kappa, e) \mid \mathtt{let}\ \mathtt{pack}(\varepsilon{:}\kappa, x) = e_1\ \mathtt{in}\ e_2 \\
\textit{Expression-level Contexts} & & \Gamma ::= \bullet \mid \Gamma, x{:}\tau
\end{array}
$$

Fig. 2. Syntax of λ^{rgnUL}

between the contexts Γ_1 and Γ_2. Splitting the context is necessary to ensure that variables are used appropriately by sub-expressions. Note that $\boxdot$ must ensure that an L assumption appears in exactly one sub-context, while U assumptions may appear in both sub-contexts.

The main typing judgement has the form $\Delta; \Gamma \vdash e : \tau$; Figure 3 gives typing rules for each of the expression forms of λ^{rgnUL}.

Finally, we assign types for each of the region primitives of λ^{rgnUL}:

$$
\begin{array}{rl}
\mathtt{newrgn} : & {}^{\mathsf{U}}({}^{\mathsf{L}}\mathbf{1}_{\otimes} \multimap {}^{\mathsf{L}}\exists \varrho.\, {}^{\mathsf{L}}({}^{\mathsf{L}}\mathsf{cap}\, \varrho \otimes {}^{\mathsf{U}}\mathsf{hnd}\, \varrho)) \\
\mathtt{freergn} : & {}^{\mathsf{U}}\forall \varrho.\, {}^{\mathsf{U}}({}^{\mathsf{L}}({}^{\mathsf{L}}\mathsf{cap}\, \varrho \otimes {}^{\mathsf{U}}\mathsf{hnd}\, \varrho) \multimap {}^{\mathsf{L}}\mathbf{1}_{\otimes}) \\
\mathtt{new} : & {}^{\mathsf{U}}\forall \varrho.\, {}^{\mathsf{U}}\forall \overline{\alpha}.\, {}^{\mathsf{U}}({}^{\mathsf{L}}({}^{\mathsf{L}}\mathsf{cap}\, \varrho \otimes {}^{\mathsf{U}}\mathsf{hnd}\, \varrho \otimes {}^{\mathsf{U}}\overline{\alpha}) \multimap {}^{\mathsf{L}}({}^{\mathsf{L}}\mathsf{cap}\, \varrho \otimes {}^{\mathsf{U}}(\mathsf{ref}\, \varrho\, {}^{\mathsf{U}}\overline{\alpha}))) \\
\mathtt{read} : & {}^{\mathsf{U}}\forall \varrho.\, {}^{\mathsf{U}}\forall \overline{\alpha}.\, {}^{\mathsf{U}}({}^{\mathsf{L}}({}^{\mathsf{L}}\mathsf{cap}\, \varrho \otimes {}^{\mathsf{U}}(\mathsf{ref}\, \varrho\, {}^{\mathsf{U}}\overline{\alpha})) \multimap {}^{\mathsf{L}}({}^{\mathsf{L}}\mathsf{cap}\, \varrho \otimes {}^{\mathsf{U}}\overline{\alpha})) \\
\mathtt{write} : & {}^{\mathsf{U}}\forall \varrho.\, {}^{\mathsf{U}}\forall \overline{\alpha}.\, {}^{\mathsf{U}}({}^{\mathsf{L}}({}^{\mathsf{L}}\mathsf{cap}\, \varrho \otimes {}^{\mathsf{U}}(\mathsf{ref}\, \varrho\, {}^{\mathsf{U}}\overline{\alpha}) \otimes {}^{\mathsf{U}}\overline{\alpha}) \multimap {}^{\mathsf{L}}({}^{\mathsf{L}}\mathsf{cap}\, \varrho \otimes {}^{\mathsf{U}}\mathbf{1}_{\otimes}))
\end{array}
$$

We have purposefully "streamlined" the type of the reference primitives in order to simplify the exposition. For example, note that we may only allocate, read, and write references whose contents are unrestricted. However, there is no fundamental difficulty in adopting a richer set of reference primitives (à la $\lambda^{\mathsf{refURAL}}$ [8]), which would allow references to contain arbitrary values.

Space again precludes us from giving a detailed description of the operational semantics for λ^{rgnUL}; however, it is entirely standard for a region-based language. The small-step operational semantics is defined by a relation between configurations of the form (ψ, e), where ψ is a global heap mapping region names to regions and regions are mappings from pointers to values.

The primitives `newrgn` and `freergn` perform the complementary actions of creating and destroying a region in the global heap. Note that the type of `newrgn`

$\boxed{\Delta; \Gamma \vdash e : \tau}$

$$\frac{}{\Delta; \bullet, x{:}\tau \vdash x : \tau} \qquad \frac{\Delta; \Gamma_1 \boxdot \Gamma_2 \rightsquigarrow \Gamma \qquad \Delta \vdash \Gamma_1 \preceq \mathsf{U} \qquad \Delta; \Gamma_2 \vdash e : \tau}{\Delta; \Gamma \vdash e : \tau}$$

$$\frac{\Delta \vdash q : \mathsf{QUAL}}{\Delta; \bullet \vdash {}^{q}\langle\rangle : {}^{q}\mathbf{1}_{\otimes}} \qquad \frac{\Delta \vdash \Gamma_1 \boxdot \Gamma_2 \rightsquigarrow \Gamma \qquad \Delta; \Gamma_1 \vdash e_1 : {}^{q}\mathbf{1}_{\otimes} \qquad \Delta; \Gamma_2 \vdash e_2 : \tau}{\Delta; \Gamma \vdash \mathtt{let}\ \langle\rangle = e_1\ \mathtt{in}\ e_2 : \tau}$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma_1 \boxdot \cdots \boxdot \Gamma_n \rightsquigarrow \Gamma \\ \Delta; \Gamma_1 \vdash e_1 : \tau_1 \qquad \Delta \vdash \tau_1 \preceq q \qquad \ldots \\ \Delta; \Gamma_n \vdash e_n : \tau_n \qquad \Delta \vdash \tau_n \preceq q\end{array}}{\Delta; \Gamma \vdash {}^{q}\langle e_1, \ldots, e_n\rangle : {}^{q}(\tau_1 \otimes \cdots \otimes \tau_n)} \qquad \frac{\begin{array}{c}\Delta \vdash \Gamma_1 \boxdot \Gamma_2 \rightsquigarrow \Gamma \\ \Delta; \Gamma_1 \vdash e_1 : {}^{q}(\tau_1 \otimes \cdots \otimes \tau_n) \\ \Delta; \Gamma_2, x_1{:}\tau_1, \ldots, x_n{:}\tau_n \vdash e_2 : \tau\end{array}}{\Delta; \Gamma \vdash \mathtt{let}\ \langle x_1, \ldots, x_n\rangle = e_1\ \mathtt{in}\ e_2 : \tau}$$

$$\frac{\begin{array}{c}\Delta \vdash \Gamma \preceq q \\ \Delta; \Gamma, x{:}\tau_x \vdash e : \tau\end{array}}{\Delta; \Gamma \vdash {}^{q}\lambda x{:}\tau_x.\, e : {}^{q}(\tau_x \multimap \tau)} \qquad \frac{\begin{array}{c}\Delta \vdash \Gamma_1 \boxdot \Gamma_2 \rightsquigarrow \Gamma \\ \Delta; \Gamma_1 \vdash e_1 : {}^{q}(\tau_x \multimap \tau) \qquad \Delta; \Gamma_2 \vdash e_2 : \tau_x\end{array}}{\Delta; \Gamma \vdash e_1\, e_2 : \tau}$$

$$\frac{\Delta \vdash \Gamma \preceq q \qquad \Delta, \varepsilon{:}\kappa; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash {}^{q}\Lambda\varepsilon{:}\kappa.\, e : {}^{q}(\forall\varepsilon{:}\kappa.\, \tau)} \qquad \frac{\Delta; \Gamma \vdash e_1 : {}^{q}(\forall\varepsilon{:}\kappa.\, \tau) \qquad \Delta \vdash \epsilon_2 : \kappa}{\Delta; \Gamma \vdash e_1\, [\epsilon_2] : \tau[\epsilon_2/\varepsilon]}$$

$$\frac{\begin{array}{c}\Delta \vdash \epsilon_1 : \kappa \qquad \Delta; \Gamma \vdash e_2 : \tau[\epsilon_1/\varepsilon] \\ \Delta \vdash \tau[\epsilon_1/\varepsilon] \preceq q\end{array}}{\Delta; \Gamma; \Sigma \vdash {}^{q}\mathtt{pack}(\epsilon_1{:}\kappa, e_2) : {}^{q}(\exists\varepsilon{:}\kappa.\, \tau)} \qquad \frac{\begin{array}{c}\Delta \vdash \Gamma_1 \boxdot \Gamma_2 \rightsquigarrow \Gamma \qquad \Delta \vdash \tau' : \star \\ \Delta; \Gamma_1 \vdash e_1 : {}^{q}(\exists\varepsilon{:}\kappa.\, \tau) \\ \Delta, \varepsilon{:}\kappa; \Gamma_2, x{:}\tau \vdash e_2 : \tau'\end{array}}{\Delta; \Gamma \vdash \mathtt{let}\ \mathtt{pack}(\varepsilon{:}\kappa, x) = e_1\ \mathtt{in}\ e_2 : \tau'}$$

Fig. 3. Static Semantics of λ^{rgnUL}

specifies that it returns an existential package, hiding the name of the fresh region. The primitives `new`, `read`, and `write` behave precisely as their counterparts in any region-based language. Additionally, their types specify that they thread $\mathsf{L}\mathsf{cap}\,\varrho$ values through the evaluation; the capability is simply presented at each access of a region and returned to allow future access. In the semantics, the capability is represented as a dummy token, which has no run-time significance.

As expected, the type system for λ^{rgnUL} is sound with respect to its operational semantics:

Theorem 1 (λ^{rgnUL} Safety). *If $\bullet; \bullet \vdash e_1 : \tau$ and $(\{\}, e_1) \longmapsto^* (\psi_2, e_2)$, then either there exists v such that $e_2 \equiv v$ or there exists ψ_3 and e_3 such that $(\psi_2, e_2) \longmapsto (\psi_3, e_3)$.*

We have formally verified this result (for a rich superset of λ^{rgnUL}) in the Twelf system [10] using its metatheorem checker [11]. The mechanized proof can be obtained at http://www.cs.cornell.edu/People/fluet/research/substruct-regions/.

4 Translation: $\mathsf{F}^{\mathsf{RGN}}$ to λ^{rgnUL}

Having introduced both our source and target calculi, we are in a position to consider a (type-preserving) translation from $\mathsf{F}^{\mathsf{RGN}}$ to λ^{rgnUL}. Before giving the details, we discuss a few of the high-level issues.

First, we note that $\mathsf{F}^{\mathsf{RGN}}$ has no notion of linearity in the syntax or type system. Rather, all variables and types are implicitly considered unrestricted. Hence, we can expect that the translation of all $\mathsf{F}^{\mathsf{RGN}}$ expressions will yield λ^{rgnUL} expressions with a U qualified type.

On the other hand, we claimed that a stateful region computation could be interpreted as a stack transformer. Recall that the type $\mathsf{rgn}\, s\, \tau$ is the type of computations which transform a stack indexed by s and deliver a value of type τ. A key characteristic of $\mathsf{F}^{\mathsf{RGN}}$ is that all primitive stack transformers are meant to use the stack in a single-threaded manner; hence, a stateful computation can update the stack in place. This single-threaded behavior is precisely the sort of resource management that may be captured by a substructural type system. Hence, we can expect that the representation of a stack of regions in λ^{rgnUL} will be a value with L qualified type. In particular, we will represent a stack of regions as a sequence of linear capabilities, formed out of nested linear tuples.

Third, we must be mindful of a slight mismatch between the hnd and ref types in $\mathsf{F}^{\mathsf{RGN}}$ and the corresponding types in λ^{rgnUL}. Recall that, in $\mathsf{F}^{\mathsf{RGN}}$, $\mathsf{hnd}\, s$ and $\mathsf{ref}\, s\, \tau$ are handles for and references allocated in the region at the top of the stack indexed by s. Whereas, in λ^{rgnUL}, $\mathsf{hnd}\, r$ and $\mathsf{ref}\, r\, \tau$ explicitly name the region of the handle or reference. This subtle distinction (whether the region is implicit or explicit) will need to be handled by the translation.

Bearing these issues in mind, we turn our attention to the translation of $\mathsf{F}^{\mathsf{RGN}}$ type-level terms given in Figure 4. $\mathcal{S}_\star [\![s]\!]$ translates a $\mathsf{F}^{\mathsf{RGN}}$ term of STACK kind to a λ^{rgnUL} term of $\star$ kind. As the STACK kind of $\mathsf{F}^{\mathsf{RGN}}$ is inhabited only by variables, the translation is trivial: in λ^{rgnUL}, ς is considered a variable of $\star$ kind.

$\mathcal{T}_\star [\![\tau]\!]$ and $\mathcal{T}_{\overline{\star}} [\![\tau]\!]$ translate a $\mathsf{F}^{\mathsf{RGN}}$ term of $\star$ kind to λ^{rgnUL} terms of $\star$ and $\overline{\star}$ kinds, respectively. As we observed above, when we translate a $\mathsf{F}^{\mathsf{RGN}}$ type to a λ^{rgnUL} type, we ensure that the result is a U qualified type. The $\mathcal{T}_{\overline{\star}} [\![\tau]\!]$ translation is straightforward on the functional types. (However, note that a $\mathsf{F}^{\mathsf{RGN}}$ variable α of $\star$ kind is translated to a λ^{rgnUL} variable $\overline{\alpha}$ of $\overline{\star}$ kind; this ensures that *every* type corresponding to a $\mathsf{F}^{\mathsf{RGN}}$ type is manifestly qualified with U.)

More interesting are the translations of the types associated with the rgn monad. In the translation of the $\mathsf{rgn}\, s\, \tau$ type, we see the familiar store (stack) passing interpretation of computations. Since the representation of a stack of regions is linear, the resulting store/value pair is qualified with L. Next, consider the translation of the $\mathsf{pf}\, (s_1 \leq s_2)$ type. Recall that it is the type of witnesses to the fact that the stack indexed by s_1 is a subtype of the stack indexed by s_2. Hence, we translate to a type that expresses the isomorphism between $\mathcal{S}_\star [\![s_2]\!]$ and $^{\mathsf{L}}(\mathcal{S}_\star [\![s_1]\!] \otimes \beta)$, for some "slack" β. Note that while the types $\mathcal{S}_\star [\![s_2]\!]$, $\mathcal{S}_\star [\![s_1]\!]$, and β may be linear, the pair of functions witnessing the isomorphism is unrestricted. This corresponds to the fact that the *proof* that s_1 is a subtype of s_2 is *persistent*, while the *existence* of the stacks s_1 and s_2 are *ephemeral*.

$\mathsf{F}^{\mathsf{RGN}}$ STACK *to* λ^{rgnUL} $\star$
$\mathcal{S}_\star \llbracket \varsigma \rrbracket = \varsigma$

$\mathsf{F}^{\mathsf{RGN}}$ $\star$ *to* λ^{rgnUL} $\star$
$\mathcal{T}_\star \llbracket \tau \rrbracket = {}^{\mathsf{U}}\mathcal{T}_{\overline{\star}} \llbracket \tau \rrbracket$

$\mathsf{F}^{\mathsf{RGN}}$ $\star$ *to* λ^{rgnUL} $\overline{\star}$ *(functional types)*

$$\begin{aligned}
\mathcal{T}_{\overline{\star}} \llbracket \alpha \rrbracket &= \overline{\alpha} \\
\mathcal{T}_{\overline{\star}} \llbracket \tau_1 \to \tau_2 \rrbracket &= \mathcal{T}_\star \llbracket \tau_1 \rrbracket \multimap \mathcal{T}_\star \llbracket \tau_2 \rrbracket \\
\mathcal{T}_{\overline{\star}} \llbracket \mathbf{1} \rrbracket &= \mathbf{1}_\otimes \\
\mathcal{T}_{\overline{\star}} \llbracket \tau_1 \times \tau_2 \rrbracket &= \mathcal{T}_\star \llbracket \tau_1 \rrbracket \otimes \mathcal{T}_\star \llbracket \tau_2 \rrbracket \\
\mathcal{T}_{\overline{\star}} \llbracket \forall \alpha{:}\star \,.\, \tau \rrbracket &= \forall \overline{\alpha}{:}\overline{\star}.\, \mathcal{T}_\star \llbracket \tau \rrbracket \\
\mathcal{T}_{\overline{\star}} \llbracket \forall \varsigma{:}\mathsf{STACK}.\, \tau \rrbracket &= \forall \varsigma{:}\star \,.\, \mathcal{T}_\star \llbracket \tau \rrbracket
\end{aligned}$$

$\mathsf{F}^{\mathsf{RGN}}$ $\star$ *to* λ^{rgnUL} $\overline{\star}$ (rgn *monad types)*

$$\begin{aligned}
\mathcal{T}_{\overline{\star}} \llbracket \mathsf{rgn}\, s\, \tau \rrbracket &= \mathcal{S}_\star \llbracket s \rrbracket \multimap {}^{\mathsf{L}}(\mathcal{S}_\star \llbracket s \rrbracket \otimes \mathcal{T}_\star \llbracket \tau \rrbracket) \\
\mathcal{T}_{\overline{\star}} \llbracket \mathsf{pf}\, (s_1 \leq s_2) \rrbracket &= \exists \beta{:}\star \,.\, \mathbf{Iso}(\mathcal{S}_\star \llbracket s_2 \rrbracket \,, {}^{\mathsf{L}}(\mathcal{S}_\star \llbracket s_1 \rrbracket \otimes \beta)) \\
\mathcal{T}_{\overline{\star}} \llbracket \mathsf{hnd}\, s \rrbracket &= \exists \varrho{:}\mathsf{RGN}.\, {}^{\mathsf{U}}({}^{\mathsf{U}}\exists \beta{:}\star \,.\, \mathbf{Iso}(\mathcal{S}_\star \llbracket s \rrbracket \,, {}^{\mathsf{L}}(\beta \otimes {}^{\mathsf{L}}(\mathsf{cap}\, \varrho))) \otimes {}^{\mathsf{U}}(\mathsf{hnd}\, \varrho)) \\
\mathcal{T}_{\overline{\star}} \llbracket \mathsf{ref}\, s\, \tau \rrbracket &= \exists \varrho{:}\mathsf{RGN}.\, {}^{\mathsf{U}}({}^{\mathsf{U}}\exists \beta{:}\star \,.\, \mathbf{Iso}(\mathcal{S}_\star \llbracket s \rrbracket \,, {}^{\mathsf{L}}(\beta \otimes {}^{\mathsf{L}}(\mathsf{cap}\, \varrho))) \otimes {}^{\mathsf{U}}(\mathsf{ref}\, \varrho\, \mathcal{T}_\star \llbracket \tau \rrbracket))
\end{aligned}$$

λ^{rgnUL} *Type-level Macros*

$$\mathbf{Iso}(\tau_1, \tau_2) = {}^{\mathsf{U}}({}^{\mathsf{U}}(\tau_1 \multimap \tau_2) \otimes {}^{\mathsf{U}}(\tau_2 \multimap \tau_1))$$

Fig. 4. Type-level Translation

The translation of the $\mathsf{hnd}\, s$ and $\mathsf{ref}\, s\, \tau$ types are similar. An existentially bound region name ϱ fixes the region for the λ^{rgnUL} handle or reference, while an isomorphism witnesses the fact that ϱ may be found within the stack $\mathcal{S}_\star \llbracket s \rrbracket$.

With the translation of $\mathsf{F}^{\mathsf{RGN}}$ type-level terms in place, the translation of $\mathsf{F}^{\mathsf{RGN}}$ expressions follows almost directly. We elide the translation of the introduction and elimination forms for the functional types in $\mathsf{F}^{\mathsf{RGN}}$ (it is simply the homomorphic mapping of the given expression translations) and focus on the translation of the rgn monad operations. For readability, we give translations for fully applied region primitives only, assuming that partially applied primitives have been eta-expanded. The translation of `return` and `then` follow directly from our store (stack) passing interpretation of $\mathsf{rgn}\, s\, \tau$ types:

$$\begin{aligned}
&\mathcal{E} \llbracket \texttt{return}\, [s]\, [\tau_\alpha]\, e \rrbracket \;= \\
&\quad \texttt{let}\ \mathit{res}{:}\mathcal{T}_\star \llbracket \tau_\alpha \rrbracket = \mathcal{E} \llbracket e \rrbracket\ \texttt{in} \\
&\quad {}^{\mathsf{U}}\lambda \mathit{stk}{:}\mathcal{S}_\star \llbracket s \rrbracket \,.\, {}^{\mathsf{L}}\langle \mathit{stk}, \mathit{res} \rangle
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E} \llbracket \texttt{then}\, [s]\, [\tau_\alpha]\, [\tau_\beta]\, e_1\, e_2 \rrbracket \;= \\
&\quad \texttt{let}\ f{:}\mathcal{T}_\star \llbracket \mathsf{rgn}\, s\, \tau_\alpha \rrbracket = \mathcal{E} \llbracket e_1 \rrbracket\ \texttt{in} \\
&\quad \texttt{let}\ g{:}\mathcal{T}_\star \llbracket \tau_\alpha \to \mathsf{rgn}\, s\, \tau_\beta \rrbracket = \mathcal{E} \llbracket e_2 \rrbracket\ \texttt{in} \\
&\quad {}^{\mathsf{U}}\lambda \mathit{stk}{:}\mathcal{S}_\star \llbracket s \rrbracket \,.\, \texttt{let}\ \langle \mathit{stk}, \mathit{res} \rangle = f\ \mathit{stk}\ \texttt{in} \\
&\qquad\qquad g\ \mathit{res}\ \mathit{stk}
\end{aligned}$$

The translation of `letRgn` is the most complicated, but breaks down into conceptually simple components. We bracket the execution of the inner computation with a `newrgn`/`freergn` pair, creating and destroying a new region. We construct the representation of the new stack stk_2 for the inner computation by pairing the old stack stk_1 with the new region capability *cap*. Finally, we construct isomorphisms witnessing the relationships between the new region capability and the new stack and between the old stack and the new stack. We carefully chose the isomorphism types so that the identity function suffices as a witness. Putting all of these pieces together, we have the following:

$$
\begin{array}{l}
\mathcal{E}\,[\![\texttt{letRgn}\,[s_1]\,[\tau_\alpha]\,e]\!] \;=\; \\
\quad \texttt{let}\ f{:}\mathcal{T}_\star\,[\![\forall\varsigma_2{:}\mathsf{STACK}.\,\mathsf{pf}\,(s_1 \le \varsigma_2) \to \mathsf{hnd}\,\varsigma_2 \to \mathsf{rgn}\,\varsigma_2\,\tau_\alpha]\!] = \mathcal{E}\,[\![e]\!]\ \texttt{in} \\
\quad {}^{\mathsf{U}}\lambda stk_1{:}\mathcal{S}_\star\,[\![s_1]\!]\,.\,\texttt{let}\ \texttt{pack}(\varrho{:}\mathsf{RGN}, \langle cap, hnd\rangle) = \texttt{newrgn}\,{}^{\mathsf{L}}\langle\rangle\ \texttt{in} \\
\qquad\qquad \texttt{let}\ id = {}^{\mathsf{U}}\lambda stk{:}{}^{\mathsf{L}}(\mathcal{S}_\star\,[\![s_1]\!] \otimes {}^{\mathsf{L}}\mathsf{cap}\,\varrho).\ stk\ \texttt{in} \\
\qquad\qquad \texttt{let}\ ppf = {}^{\mathsf{U}}\texttt{pack}({}^{\mathsf{L}}(\mathsf{cap}\,\varrho){:}\star, {}^{\mathsf{U}}\langle id, id\rangle)\ \texttt{in} \\
\qquad\qquad \texttt{let}\ phnd = {}^{\mathsf{U}}\texttt{pack}(\varrho{:}\mathsf{RGN}, {}^{\mathsf{U}}\langle {}^{\mathsf{U}}\texttt{pack}(\mathcal{S}_\star\,[\![s_1]\!]{:}\star, {}^{\mathsf{U}}\langle id, id\rangle), hnd\rangle)\ \texttt{in} \\
\qquad\qquad \texttt{let}\ stk_2 = {}^{\mathsf{L}}\langle stk_1, cap\rangle\ \texttt{in} \\
\qquad\qquad \texttt{let}\ \langle stk_2, res\rangle = f\,[{}^{\mathsf{L}}(\mathcal{S}_\star\,[\![s_1]\!] \otimes {}^{\mathsf{L}}(\mathsf{cap}\,\varrho))]\ ppf\ phnd\ stk_2\ \texttt{in} \\
\qquad\qquad \texttt{let}\ \langle stk_1, cap\rangle = stk_2\ \texttt{in} \\
\qquad\qquad \texttt{let}\ \langle\rangle = \texttt{freergn}\,[\varrho]\,{}^{\mathsf{L}}\langle cap, hnd\rangle\ \texttt{in} \\
\qquad\qquad {}^{\mathsf{L}}\langle stk_1, res\rangle
\end{array}
$$

We can see the isomorphisms in action in the translation of `coerceRgn`:

$$
\begin{array}{l}
\mathcal{E}\,[\![\texttt{coerceRgn}\,[s_1]\,[s_2]\,[\tau_\alpha]\,e_1\,e_2]\!] \;=\; \\
\quad \texttt{let}\ ppf{:}\mathcal{T}_\star\,[\![\mathsf{pf}\,(s_1 \le s_2)]\!] = \mathcal{E}\,[\![e_1]\!]\ \texttt{in} \\
\quad \texttt{let}\ f{:}\mathcal{T}_\star\,[\![\mathsf{rgn}\,s_1\,\tau_\alpha]\!] = \mathcal{E}\,[\![e_2]\!]\ \texttt{in} \\
\quad {}^{\mathsf{U}}\lambda stk_2{:}\mathcal{S}_\star\,[\![s_2]\!]\,.\,\texttt{let}\ \texttt{pack}(\beta{:}\star, \langle spl, cmb\rangle) = ppf\ \texttt{in} \\
\qquad\qquad \texttt{let}\ \langle stk_1, stk_\beta\rangle = spl\ stk_2\ \texttt{in} \\
\qquad\qquad \texttt{let}\ \langle stk_1, res\rangle = f\ stk_1\ \texttt{in} \\
\qquad\qquad \texttt{let}\ stk_2 = cmb\,{}^{\mathsf{L}}\langle stk_1, stk_\beta\rangle\ \texttt{in} \\
\qquad\qquad {}^{\mathsf{L}}\langle stk_2, res\rangle
\end{array}
$$

Note how the the stack "slack" stk_β is split out and then combined in, bracketing the execution of the $\mathsf{rgn}\,s_1\,\tau_\alpha$ computation.

As a final example, we can see an "empty" stack (represented by a ${}^{\mathsf{L}}\mathbf{1}_\otimes$ value) being provided as the initial stack in the translation of `runRgn`:

$$
\begin{array}{l}
\mathcal{E}\,[\![\texttt{runRgn}\,[\tau_\alpha]\,e]\!] \;=\; \\
\quad \texttt{let}\ f{:}\mathcal{T}_\star\,[\![\forall\varsigma{:}\mathsf{STACK}.\,\mathsf{rgn}\,\varsigma\,\tau_\alpha]\!] = \mathcal{E}\,[\![e]\!]\ \texttt{in} \\
\quad \texttt{let}\ \langle\langle\rangle, res\rangle = f\,[{}^{\mathsf{L}}\mathbf{1}_\otimes]\,{}^{\mathsf{L}}\langle\rangle\ \texttt{in}\ res
\end{array}
$$

The translations of the remaining rgn monad primitives are given in Figure 5. We strongly believe, but have not mechanically verified, that the translation is type preserving.

5 Extensions

The primary advantage of working at the target level is that we can expose the capabilities for regions as first-class objects instead of indirectly manipulating a stack of regions. In turn, this allows us to avoid the last-in-first-out lifetimes dictated by a lexically-scoped `letRgn`. For example, we can now explain the semantics for Cyclone's *dynamic regions* and *unique pointers* using the concepts in the target language.

Dynamic Regions In Cyclone, a dynamic region r is represented by a *key* ($\mathsf{key}\,r$) which is treated linearly by the type system. At the target level, a key can be represented by a pair of the capability for the region and its handle:

$$\mathsf{key}\,r = {}^{\mathsf{L}}({}^{\mathsf{L}}\mathsf{cap}\,r \otimes {}^{\mathsf{U}}\mathsf{hnd}\,r)$$

Then creating a new key is accomplished by calling `newrgn`, and destroying the key is accomplished by calling `freergn`.

$$
\begin{array}{l}
\mathcal{E}\,[\![\texttt{new}\,[s]\,[\tau_\alpha]\,e_1\,e_2]\!] \;=\\
\quad \texttt{let}\ \mathit{phnd}{:}\mathcal{T}_\star\,[\![\mathsf{hnd}\,s]\!] = \mathcal{E}\,[\![e_1]\!]\ \texttt{in}\\
\quad \texttt{let}\ x{:}\mathcal{T}_\star\,[\![\tau_\alpha]\!] = \mathcal{E}\,[\![e_2]\!]\ \texttt{in}\\
\quad {}^{\mathsf{U}}\lambda \mathit{stk}{:}\mathcal{S}_\star\,[\![s]\!]\,.\,\texttt{let}\ \mathtt{pack}(\varrho{:}\mathsf{RGN}, \langle \mathtt{pack}(\beta{:}\star, \langle \mathit{prj}, \mathit{inj}\rangle), \mathit{hnd}\rangle) = \mathit{phnd}\ \texttt{in}\\
\qquad\qquad \texttt{let}\ \langle \mathit{stk}_\beta, \mathit{cap}\rangle = \mathit{prj}\ \mathit{stk}\ \texttt{in}\\
\qquad\qquad \texttt{let}\ \langle \mathit{cap}, \mathit{ref}\rangle = \texttt{new}\,[\varrho]\,[\mathcal{T}_\star\,[\![\tau_\alpha]\!]]\,{}^{\mathsf{L}}\langle \mathit{cap}, \mathit{hnd}, x\rangle\ \texttt{in}\\
\qquad\qquad \texttt{let}\ \mathit{pref} = {}^{\mathsf{U}}\mathtt{pack}(\rho{:}\mathsf{RGN}, {}^{\mathsf{U}}\langle {}^{\mathsf{U}}\mathtt{pack}(\beta{:}\star, {}^{\mathsf{U}}\langle \mathit{prj}, \mathit{inj}\rangle), \mathit{ref}\rangle)\ \texttt{in}\\
\qquad\qquad \texttt{let}\ \mathit{stk} = \mathit{inj}\,{}^{\mathsf{L}}\langle \mathit{stk}_\beta, \mathit{cap}\rangle\ \texttt{in}\\
\qquad\qquad {}^{\mathsf{L}}\langle \mathit{stk}, \mathit{pref}\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}\,[\![\texttt{read}\,[s]\,[\tau_\alpha]\,e]\!] \;=\\
\quad \texttt{let}\ \mathit{pref}{:}\mathcal{T}_\star\,[\![\mathsf{ref}\,s\,\tau_\alpha]\!] = \mathcal{E}\,[\![e]\!]\ \texttt{in}\\
\quad {}^{\mathsf{U}}\lambda \mathit{stk}{:}\mathcal{S}_\star\,[\![s]\!]\,.\,\texttt{let}\ \mathtt{pack}(\varrho{:}\mathsf{RGN}, \langle \mathtt{pack}(\beta{:}\star, \langle \mathit{prj}, \mathit{inj}\rangle), \mathit{ref}\rangle) = \mathit{pref}\ \texttt{in}\\
\qquad\qquad \texttt{let}\ \langle \mathit{stk}_\beta, \mathit{cap}\rangle = \mathit{prj}\ \mathit{stk}\ \texttt{in}\\
\qquad\qquad \texttt{let}\ \langle \mathit{cap}, \mathit{res}\rangle = \texttt{read}\,[\varrho]\,[\mathcal{T}_\star\,[\![\tau_\alpha]\!]]\,{}^{\mathsf{L}}\langle \mathit{cap}, \mathit{ref}\rangle\ \texttt{in}\\
\qquad\qquad \texttt{let}\ \mathit{stk} = \mathit{inj}\,{}^{\mathsf{L}}\langle \mathit{stk}_\beta, \mathit{cap}\rangle\ \texttt{in}\\
\qquad\qquad {}^{\mathsf{L}}\langle \mathit{stk}, \mathit{res}\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}\,[\![\texttt{write}\,[s]\,[\tau_\alpha]\,e_1\,e_2]\!] \;=\\
\quad \texttt{let}\ \mathit{pref}{:}\mathcal{T}_\star\,[\![\mathsf{ref}\,s\,\tau_\alpha]\!] = \mathcal{E}\,[\![e_1]\!]\ \texttt{in}\\
\quad \texttt{let}\ x{:}\mathcal{T}_\star\,[\![\tau_\alpha]\!] = \mathcal{E}\,[\![e_2]\!]\ \texttt{in}\\
\quad {}^{\mathsf{U}}\lambda \mathit{stk}{:}\mathcal{S}_\star\,[\![s]\!]\,.\,\texttt{let}\ \mathtt{pack}(\varrho{:}\mathsf{RGN}, \langle \mathtt{pack}(\beta{:}\star, \langle \mathit{prj}, \mathit{inj}\rangle), \mathit{ref}\rangle) = \mathit{pref}\ \texttt{in}\\
\qquad\qquad \texttt{let}\ \langle \mathit{stk}_\beta, \mathit{cap}\rangle = \mathit{prj}\ \mathit{stk}\ \texttt{in}\\
\qquad\qquad \texttt{let}\ \langle \mathit{cap}, \mathit{res}\rangle = \texttt{write}\,[\varrho]\,[\mathcal{T}_\star\,[\![\tau_\alpha]\!]]\,{}^{\mathsf{L}}\langle \mathit{cap}, \mathit{ref}, x\rangle\ \texttt{in}\\
\qquad\qquad \texttt{let}\ \mathit{stk} = \mathit{inj}\,{}^{\mathsf{L}}\langle \mathit{stk}_\beta, \mathit{cap}\rangle\ \texttt{in}\\
\qquad\qquad {}^{\mathsf{L}}\langle \mathit{stk}, \mathit{res}\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}\,[\![\texttt{reflSub}\,[s]]\!] \;=\\
\quad \texttt{let}\ \mathit{spl} = {}^{\mathsf{U}}\lambda \mathit{stk}{:}\mathcal{S}_\star\,[\![s]\!]\,.\,\texttt{let}\ \mathit{su} = {}^{\mathsf{L}}\langle \mathit{stk}, {}^{\mathsf{L}}\langle\rangle\rangle\ \texttt{in}\ \mathit{su}\ \texttt{in}\\
\quad \texttt{let}\ \mathit{cmb} = {}^{\mathsf{U}}\lambda \mathit{su}{:}{}^{\mathsf{L}}(\mathcal{S}_\star\,[\![s]\!] \otimes {}^{\mathsf{L}}\mathbf{1}_\otimes).\,\texttt{let}\ \langle \mathit{stk}, \langle\rangle\rangle = \mathit{su}\ \texttt{in}\ \mathit{stk}\ \texttt{in}\\
\quad {}^{\mathsf{U}}\mathtt{pack}({}^{\mathsf{L}}\mathbf{1}_\otimes{:}\star, {}^{\mathsf{U}}\langle \mathit{spl}, \mathit{cmb}\rangle)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{E}\,[\![\texttt{transSub}\,[s_1]\,[s_2]\,[s_3]\,e_{1\rightsquigarrow 2}\,e_{2\rightsquigarrow 3}]\!] \;=\\
\quad \texttt{let}\ \mathit{ppf}_{1\rightsquigarrow 2}{:}\mathcal{T}_\star\,[\![\mathsf{pf}\,(s_1 \le s_2)]\!] = \mathcal{E}\,[\![e_{1\rightsquigarrow 2}]\!]\ \texttt{in}\\
\quad \texttt{let}\ \mathit{ppf}_{2\rightsquigarrow 3}{:}\mathcal{T}_\star\,[\![\mathsf{pf}\,(s_2 \le s_3)]\!] = \mathcal{E}\,[\![e_{2\rightsquigarrow 3}]\!]\ \texttt{in}\\
\quad \texttt{let}\ \mathtt{pack}(\alpha{:}\star, \langle \mathit{spl}_{2\rightsquigarrow 1\otimes\alpha}, \mathit{cmp}_{1\otimes\alpha\rightsquigarrow 2}\rangle) = \mathit{ppf}_{1\rightsquigarrow 2}\ \texttt{in}\\
\quad \texttt{let}\ \mathtt{pack}(\beta{:}\star, \langle \mathit{spl}_{3\rightsquigarrow 2\otimes\beta}, \mathit{cmp}_{2\otimes\beta\rightsquigarrow 3}\rangle) = \mathit{ppf}_{2\rightsquigarrow 3}\ \texttt{in}\\
\quad \texttt{let}\ \mathit{spl} = {}^{\mathsf{U}}\lambda \mathit{stk}_3{:}\mathcal{S}_\star\,[\![s_3]\!]\,.\,\texttt{let}\ \langle \mathit{stk}_2, \mathit{stk}_\beta\rangle = \mathit{spl}_{3\rightsquigarrow 2\otimes\beta}\ \mathit{stk}_3\ \texttt{in}\\
\qquad\qquad\qquad\qquad \texttt{let}\ \langle \mathit{stk}_1, \mathit{stk}_\alpha\rangle = \mathit{spl}_{2\rightsquigarrow 1\otimes\alpha}\ \mathit{stk}_2\ \texttt{in}\\
\qquad\qquad\qquad\qquad \texttt{let}\ \mathit{sss} = {}^{\mathsf{L}}\langle \mathit{stk}_1, {}^{\mathsf{L}}\langle \mathit{stk}_\beta, \mathit{stk}_\alpha\rangle\rangle\ \texttt{in}\\
\qquad\qquad\qquad\qquad \mathit{sss}\qquad \texttt{in}\\
\quad \texttt{let}\ \mathit{cmb} = {}^{\mathsf{U}}\lambda \mathit{sss}{:}{}^{\mathsf{L}}(\mathcal{S}_\star\,[\![s_1]\!] \otimes {}^{\mathsf{L}}(\beta \otimes \alpha)).\,\texttt{let}\ \langle \mathit{stk}_1, \langle \mathit{stk}_\beta, \mathit{stk}_\alpha\rangle\rangle = \mathit{sss}\ \texttt{in}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{let}\ \mathit{stk}_2 = \mathit{cmb}_{1\otimes\alpha\rightsquigarrow 2}\,{}^{\mathsf{L}}\langle \mathit{stk}_1, \mathit{stk}_\alpha\rangle\ \texttt{in}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{let}\ \mathit{stk}_3 = \mathit{cmb}_{2\otimes\beta\rightsquigarrow 3}\,{}^{\mathsf{L}}\langle \mathit{stk}_2, \mathit{stk}_\beta\rangle\ \texttt{in}\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{stk}_3\qquad \texttt{in}\\
\quad {}^{\mathsf{U}}\mathtt{pack}({}^{\mathsf{L}}(\beta \otimes \alpha){:}\star, {}^{\mathsf{U}}\langle \mathit{spl}, \mathit{cmb}\rangle)
\end{array}
$$

Fig. 5. Translation of rgn Monad Operations

To access a value allocated in a dynamic region, or to allocate a value in a dynamic region, Cyclone requires that the region be *opened* by presenting its key. The `openDRgn` is similar to a `letRgn` in that it conceptually pushes the dynamic region onto the stack of regions, executes the body, and then pops the region off the stack. During execution of the `openDRgn`'s body, the key becomes inaccessible, ensuring that the region cannot be deallocated. At the end of the `openDRgn` scope, the key is given back. The programmer is then able to destroy the region or later re-open it.

The `openDRgn` primitive can be implemented as a higher-order function with a signature like this (eliding the L and U qualifiers, and using source-level rgn to abbreviate the store passing translation):

$$\texttt{openDRgn} : \forall \varrho, \varsigma, \alpha.\mathsf{key}\ \varrho \multimap (\mathsf{hnd}\ \varrho \multimap \mathsf{rgn}\ (\varsigma \otimes \mathsf{cap}\ \varrho)\ \alpha) \multimap \mathsf{rgn}\ \varsigma\ (\alpha \otimes \mathsf{key}\ \varrho)$$

The function takes the key for ϱ and a computation, which, when given the handle for ϱ, expects to run on a stack of the form $\varsigma \otimes \mathsf{cap}\ \varrho$ for some ς. Once applied, `openDRgn` returns a computation, which, when run on a stack ς, opens up the key to get the capability and handle, pushes the capability for ϱ on the stack, passes the handle to the computation and runs it in the extended stack to produce an α value. Then, it pops the capability, and returns a (linear) pair of the result and the re-packaged key. (We leave the definition of `openDRgn` as an exercise for the reader.)

Furthermore, since keys are first-class objects, they can be placed in data structures. For example, in our space-conscious web server [3], we use a list of dynamic regions, each of which holds data corresponding to a particular connection. When we receive data from a connection, we find the corresponding key, open it up, and then place the data in the region. When a connection is terminated, we pull the corresponding key out of the queue, perform `freeDRgn` on it, and thus deallocate all of the data associated with the connection. The price paid for this flexibility is that the list of keys must be treated linearly to avoid creating multiple aliases to the keys.

Unique Pointers Cyclone's unique pointers are anonymous dynamic regions, without the handle. Like the keys of dynamic regions, they can be destroyed at any time and the type system treats them linearly. At the target level, a unique pointer to a τ object can be represented as a term with type:

$$^{\mathsf{L}}\exists\varrho.\,^{\mathsf{L}}(^{\mathsf{L}}\mathsf{cap}\ \varrho \otimes\ ^{\mathsf{U}}(^{\mathsf{L}}\mathsf{cap}\ \varrho \multimap\ ^{\mathsf{U}}\mathbf{1}_{\otimes}) \otimes\ ^{\mathsf{U}}(\mathsf{ref}\ \varrho\,\tau))$$

Note that the actual reference is unrestricted, whereas the capability is linear; the handle is not available for further allocations, but is caught up in the function closure, which may be applied to free the region. This encoding allows us to "open" a unique pointer, just as we do dynamic regions, and for a limited scope, freely access (and duplicate) the underlying reference. Of course, during the scope of the open, we temporarily lose the capability to deallocate the object, but regain the capability upon exit from the scope.

In practice, the ability to open dynamic regions and unique pointers has proven crucial for integrating these facilities into the language. They make it relatively easy to access a data structure and mitigate some of the pain of threading linear resources through the program. Furthermore, they make it possible

to write re-usable libraries that manipulate data allocated in lexically-scoped regions, dynamic regions, or as unique objects.

Phase-Splitting In our target language, we represented capabilities and proof witnesses as explicit terms. But we have also crafted the language and translation so that these values are never actually needed at run-time. For instance, our witnesses only manipulate (products of) capabilities, which are themselves operationally irrelevant. These objects are only used to make the desired safety properties easy to check statically. So in principle, we should be able to erase the capabilities and witnesses before running a program.

To realize this goal, we should introduce a phase distinction via another modality, where we treat capabilities and proof witnesses as static objects, and all other terms as dynamic. The modality would demand that, as usual, static computations cannot depend upon dynamic values. Furthermore, we must be sure that witness functions (i.e., proof objects) are in fact total, effect-free functions so that they and their applications to capabilities may be safely erased. This sort of phase-splitting is effectively used in other settings that mix programming languages and logics, such as Xi *et al.*'s Applied Type System [12] and Sheard's Omega [13]. Perhaps the most promising approach is suggested by Mandelbaum, Walker, and Harper's work [14], where they developed a two-level language for reasoning about effectful programs.

The primary reason we did not introduce phase splitting here is that it complicates the translation and the target language, and thus obscures what is actually a relatively simple and straightforward encoding. A secondary reason is that, as demonstrated by the cited work above, there are many domains that would benefit from a general solution to the problem of type relevant, but operationally irrelevant, values.

6 Related Work and Open Issues

There has been much prior work aimed at relaxing the stack discipline imposed on region lifetimes by the Tofte-Talpin (TT) approach. The ML Kit [15] uses a storage-mode analysis to determine when it is safe to deallocate data in a region (known as region resetting) prior to the deallocation of the region itself. The safety of the storage-mode analysis has not been established formally.

Aiken *et al.* [16] eliminate the requirement that region allocation and deallocation should coincide with the beginning and end of the scope of region variables introduced by the `letregion` construct. They use a *late allocation/early deallocation* approach that delays the allocation of a region until just before its first access, and deallocates the region just after its last access. We believe that the results of their analysis can be encoded explicitly in our target language.

Unlike the previous two approaches which build on TT, Henglein *et al.* [17] present a region system that (like ours) replaces the `letregion` primitive with explicit commands to create and free a region. To ensure safety, they use a Hoare-logic-based region type system and consequently have no support for higher-order

functions. While they provide an inference algorithm to annotate programs with region manipulation commands, we intend for our system to serve as a target language for programs annotated using TT region inference, or those written in languages like Cyclone. The Calculus of Capabilities [18] is also intended as a target for TT-annotated programs, but unlike λ^{rgnUL}, it is defined in terms of a continuation-passing style language and does not support first-class regions.

The region system presented by Walker and Watkins [19] is perhaps the most closely related work. Like our target, they require a linear capability to be presented upon each access to a region. However, they provide a primitive, similar to `letregion`, that allows a capability to be temporarily treated as unrestricted for convenience's sake. We have shown that no such primitive is needed. Rather, we use a combination of monadic encapsulation (to thread capabilities) coupled with *unrestricted witnesses* to achieve the same flexibility. In particular, our open construct for dynamic regions (and unique pointers) achieves the same effect as the Walker-Watkin's primitive.

A related body of work has used regions as a low-level primitive on which to build type-safe garbage collectors [20–22]. Each of these approaches requires non-lexical regions, since, in a copying collector, the from- and to-spaces have non-nested lifetimes. Hawblitzel *et al.* [22] introduce a very low-level language in which they begin with a single linear array of words, construct lists and arrays out of the basic linear memory primitives, introduce *type sequences* for building regions of nonlinear data. Such a foundational approach is admirable, but there is a large semantic gap between a high-level language and such a target. Hence, λ^{rgnUL} serves as a useful intermediate point, and we may envision further translation from λ^{rgnUL} to such a low-level language.

The Vault language [23, 24] includes many of the features described in our target, including linear capabilities for accessing resources and a mechanism, called adoption, for temporarily transferring ownership of a capability to another capability, for a limited scope. But Vault also includes support for strong (i.e., type-changing) updates on linear resources, as well as features for temporarily treating an unrestricted resource as if it were linear. On the other hand, to the best of our knowledge, there exists no formal model that justifies the soundness of all of these mechanisms. We believe that it may be possible to combine λ^{rgnUL} with our previous work on strong updates [25, 26] to achieve this.

References

1. Cyclone, version 0.9. (2005) http://www.eecs.harvard.edu/ greg/cyclone/.
2. Tofte, M., Talpin, J.P.: Region-based memory management. Information and Computation **132**(2) (1997) 109–176
3. Hicks, M., Morrisett, G., Grossman, D., Jim, T.: Experience with safe manual memory-management in Cyclone. In: Proc. International Symposium on Memory Management. (2004) 73–84
4. Fluet, M., Wang, D.: Implementation and performance evaluation of a safe runtime system in Cyclone. In: Proc. SPACE Workshop. (2004)

5. Grossman, D., Morrisett, G., Jim, T., Hicks, M., Wang, Y., Cheney, J.: Region-based memory management in Cyclone. In: Proc. Programming Language Design and Implementation. (2002) 282–293
6. Fluet, M., Morrisett, G.: Monadic regions. In: Proc. International Conference on Functional Programming. (2004) 103–114
7. Launchbury, J., Peyton Jones, S.: State in Haskell. Lisp and Symbolic Computation **8**(4) (1995) 293–341
8. Ahmed, A., Fluet, M., Morrisett, G.: A step-indexed model of substructural state. In: Proc. International Conference on Functional Programming. (2005) 78–91
9. Walker, D.: Substructural type systems. In Pierce, B., ed.: Advanced Topics in Types and Programming Languages. MIT Press, Cambridge, MA (2005) 3–43
10. Pfenning, F., Schürmann, C.: Twelf – a meta-logic framework for deductive systems. In: Proc. Conference on Automated Deduction. (1999) 202–206
11. Schürmann, C., Pfenning, F.: A coverage checking algorithm for LF. In: Proc. Theorem Proving in Higher Order Logics. (2003) 120–135 LNCS 2758.
12. Chen, C., Xi, H.: Combining programming with theorem proving. In: Proc. International Conference on Functional Programming. (2005) 66–77
13. Sheard, T., Pasalic, E.: Meta-programming with built-in type equality (extended abstract). In: International Workshop on Logical Frameworks and Meta-Languages. (2004)
14. Mandelbaum, Y., Walker, D., Harper, R.: An effective theory of type refinements. In: Proc. International Conference on Functional Programming. (2003) 213–225
15. Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N., Olesen, T.H., Sestoft, P.: Programming with regions in the ML Kit (for version 4). Technical report, IT University of Copenhagen (2002)
16. Aiken, A., Fähndrich, M., Levien, R.: Better static memory management: Improving region-based analysis of higher-order languages. In: Proc. Programming Language Design and Implementation. (1995) 174–185
17. Henglein, F., Makholm, H., Niss, H.: A direct approach to control-flow sensitive region-based memory management. In: Proc. Principles and Practice of Declarative Programming. (2001) 175–186
18. Walker, D., Crary, K., Morrisett, G.: Typed memory management in a calculus of capabilities. ACM Transactions on Programming Languages and Systems **24**(4) (2000) 701–771
19. Walker, D., Watkins, K.: On regions and linear types. In: Proc. International Conference on Functional Programming. (2001) 181–192
20. Wang, D., Appel, A.: Type-preserving garbage collectors. In: Proc. Principles of Programming Languages. (2001) 166–178
21. Monnier, S., Saha, B., Shao, Z.: Principled scavenging. In: Proc. Programming Language Design and Implementation. (2001) 81–91
22. Hawblitzel, C., Wei, E., Huang, H., Krupski, E., Wittie, L.: Low-level linear memory management. In: Proc. SPACE Workshop. (2004)
23. DeLine, R., Fähndrich, M.: Enforcing high-level protocols in low-level software. In: Proc. Programming Language Design and Implementation. (2001) 59–69
24. Fähndrich, M., DeLine, R.: Adoption and focus: Practical linear types for imperative programming. In: Proc. Programming Language Design and Implementation. (2002) 13–24
25. Morrisett, G., Ahmed, A., Fluet, M.: $\mathbf{L}^3$: A linear language with locations. In: Proc. Typed Lambda Calculi and Applications. (2005) 293–307
26. Ahmed, A., Fluet, M., Morrisett, G.: $\mathbf{L}^3$: A linear language with locations. Technical Report TR-24-04, Harvard University (2004)