Spegion: Implicit and Non-Lexical Regions with Sized Allocations

Jack Hughes
School of Computing, University of Kent, Canterbury, UK

Michael Vollmer
School of Computing, University of Kent, Canterbury, UK

Mark Batty
School of Computing, University of Kent, Canterbury, UK

Abstract

Region based memory management is a powerful tool designed with the goal of ensuring memory safety statically. The region calculus of Tofte and Talpin is a well known example of a region based system, which uses regions to manage memory in a stack-like fashion. However, the region calculus is lexically scoped and requires explicit annotation of memory regions, which can be cumbersome for the programmer. Other systems have addressed non-lexical regions, but these approaches typically require the use of a substructural type system to track the lifetimes of regions. We present Spegion, a language with implicit non-lexical regions, which provides these same memory safety guarantees for programs that go beyond using memory allocation in a stack-like manner. We are able to achieve this with a concise syntax, and without the use of substructural types, relying instead on an effect system to enforce constraints on region allocation and deallocation. These regions may be divided into sub-regions, i.e., **Sp**littable r**Egion**s, allowing fine grained control over memory allocation. Furthermore, Spegion permits *sized* allocations, where each value has an associated size which is used to ensure that regions are not over-allocated into. We present a type system for Spegion and prove it is type safe with respect to a small-step operational semantics.

2012 ACM Subject Classification Software and its engineering → Allocation / deallocation strategies; Theory of computation → Type theory

Keywords and phrases Regions, Type Systems, Effect Systems, Programming Languages, Memory

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.15

Related Version *Full Version*: `https://arxiv.org/abs/2506.02182`

Funding This work is supported by the EPSRC under grants *EP/X021173/1* and *EP/V000470/1*.

1 Introduction

Writing error-free code is hard. Writing error-free systems code is even harder. Possibly the most common source of bugs in systems programming code is the traditional approach of manual memory management. Use-after-free bugs, where a program attempts to access memory that has been deallocated, are pervasive in C code, and can lead to undefined behaviour, crashes, and security vulnerabilities. Nevertheless, manual memory management is still widely used in systems code as it provides the programmer with fine-grained control over memory usage. A vast swathe of research has been dedicated to the problem of memory safety in systems programming languages. One approach is to use a garbage collector to automatically manage memory. However, garbage collectors can introduce unpredictable pauses in the execution of a program, which is unacceptable in systems programming where performance is critical. Another approach is to ensure memory safety statically using a type system, one of the most well-studied verification tools in computer science.

The region calculus of Tofte and Talpin [29] is a type system designed for ensuring memory safety in ML [27] through the use of memory *regions*. Memory is divided into regions which are allocated in a stack-like manner, using the language's **letregion** construct. This allows deallocation to be done in a single operation by popping the topmost region from the stack:

letregion ρ **in** e

The memory region ρ is created and pushed onto the region stack. After the evaluation of **letregion**, ρ is popped from the stack, deallocating any memory allocated during the evaluation of e. Region based memory management has since been applied to languages other than ML, such as Java [6,9] and Prolog [21]. However, these region calculi are lexically scoped, restricting the programmer from expressing more complex memory usage patterns that may arise in code. Lexically-scoped regions are tied to the static structure of the program and are deallocated in a stack-like manner. This limits flexibility in memory management, as regions cannot be split or partially deallocated. Consider the following C program, for example: Here

```
struct task {int n; struct task* next;};

struct task* build_tasks () {
    struct task* t      = malloc(sizeof(struct task));
    t->next             = malloc(sizeof(struct task));
    t->next->next       = malloc(sizeof(struct task));
    t->next->next->n    = 1;
    t->next->next->next = NULL;
    return t;
  }

int main () {
  struct task* t_head = build_tasks();
  struct task* t_last = t_head->next->next;
  free(t_head->next);
  free(t_head);
  return t_last->n;
}
```

Figure 1 Non-lexically-scoped memory use.

we have a linked list of tasks, where each task has a pointer to the next task. The function `build_tasks` allocates three tasks and links them together. The main function then frees the first two tasks, leaving the third task dangling. Traditional region calculi are unable to express this pattern as the regions are allocated and de-allocated non-lexically.

There has been prior work exploring the safe use of non-lexically scoped regions, though this work has often focused on inferring region annotations for functional programs, or involved the use of substructural types (requiring that code is written to use memory linearly or avoid aliasing, for example), or imposed constraints on the runtime of programs like requiring reference counting. This research is covered in detail in Section 6.2.

We present an approach to non-lexically scoped region which allows for the same memory safety guarantees as prior systems, without the need for linear or substructural types and with a concise syntax. We refer to these as *implicit* regions, as they are not first-class values in the language's syntax and are not tied to the lexical structure of the program. Our system, named Spegion, removes the need for first class regions in the language's source syntax (with the exception of polymorphic constructs). Instead, when allocating a piece of data, the programmer needs only to provide an existing value in the region they wish to allocate the new value into. For example,

let $x =$ **newrgn in**
  v **at** x
  freergn x;

The **newrgn** construct returns a null pointer for a new region, which is then used to allocate v into that region. The type system then ensures that the region associated with x is live when v is allocated into it. Eliminating first-class regions from the syntax provides natural compatibility with C-like languages, where regions are not first-class values. Spegion sidesteps the need to retrofit complex region syntax onto C by using a special syntactic form for allocation (v **at** x) to determine which region to allocate a value into. Moreover, implicit regions eliminate the need for substructural type constraints, a common source of complexity in region type systems and a feature which C does not support.

Our non-lexical approach to regions in Spegion also permits the splitting of regions into sub-regions, allowing unused memory to be reclaimed without deallocating the entire region. This allows the typing of patterns which are difficult to express, even in C.

Furthermore, our system introduces the notion of *sized* regions. Each value in our language has an associated size, which is an abstract representation of the amount of memory that the value occupies. We use this information to ensure that regions are not over-allocated into, as each region has an associated maximum size constraint. These constraints are provided explicitly by the programmer as annotations on region creation and allocation sites, giving them fine-grained control over memory usage in their program. In the previous example, we allocated v into the region associated with x. This allocation carries the implicit constraint that the region of x must be of an arbitrary size. A more fine-grained allocation is possible using size annotations:

let $x =$ **newrgn** [10] **in**
  v [5] **at** x

Here, 10 is a size representing 10 abstract units of memory. The region associated with x has a size of 5 and we can only allocate into this region if the total size of allocations into the region does not exceed 10. In the above code, we allocate 5 units of memory into the region. as indicated by the size annotation on the allocation site. Contrarily, the following code would be invalid:

let $x =$ **newrgn** [10] **in**
  v [10] **at** x;
  v [5] **at** x

Since the sum of allocations (15) is greater than the region's bound size. This program is thus rejected by the type checker.

Spegion's size annotations are abstract units of memory, but the principle can be applied to concrete memory sizes. In performance critical applications, such as embedded systems, where the programmer has clear static constraints on the size of data structures, the programmer can use these size annotations to ensure that the program does not exceed the available memory.

Size annotations are an optional feature of Spegion, as unannotated regions and allocations default to being of unbounded size. These size constraints on allocations can be enforced using an effect system, and present an opportunity for further research into type-level reasoning about memory usage.

1.1 Contributions

This paper makes the following contributions:

- We introduce Spegion, a novel core calculus and formal semantics for a language featuring implicit, non-lexical regions with explicit size constraints. Unlike traditional region calculi which are lexically scoped and require first-class region syntax, Spegion removes first-class regions from the source language syntax (except for polymorphic constructs), enabling natural compatibility with C-like languages.
- We develop a type system that leverages effects to ensure safe allocation into implicit regions, eliminating the need for substructural or linear types commonly required in prior non-lexical region systems such as Cyclone. This approach supports flexible memory management patterns including region splitting and partial deallocation, which are difficult to express in lexically scoped systems.
- We extend the concept of region-based memory management by introducing sized regions, where each region and allocation carries an abstract size annotation. This allows static enforcement of memory usage bounds via an effect system, providing fine-grained control over memory consumption and enabling static reasoning about memory usage in performance-critical applications.
- We provide a formal proof of type safety for Spegion, building on and adapting the syntactic proof techniques from Helsen and Thiemann's work on the region calculus.
- We demonstrate the expressiveness and practical applicability of Spegion through a series of illustrative examples, including a sketch of an extension combining refinement types with region sizes to further enhance static guarantees about memory usage.

1.2 Overview

The rest of the paper is structured as follows. Section 2 introduces the syntax and typing rules of Spegion. In Section 3 we present the dynamic semantics of the language as reduction rules. In Section 4 we present the type safety properties of the language in the form of progress and preservation theorems. Section 5 explores the potential applications of Spegion in several domains, including a sketch of how refinement types can be combined with our system to provide powerful static guarantees about memory usage. We also discuss the application of our system in the context of memory safety in C-like languages. Section 6 discusses related work, and Section 7 concludes.

2 Static Semantics

We define a core calculus for Spegion, drawing from the region calculus of [29] and the presentation by [17]. The syntax of types is given by:

$$\begin{aligned} \tau &::= \quad \alpha \mid \text{Int} \mid \text{Unit} \mid \text{Bool} \mid \text{Ref}\ \tau \mid \mu \xrightarrow{\varphi} \mu \mid \forall\{\alpha, \rho, \epsilon\}.\mu \xrightarrow{\varphi} \mu \\ \mu &::= \quad (\tau, \rho) \end{aligned} \tag{types}$$

Types τ are either type variables α, integers, units, booleans, references to other types, function types, or polymorphic types. A program expression in our calculus is assigned a "type-with-place", i.e., a pair of a type and a region, denoted by μ.

Function types are written as $\mu_1 \xrightarrow{\varphi} \mu_2$. Above the function arrow is an *arrow effect*, i.e. the latent effect that happens on application of the function, as in [29]. Polymorphic functions are typed by type schemes $\forall\{\alpha, \rho, \epsilon\}.\mu$ which bind type α, region ρ, and effect ϵ variables. We refer to these as kind-annotated type variables, however, in the syntax of our calculus we typically omit the kind annotations for brevity.

Kinds are given by the grammar:

$$\kappa ::= \quad \text{Type} \mid \kappa \to \kappa \mid \text{Region} \mid \text{Effect} \mid \text{Size} \qquad \text{(kinds)}$$

That is a kind is either a type, a function kind, a region, an effect, or a size. Kinding rules for types are given in Figure 2. These rules are mostly straightforward and ensure that type, regions, effects, and sizes are well-structured.

$$\frac{\text{Type} = K(\alpha)}{K \vdash \alpha : \text{Type}}\ \kappa\text{-VAR} \qquad \frac{K, \alpha : \text{Type}, \rho : \text{Region}, \epsilon : \text{Effect} \vdash \tau : \text{Type}}{K \vdash \forall\{\alpha, \rho, \epsilon\}.\tau : \text{Type}}\ \kappa\text{-FORALL} \qquad \frac{\text{Region} = K(\rho)}{K \vdash \rho : \text{Region}}\ \kappa\text{-REG}$$

$$\frac{}{K \vdash (\to) : \text{Type} \to \text{Effect} \to \text{Type} \to \text{Type}}\ \kappa\text{-}\to \qquad \frac{K \vdash \kappa_1 : \text{Type} \qquad K \vdash \tau : \text{Type}}{K \vdash (\kappa_1 \to \kappa_2)\ \tau : \kappa_2}\ \kappa\text{-APP}_1$$

$$\frac{K \vdash \kappa_1 : \text{Effect} \qquad K \vdash \varphi : \text{Effect}}{K \vdash (\kappa_1 \to \kappa_2)\ \varphi : \kappa_2}\ \kappa\text{-APP}_2 \qquad \frac{}{K \vdash \text{Int} : \text{Type}}\ \kappa\text{-INT} \qquad \frac{}{K \vdash \text{Unit} : \text{Type}}\ \kappa\text{-UNIT}$$

$$\frac{K \vdash \tau : \text{Type}}{K \vdash \text{Ref}\ \tau : \text{Type}}\ \kappa\text{-REF} \qquad \frac{}{K \vdash \text{Bool} : \text{Type}}\ \kappa\text{-BOOL} \qquad \frac{}{K \vdash s : \text{Size}}\ \kappa\text{-SIZE}$$

$$\frac{K \vdash \kappa_1 : \text{Size} \qquad K \vdash \kappa_2 : \text{Size}}{K \vdash \kappa_1\ op\ \kappa_2 : \text{Size}}\ \kappa\text{-OP} \qquad \frac{K \vdash \tau : \text{Type} \qquad K \vdash \rho : \text{Region}}{K \vdash (\tau, \rho) : \text{Type}}\ \kappa\text{-TyWithPlace}$$

$$\frac{}{K \vdash_\kappa \{\bot\} : \text{Effect}}\ \kappa\text{-BOT} \qquad \frac{K \vdash \varphi_1 : \text{Effect} \quad K \vdash \varphi_2 : \text{Effect}}{K \vdash \varphi_1 \times \varphi_2 : \text{Effect}}\ \kappa\text{-}\times \qquad \frac{K \vdash \varphi_1 : \text{Effect} \quad K \vdash \varphi_2 : \text{Effect}}{K \vdash \varphi_1 \sqcup \varphi_2 : \text{Effect}}\ \kappa\text{-}\sqcup$$

$$\frac{K \vdash s : \text{Size} \qquad K \vdash \rho : \text{Region}}{K \vdash \{\textbf{alloc}\ s\ \rho\} : \text{Effect}}\ \kappa\text{-ALLOC} \qquad \frac{K \vdash \rho : \text{Region} \qquad K \vdash s : \text{Size}}{K \vdash \{\textbf{fresh}\ \rho\ s\} : \text{Effect}}\ \kappa\text{-FRESH}$$

$$\frac{K \vdash \rho : \text{Region} \qquad K \vdash s : \text{Size} \qquad K \vdash \rho' : \text{Region}}{K \vdash \{\textbf{split}\ \rho\ s\ \rho'\} : \text{Effect}}\ \kappa\text{-SPLIT}$$

$$\frac{K \vdash \rho : \text{Region}}{K \vdash \{\textbf{free}\ \rho\} : \text{Effect}}\ \kappa\text{-FREE} \qquad \frac{\text{Effect} = K(\epsilon)}{K \vdash \{\epsilon\} : \text{Effect}}\ \kappa\text{-}\epsilon \qquad \frac{K \vdash \epsilon : \text{Effect} \qquad K \vdash \varphi : \text{Effect}}{K \vdash \{\textbf{rec}\ \epsilon\ \varphi\} : \text{Effect}}\ \kappa\text{-REC}$$

Figure 2 Kinding rules in Spegion.

2.1 Effects and Sizes

An effect φ describes the sequence of actions that are performed on a region when an expression is evaluated. Many expressions in the calculus convey some effect, which form part of the expression's typing judgement. The following grammar defines these actions:

$$\begin{aligned}\varphi ::= \quad & \varphi \times \varphi \mid \{\bot\} \mid \{\textbf{fresh } \rho\ s\} \mid \{\textbf{free } \rho\} \mid \{\textbf{split } \rho\ s\ \rho'\} \mid \{\textbf{alloc } s\ \rho\} \mid \varphi \sqcup \varphi \\ \mid \quad & \{\epsilon\} \mid \{\textbf{rec } \epsilon\ \varphi\}\end{aligned} \tag{effects}$$

Effects are composed using the $\times$ operator, which describes the sequential composition of two effects. This operator also ensures constraints on memory usage are respected when two effects are composed. The $\{\bot\}$ effect describes an empty effect, i.e., no actions are performed, and is used in the typing of expressions which are pure computations. The creation of a fresh region is denoted by $\{\textbf{fresh } \rho\ s\}$, where ρ is a fresh region name and s is the region's *size*.

A size is an abstract unit of memory. For simplicity, we consider sizes to be natural numbers, extended with a special size ω which represents an unknown size. Sizes thus comprise an extended natural numbers preordered semiring $(\overline{\mathbb{N}}, +, \cdot, \dot{-}, 0, 1, \sqsubseteq)$ where $\overline{\mathbb{N}} = \mathbb{N} \cup \{\omega\}$, such that $\forall n \in \overline{\mathbb{N}}. n + \omega = \omega = \omega + n = \omega$. The preorder $\sqsubseteq$ is the standard preorder on $\mathbb{N}$ with ω as the greatest element. The semiring is also equipped with a truncated subtraction (or *monus*) operation $\dot{-}$, given by Definition 2.1:

▶ **Definition 2.1** (Monus ($\dot{-}$)). For any two numbers $n, n' \in \overline{\mathbb{N}}$, $\dot{-}$ is defined:

$$n \dot{-} n' = \begin{cases} n - n' & n, n' \sqsubset \omega \wedge n \sqsupseteq n', \\ 0 & n, n' \sqsubset \omega \wedge n \sqsubset n' \\ n & n' = 0 \\ 0 & n \sqsubset n' \wedge n' = \omega, \\ \omega & \text{if } n = \omega \end{cases}$$

The typing rules do not make use of subtraction, however, it is required during evaluation.

The effect $\{\textbf{free } \rho\}$ describes the freeing of a region ρ, which deallocates all memory in the region. The creation of sub-regions is denoted by $\{\textbf{split } \rho\ s\ \rho'\}$, where ρ' is a fresh region name for a sub-region of ρ with size s. Allocation of a value into a region is described by $\{\textbf{alloc } s\ \rho\}$, where s is the size of the value being allocated, and ρ is the region it is being allocated into. If statements introduce branching into our calculus, with the different cases potentially having different effects. This is captured by the $\{\varphi_1 \sqcup \varphi_2\}$ effect, where φ_1 denotes the effect of the true branch and φ_2 the effect of the false branch. Recursion is described in effects through the $\{\epsilon\}$ and $\{\textbf{rec } \epsilon\ \varphi\}$ effects. The former denotes the usage of a recursive function definition, while the latter captures the latent effect of this function upon application. We explain further when we discuss the typing rules for recursive functions.

2.2 Syntax and Typing

The term syntax of our calculus comprises the λ-calculus, with the addition of references, sequential composition, polymorphic recursion, as well as constructs for region manipulation. These include expressions for creating, freeing, and splitting regions, as well as primitives for allocating data into regions and copying data between regions. The full syntax is given by the following grammar, which divides constructs between values v and expressions e:

$$\begin{aligned} v ::= \quad & n \mid \text{true} \mid \text{false} \mid \Lambda\{\alpha, \rho, \epsilon\}.\lambda x.e \mid \lambda x.e \mid () \mid l_\rho \\ e ::= \quad & x \mid l_\rho \mid v\ [s]\ \textbf{at}\ e \mid e\ e \mid \textbf{ref}\ e \mid\ !e \mid e := e \mid e;\ e \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \\ \mid \quad & \textbf{let}\ x = e\ \textbf{in}\ e \mid e\ @\ \mu \mid \textbf{let}\ f = \textbf{fix}(f, (\Lambda\{\alpha, \rho, \epsilon\}.\lambda x.e_1)\ [s]\ \textbf{at}\ e_2)\ \textbf{in}\ e_3 \\ \mid \quad & \textbf{newrgn}\ [s] \mid \textbf{freergn}\ e \mid \textbf{split}\ [s]\ e \mid \textbf{copy}\ e\ \textbf{into}\ e \end{aligned} \tag{terms}$$

The syntax of values v includes integers n, booleans true and false, polymorphic functions $\Lambda\{\alpha, \rho, \epsilon\}.\lambda x.e$, functions $\lambda x.e$, unit $()$, and locations l_ρ. Expressions e include variables x, locations l_ρ, value allocations $v\ [s]\ \textbf{at}\ e$, function application $e\ e$, reference creation $\textbf{ref}\ e$,

$$\frac{}{K \mid \Gamma \mid \Sigma \vdash c : \mathrm{Int}}\ \textsc{t-int} \qquad \frac{}{K \mid \Gamma \mid \Sigma \vdash () : \mathrm{Unit}}\ \textsc{t-unit}$$

$$\frac{}{K \mid \Gamma \mid \Sigma \vdash \mathrm{true} : \mathrm{Bool}}\ \textsc{t-true} \qquad \frac{}{K \mid \Gamma \mid \Sigma \vdash \mathrm{false} : \mathrm{Bool}}\ \textsc{t-false}$$

$$\frac{\tau = \Sigma(l_\rho) \qquad K \vdash \tau : \mathrm{Type}}{K \mid \Gamma \mid \Sigma \vdash l_\rho : \tau}\ \textsc{t-loc} \qquad \frac{K \vdash \mu_1 : \mathrm{Type} \quad K \mid \Gamma, x : \mu_1 \mid \Sigma \vdash e : \mu_2 \mid \varphi}{K \mid \Gamma \mid \Sigma \vdash \lambda x.e : \mu_1 \xrightarrow{\varphi} \mu_2}\ \textsc{t-}\lambda$$

$$\frac{K, \alpha : \mathrm{Type}, \rho : \mathrm{Region}, \epsilon : \mathrm{Effect} \mid \Gamma, x : \mu_1 \mid \Sigma \vdash e : \mu_2 \mid \varphi}{K \mid \Gamma \mid \Sigma \vdash \Lambda\{\alpha, \rho, \epsilon\}.\lambda x.e : \forall\{\alpha : \mathrm{Type}, \rho : \mathrm{Region}, \epsilon : \mathrm{Effect}\}.\mu_1 \xrightarrow{\varphi} \mu_2}\ \textsc{t-}\Lambda$$

Figure 3 Value typing rules in Spegion.

dereferencing $!e$, assignment $e := e$, sequential composition $e;\ e$, conditionals **if** e **then** e **else** e, let-binding **let** $x = e$ **in** e, type applications $e\ @\ \mu$, and recursive function definitions **let** $f =$ **fix**$(f, (\Lambda\{\alpha, \rho, \epsilon\}.\lambda x.e_1)\ [s]$ **at** $e_2)$ **in** e_3. Region manipulation constructs include creating a new region **newrgn** $[s]$, freeing a region **freergn** e, splitting a sub-region from another region **split** $[s]\ e$, and copying data between regions **copy** e **into** e.

Expressions are typed by the judgement:

$$K \mid \Gamma \mid \Sigma \vdash e : (\tau, \rho) \mid \varphi$$

assigning an expression e the type τ in region ρ with effect φ. A context of kind-annotated type variables K provides a mapping from type variables to kinds, given by:

$$K ::= \quad \emptyset \mid K, \alpha : \mathrm{Type} \mid K, \rho : \mathrm{Region} \mid K, \epsilon : \mathrm{Effect} \qquad \text{(kinding contexts)}$$

That is, a context may be empty $\emptyset$, extended with a type variable $\alpha : \mathrm{Type}$, a region variable $\rho : \mathrm{Region}$, or an effect variable $\epsilon : \mathrm{Effect}$. Free variable contexts Γ are given by:

$$\Gamma ::= \quad \emptyset \mid \Gamma, x : \mu \qquad \text{(free variable contexts)}$$

A store typing context Σ is a mapping from locations to types, defined:

$$\Sigma ::= \quad \emptyset \mid \Sigma, l_\rho : \tau \qquad \text{(store typing)}$$

That is l_ρ is a location in region ρ with type τ. Figure 4 provides typing rules for the expressions. Values, which are non-computational objects, are typed by a separate judgement

$$K \mid \Gamma \mid \Sigma \vdash v : \tau$$

This judgement is similar to that of expressions but does not include an effect. Figure 3 gives the typing rules for values, relating terms to types.

The rules for integers (T-INT), unit (T-UNIT), and bools (T-TRUE, T-FALSE) are standard. For (T-λ), the type of the argument x is added to the variable context Γ and the expression e is typed. The type of the function is then a function type from the type of the argument to the type of the body. The effect of the function is the latent effect of the body. A location l_ρ is typed by (T-LOC), where the location's type τ is looked up in the store typing Σ. Finally, polymorphic functions are typed by (T-Λ), which also adds the kind-annotated type, region, and effect variables to the kind variable context K.

$$\text{T-USE-VAL}\ \frac{K \mid \Gamma \mid \Sigma \vdash l_\rho : \tau}{K \mid \Gamma \mid \Sigma \vdash l_\rho : (\tau, \rho) \mid \{\bot\}} \qquad \frac{K \mid \Gamma \mid \Sigma \vdash e : (\tau', \rho) \mid \varphi \qquad K \mid \Gamma \mid \Sigma \vdash v : \tau \qquad K \vdash s : \text{Size}}{K \mid \Gamma \mid \Sigma \vdash v\ [s]\ \textbf{at}\ e : (\tau, \rho) \mid \varphi \times \{\textbf{alloc}\ s\ \rho\}}\ \text{T-VAL}$$

$$\frac{s \sqsupseteq 1 \qquad K, \rho : \text{Region} \vdash s : \text{Size}}{K \mid \Gamma \mid \Sigma \vdash \textbf{newrgn}\ [s] : (\text{Unit}, \rho) \mid \{\textbf{fresh}\ \rho\ s\} \times \{\textbf{alloc}\ 1\ \rho\}}\ \text{T-NEWRGN}$$

$$\frac{K \mid \Gamma \mid \Sigma \vdash e : (\tau, \rho) \mid \varphi}{K \mid \Gamma \mid \Sigma \vdash \textbf{freergn}\ e : (\text{Unit}, \rho_{\text{glob}}) \mid \varphi \times \{\textbf{free}\ \rho\}}\ \text{T-FREERGN}$$

$$\frac{K, \rho' : \text{Region} \mid \Gamma \mid \Sigma \vdash e : (\tau, \rho) \mid \varphi \qquad s \sqsupseteq 1 \qquad K \vdash s : \text{Size}}{K \mid \Gamma \mid \Sigma \vdash \textbf{split}\ [s]\ e : (\text{Unit}, \rho') \mid \varphi \times \{\textbf{split}\ \rho\ s\ \rho'\} \times \{\textbf{alloc}\ 1\ \rho'\}}\ \text{T-SPLIT}$$

$$\frac{K \mid \Gamma \mid \Sigma \vdash e_1 : (\tau, \rho) \mid \varphi_1 \qquad K \mid \Gamma \mid \Sigma \vdash e_2 : (\tau', \rho') \mid \varphi_2}{K \mid \Gamma \mid \Sigma \vdash \textbf{copy}\ e_1\ \textbf{into}\ e_2 : (\tau, \rho') \mid \varphi_1 \times \varphi_2 \times \{\textbf{alloc}\ 1\ \rho'\}}\ \text{T-COPY}$$

$$\text{T-VAR}\ \frac{K \vdash \mu : \text{Type}}{K \mid \Gamma, x : \mu \mid \Sigma \vdash x : \mu \mid \{\bot\}} \qquad \text{T-LET}\ \frac{K \mid \Gamma \mid \Sigma \vdash e_1 : \mu_1 \mid \varphi_1 \qquad K \mid \Gamma, x : \mu_1 \mid \Sigma \vdash e_2 : \mu_2 \mid \varphi_2}{K \mid \Gamma \mid \Sigma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \mu_2 \mid \varphi_1 \times \varphi_2}$$

$$\frac{K \mid \Gamma \mid \Sigma \vdash e_1 : (\text{Bool}, \rho) \mid \varphi_1 \qquad K \mid \Gamma \mid \Sigma \vdash e_2 : \mu_2 \mid \varphi_2 \qquad K \mid \Gamma \mid \Sigma \vdash e_3 : \mu_2 \mid \varphi_3}{K \mid \Gamma \mid \Sigma \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \mu_2 \mid \varphi_1 \times (\varphi_2 \sqcup \varphi_3)}\ \text{T-IF}$$

$$\text{T-REF}\ \frac{K \mid \Gamma \mid \Sigma \vdash e : (\tau, \rho) \mid \varphi}{K \mid \Gamma \mid \Sigma \vdash \textbf{ref}\ e : (\text{Ref}\ \tau, \rho) \mid \varphi \times \{\textbf{alloc}\ 1\ \rho\}} \qquad \text{T-DEREF}\ \frac{K \mid \Gamma \mid \Sigma \vdash e : (\text{Ref}\ \tau, \rho) \mid \varphi}{K \mid \Gamma \mid \Sigma \vdash !e : (\tau, \rho) \mid \varphi}$$

$$\text{T-ASSIGN}\ \frac{K \mid \Gamma \mid \Sigma \vdash e_1 : (\text{Ref}\ \tau, \rho') \mid \varphi_1 \qquad K \mid \Gamma \mid \Sigma \vdash e_2 : \mu \mid \varphi_2}{K \mid \Gamma \mid \Sigma \vdash e_1 := e_2 : (\text{Unit}, \rho_{\text{glob}}) \mid \varphi_1 \times \varphi_2} \qquad \text{T-SEQ}\ \frac{K \mid \Gamma \mid \Sigma \vdash e_1 : (\text{Unit}, \rho) \mid \varphi_1 \qquad K \mid \Gamma \mid \Sigma \vdash e_2 : \mu \mid \varphi_2}{K \mid \Gamma \mid \Sigma \vdash e_1; e_2 : \mu \mid \varphi_1 \times \varphi_2}$$

$$\frac{K \mid \Gamma \mid \Sigma \vdash e_1 : (\mu_1 \xrightarrow{\varphi} \mu_2, \rho) \mid \varphi_1 \qquad K \mid \Gamma \mid \Sigma \vdash e_2 : \mu_1 \mid \varphi_2}{K \mid \Gamma \mid \Sigma \vdash e_1\ e_2 : \mu_2 \mid \varphi_1 \times \varphi_2 \times \varphi}\ \text{T-APP}$$

$$\frac{K \mid \Gamma \mid \Sigma \vdash e : \forall\{\alpha, \rho, \epsilon : \text{Effect}\}.\mu_1 \xrightarrow{\varphi} \mu_2 \mid \{\bot\} \qquad K \vdash (\tau, \rho') : \text{Type}}{K \mid \Gamma \mid \Sigma \vdash e\ @\ (\tau, \rho', \varphi) : [\alpha \mapsto \tau, \rho \mapsto \rho'](\mu_1 \xrightarrow{\varphi} \mu_2) \mid \{\bot\}}\ \text{T-TYAPP}$$

$$\text{T-FIX}\ \frac{\begin{array}{c}\mu_f = (\forall\{\alpha, \rho, \epsilon\}.(\alpha, \rho) \xrightarrow{\{\epsilon\}} \mu_1, \rho_f) \\ K \mid \Gamma, f : \mu_f \mid \Sigma \vdash (\Lambda\{\alpha, \rho, \epsilon\}.\lambda x.e_1)\ [s]\ \textbf{at}\ e_2 : (\forall\{\alpha, \rho, \epsilon\}.(\alpha, \rho) \xrightarrow{\varphi} \mu_1, \rho_f) \mid \varphi_1 \\ \varphi' = [\{\epsilon\} \mapsto \{\textbf{rec}\ \epsilon\ \varphi\}]\varphi \qquad K \mid \Gamma, f : (\forall\{\alpha, \rho, \epsilon\}.(\alpha, \rho) \xrightarrow{\varphi'} \mu_1, \rho_f) \mid \Sigma \vdash e_3 : \mu_2 \mid \varphi_2\end{array}}{K \mid \Gamma \mid \Sigma \vdash \textbf{let}\ f = \textbf{fix}(f, (\Lambda\{\alpha, \rho, \epsilon\}.\lambda x.e_1)\ [s]\ \textbf{at}\ e_2)\ \textbf{in}\ e_3 : \mu_2 \mid \varphi_1 \times \varphi_2}$$

Figure 4 Expression typing rules in Spegion.

In the typing of expressions, the rule for variables (T-VAR) is straightforward, if the variable is present in the free variable context Γ with type μ then $x : \mu$. The rule for locations (T-USE-VAL) is also straightforward, if the location l_ρ is present in the store typing Σ with type τ then $l_\rho : (\tau, \rho)$.

Creating a new region is handled by the (T-NEWRGN) rule. The **newrgn** construct allocates a new region of size s and has type (Unit, ρ) where ρ is the freshly allocated region. This behaviour is captured in the rule's effect $\{\textbf{fresh}\ \rho\ s\}$, which binds a fresh region name ρ to the size s. Evaluating this expression returns a unit value, which acts as a pointer into the region, allowing this region to be referenced in the subsequent program. This eliminates the need for syntactic region variables in all non-polymorphic code. The allocation of this null value is described in the second part of the effect $\{\textbf{alloc}\ 1\ \rho\}$, i.e., allocate a value of size 1 into ρ (since a value cannot be 0-sized). Thus, we have the additional constraint $s \sqsupseteq 1$ ensuring regions must be able to allocate at least 1 unit of memory.

The rule for freeing a region (T-FREERGN) deletes the region ρ associated with the expression e, freeing its associated memory locations. This yields an effect comprised of the effect of typing e (φ) followed by the effect $\{\textbf{free}\ \rho\}$. Evaluating a **freergn** expression returns a pointer to value of type Unit in the global region ρ_{glob} which is always available.

A sub-region may be split from a region using the (T-SPLIT) rule. As with **freergn**, the region to be split is associated with the sub-expression e. The programmer-defined size s determines the size of the new sub-region, represented by the effect $\{\textbf{split}\ \rho\ s\ \rho'\}$. As in (T-NEWRGN), a null pointer into the new region is created (thus the rule also has an **alloc** effect), and the overall expression has the type (Unit, ρ'), where ρ' is the fresh sub-region.

Copying data between regions is handled by the (T-COPY) rule. The behaviour of **copy** can also be simulated in terms of (T-ASSIGN) and (T-VAL). However, we include the **copy** construct as a useful primitive. The expression **copy** e_1 **into** e_2 copies the location obtained from evaluating e_1 from its region ρ into the region ρ' associated with e_2. The effect of this operation is the composition of the effects obtained from typing e_1 and e_2, followed by the effect $\{\textbf{alloc}\ 1\ \rho'\}$. Locations have a size of 1, thus this effect conveys the creation of a new location in ρ' pointing to location value.

Allocating a value into a region is handled by the T-VAL rule. A programmer is required to annotate the allocation of a value with a size, which is used in the effect $\{\textbf{alloc}\ s\ \rho\}$. Unlike [29], where the region is explicitly passed as an argument to the allocation primitive, our rule allocates the value into the region associated with the sub-expression e. For example, consider the following trivial program:

$$\textbf{let}\ x = \textbf{newrgn}\ [3]\ \textbf{in}\ ()\ [2]\ \textbf{at}\ x$$

which creates a new region of size 3 and allocates a unit value of size 2 into it. This program has the type (Int, ρ) with the effect:

$$\{\textbf{fresh}\ \rho\ 3\} \times \{\textbf{alloc}\ 1\ \rho\} \times \{\textbf{alloc}\ 2\ \rho\}$$

conveying the creation and allocation into a region without ever giving it an explicit name in the program. This allows our calculus to model region-based memory management in languages where these first-class regions are not present in the source language.

Typing rules for let-binding (T-LET), creating a reference (T-REF), dereferencing (T-DEREF), assigning to a reference (T-ASSIGN), and sequential composition (T-SEQ) thus handle the sequencing of the effects of their sub-expressions, but are otherwise straightforward. Similarly, application (T-APP) sequences the effects of the sub-expressions, followed by the function type's latent effect. If statements (T-IF) are typed using a join effect $\{\varphi_1 \sqcup \varphi_2\}$, which is the effect of the true branch φ_1 and the false branch φ_2 combined.

Recursive function definitions are typed by the (T-FIX) rule. The body of the definition is typed with an effect variable in place of the functions actual latent effect. In the rule's third premise, this variable is then substituted for the recursive effect $\{\textbf{rec}\ \epsilon\ \varphi\}$ yielding φ'. In the typing of e_3, f is then bound with the same function type as before but with this new latent effect φ'. This recursive effect conveys that the the effect φ may be repeated an unbounded number of times. This is akin to the notion of "Kleene star effects" [23, 24] and iterable sequential effects [15].

A consequence of recursion is that regions which appear freely in e_1 must be capable of allocating an unbounded amount of memory. This constraint is enforced at the point where the recursive variable effect is composed with another effect in the typing of the function body (via $\times$), since the presence of a ϵ in φ indicates that the function recurses. We provide the definition of valid effect composition in detail in Section 2.3.

Recursive variable usage thus requires that the variable's type scheme be instantiated via the (T-TYAPP) rule, which instantiates the type scheme and region at the type and region provided by the programmer. This is the only rule in the syntax where a region name is required, which can be obtained using a primitive operator regionOf(e).

2.3 Effect Composition

The typing rules above define the individual effects of expressions. However, taken alone an effect does not provide a complete picture of the memory usage of a program. For example, the effect of an allocation is the $\{\textbf{alloc}\ s\ \rho\}$ effect preceded by the effect of the expression which identifies the region being allocated into. However, at the point of typing this allocation, there is no way of knowing if the allocation described by this effect is valid – this can only be known when the effect is composed with the effect which describes the creation of ρ. We describe below the behaviour of the $\times$ operator, which composes two effects. Effects in our calculus are sequential, thus the effect $\varphi \times \varphi'$ describes the effect of φ followed by the effect of φ'. Figure 5 provides the rules for valid effect compositions.

In the ($\times$-$\bot$) and ($\times$-FRESH) rules, composing an effect with $\{\bot\}$ or $\{\textbf{fresh}\ \rho\ s\}$ is always valid. In the latter case we thus assume that SPEGION always has the ability to allocate new regions. An effect which frees a region $\{\textbf{free}\ \rho\}$ is valid only if the region is not already freed in the preceding effect ($\times$-FREE).

Effects which handle allocation must consider three cases. The first ($\times$-FRESHALLOC) is when the entire lifetime of the region up until the point of the current allocation is described in the preceding effects. In this case, the size of the new allocation effect s' must not exceed the current total size of allocations in the ρ. This is enforced by the constraint $\exists s''.s'' + s' + \text{sumAllocs}(\rho, \varphi) \sqsubseteq s$, i.e., there exists some leftover space s'' after adding the size of the new allocation s' to the total size of allocations in ρ. Definition 2.2 defines the function calculating this total size, mapping a region and effect to a size.

▶ **Definition 2.2** (Total Size of Allocations)**.**

$$\text{sumAllocs}(\rho, \varphi) = \begin{cases} s + \text{sumAllocs}(\rho, \varphi') & \varphi = \varphi' \times \{\textbf{alloc}\ s\ \rho\} \\ s + \text{sumAllocs}(\rho, \varphi') & \varphi = \varphi' \times \{\textbf{split}\ \rho\ s\ \rho'\} \wedge \\ s + s' & \varphi = \varphi' \times \varphi'' \wedge \\ & s\ = \text{sumAllocs}(\rho, \varphi') \wedge \\ & s' = \text{sumAllocs}(\rho, \varphi'') \\ \max(s, s') & \varphi = \varphi' \sqcup \varphi'' \wedge \\ & s\ = \text{sumAllocs}(\rho, \varphi') \wedge \\ & s' = \text{sumAllocs}(\rho, \varphi'') \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{}{\varphi \times \{\bot\}}\ \times\text{-}\bot \qquad \frac{}{\varphi \times \{\mathbf{fresh}\ \rho\ s\}}\ \times\text{-Fresh} \qquad \frac{\{\mathbf{free}\ \rho\} \notin \varphi}{\varphi \times \{\mathbf{free}\ \rho\}}\ \times\text{-Free}$$

$$\times\text{-FreshAlloc}\ \frac{\exists s''.s' + s'' + \mathrm{sumAllocs}(\rho, \varphi) \sqsubseteq s}{\{\mathbf{fresh}\ \rho\ s\} \times \varphi \times \{\mathbf{alloc}\ s'\ \rho\}} \qquad \times\text{-Alloc}\ \frac{\{\mathbf{free}\ \rho\} \notin \varphi \qquad \{\mathbf{fresh}\ \rho\ s'\} \notin \varphi}{\varphi \times \{\mathbf{alloc}\ s\ \rho\}}$$

$$\times\text{-SplitAlloc}\ \frac{\exists s''.s' + s'' + \mathrm{sumAllocs}(\rho, \varphi) \sqsubseteq s}{\{\mathbf{split}\ \rho\ s\ \rho'\} \times \varphi \times \{\mathbf{alloc}\ s'\ \rho\}} \qquad \times\text{-FreshSplit}\ \frac{\exists s''.s' + s'' + \mathrm{sumAllocs}(\rho, \varphi) \sqsubseteq s}{\{\mathbf{fresh}\ \rho\ s\} \times \varphi \times \{\mathbf{split}\ \rho\ s'\ \rho'\}}$$

$$\times\text{-Split}\ \frac{\{\mathbf{free}\ \rho\} \notin \varphi}{\varphi \times \{\mathbf{split}\ \rho\ s\ \rho'\}} \qquad \times\text{-SplitSplit}\ \frac{\exists s''.s' + s'' + \mathrm{sumAllocs}(\rho, \varphi) \sqsubseteq s}{\{\mathbf{split}\ \rho\ s\ \rho'\} \times \varphi \times \{\mathbf{split}\ \rho'\ s'\ \rho''\}} \qquad \times\text{-}\sqcup\ \frac{\varphi_1 \times \varphi_2 \qquad \varphi_1 \times \varphi_3}{\varphi_1 \times (\varphi_2 \sqcup \varphi_3)}$$

$$\times\text{-VarR}\ \frac{\forall(\rho \times s) \in \mathrm{freeAllocs}(\varphi).s \sqsupseteq \omega}{\varphi \times \{\epsilon\}} \qquad \times\text{-VarL}\ \frac{\forall(\rho \times s) \in \mathrm{freeAllocs}(\varphi).s \sqsupseteq \omega}{\{\epsilon\} \times \varphi} \qquad \times\text{-Rec}\ \frac{\times[\epsilon \mapsto \{\bot\}]\varphi'}{\varphi \times \{\mathbf{rec}\ \epsilon\ \varphi'\}}$$

Figure 5 Effect composition rules in Spegion ($\times$).

A similar case applies when allocating into a sub-region where the entire lifetime of the region is described in the preceding effects ($\times$-SplitAlloc).

The second case for allocation ($\times$-Alloc) is when the region is not created in the preceding effect, i.e. $\varphi \times \{\mathbf{alloc}\ s\ \rho\}$ where $\{\mathbf{fresh}\ s'\ \rho\} \notin \varphi$. This situation arises when combining two effects in the typing of a sub-term of the expression where the region is created. In this case, the current total size of allocations in ρ is unknown, thus the permissibility of the current allocation effect is unknown. Thus, the allocation is simply accepted as valid with the knowledge that $\varphi \times \{\mathbf{alloc}\ s\ \rho\}$ will eventually be composed with some effect describing the preceding lifetime of ρ, at which point it will be validated by the previous case.

The rules for composing split effects $\{\mathbf{split}\ \rho\ s\ \rho'\}$ behave in a similar manner as allocation ($\times$-FreshSplit), ($\times$-SplitSplit), and ($\times$-Split). The first two rules handle the cases where the entire lifetime of the region is described in the preceding effects, whilst the last rule handles the case where this information is not available.

Composing a join effect $\varphi_2 \sqcup \varphi_3$ with another effect φ_1 is straightforward ($\times$-$\sqcup$). Since both branches of the program must be well-typed, the effect φ_1 is simply composed with both effects. If this is valid then so the composition of the join: $\varphi_1 \times \{\varphi_2 \sqcup \varphi_3\}$.

Effect variables ϵ occur as part of an effect when a recursive function definition is typed. Composition of these effects are handled by the ($\times$-VarR) and ($\times$-VarL) rules. A constraint of recursive function definitions is that regions which occur freely of the function body must be created with an unbounded size ω since it is not possible to know statically how many times the function will recurse. Regions which are created inside the function body itself may be given any size. The addition of a lightweight refinement types would allow us to express and type programs which allocate recursively into a free region under some constraints. We consider this idea in more depth in Section 5.2.

The free regions of the effects $\varphi \times \{\epsilon\}$ and $\{\epsilon\} \times \varphi$ are collected by the function freeAllocs, given by Definition 2.3. This function maps an effect to the set of allocations (and splits) of free regions which occur in the effect. A region is free if there is no $\{\mathbf{fresh}\ s\ \rho\}$ effect in φ.

This set takes the form of pairs of regions and sizes, where $\bot$ is used to indicate that the region was freed inside φ. Since regions of unbounded size are represented by size ω, the only sized allocations that are permitted must also be those of ω. This is enforced by the constraint $\forall(\rho \times s) \in \text{freeAllocs}(\varphi).s \sqsupseteq \omega$.

▶ **Definition 2.3** (Free Region Allocations of an Effect)**.** Given an effect φ, the allocations of regions which are free in φ is given by:

$$\text{freeAllocs}(\varphi) = \begin{cases} \{(\rho \times s)\} & \varphi = \{\textbf{alloc}\ s\ \rho\} \\ \{(\rho \times s)\} \cup \text{freeAllocs}(\varphi') & \varphi = \varphi' \times \{\textbf{alloc}\ s\ \rho\} \wedge \\ & \{\textbf{fresh}\ \rho\ s'\} \notin \varphi' \\ \{(\rho \times s)\} & \varphi = \{\textbf{split}\ \rho\ s\ \rho'\} \\ \{(\rho \times s)\} \cup \text{freeAllocs}(\varphi') & \varphi = \varphi' \times \{\textbf{split}\ \rho\ s\ \rho'\} \wedge \\ & \{\textbf{fresh}\ \rho\ s'\} \notin \varphi' \\ \{(\rho \times 0)\} & \varphi = \{\textbf{free}\ \rho\} \\ \{(\rho \times 0)\} \cup \text{freeAllocs}(\varphi') & \varphi = \varphi' \times \{\textbf{free}\ \rho\} \wedge \\ & \{\textbf{fresh}\ \rho\ s\} \notin \varphi' \\ \text{freeAllocs}(\varphi') & \varphi = \{\textbf{rec}\ \epsilon\ \varphi'\} \\ \text{freeAllocs}(\varphi') \cup \text{freeAllocs}(\varphi'') & \varphi = \varphi' \times \varphi'' \\ \text{freeAllocs}(\varphi') \cup \text{freeAllocs}(\varphi'') & \varphi = \varphi' \sqcup \varphi'' \\ \emptyset & \text{otherwise} \end{cases}$$

Finally, the usage of a recursive function definition is handled via the $\{\textbf{rec}\ \epsilon\ \varphi''\}$ effect. This rule ($\times$-Rec) is straightforward, since the burden of ensuring that free regions in φ'' are unbounded is placed on the ($\times$-VarR) and ($\times$-VarL) rules. The latent effect of the recursive function definition φ'' is simply composed with the preceding effect φ. Effect variables in φ'' are substituted for $\{\bot\}$ since there is no need to recheck the validity of $\{\epsilon\}$ effects.

Effect combination is a partial function, since the combination of some effects is not valid. For example, the combination of two effects which free the same region is invalid. In this case, the function $\times$ is undefined. Undefined compositions are rejected during typing.

We conclude this section with an example of a Spegion program and its typing:

```
let x = newrgn [5] in
in
   (λz.newrgn [5];
      newrgn [5]) [1] at x
```

which defines a function that creates two fresh regions, and is typed as:

$$((\text{Unit}, \rho')\ \xrightarrow{\{\textbf{fresh}\ \rho_1\ 5\} \times \{\textbf{fresh}\ \rho_2\ 5\}}\ (\text{Unit}, \rho_2), \rho) \mid \{\textbf{alloc}\ 1\ \rho\}$$

3 Dynamic Semantics

The relation $\langle e \mid \sigma \rangle \to \langle e' \mid \sigma' \rangle$ means that e reduces to e' in a single step. Unlike the calculus of [17], our reduction rules include an explicit store to enable the use of references. This store is dual layered, with an outer store σ mapping region names ρ to pairs of inner stores $\sigma_\rho^{in\cdot}$, and maximum sizes s:

$$\sigma ::= \quad \emptyset \mid \rho \mapsto (\sigma_\rho^{in\cdot}, s) \qquad \text{(outer store)}$$

This maximum size which is specified by the programmer as an annotation when the store is allocated initially via a **newrgn** expression. Inner stores thus map locations to values:

$$\sigma_\rho^{in\cdot} ::= \quad \emptyset \mid l \mapsto v \qquad \text{(inner store)}$$

Using our store typing context Σ which maps locations to types, we define the following judgement for well-typed stores:

$$K \mid \Gamma \mid \Sigma \vdash \sigma$$

This judgement asserts that all values in the store are coherent with the store typing, and that the total size allocated for each region does not exceed the region's specified maximum size. Figure 6 gives the typing rules for stores.

$$\frac{}{K \mid \Gamma \mid \Sigma \vdash \emptyset}\ \text{ST-OUTER-EMPTY} \qquad \frac{\text{currentSize}(\sigma(\rho)) \sqsubseteq s \qquad K \mid \Gamma \mid \Sigma \vdash \sigma \qquad K \mid \Gamma \mid \Sigma \vdash \sigma_\rho^{in\cdot}}{K \mid \Gamma \mid \Sigma \vdash \sigma, \rho \mapsto (\sigma_\rho^{in\cdot}, s)}\ \text{ST-OUTER-REGION}$$

Figure 6 Typing rules of σ.

From these rules a store is well-typed if it is empty (ST-OUTER-EMPTY) or (ST-OUTER-REGION) if each of its inner stores are well-typed, and the total size of values allocated in a region does not exceed the region's maximum size s.

The currentSize function is given by Definition 3.1, mapping inner stores to the total size of values allocated in that inner store, the dynamic counterpart to Definition 2.2 in Section 2:

▶ **Definition 3.1** (Current Size).

$$\text{currentSize}(\sigma_\rho^{in\cdot}) = \begin{cases} \text{currentSize}(\emptyset, s) & = 0 \\ \text{currentSize}((\sigma_\rho^{in\cdot\prime}, l_\rho \mapsto v), s) & = \text{sizeOf}(v) \\ & \quad + \text{currentSize}(\sigma_\rho^{in\cdot\prime}, s) \end{cases}$$

The function sizeOf(v) maps values v to sizes, given by Definition 3.2:

▶ **Definition 3.2** (Value Size). Given a value v, sizeOf(v) is defined:

$$\text{sizeOf}(v) = \begin{cases} 1 + |\text{freeLocs}(e)| & v = (\lambda x.e) \\ 1 + |\text{freeLocs}(e)| & v = (\Lambda\{\alpha, \rho, \epsilon\}.\lambda x.e) \\ 1 & \text{otherwise} \end{cases}$$

where freeLocs(e) returns a list of locations in an expression e.

A separate judgement types the inner store:

$$K \mid \Gamma \mid \Sigma \vdash \sigma_\rho^{in\cdot}$$

asserting that all values in the inner store are coherent with the store typing. The typing rules for inner stores are straightforward and are given in Figure 7.

$$\frac{}{K \mid \Gamma \mid \Sigma \vdash \emptyset}\ \text{ST-INNER-EMPTY} \qquad \frac{\tau = \Sigma(l_\rho) \qquad K \mid \Gamma \mid \Sigma \vdash \sigma_\rho^{in\cdot} \qquad K \mid \Gamma \mid \Sigma \vdash v : \tau}{K \mid \Gamma \mid \Sigma \vdash \sigma_\rho^{in\cdot}, l_\rho \mapsto v}\ \text{ST-INNER-LOC}$$

Figure 7 Typing rules of $\sigma_\rho^{in\cdot}$.

$$\textsc{e-newrgn}\ \frac{\rho = \text{freshRegion}() \qquad l_\rho = \text{freshLoc}(\rho)}{\langle \textbf{newrgn}\ [s] \mid \sigma \rangle \longrightarrow \langle l_\rho \mid \sigma, \rho \mapsto (l_\rho \mapsto (), s) \rangle}$$

$$\frac{}{\langle \textbf{freergn}\ l_\rho \mid \sigma \rangle \longrightarrow \langle l^1_{\rho_{\text{glob}}} \mid \sigma \setminus \rho \rangle}\ \textsc{e-freergnL}$$

$$\frac{\rho' = \text{freshRegion}() \qquad (\sigma^{in.}_\rho, s_a) = \sigma(\rho) \qquad l_{\rho'} = \text{freshLoc}(\rho')}{\langle \textbf{split}\ [s]\ l_\rho \mid \sigma \rangle \longrightarrow \langle l'_\rho \mid [\rho \mapsto (\sigma^{in.}_\rho, s_a \dot{-} s)]\sigma, \rho' \mapsto (l_{\rho'} \mapsto (), s) \rangle}\ \textsc{e-splitL}$$

$$\frac{\begin{array}{c}(\sigma^{in.}_\rho, s_a) = \sigma(\rho) \qquad (\sigma^{in.}_{\rho'}, s'_a) = \sigma(\rho') \\ v = \sigma^{in.}_\rho(l_\rho) \qquad l'_{\rho'} = \text{freshLoc}(\rho')\end{array}}{\langle \textbf{copy}\ l_\rho\ \textbf{into}\ l_{\rho'} \mid \sigma \rangle \longrightarrow \langle l'_{\rho'} \mid [\rho' \mapsto ((\sigma^{in.}_{\rho'}, l'_{\rho'} \mapsto l_\rho), s'_a)]\sigma \rangle}\ \textsc{e-copyL}$$

$$\frac{\begin{array}{c}(\sigma^{in.}_\rho, s_a) = \sigma(\rho) \qquad s_v = \text{sizeOf}(v) \\ s_v \sqsubseteq s \qquad l'_\rho = \text{freshLoc}(\rho)\end{array}}{\langle v\ [s]\ \textbf{at}\ l_\rho \mid \sigma \rangle \longrightarrow \langle l'_\rho \mid [\rho \mapsto ((\sigma^{in.}_\rho, l'_\rho \mapsto v), s_a)]\sigma \rangle}\ \textsc{e-valL}$$

$$\frac{(\sigma^{in.}_\rho, s_a) = \sigma(\rho) \qquad (\lambda x.e) = \sigma^\rho_{in.}(l_\rho)}{\langle l_\rho\ l'_{\rho'} \mid \sigma \rangle \longrightarrow \langle [x \mapsto l'_{\rho'}]e \mid \sigma \rangle}\ \textsc{e-appL}$$

$$\frac{}{\langle \textbf{let}\ x : \mu = l_\rho\ \textbf{in}\ e_2 \mid \sigma \rangle \longrightarrow \langle [x \mapsto l_\rho]e_2 \mid \sigma \rangle}\ \textsc{e-letL}$$

$$\textsc{e-ifFalse}\ \frac{\sigma^{in.}_\rho = \sigma(\rho) \qquad \text{false} = \sigma^{in.}_\rho(l_\rho)}{\langle \textbf{if}\ l_\rho\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \mid \sigma \rangle \longrightarrow \langle e_3 \mid \sigma \rangle} \qquad \textsc{e-ifTrue}\ \frac{\sigma^{in.}_\rho = \sigma(\rho) \qquad \text{true} = \sigma^{in.}_\rho(l_\rho)}{\langle \textbf{if}\ l_\rho\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \mid \sigma \rangle \longrightarrow \langle e_2 \mid \sigma \rangle}$$

$$\frac{(\sigma^{in.}_\rho, s_a) = \sigma(\rho) \qquad l'_\rho = \text{freshLoc}(\rho)}{\langle \textbf{ref}\ l_\rho \mid \sigma \rangle \longrightarrow \langle l'_\rho \mid [\rho \mapsto ((\sigma^{in.}_\rho, l'_\rho \mapsto l_\rho), s_a)]\sigma \rangle}\ \textsc{e-refL}$$

$$\frac{(\sigma^{in.}_\rho, s_a) = \sigma(\rho) \qquad l'_\rho = \sigma^\rho_{in.}(l_\rho)}{\langle !l_\rho \mid \sigma \rangle \longrightarrow \langle l'_\rho \mid \sigma \rangle}\ \textsc{e-derefL}$$

$$\textsc{e-assignL}\ \frac{(\sigma^{in.}_\rho, s_a) = \sigma(\rho)}{\langle l_\rho := l_{\rho'} \mid \sigma \rangle \longrightarrow \langle l^1_{\rho_{\text{glob}}} \mid [\rho \mapsto ((\sigma^{in.}_\rho, l_\rho \mapsto l_{\rho'}), s_a)]\sigma \rangle} \qquad \textsc{e-seqNext}\ \frac{}{\langle l_\rho; e_2 \mid \sigma \rangle \longrightarrow \langle e_2 \mid \sigma \rangle}$$

$$\textsc{e-tyApp}\ \frac{}{\langle e\ @\ (\tau, \rho') \mid \sigma \rangle \longrightarrow \langle e \mid \sigma \rangle} \qquad \textsc{e-fixL}\ \frac{\sigma^{in.}_\rho = \sigma(\rho) \qquad \Lambda\{\alpha, \rho, \epsilon\}.\lambda x.e_1 = \sigma^{in.}_\rho(l_\rho)}{\langle \textbf{let}\ f = (f, l_\rho)\ \textbf{in}\ e_3 \mid \sigma \rangle \longrightarrow \langle [x \mapsto l_\rho]e_3 \mid \sigma \rangle}$$

Figure 8 Evaluation rules of Spegion.

Figure 8 gives the reduction rules for Spegion. For brevity, we omit congruence rules which are standard. The complete set of rules can be found in the appendix.

The (E-NEWRGN) rule allocates a new region in the store, with the specified size s, and allocates and steps to a new location that points to a unit value in the fresh region. The (E-FREERGNL) rule simply frees the region ρ from the store and steps to a pointer to the unit value in the global region $l^1_{\rho_{\text{glob}}}$.

Splitting a sub-region off from another region is handled by the (E-SPLITL) rule. The rule makes use of the size semiring's truncated subtraction operation, given by Definition 2.1, to calculate the new reduced maximum size of the parent region ρ. As in (E-NEWRGN), the rule then allocates a new region ρ', then allocates and steps to a location in ρ' pointing to a unit value to act as a pointer to the region.

The (E-COPYL) rule copies a location from one region to another. A fresh location $l'_{\rho'}$ is allocated in the region being copied to (ρ') pointing to the location being copied from ρ (l_ρ).

The reduction rule for allocation (E-VALL) compares the programmer annotated size against the actual size of the value being allocated (calculated via 3.2). If the annotated size is greater than or equal to the actual size, the value is allocated into the region at a fresh location l'_ρ in the store. Type safety guarantees that ρ has enough space to allocate a value of the annotated size. If the annotated size is less than the actual size, the program gets stuck. Type safety also ensures that such a situation should never arise.

The reduction rules for application (E-APPL), let binding (E-LETL), if statements (E-IFFALSE), (E-IFTRUE), referencing (E-REFL), dereferencing (E-DEREFL), assignment (E-ASSIGNL), and sequential composition (E-SEQNEXT) are standard.

Type application (E-BIGAPP) is handled by the rule of the same name, which is a no-op, since it is a purely static construct. Recursive functions are handled by the (E-FIXL) rule, which substitutes the location l_ρ, where the recursive function $\Lambda\{\alpha, \rho, \epsilon\}.\lambda x.e_1$ is stored, for the variable f in e_3.

4 Properties of the Type System

Type safety is guaranteed by Theorems 1 and 2. Theorem 1 states the property of *progress*, i.e, a well-typed closed term is either a value, or can be further reduced. Theorem 2 states the property of type *preservation* (or *subject reduction*), which states that if a term is well-typed and can take a step of evaluation, then the resulting term is also well-typed. The proofs are based on the syntactic type soundness proofs of Helsen and Thiemann [17], and can be found in the extended version of this paper.

▶ **Theorem 1** (Progress). *If*

$$K \mid \Gamma \mid \Sigma \vdash e : (\tau, \rho) \mid \varphi$$

and

$$K \mid \Gamma \mid \Sigma \vdash \sigma$$

then either

i *e is a value or*
ii *e has the form (x) (a variable), with $x \in fv(e)$ or*
iii *there exists an e' such that $\langle e \mid \sigma\rangle \longrightarrow \langle e' \mid \sigma'\rangle$*

▶ **Theorem 2** (Preservation). *If*

$$K \mid \Gamma \mid \Sigma \vdash e : \mu \mid \varphi$$
$$K \mid \Gamma \mid \Sigma \vdash \sigma$$
$$\langle e \mid \sigma\rangle \longrightarrow \langle e' \mid \sigma'\rangle$$

for some Σ' such that

$$\Sigma' \supseteq \Sigma$$

we have that

$$K \mid \Gamma \mid \Sigma' \vdash e' : \mu \mid \varphi'$$
$$K \mid \Gamma \mid \Sigma' \vdash \sigma'$$

where

$$K \vdash \varphi' \sqsubseteq \varphi : \mathit{Effect}$$

From our assumption of store typing correctness, we assert that e is a sub-derivation of some larger derivation e″ for which correctness with regard to region size holds, i.e., given

$$K \mid \Gamma \mid \Sigma \vdash \sigma$$

we have that:

$$\forall \rho \in \sigma . \forall l_\rho \in \sigma^{in \cdot}_\rho . l_\rho \in dom(\Sigma) \wedge sizeOf(\Sigma(l_\rho)) \sqsupseteq sizeOf(\sigma^{in \cdot}_\rho(l_\rho))$$

Finally, we assume the existence of a global region parametrising the calculus which cannot be freed, as well as a location inside this global region of type Unit:

$$\forall \Sigma . l^1_{\rho_{glob}} : \mathit{Unit} \in \Sigma$$

Preservation states that if an expression e is well-typed and has effect φ, and a reduction step can be made, then the resulting expression e' is also well-typed with an updated store typing Σ' and an effect φ' such that $\varphi' \sqsubseteq \varphi$.

$$\frac{K \vdash \varphi_1 \equiv \varphi_2 : \text{Effect}}{K \vdash \varphi_1 \sqsubseteq \varphi_2 : \text{Effect}}\ \text{SB-}\equiv$$

$$\frac{K \vdash \varphi : \text{Effect}}{K \vdash \{\bot\} \sqsubseteq \varphi}\ \text{SB-}\bot$$

$$\frac{K \vdash \varphi_1 \sqsubseteq \varphi_3 : \text{Effect} \qquad K \vdash \varphi_2 \sqsubseteq \varphi_3 : \text{Effect}}{K \vdash \varphi_1 \times \varphi_2 \sqsubseteq \varphi_3 : \text{Effect}}\ \text{SB-}\times_{above}$$

$$\frac{K \vdash \varphi_1 \sqsubseteq \varphi_2 : \text{Effect} \qquad K \vdash \varphi_3 : \text{Effect}}{K \vdash \varphi_1 \sqsubseteq \varphi_2 \times \varphi_3 : \text{Effect}}\ \text{SB-}\times^1_{below}$$

$$\frac{K \vdash \varphi_2 : \text{Effect} \qquad K \vdash \varphi_1 \sqsubseteq \varphi_3 : \text{Effect}}{K \vdash \varphi_1 \sqsubseteq \varphi_2 \times \varphi_3 : \text{Effect}}\ \text{SB-}\times^2_{below}$$

$$\frac{K \vdash \varphi_1 \sqsubseteq \varphi_3 : \text{Effect} \qquad K \vdash \varphi_2 \sqsubseteq \varphi_3 : \text{Effect}}{K \vdash \varphi_1 \sqcup \varphi_2 \sqsubseteq \varphi_3 : \text{Effect}}\ \text{SB-}\sqcup_{above}$$

$$\frac{K \vdash \varphi_1 \sqsubseteq \varphi_2 : \text{Effect} \qquad K \vdash \varphi_3 : \text{Effect}}{K \vdash \varphi_1 \sqsubseteq \varphi_2 \sqcup \varphi_3 : \text{Effect}}\ \text{SB-}\sqcup^1_{below}$$

$$\frac{K \vdash \varphi_2 : \text{Effect} \qquad K \vdash \varphi_1 \sqsubseteq \varphi_3 : \text{Effect}}{K \vdash \varphi_1 \sqsubseteq \varphi_2 \sqcup \varphi_3 : \text{Effect}}\ \text{SB-}\sqcup^2_{below}$$

Figure 9 Subsumption rules for Spegion ($\sqsubseteq$).

The relation $\sqsubseteq$ defines effect *subsumption*. Effect subsumption is used to relate the effects of intermediate terms as a program is evaluated, since evaluation does not preserve syntactic effects. For example, an allocation expression has an effect which contains an allocation effect $\{\mathbf{alloc}\ s\ \rho\}$. However, the reduction step for allocation expressions produces a location, and locations are typed with an empty effect $\{\bot\}$. To the programmer, these intermediate effects are not of interest – the effect of importance is the overall effect of the program. In preservation, however, we ensure that the intermediate effects are related to this overall effect via the subsumption judgement:

$$K \vdash \varphi_1 \sqsubseteq \varphi_2 : \text{Effect}$$

These rules are given in Figure 9. These rules are standard, however, note that we include

$$\frac{K \vdash \varphi : \text{Effect}}{K \vdash \varphi \equiv \varphi : \text{Effect}}\ \equiv\text{-REFL} \qquad \frac{K \vdash \varphi_1 : \text{Effect} \quad K \vdash \varphi_2 : \text{Effect} \quad K \vdash \varphi_1 \equiv \varphi_2 : \text{Effect}}{K \vdash \varphi_2 \equiv \varphi_1 : \text{Effect}}\ \equiv\text{-SYM}$$

$$\frac{K \vdash \varphi_1 \equiv \varphi_2 : \text{Effect} \quad K \vdash \varphi_2 \equiv \varphi_3 : \text{Effect}}{K \vdash \varphi_1 \equiv \varphi_3 : \text{Effect}}\ \equiv\text{-TR}$$

$$\frac{K \vdash \varphi_1 \equiv \varphi_1' : \text{Effect} \quad K \vdash \varphi_2 \equiv \varphi_2' : \text{Effect}}{K \vdash \varphi_1 \times \varphi_2 \equiv \varphi_1' \times \varphi_2' : \text{Effect}}\ \equiv\text{-}\times\text{-CONG}$$

$$\frac{K \vdash \varphi_1 : \text{Effect} \quad K \vdash \varphi_2 : \text{Effect} \quad K \vdash \varphi_3 : \text{Effect}}{K \vdash \varphi_1 \times (\varphi_2 \times \varphi_3) \equiv (\varphi_1 \times \varphi_2) \times \varphi_3 : \text{Effect}}\ \equiv\text{-}\times\text{-ASSOC}$$

$$\frac{K \vdash \varphi_1 : \text{Effect} \quad K \vdash \varphi_2 : \text{Effect} \quad K \vdash \varphi_3 : \text{Effect}}{K \vdash \varphi_1 \sqcup (\varphi_2 \sqcup \varphi_3) \equiv (\varphi_1 \sqcup \varphi_2) \sqcup \varphi_3 : \text{Effect}}\ \equiv\text{-}\sqcup\text{-ASSOC}$$

$$\frac{K \vdash \varphi_1 \equiv \varphi_1' : \text{Effect} \quad K \vdash \varphi_2 \equiv \varphi_2' : \text{Effect}}{K \vdash \varphi_1 \sqcup \varphi_2 \equiv \varphi_1' \sqcup \varphi_2' : \text{Effect}}\ \equiv\text{-}\sqcup\text{-CONG}$$

Figure 10 Effect equality rules for Spegion ($\equiv$).

kinding of effects and we include rules for the join operator $\sqcup$. The (SB-$\equiv$) rule requires a judgement of effect equivalence:

$$K \vdash \varphi_1 \equiv \varphi_2 : \text{Effect}$$

which is given by the rules in Figure 10. Again, these rules are completely standard.

5 Applications

5.1 System Code Examples

Various systems programming idioms are expressible in the language. Below, we present examples first in C and then in Spegion. In each case, the translation is shallow: region-size annotations required by our language can be inferred from the C program. In one example, our language exceeds what can be expressed in C. In all cases, the type system statically recognises memory safety errors.

Use After Free. In the C code below, a struct with two members is allocated, written to, freed, and then erroneously read from. This is *use after free*, a failure of memory safety.

```
struct mine {int a; int b;};

int free_help(int* p) { free(p); }

int main () {
  struct mine *mp = malloc(sizeof(struct mine));
  mp->a = 0;
  mp->b = 1;
  int *bp = &(mp->b);
  free_help(mp);
  return *bp;    // uaf
}
```

The translation of this code to our language is direct: the size of the new region is taken from the call to `malloc` in the C code.

let $r =$ **newrgn** $[2]$ **in**
let $(x, y) = (\mathbf{ref}\ 0, \mathbf{ref}\ 0)\ [2]$ **at** r **in**
let $x := 0\ [1]$ **at** ρ_{glob} **in**
let $y := 1\ [1]$ **at** ρ_{glob} **in**
let $bp = y$ **in**
freergn r;
let $b = !bp$

This program does not pass the static semantics because on the last line, the type effects record that region r was previously freed, and in typing the final line, the T-REF rule requires the region of bp to be live.

Recursion. Below, a function allocates, calls itself recursively, and then frees its prior allocation. There is no memory safety error: despite being nested by the recursion, all of the allocations are matched by frees.

```
int alloc_rec_free (int n) {
  if (n) {
    int *p = malloc(sizeof(int));
    int  v = alloc_rec_free (n - 1);
    free(p);
    return v;
  } else
    return 0;
}

int main () {
  return alloc_rec_free(10);
}
```

The translation uses the **fix** constructor. Again the translation is shallow: none of the additional annotations in our language require any analysis to concoct. Indeed the function's type scheme is empty because it conveys no change to the live regions to its context.

let arf $=$ **fix**(arf, $(\Lambda\emptyset.\lambda n.$
  if $(n == 0\ [1]$ **at** $\rho_{\text{glob}})$ **then**
    let $r =$ **newrgn** $[1]$ **in**

$$
\begin{array}{l}
\quad\quad \textbf{let } p = \textbf{ref } 0\ [1]\ \textbf{at } r \textbf{ in} \\
\quad\quad \textbf{let } v = (\text{arf} \ @\ \emptyset)\ (n - 1\ [1]\ \textbf{at } \rho_{\text{glob}}) \textbf{ in} \\
\quad\quad \textbf{freergn } p; \\
\quad\quad v \\
\quad \textbf{else} \\
\quad\quad 0\ [1]\ \textbf{at } \rho_{\text{glob}} \\
)\ [1]\ \textbf{at } \rho_{\text{glob}}) \textbf{ in} \\
(\text{arf} \ @\ \emptyset)\ (10\ [1]\ \textbf{at } \rho_{\text{glob}})
\end{array}
$$

The static semantics accepts this program and it is memory safe.

Modelling Loops. Loops can be translated to recursive functions. The C code below is analogous to the code above but it uses a loop. The loop body allocates, it records the value of the index and then frees the prior allocation.

```
int main () {
  int v;
  for (int n = 10; n > 0; n--) {
    int *p = malloc(sizeof(int));
    v = n;
    free(p);
  }
  return v;
}
```

The loop is translated into a recursive function whose argument serves as the loop index.

$$
\begin{array}{l}
\textbf{let } \text{loop} = \textbf{fix}(\text{loop}, (\Lambda\emptyset.\lambda n. \\
\quad \textbf{if } (n > 0\ [1]\ \textbf{at } \rho_{\text{glob}}) \textbf{ then} \\
\quad\quad \textbf{let } r = \textbf{newrgn } [1] \textbf{ in} \\
\quad\quad \textbf{let } p = \textbf{ref } (0\ [1]\ \textbf{at } \rho_{\text{glob}})\ [1]\ \textbf{at } r \textbf{ in} \\
\quad\quad \textbf{let } v := n \textbf{ in} \\
\quad\quad \textbf{freergn } r; \\
\quad\quad (\text{loop} \ @\ \emptyset)\ (n - 1\ [1]\ \textbf{at } \rho_{\text{glob}}) \\
\quad \textbf{else } () \\
)\ [1]\ \textbf{at } \rho_{\text{glob}}) \textbf{ in} \\
(\text{loop} \ @\ \emptyset)\ (10\ [1]\ \textbf{at } \rho_{\text{glob}}); \\
v
\end{array}
$$

Pointer Arithmetic and Finite Buffers. The C code below produces and consumes packets, serialising them in a buffer. SPEGION tracks pointer arithmetic and will catch patterns of use that exhaust the size of the buffer.

```
struct packet {int len; int payload[];};

struct packet * produce (struct packet *p, int data) {
  p->len = sizeof(int);  p->payload[0] = data;
  return p->payload + p->len;
}
```

```
struct packet * consume (struct packet *p, int* sum) {
  *sum += p->payload[0];
  return p->payload + p->len;
}

int main () {
  int sum = 0;
  void *buf = malloc(1000);
  struct packet *pp = buf, *cp = buf;
  pp = produce(pp, 1);  pp = produce(pp, 2);
  cp = consume(cp, &sum);  cp = consume(cp, &sum);
  return sum;
}
```

`main` function translates into the following Spegion. Statically, the loads and stores of calls to produce and consume are checked against the bounds of the region of buf. The return pointers type check, even if they escape the region of buf, matching idiomatic use of pointer arithmetic in C.

let r1 = **newrgn in**
let sum = **ref** (0 **at** r1) **in**
let buf = **newrgn** [1000] **in**
let init = **ref** (initPacket [1] **at** buf) **in**
let (pp, cp) =!init **in**
let pp = produce(pp, 1) **in**
let pp = produce(pp, 2) **in**
let cp = consume(cp, sum) **in**
let cp = consume(cp, sum) **in**
!sum

Non-Lexically-Scoped Lifetimes. The C code of Figure 1 allocates a linked list of three tasks in list order, takes a reference to the last task, and then frees the first two tasks. Spegion can represent and statically type programs like this. The translation is direct:

let re = **newrgn** [1] **in**
let end = 0 [1] **at** re **in**

let r_0 = **newrgn** [2] **in**
let t_0 = (0 [1] **at** ρ_{glob}, end) [2] **at** r_0 **in**
let $(n_0, \text{next}_0) = t_0$ **in**

let r_1 = **newrgn** [2] **in**
let t_1 = (0 [1] **at** ρ_{glob}, end) [2] **at** r_1 **in**
let $(n_1, \text{next}_1) = t_1$ **in**
let $\text{next}_0 = t_1$ **in**

let r_2 = **newrgn** [2] **in**
let t_2 = (0 [1] **at** ρ_{glob}, end) [2] **at** r_2 **in**
let $(n_2, \text{next}_2) = t_2$ **in**
let $\text{next}_1 = t_2$ **in**

let $n_2 = 1$ [1] **at** ρ_{glob} **in**

freergn t_0;

freergn t_1;

let $(n, \_) = !t_2$ **in**

n

Splitting. Our language represents idioms that C cannot represent, in particular *splitting*. In this idiom, the programmer creates a region. Subsequently they split the region into a part that is freed, and a part that is kept and used.

C cannot represent this idiom, but the C code below demonstrates the idea. The `alloc2` function mirrors the initial allocation, but to match the limitations of C it is implemented with an indirection and two allocations matching the subsequent split. The first part of the allocation made by `alloc2` is freed, and the second part is used.

```
struct indirect {int* p1; int* p2;};

struct indirect * alloc2 () {
  struct indirect *p = malloc(sizeof(struct indirect));
  p->p1 = malloc(sizeof(int));
  p->p2 = malloc(sizeof(int));
  return p;
}

int main () {
  struct indirect *pr = alloc2();
  *(pr->p2) = 1;
  free(pr->p1);
  return *(pr->p2);
}
```

Our language can represent this programming idiom. The region r is created in one call to **newrgn**, and then split into two parts. These parts can be used as a pair, but they can also be independently freed.

let $r =$ **newrgn** [2] **in**

let $r_1 =$ **split** [1] **in**

let $v_1 = (0$ [1] **at** $\rho_{\text{glob}})$ [1] **at** r_1 **in**

let $v_2 = (0$ [1] **at** $\rho_{\text{glob}})$ [1] **at** r **in**

let $(p_1, p_2) = (v_1, v_2)$ [2] **at** ρ_{glob} **in**

let $p_2 := 1$ [1] **at** ρ_{glob}**in**

freergn p_1;

$!p_2$

5.2 Refinement Types

Refinement types give the programmer the ability to use *predicates* in the type as a means of restricting the values described by the type [11, 22]. For example, the type:

$n : \text{Int} \to \text{Int}\ [n' \mid n' \leq n]$

describes a function from integers to integers which returns a value that must be less than or equal to the input. This is a simple example of a refinement type, but they can be used to encode many forms of specification.

A natural extension of our work would be to incorporate refinement types into our type system, allowing the programmer to reason about region sizes in the types of their programs. In this section we sketch out a simple refinement system based on the approach of Jhala and Vazou [18] that could be used in tandem with SPEGION's region sizes and sized allocations. We define a new sort of *predicates* which can be used in the type system to restrict the values described by a type. This section is mostly a sketch of the ideas, and we leave the full development of the system to future work. We begin with the syntax of predicates:

$$p := \quad x \mid n \mid \text{true} \mid \text{false} \mid p + p \mid p \cdot p \mid \neg p \mid \textbf{if } p \textbf{ then } p \textbf{ else } p \mid f(\overline{p}) \qquad \text{(predicates)}$$

Predicates are drawn from quantifier-free fragment of linear arithmetic and uninterpreted functions [3]. That is, a predicate p may be a variable x, a natural number constant n, boolean constants true and false, the sum or product of two predicates, the negation of a predicate, a conditional, or the application of an uninterpreted function f to a list of predicates. For this section, we extend SPEGION with type annotations and a new list datatype:

$$\text{List } (\tau, \rho) = \text{Nil} : \text{List } (\tau, \rho) \mid \text{Cons} : (\tau, \rho) \xrightarrow{\{\bot\}} \text{List } (\tau, \rho) \xrightarrow{\{\bot\}} \text{List } (\tau, \rho)$$

That is, a list is a value which is either empty or a pair of a value of type (τ, ρ) and a reference to another list. For simplicity the payload of the list must be stored in the same region as the list itself. One could envision a more complex type where the payload and indeed the tail of the list could be stored in separate regions. With the above type we can construct a function which takes an integer n and creates a list in a region of arbitrary size that has length n:

```
let r = newrgn
  in
  let buildList = fix(buildList, Λ{α, ρ, ϵ}λn.
    if n == 0
      then
        Nil at r
      else
        Cons (n, buildList @(Int, regionOf(r)) n − 1)) at r
in ...
```

With refinement types we could constrain this function to only return lists where the size of the list is less than or equal to the size of the region:

```
let r = newrgn [5]
in
  let buildList : ((n : Int, regionOf(r)) -φ-> (List (Int, regionOf(r))) [1 + (2 · n) < 5]) =
    fix(buildList, Λ{α, ρ, ϵ}λn.
      if n == 0
        then
          Nil [1] at r
        else
          Cons (n at [1], buildList @(Int, regionOf(r)) n − 1)) [1] at r
  in ...
```

where $\varphi = \{\textbf{alloc } 1 \text{ regionOf}(r)\} \sqcup (\{\epsilon\} \times \{\textbf{alloc } 1 \text{ regionOf}(r)\})$. Note we remove superfluous $\{\bot\}$ effects from the type annotation for brevity. Thus if we try to apply buildList to the integer 10, this application will fail as the size of the list produced is greater than the

size of the region associated with r. This is just a taste of what the interaction between refinement types and region sizes can offer. We believe that adding refinements to Spegion is a natural extension of the language, with few changes required to the core calculus.

6 Related Work

Key areas of related work for this research relate to the Rust programming language, and effect and region types in programming language design.

6.1 Rust

Rust is a popular systems programming language that aims to statically enforce memory safety, and it has also been the subject of significant academic research. Notable formal work includes Rust Belt [19] and Oxide [31]. The key concepts involved in Rust's approach to enforcing memory safety are *ownership* and *lifetimes*, with the latter corresponding to regions. As a key component of enforcing safety guarantees, Rust often limits or prohibits the introduction of *aliases* of pointers to mutable data, giving it a similar flavour to the substructural type system approaches detailed later in this section. This is, in part, because Rust is also concerned with preventing data races in concurrent code.

The type system might be extended to the concurrent context by reconciling the sequence of effects with the orderings imposed by a declarative concurrency semantics, like those of sequential consistency, ARMv8, or C++, but this is left to future work. Even in sequential code, however, there are difficulties involved in aliasing pointers and (for example) building cyclic data structures.

The following program is rejected by the Rust compiler:

```
fn main() {
    let mut a: i32 = 10;
    let b: &i32 = &a;

    {
        let c: &mut i32 = &mut a;
        *c = 20;
    }
    // use a,b ...
}
```

The compiler will point out `a` being borrowed multiple times (immutably by `b` and mutably by `c`). This program can be represented in our calculus with no issue, as our approach allows arbitray aliasing of mutable references. In general, Rust's expressive type system supports many patterns of memory usage, and we believe there is potential in combining ideas from Rust and from our effect-based approach.

6.2 Effect Types and Region Calculi

There is extensive prior work on using effects and regions in programming language design. Starting in the 1980s, Lucassen and Gifford, among others, explored the development of polymorphic effect type systems [14, 20]. They made use of a *kind* system consisting of types, which describe what sort of value an expression may evaluate to; effects, which describe what side effects may be performed when evaluating an expression; and regions, which describe groups of values in memory that are related. Building upon the idea of associating types with regions, Tofte and Talpin introduced the region calculus and region-based memory management [28, 29], which provided a structured way to control memory in higher order programs by associating types with parameters to statically track their lifetimes.

Region-based memory management formed the basis of ML-Kit, a compiler for Standard ML that used *region inference* to infer region annotations of source programs and statically determine memory usage [4, 25]. Rather than requiring programmers to manually annotate their programs with region/lifetime parameters, the compiler determines them automatically. Subsequent work investigated improving region inference, as it was prone to potentially producing programs that leak memory or unnecessarily extend the lifetime of objects [1, 26]. ML-Kit was augmented with a runtime garbage collector in order to collect memory left around inside regions which had become garbage but had not yet been de-allocated because it lived inside a still-alive region [16].

Existing papers have proven type safety theorems for variations of the region calculus. Helsen and Thiemann give an elegant syntactic proof of type safety for a region calculus with polymorphic types and recursive let-bindings, which was a primary source for the formalism in this paper [17]. They represent region de-allocation by introducing a special "dead" region identifier, and when a region goes out of scope they substitute this special identifier for the now-dead reagion. This also means they do not need to have a phase distinction between handles and region identifiers. However, they do not have a store in their operational semantics, which leads to a straightforward proof but unfortunately does not allow for mutable data. Subsequent work extended this approach with an explicit store, but had to remove polymorphic types [7]. Prior work also investigated proof of soundness for a fragment of the region calculus via a translation into a target language [2].

One of the major issues with using region-based memory management as proposed in the original region calculus is that the lexical scoping of regions means they are required to follow a stack or last-in-first-out (LIFO) ordering. This was observed by Aiken et al., demonstrating that extending the traditional region calculus by removing the LIFO restriction dramatically reduces memory usage (sometimes asymptotically) in functional programs [1]. Like our approach, they de-couple allocation and de-allocation, though their approach is for functional programs without mutable memory. They are also focused on region annotation inference in order to statically determine memory lifetimes of ML programs, and so they do not consider bounding or splitting regions and cannot track fine-grained effect details like an allocation consuming a specific amount of memory in a region.

Relatedly, Boudol presented a monomorphic variant of the region calculus that permitted early de-allocation [5]. A type system tracks both negative de-allocation effects and positive effects such as reads and writes, ensuring that the positive effects on a region cannot occur after the negative effect.

Further work explored adding explicit region allocation and de-allocation to a C-like language, where the safety of region de-allocation relied on keeping track of a dynamic *reference count* for objects in the heap [12, 13].

A prominent line of research that addressed the problem of LIFO lifetimes of regions is the use of *substructural types* in region type systems [10]. The Cyclone programming language, and related work on the calcluls of capabilities, allowed for type-safe manual memory management using a combination of type system features including substuctural types, allowing for both LIFO regions as well as first-class dynamic regions and unique pointers [8, 10]. This style of type system requires that programmers thread resources linearly through programs and avoid creating aliases. In contrast, a primary goal of our system is to avoid imposing this requirement of linearity in programs, and allowing for the aliasing of pointers that occurs in idiomatic imperative code.

Region types have also been used in programming language design for reasons beyond memory management. For example, in Gibbon, region types are used in combination with an effect type system to type programs that create and manipulate serialised data, where each region contains a serialised data structure [30].

7 Conclusion

We have proposed a type system for region-based memory management that allows implicit regions with non-lexical scoping and explicit size constraints. We have proven type safety for this system, and shown several directions in which this system can be extended. Based on this type system, we are currently developing an implementation of Spegion in the form of both an interpreter for Spegion as well as a static analysis tool targeting the C language.

References

1 Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: improving region-based analysis of higher-order languages. *SIGPLAN Not.*, 30(6):174–185, June 1995. doi:10.1145/223428.207137.

2 Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Region analysis and the polymorphic lambda calculus. In *Proceedings of the 14th Annual IEEE Symposium on Logic in Computer Science*, LICS '99, page 88, USA, 1999. IEEE Computer Society.

3 Clark W. Barrett, Aaron Stump, and Cesare Tinelli. The smt-lib standard version 2.0, 2010. URL: https://smt-lib.org/papers/smt-lib-reference-v2.0-r10.12.21.pdf.

4 Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 171–183, New York, NY, USA, 1996. Association for Computing Machinery. doi:10.1145/237721.237771.

5 Gérard Boudol. Typing safe deallocation. In Sophia Drossopoulou, editor, *Programming Languages and Systems*, pages 116–130, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. doi:10.1007/978-3-540-78739-6_10.

6 Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time java. *SIGPLAN Not.*, 38(5):324–337, May 2003. doi:10.1145/780822.781168.

7 Christiano Calcagno, Simon Helsen, and Peter Thiemann. Syntactic type soundness results for the region calculus. *Inf. Comput.*, 173(2):199–221, March 2002. doi:10.1006/inco.2001.3112.

8 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*, POPL '99, pages 262–275, New York, NY, USA, 1999. Association for Computing Machinery. doi:10.1145/292540.292564.

9 Morgan Deters and Ron K. Cytron. Automated discovery of scoped memory regions for real-time java. *SIGPLAN Not.*, 38(2 supplement):25–35, June 2002. doi:10.1145/773039.512433.

10 Matthew Fluet, Greg Morrisett, and Amal Ahmed. Linear regions are all you need. In *Proceedings of the 15th European Conference on Programming Languages and Systems*, ESOP'06, pages 7–21, Berlin, Heidelberg, 2006. Springer-Verlag. doi:10.1007/11693024_2.

11 Tim Freeman and Frank Pfenning. Refinement types for ml. *SIGPLAN Not.*, 26(6):268–277, May 1991. doi:10.1145/113446.113468.

12 David Gay and Alex Aiken. Memory management with explicit regions. *SIGPLAN Not.*, 33(5):313–323, May 1998. doi:10.1145/277652.277748.

13 David Gay and Alex Aiken. Language support for regions. *SIGPLAN Not.*, 36(5):70–80, May 2001. doi:10.1145/381694.378815.

14 David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 28–38, New York, NY, USA, 1986. Association for Computing Machinery. doi:10.1145/319838.319848.

15 Colin S. Gordon. Polymorphic iterable sequential effect systems. *ACM Trans. Program. Lang. Syst.*, 43(1), April 2021. doi:10.1145/3450272.

16 Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. *SIGPLAN Not.*, 37(5):141–152, May 2002. doi:10.1145/543552.512547.

17 Simon Helsen and Peter Thiemann. Syntactic type soundness for the region calculus. *Electronic Notes in Theoretical Computer Science*, 41(3):1–19, 2001. HOOTS 2000, 4th International Workshop on Higher Order Operational Techniques in Semantics (Satellite to PLI 2000). doi:10.1016/S1571-0661(04)80870-3.

18 Ranjit Jhala and Niki Vazou. Refinement types: A tutorial. *Found. Trends Program. Lang.*, 6(3–4):159–317, October 2021. doi:10.1561/2500000032.

19 Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158154.

20 J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. Association for Computing Machinery. doi:10.1145/73560.73564.

21 Henning Makholm. A region-based memory manager for prolog. *SIGPLAN Not.*, 36(1):25–34, October 2000. doi:10.1145/362426.362434.

22 Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements. *SIGPLAN Not.*, 38(9):213–225, August 2003. doi:10.1145/944746.944725.

23 Alan Mycroft, Dominic Orchard, and Tomas Petricek. Effect systems revisited–control-flow algebra and semantics. In *Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays on Semantics, Logics, and Calculi - Volume 9560*, pages 1–32, Berlin, Heidelberg, 2015. Springer-Verlag. doi:10.1007/978-3-319-27810-0_1.

24 Dominic Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. *SIGPLAN Not.*, 51(1):568–581, January 2016. doi:10.1145/2914770.2837634.

25 Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4):724–767, July 1998. doi:10.1145/291891.291894.

26 Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher Order Symbol. Comput.*, 17(3):245–265, September 2004. doi:10.1023/B:LISP.0000029446.78563.a4.

27 Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, and Peter Sestoft. Programming with regions in the ml kit (for version 4), October 2001.

28 Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 188–201, New York, NY, USA, 1994. Association for Computing Machinery. doi:10.1145/174675.177855.

29 Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997. doi:10.1006/inco.1996.2613.

30 Michael Vollmer, Chaitanya Koparkar, Mike Rainey, Laith Sakka, Milind Kulkarni, and Ryan R. Newton. Local: a language for programs operating on serialized data. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 48–62, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3314221.3314631.

31 Aaron Weiss, Olek Gierczak, Daniel Patterson, and Amal Ahmed. Oxide: The essence of rust, 2021. arXiv:1903.00982.