Typed Regions

Stefan Monnier

Département d'informatique et recherche opérationnelle
Université de Montréal
Montréal, QC, H3C 3J7, Canada
monnier@iro.umontreal.ca

Abstract

Standard type systems are not sufficiently expressive when applied to low-level memory-management code. Such code often requires some form of *strong update* (i.e. assignments that change the type of the affected location) and needs to reason about the relative position of objects in memory. We present a novel type system which, like alias types, provides a form of strong update, but with the advantage that it does not require the aliasing pattern to be statically described. It can also provide operations over sequential memory locations and allows covariant reference casts, both of which are required to implement a type-preserving stop&copy garbage collector that can properly collect cyclic data-structures. Finally, this type system is able to keep track of almost arbitrary properties of values and state, giving it a power formerly reserved to Hoare logic.

As the technology of certifying compilation and proof carrying code [16, 1, 8] progresses, the need to ensure the safety of the runtime system increases: if you go through the trouble of writing a foundational proof of safety of your code, you would rather not trust an unverified conservative garbage collector (GC) with your data. For this reason, it is important to be able to write a type-safe GC, but the state of the art in this matter is still completely impractical: it cannot even handle cyclic data-structures. This paper's main goals are thus:

- Argue that, in order to type-check a GC that can collect cyclic data-structures, the type system has to provide a form of assignment that can change the type of a location (i.e. a *strong update* [3]) even if the set of aliases to this location is unknown.
- Present a type system that provides such a facility. This type system allows the programmer to choose any mix of linear or intuitionistic typing of references and to seamlessly change this choice over time to adapt it to the current needs.

Traditional type systems are not well-suited to reason about type safety of low-level memory management such as explicit memory allocation, initialization, deallocation, or reuse. Existing solutions to these problems either have a very limited applicability or rely on some form of *linearity* constraint. Such constraints tend to be inconvenient and a lot of work has gone into relaxing them. For example, the alias-types system [22] is able to cleanly handle several of the points above, even in the presence of arbitrary aliasing, as long as the aliases can be statically tracked by the type system.

The reason why it is challenging to show type safety of low-level memory management is that for this kind of code, we end up having to prove some non-trivial properties about the code just to show its type safety. For example, type safety of a generational GC depends on the correct processing of the remembered-set (a data-structure holding the set of pointers from the old generation to the new).

An alternative approach would be to use Hoare logic [10] to show the correctness of the low-level code and then provide a type-safe interface to it. But it is not clear how those two parts would interact: the low-level code might be spread in pieces over a lot of code and might need to propagate complex invariants to the various pieces through the type-safe interface, as is the case for the code that maintains the remembered-set in the mutator. Furthermore as we start to encode more properties than basic type safety into our type systems, the difficulties we are seeing here will start to appear for more mundane code as well. This tendency can already be seen in the Vault project which uses an approach taken from alias-types to prove other properties of their code than just type safety.

The present work is thus an attempt to provide a middle ground between Hoare logic and traditional type systems. Additionally to the above stated goals, we make the following contributions:

- A language that subsumes traditional region calculi as well as alias-types calculi to simultaneously combine the benefits of traditional *intuitionistic* references and *linear* references.
- We introduce type cast on memory locations and strong update operations that work in the absence of any static aliasing information.
- Those operations generalize and enhance the widen operator used in [14] while relying on a much simpler soundness proof.
- We show how to use the calculus of inductive constructions (CiC) to track properties of state. This extends the work of Shao et al. [18] where they used CiC as their type language to track arbitrary properties of values.

Section 1 gives a quick preview of the basic idea developed in this paper. Section 2 introduces the problem of cyclic data-structures as well as two type systems on which our work is built. Section 3 describes the new language. Section 4 shows some examples of what the language can do. We then discuss related work and conclude.

1. Overview

The typed regions system presented in this paper is basically a hybrid between traditional region systems and alias types systems. In a traditional region system, the type of a pointer completely determines the type of the object to which it points. In alias types systems on the other hand, the type of the pointer does not carry any information about the type of the object to which it points. Instead, the type of the pointer only indicates the location to which it points, and a separate environment is used to look up the type of that location. Typed regions combines those two such that the type of a location is partly determined by the type of the pointers and partly by a separate environment.

$$
\begin{array}{lll}
(\textit{types}) & \sigma ::= t \mid \mathsf{int} \mid \sigma \times \sigma \mid \sigma \ \mathsf{at}\ \rho \mid \forall[\Delta]\{\Theta\}(\vec{\sigma}) \to 0 & \\
(\textit{values}) & v ::= x \mid i \mid (v, v) \mid \nu.n \mid \lambda[\Delta]\{\Theta\}(\Gamma).e & \\
(\textit{ops}) & \mathit{op} ::= v \mid \pi_i v \mid \mathsf{put}[\rho]\ v \mid \mathsf{get}\ v & \\
(\textit{terms}) & e ::= v[\vec{\sigma}](\vec{v}) \mid \mathsf{halt}\ v \mid \mathsf{let}\ x = \mathit{op}\ \mathsf{in}\ e \mid \mathsf{freergn}\ \rho; e & \\
 & \quad \mid \mathsf{let}\ r = \mathsf{newrgn}\ \mathsf{in}\ e \mid \mathsf{set}\ v := v; e & \\
(\textit{kinds}) & \kappa ::= \Omega \mid \mathsf{R} & \\
(\textit{regions}) & \rho ::= r \mid \nu & \\
(\textit{kind env}) & \Delta ::= \bullet \mid \Delta, t:\kappa & \\
(\textit{type env}) & \Gamma ::= \bullet \mid \Gamma, x:\sigma & \\
(\textit{heap type}) & \Psi ::= \bullet \mid \Psi, \nu.n \mapsto \sigma & \\
(\textit{region env}) & \Theta ::= \bullet \mid \Theta, \rho &
\end{array}
$$

Figure 1: Syntax of a region-based language.

The key idea is to introduce the concept of the *intended type* of a memory location, which stays constant throughout the lifetime of that location and thus corresponds to a traditional (intuitionistic) type, supplemented with a non-constant map that translates the intended type of each location to its *actual type*.

The intended type of a location is typically a high-level view of the type of objects that it can hold, that abstracts away time-varying details such as the fact that some fields might be temporarily uninitialized, or that the object is currently replaced by a forwarding pointer.

Let's say we have a pointer of type $\tau\ \mathsf{at}\ \rho.n$: [1] that means that it points to an object of intended type τ at location n in region ρ. Each region has a type φ. This type is a function that takes two arguments, the location n of an object and its intended type τ, and returns the actual type of the object $\varphi\ n\ \tau$.

Because the intended types are reflected in the types of pointers, they are kept immutable, since changing the intended type of a location would require updating the type of all the pointers to that location. On the other hand, assignments can change the actual type of a location by modifying the region's type φ.

If φ is a simple identity function that ignores n, then intended types and actual types are the same and we have basically a traditional region system. On the other hand, if φ ignores τ and only uses n to determine the actual type of a location, we have a system reminiscent of alias types.

The difference in power between typed regions and alias types is similar to the difference between destructive update and a simulation of it using functional update: when functionally updating an element shared by several data-structures, one needs to rebuild the spine that leads to this element for each data-structure where it is used, which requires one to keep track of those spines, whereas with destructive updates, the operation can be done without any knowledge of where this object is currently referenced.

2. Background

2.1 Cyclic structures

In the course of writing the *copy* routine of a garbage collector, we discovered that although current type systems can handle the case where the graph is acyclic, generalizing the code to properly handle cycles proves difficult. After experimenting with various algorithms, it became clear that the problem is more fundamental: current type systems are unable to type-check some generic code that can build arbitrary cyclic data-structures. By *generic*, we mean that it can apply to objects of any type. In other contexts, it could be called *polytypic*, or *intentionally polymorphic*. The most obvious examples are *gccopy* and *unpickle*.

[1] By convention, type-level expressions will use the meta-variable ρ for regions, τ for intended types, σ for actual types, and φ for other kinds of types such as region types.

To see this, let us look at a classic example, a datatype for doubly-linked lists:

$$\mathsf{datatype}\ \alpha\ \mathit{dlist} = \mathit{Node}\ \mathsf{of}\ \alpha * \alpha\ \mathit{dlist\ ref} * \alpha\ \mathit{dlist\ ref}$$

The SML type system allows us to declare this datatype and write functions to manipulate it, but does not offer us any way to create such an object because there is no base case to start from. But even if we have a base case, the type system can get in the way. Let us take another example:

$$\mathsf{datatype}\ \alpha\ \mathit{tree} = \mathit{Node}\ \mathsf{of}\ \alpha \mid \mathit{Branch}\ \mathsf{of}\ \alpha\ \mathit{tree\ ref} * \alpha\ \mathit{tree\ ref}$$

This time, we can construct such trees since we do have a base case, but if we want to construct an "infinite" tree with no actual *Node* in it, we still first need to build a *Node*. More specifically, in order to create a cyclic data-structure, we always need a base case to start from, even if the data-structure we want to get in the end does not contain any node of this base case any more. But in the above example, we can only build a base case if we have an object of the proper type α. Which means that a generic routine such as an unpickler or a copying GC needs to be able to construct from scratch the base case of any type that could be involved in a cyclic data-structure. In the tree example above, that means creating a *Node* of α for any α. Clearly this is not possible.

OCaml provides special support to build cyclic data-structures, such as the *dlist* example above, with $\mathsf{val\ rec}\ n = \mathit{Node}(0, n, n)$. This helps for specific code, but it only works for pre-determined cycles, whereas a copying GC simply does not even know when it is creating cycles.

Type systems that can decouple allocation from initialization are the key to solving this problem, but none of the systems developed so far are sufficiently flexible to handle the case of a generic function such as *unpickle*. More specifically, none of them know how to handle the case where the pointer to the allocated object *escapes* (i.e. is passed around and stored at arbitrary locations) before the object is initialized: when we allocate a new object, we obviously know its one and only alias, but we cannot fully initialize it yet because some of the values might not exist yet, and by the time we are done unpickling the children such that initialization can complete, there can be any number of aliases and we do not statically know them because the function is generic.

In order to type-check a practical copying GC, we need a new type system that is able to update the type (e.g. from uninitialized to initialized) of all the aliases to a particular object even when those aliases are not statically known.

2.2 Regions

Region-based type systems [20, 5] are the most practical systems offering type-safe explicit memory management. They provide a solution to the problem of safe deallocation, with a minimum of added constraints. Even though they do not offer any help when trying to type-check low-level code such as object initialization, their practicality makes them very attractive as a starting point. The

$$
\begin{array}{lll}
\textit{(types)} & \sigma & ::= t \mid \mathsf{int} \mid \rho \mid \forall[\Delta]\{\Theta\}(\vec{\sigma}) \to 0 \\
\\
\textit{(values)} & v & ::= x \mid i \mid \nu \mid \lambda[\Delta]\{\Theta\}(\Gamma).e \\
\textit{(ops)} & op & ::= v \mid \pi_i v \\
\textit{(terms)} & e & ::= v[\vec{\sigma}](\vec{v}) \mid \mathsf{halt}\ v \mid \mathsf{let}\ x = op\ \mathsf{in}\ e \mid \mathsf{free}\ \rho; e \\
 & & \quad \mid \mathsf{let}\ (r, x) = \mathsf{new}\ n\ \mathsf{in}\ e \mid \mathsf{set}\ \pi_i v := v; e
\end{array}
$$

$$
\begin{array}{lll}
\textit{(kinds)} & \kappa & ::= \Omega \mid \mathsf{Heap} \mid \mathsf{Loc} \\
\textit{(locations)} & \rho & ::= r \mid \nu \\
\textit{(type env)} & \Gamma & ::= \bullet \mid \Gamma, x{:}\sigma \\
\textit{(kind env)} & \Delta & ::= \bullet \mid \Delta, t{:}\kappa \\
\textit{(mem env)} & \Theta & ::= \bullet \mid \Theta, \rho{\mapsto}(\sigma, ..., \sigma)
\end{array}
$$

Figure 2: Syntax of an alias-types language.

idea behind region calculi is to only provide bulk deallocation of a whole region (group of objects) at a time. This way the type system only needs to keep track of regions rather than individual objects: the type of every pointer is simply annotated with the region that it references.

Figure 1 shows an example of such a language. $\mathsf{put}[\rho]\ v$ allocates v in region ρ; get v dereferences v and returns the object it points to; newrgn creates a new region; freergn deallocates it; set $v_1 := v_2$ places v_2 at the location to which v_1 points; $\nu.n$ is a pointer to the n^{th} object created by put in region ν and has type σ at ν if the object referenced has type σ; $\lambda[\Delta]\{\Theta\}(\Gamma).e$ is a function of type $\forall[\Delta]\{\Theta\}(\vec{\sigma}) \to 0$; functions are fully closed and use continuation passing style, so they never return (hence the $\to 0$ in the type); Δ is the list of type parameters; Θ lists the regions that need to be live at the time of the call; and $\vec{\sigma}$ and Γ lists the types of the value parameters. A function call $v[\vec{\sigma}](\vec{v})$ passes types $\vec{\sigma}$ and values $\vec{v}$ to function v. Type variables can have two kinds, depending on whether they range over the types σ of kind Ω of objects or over the types ρ of kind R of regions; Ψ does not appear in the terms but is used in the typing rules (not shown here) where it keeps track of the type of each memory location, such that for all reference value $\nu.n$ its type is $\Psi(\nu.n)$ at ν.

Region calculi do not all look like the above, of course. They do not all use fully closed functions and continuation passing style, for example, but a direct-style presentation would be more complex, as is the correct treatment of closure allocation.

Here is a sample function that creates a cyclic node of the *tree* datatype presented previously, assuming the language has been extended with support for datatypes:

$$
\begin{array}{l}
\textbf{\textit{mktree}}[r, t]\{r\}(x{:}t, k : \forall[r, t]\{r\}((\textit{tree}\ t)\ \mathsf{at}\ r) \to 0) \\
= \mathsf{let}\ y = \mathsf{put}[r]\ (\textit{Node}\ x)\ \mathsf{in} \\
\quad \mathsf{set}\ y := \textit{Branch}\ y\ y; k[r, t](y)
\end{array}
$$

The function expects two type arguments r and t, it expects the region r to be live, and expects an argument x of type t (which is only used temporarily to create the dummy *Node*) and a continuation argument k. The (omitted) kind of r is R and the kind of t is Ω. The put operation allocates memory and temporarily puts a dummy *Node* into it, while the set operation creates the actual cycle. The continuation k also expects two types r and t, it also expects region r to be live and expects a single value argument which is a pointer to a tree in region r. If k's type had $\{\}$ in place of $\{r\}$, it would force us to deallocate the region r before calling it and it would make n into a dangling pointer, which is allowed because liveness of the region is only needed and checked when dereferencing with get.

2.3 Alias types

The alias-types system [19, 22] was developed precisely to handle low-level code such as object initialization, memory reuse, and safe deallocation at the object level. To that end, the type of pointers is changed to carry no information about the type of the referenced object. Instead, the type of a pointer is just the location it is pointing to, so it does not need to change when the location's type or liveness changes. While it provides a lot of power when dealing with low-level code, it relies on an amount of static information which is not always available and definitely not available in our copying GC: if we had this information, we could also statically decide when to deallocate, so we would not need a GC in the first place.

Figure 2 shows the syntax of a very simple alias-types language. It can be thought of as a region-based language where the pointers can only point to regions rather than to objects inside them, where regions have been turned into tuples, and where objects inside regions are instead just fields of those tuples. put has disappeared since we cannot add fields to a tuple; get is replaced by π_i; set now only mutates a field of a tuple; pointers ν now just have type ν. The environment Ψ mapping locations to their types has been merged into Θ. When dereferencing a pointer of type ρ, we thus have to check the liveness and the type of the corresponding location by looking up ρ in Θ. $\mathsf{let}\ (r, x) = \mathsf{new}\ n\ \mathsf{in}\ e$ allocates a new object of size n and returns the location as both a value x and a type r. We could also have done this for regions so as to distinguish between the region type and the region value passed to put at runtime, but we conflated the two for simplicity.

Here is a sample code that takes a value of type t and creates an infinite list of this element (a 1-element circular list):

$$
\begin{array}{l}
\textbf{\textit{mklist}}[\epsilon, t]\{\epsilon\}(x : t, k{:}\forall[\epsilon, t, r]\{\epsilon, r \mapsto (t, r)\}(r) \to 0) \\
= \mathsf{let}\ (r, n) = \mathsf{new}\ 2\ \mathsf{in} \\
\quad \mathsf{set}\ \pi_0 n := x; \mathsf{set}\ \pi_1 n := n; k[\epsilon, t, r](n)
\end{array}
$$

The function expects two arguments ϵ and t where ϵ has kind Heap; it expects also that ϵ is live, and it expects two value arguments x of type t and k, the continuation. The type of the continuation shows that it expects three type arguments, where r is a location; it expects the heap ϵ to still be live and extended with a pair at location r holding the infinite list; and it expects a single value argument which is the pointer to that list. Since new only knows about the size of the object, it can only do allocation and the type at location r is originally set to $(\mathsf{int}, \mathsf{int})$ and is then incrementally updated by each set operation to (t, int) and then (t, r).

The ability to update a location's type is the key power of alias-types. But for that it relies crucially on the fact that the type system keeps track of pointer values. In particular, the types need to statically but precisely describe the shape of the heap. Witness the fact in the above example that the type of the circular list is not just *list* t but instead explicitly describes a 1-element cycle and thus disallows any other shape. The type language of [22] is of course much richer than what we show here, providing a lot more flexibility in the kind of heap shapes you can describe.

2.4 Calculus of inductive constructions

The calculus of inductive constructions (CiC) [17] that we use as our type language is an extension of the calculus of constructions (CC) [4], which is a higher-order typed λ-calculus. Addi-

$$
\begin{array}{llcl}
(\textit{regions}) & \rho:\mathsf{R}\ \kappa & ::= & r \mid \nu \\
(\textit{types}) & \sigma:\Omega & ::= & t \mid \mathsf{int} \mid \sigma\times\sigma \mid \tau\ \mathsf{at}\ \rho.n \\
 & & \mid & \forall[\Delta]\{\Theta\}(\vec{\sigma}) \to 0 \\
(\textit{values}) & v & ::= & x \mid i \mid (v,v) \mid \nu.n \mid \lambda[\Delta]\{\Theta\}(\Gamma).e \\
(\textit{operations}) & op & ::= & v \mid \pi_i v \mid \mathsf{get}\ v \\
(\textit{terms}) & e & ::= & v[\vec{\varphi}](\vec{v}) \mid \mathsf{halt}\ v \mid \mathsf{let}\ x = op\ \mathsf{in}\ e \\
 & & \mid & \mathsf{let}\ x = \mathsf{put}[\rho,\tau]\ v\ \mathsf{in}\ e \\
 & & \mid & \mathsf{let}\ r = \mathsf{newrgn}\ \varphi\ \mathsf{in}\ e \mid \mathsf{freergn}\ \rho; e \\
 & & \mid & \mathsf{set}\ v \stackrel{\varphi}{:=} v; e \mid \mathsf{cast}[P]\ \rho \mapsto \varphi; e
\end{array}
$$

$$
\begin{array}{llcl}
(\textit{sort}) & s & ::= & \mathsf{Kind} \mid \mathsf{Kscm} \mid \mathsf{Ext} \\
(\textit{ptm}) & \varphi,\tau,\kappa,P & ::= & s \mid x \mid \lambda x:\varphi.\ \varphi \mid \varphi\ \varphi \mid \Pi x:\varphi.\ \varphi \\
 & & \mid & \mathsf{Ind}(x:\varphi)\{\vec{\varphi}\} \mid \mathsf{Ctor}\ (i,\varphi) \\
 & & \mid & \mathsf{Elim}[\varphi](\varphi)\{\vec{\varphi}\} \\
(\textit{memory}) & M & ::= & \bullet \mid M, \nu.n \mapsto v \\
(\textit{heap type}) & \Psi & ::= & \bullet \mid \Psi, \nu.n \mapsto \tau \\
(\textit{type env}) & \Gamma & ::= & \bullet \mid \Gamma, x:\sigma \\
(\textit{kind env}) & \Delta & ::= & \bullet \mid \Delta, t:\kappa \\
(\textit{region env}) & \Theta & ::= & \bullet \mid \Theta, \rho \mapsto (\varphi, n)
\end{array}
$$

Figure 3: Syntax of the language.

tionally to being a powerful programming language, CC can encode Church's higher-order predicate logic via the Curry-Howard isomorphism [11]. Understanding the details of this language is not necessary for this paper, so we will only give a brief overview, starting with its syntax (as a pure type system [2]):

$$
\begin{array}{ll}
(\textit{sort}) & s ::= \mathsf{Kind} \mid \mathsf{Kscm} \mid \mathsf{Ext} \\
(\textit{ptm}) & \varphi ::= s \mid x \mid \lambda x:\varphi.\ \varphi \mid \varphi\ \varphi \mid \Pi x:\varphi.\ \varphi \\
 & \quad \mid \mathsf{Ind}(x:\varphi)\{\vec{\varphi}\} \mid \mathsf{Ctor}\ (i,\varphi) \mid \mathsf{Elim}[\varphi](\varphi)\{\vec{\varphi}\}
\end{array}
$$

x is a variable; $\varphi_1\ \varphi_2$ is a function application; $\lambda x:\varphi_1.\ \varphi_2$ is a function with argument x of type φ_1 and body φ_2; $\Pi x:\varphi_1.\ \varphi_2$ is the type of a function taking an argument of type φ_1 and returning a value of type φ_2. This is called a dependent product type and subsumes both the usual function type $\varphi_1 \to \varphi_2$ and the universal quantifier $\forall x:\varphi_1.\varphi_2$. When the bound variable x does not occur in φ_2, it can be abbreviated $\varphi_1 \to \varphi_2$.

The forms Ind, Ctor, and Elim, allow to resp. define, construct, and analyse inductive definitions, which are variants of ML datatypes and can be used to define integers and lists, for example. We will skip the details since in the remainder of this paper, we will use a more familiar ML-style notations. We will also sometimes abuse the BNF notation to informally define an inductive definition. We will, however, retain the Π notation, which can generally be read as a "for all" quantifier.

CiC has been shown to be strongly normalizing [24], hence the corresponding logic is consistent. It is supported by the Coq proof assistant [12], which we used to experiment with a prototype of the system presented in this paper.

3. Typed regions

Our new system of typed regions can be thought of as a hybrid between alias-types and the calculus of capabilities [5] supplemented with the calculus of inductive constructions (CiC), similarly to λ_H [18]. Where alias-types rely on a *linear* map of live locations's types and the calculus of capabilities relies on a linear set of live regions, we rely on a linear map of regions's types.

In a typical region calculus, the type of the object reachable from a pointer (its target) is entirely given by the type of the pointer. In contrast, in the alias-types system, the type of the pointer does not provide any direct information about the type of the target; instead, the target's type is kept in a linearly managed *type map* indexed by the pointer's type, which is the singleton type holding the object's location. Our new type system mixes the two, such that the pointer's type holds both the location and some information (called the *intended type*) about the object to which it points, while the remaining information is kept in a map of regions's types. The type of a region is a function that maps an object's location and its intended type to its actual type.

3.1 The language

The syntax of our language is shown in Fig. 3. The language uses continuation passing style and fully closed functions. $M, \Psi, \Gamma, \Delta, \Theta$ are environments used in the typing rules and operational semantics. Values v can be integers, pairs ($(v_1, v_2) : \sigma_1 \times \sigma_2$), references ($\nu.n : \tau$ at $\nu.n$), and functions. The terms do the following:

halt v	halt the machine, returning v as the result.
$\pi_i v$	select i^{th} from tuple v.
$v[\vec{\varphi}](\vec{v})$	make a tail-call to function v.
newrgn φ	allocate a new region of type φ.
freergn $\rho; e$	free the region ρ.
put$[\rho, \tau]$ v	allocate object v of intended type τ in region ρ.
get v	fetch the object pointed to by v.
set $v_1 \stackrel{\varphi}{:=} v_2; e$	update location v_1 with value v_2. φ is the new type of the location.
cast$[P]$ $\rho \mapsto \varphi; e$	Set the type of region ρ to φ. P is a proof that φ is a valid replacement.

The language is very similar to the simple region calculus presented before, except for the following differences:

- The type language is CiC. The kinds such as Ω (the kind of types σ) and R (the kind of regions ρ) along with the basic type constructors of σ are now defined directly as inductive definitions in CiC.
- The region environment Θ now contains not only a list of live regions, but a map from live regions to their type φ and size n. The type φ is a CiC function that maps the index of a location and its *intended type* to the actual type of that location.
- Pointer types have the form τ at $\rho.n$ rather than just σ at ρ, where n is the offset inside the region. Such a pointer points to an object of *intended type* τ but whose actual type can only be discovered by an indirection through the region's type: the target's type is $\varphi\ n\ \tau$ where $\Theta(\rho) = (\varphi, n)$.
- Just as before, a value $\nu.n$ has type $(\Psi(\nu.n))$ at $\nu.n$ but $\Psi(\nu.n)$ is an *intended type* and can be of any kind rather than only Ω.
- The region kind R now takes a parameter κ specifying the kind of intended types in this region. If $\rho:\mathsf{R}\ \kappa$, then the region's type φ will be of kind $\mathsf{Nat} \to \kappa \to \Omega$ and the term τ in τ at $\rho.n$ will have to have kind κ.
- put takes an additional parameter τ and returns a pointer of type τ at $\rho.n$ after checking that $v:\varphi\ n\ \tau$.
- set is now a strong update: it changes the type of the location to φ which has kind $\kappa \to \Omega$. This type needs to be provided

$$
\begin{array}{ll}
(M;\Theta;\mathsf{let}\ x = v\ \mathsf{in}\ e) & \Longrightarrow (M;\Theta;e[^{v}/_{x}]) \\
(M;\Theta;\mathsf{let}\ x = \pi_i(v_1,v_2)\ \mathsf{in}\ e) & \Longrightarrow (M;\Theta;e[^{v_i}/_{x}]) \\
(M;\Theta;\nu.n[\vec{\varphi}](\vec{v})) \quad \text{where} \\
\quad M(\nu.n) = \lambda[\overrightarrow{t:\kappa}]\{\Theta'\}(\overrightarrow{x:\sigma}).e & \Longrightarrow (M;\Theta;e[^{\vec{\varphi},\vec{v}}/_{\vec{t},\vec{x}}]) \\
(M;\Theta;\mathsf{let}\ x = \mathsf{get}\ \nu.n\ \mathsf{in}\ e) & \Longrightarrow (M;\Theta;e[^{M(\nu.n)}/_{x}]) \\
(M;\Theta,\nu\mapsto(\varphi,n);\mathsf{freergn}\ \nu;e) & \Longrightarrow (M\backslash\nu;\Theta;e) \\
(M;\Theta;\mathsf{let}\ r = \mathsf{newrgn}\ \varphi\ \mathsf{in}\ e) & \Longrightarrow (M;\Theta,\nu\mapsto(\varphi,0);e[^{\nu}/_{r}]) \\
(M;\Theta,\nu\mapsto(\varphi,n) & \\
\ ;\mathsf{let}\ x = \mathsf{put}[\nu,\tau]\ v\ \mathsf{in}\ e) & \Longrightarrow (M,\nu.n\mapsto v \\
 & \quad ;\Theta,\nu\mapsto(\varphi,n+1);e[^{\nu.n}/_{x}]) \\
(M;\Theta,\nu\mapsto(\varphi',n') & \\
\ ;\mathsf{set}\ \nu.n :\overset{\varphi}{=} v;e) & \Longrightarrow (M,\nu.n\mapsto v \\
 & \quad ;\Theta,\nu\mapsto(\mathit{upd}\ \varphi'\ n\ \varphi,n');e) \\
(M;\Theta,\nu\mapsto(\varphi,n) & \\
\ ;\mathsf{cast}[P]\ \nu\mapsto\varphi';e) & \Longrightarrow (M;\Theta,\nu\mapsto(\varphi',n);e)
\end{array}
$$

Figure 4: Operational semantics of the language.

because there are many valid choices and they are not all equivalent.

- newrgn now takes a parameter φ which is the initial type of the region.
- cast is a new operator made necessary by the fact that some of our types are functions and since function equivalence is undecidable in our type language, we sometimes need to manually help the type-checker.

We have decided to manipulate whole objects rather than words and not to split allocation from initialization. It is easy to change the language to provide alloc, load, and store instead of put, get, and set, but the typing rules become more verbose and so does the code in the examples.

3.2 Semantics

Figure 4 shows the typed operational semantics. The machine state is defined as the 3-tuple $(M;\Theta;e)$. The region environment Θ is only marginally used in put where we need to find the next free location in the region. After type erasure, Θ would only keep track of the size of each region. The rules use the auxiliary type function *upd* which takes a function φ taking an argument of type Nat and returns a function equal to it, except at point n where it now returns φ' instead:

$$\mathit{upd}\ \varphi\ n\ \varphi' = \lambda i.\mathsf{if}\ (i=n)\ (\varphi')\ (\varphi\ i)$$

Worth noting in the reduction rules is that cast is indeed a no-op which does not affect anything other than types. Also the typed region environment after a function call is Θ rather than Θ': the two should be equivalent and e has been type-checked using Θ', but the use of Θ makes it more obvious that the operation is just a jump.

The formation rules for environments are given in Fig. 5 together with the definition of a well-formed machine state $\vdash (M;\Theta;e)$.The judgment $\Delta \vdash^{CiC} \varphi : \kappa$ used in those rules, taken directly from CiC and not shown here, states that φ has kind κ in environment Δ. The judgment $\vdash \Delta$ is also taken directly from CiC. The judgment $\vdash \Psi$ checks that each intended type has a kind consistent with its region. The judgment $\Delta \vdash \Gamma$ checks that each variable's type has kind Ω. The judgment $\Delta \vdash \Theta$ checks that the type of each region is consistent with its kind and that each region has only one binding in

$\vdash \Psi$	memory type Ψ is well-formed
$\vdash \Delta$	type environment Δ is well-formed
$\Delta \vdash \Theta$	region type env Θ is well-formed in context Δ
$\Delta \vdash \Gamma$	value env Γ is well-formed in context Δ
$\Psi;\Theta \vdash M$	memory M is well formed in Ψ and Θ
$\vdash (M;\Theta;e)$	machine state is well-formed

$$\frac{\bullet \vdash^{CiC} \varphi_{ij} : \kappa \qquad \bullet \vdash^{CiC} \nu_i : \mathsf{R}\ \kappa}{\vdash \nu_0.0 \mapsto \varphi_{00},\ldots,\nu_m.n \mapsto \varphi_{mn}} \qquad \frac{}{\vdash \bullet}$$

$$\frac{\vdash \Delta \qquad \Delta \vdash^{CiC} \kappa : s}{\vdash \Delta, t:\kappa}$$

$$\frac{}{\Delta \vdash \bullet} \qquad \frac{\begin{array}{c}\Delta \vdash \Theta \qquad \rho \notin \mathit{Dom}(\Theta) \qquad \Delta \vdash^{CiC} \rho : \mathsf{R}\ \kappa \\ \Delta \vdash^{CiC} \varphi : \mathsf{Nat} \to \kappa \to \Omega \qquad \Delta \vdash^{CiC} n : \mathsf{Nat}\end{array}}{\Delta \vdash \Theta, \rho \mapsto (\varphi, n)}$$

$$\frac{\Delta \vdash^{CiC} \sigma_i : \Omega}{\Delta \vdash x_0:\sigma_0,\ldots,x_n:\sigma_n}$$

$$\frac{\begin{array}{l}\forall i:0..n\ .\ \forall j:\mathsf{Nat}\ .\ \text{if } j \geq n_i \text{ then } \nu_i.j \notin \mathit{Dom}(\Psi) \\ \qquad\qquad \text{else } \Psi;\bullet;\Theta;\bullet \vdash M(\nu_i.j) : \varphi_i\ j\ (\Psi(\nu_i.j))\end{array}}{\Psi;\Theta = \nu_0 \mapsto (\varphi_0,n_0),\ldots,\nu_n \mapsto (\varphi_n,n_n) \vdash M}$$

$$\frac{\Psi;\Theta \vdash M \qquad \bullet \vdash \Theta \qquad \vdash \Psi \qquad \Psi;\bullet;\Theta;\bullet \vdash e}{\vdash (M;\Theta;e)}$$

Figure 5: Environment formation rules.

Θ (a linearity constraint). This is important because in order to be able to free region ρ or to change its type, we need to be sure that the same physical region is not referred to somewhere else under some other name. The judgment $\Psi;\Theta \vdash M$ is the one that expresses the invariant that needs to hold so that intended types, actual types and region types are all consistent with one another. It checks that the actual type of each object in M indeed matches the result of applying to its intended type (stored in Ψ) the corresponding region's type (stored in Θ). In a simple region system, the well-formedness of M is sometimes written as $\vdash M : \Psi$ so in our case the judgment $\Psi;\Theta \vdash M$ could be thought of as $\vdash M : \Theta(\Psi)$ where Θ is taken as a function that interprets the intended memory type Ψ and returns an actual memory type. An important detail about the rule is that it verifies that Ψ has no binding for not-yet-allocated locations. This is needed because the intended type is immutable, so Ψ can only be extended but none of its existing bindings can be modified.

Figure 6 shows the formation rules for types and terms. The equivalence rule $\Theta \lhd \Theta'$ is used to formalize the fact that Θ is not ordered. In the rule for pointer values $\nu.n$, dangling pointers to dead regions can have any type (because the rule for get prevents dereferencing them), but pointers past the allocation line of a region are disallowed by checking that they have a binding in Ψ. This way, pointers are live iff their region is live. In the rule for functions, the function is forced to be fully closed by typing its body in an environment that does not include Γ or Δ'. Note that typed regions only have an impact on the typing of references and function values: any other standard types such as sum types or existential packages can be added without any difficulty.

The rule for the trivial operation where *op* is v is not shown but just delegates to the rule for values. The auxiliary rule for $v \mapsto \sigma$

$\Delta \vdash \vec{\sigma} : \overrightarrow{t:\kappa}$	actual type arguments $\vec{\sigma}$ match formal arguments $\overrightarrow{t:\kappa}$
$\Theta \sim \Theta'$	Θ is equivalent to Θ'

$$\frac{\Delta \vdash^{CiC} \sigma : \kappa \qquad \Delta \vdash \vec{\sigma} : \overrightarrow{t:\kappa}[^{\sigma}/_{t}]}{\Delta \vdash \sigma, \vec{\sigma} : t:\kappa, \overrightarrow{t:\kappa}} \qquad \frac{}{\Theta \sim \Theta} \qquad \frac{\Theta \sim \Theta'}{\Theta, \rho \mapsto (\varphi, n) \sim \Theta', \rho \mapsto (\varphi, n)}$$

$\Psi; \Delta; \Theta; \Gamma \vdash v : \sigma$	value v has type σ
$\Psi; \Delta; \Theta; \Gamma \vdash v \mapsto \sigma$	value v is a pointer to an object of type σ
$\Psi; \Delta; \Theta; \Gamma \vdash op : \sigma$	operation op returns a value of type σ

$$\frac{}{\Psi; \Delta; \Theta; \Gamma \vdash n : \mathsf{int}} \qquad \frac{\Psi; \Delta; \Theta; \Gamma \vdash v_i : \sigma_i}{\Psi; \Delta; \Theta; \Gamma \vdash (v_1, v_2) : \sigma_1 \times \sigma_2} \qquad \frac{}{\Psi; \Delta; \Theta; \Gamma \vdash x : \Gamma(x)} \qquad \frac{\nu \in \mathit{Dom}(\Theta) \Rightarrow \tau = \Psi(\nu.n)}{\Psi; \Delta; \Theta; \Gamma \vdash \nu.n : \tau \ \mathsf{at} \ \nu.n}$$

$$\frac{\vdash \Delta \qquad \Delta \vdash \Theta \qquad \Delta \vdash \overrightarrow{x:\sigma} \qquad \Psi; \Delta; \Theta; \overrightarrow{x:\sigma} \vdash e}{\Psi; \Delta'; \Theta'; \Gamma \vdash \lambda[\Delta]\{\Theta\}(\overrightarrow{x:\sigma}).e : \forall[\Delta]\{\Theta\}(\vec{\sigma}) \to 0} \qquad \frac{\Theta(\rho) = (\varphi, m) \quad \Psi; \Delta; \Theta; \Gamma \vdash v : \tau \ \mathsf{at} \ \rho.n}{\Psi; \Delta; \Theta; \Gamma \vdash v \mapsto \varphi \ n \ \tau} \qquad \frac{\Psi; \Delta; \Theta; \Gamma \vdash v \mapsto \sigma}{\Psi; \Delta; \Theta; \Gamma \vdash \mathsf{get} \ v : \sigma}$$

$$\frac{\Psi; \Delta; \Theta; \Gamma \vdash v : \sigma_1 \times \sigma_2}{\Psi; \Delta; \Theta; \Gamma \vdash \pi_i v : \sigma_i}$$

$\Psi; \Delta; \Theta; \Gamma \vdash e$	expression e is well-formed

$$\frac{\Psi; \Delta; \Theta; \Gamma \vdash op : \sigma \qquad \Psi; \Delta; \Theta; \Gamma, x:\sigma \vdash e}{\Psi; \Delta; \Theta; \Gamma \vdash \mathsf{let} \ x = op \ \mathsf{in} \ e} \qquad \frac{\Psi; \Delta; \bullet; \Gamma \vdash v : \mathsf{int}}{\Psi; \Delta; \bullet; \Gamma \vdash \mathsf{halt} \ v}$$

$$\frac{\Delta \vdash^{CiC} \rho : \mathsf{R} \ \kappa \qquad \Delta \vdash^{CiC} \tau : \kappa \qquad \Psi; \Delta; \Theta; \Gamma \vdash v : \varphi \ n \ \tau \qquad \Psi; \Delta; \Theta, \rho \mapsto (\varphi, n+1); \Gamma, x:\tau \ \mathsf{at} \ \rho.n \vdash e}{\Psi; \Delta; \Theta, \rho \mapsto (\varphi, n); \Gamma \vdash \mathsf{let} \ x = \mathsf{put}[\rho, \tau] \ v \ \mathsf{in} \ e} \qquad \frac{\Psi; \Delta; \Theta; \Gamma \vdash v \mapsto \forall[\overrightarrow{t:\kappa}]\{\Theta'\}(\vec{\sigma}) \to 0 \qquad \Delta \vdash \vec{\varphi} : \overrightarrow{t:\kappa} \quad \Theta \sim \Theta'[^{\vec{\varphi}}/_{\vec{t}}] \quad \Psi; \Delta; \Theta; \Gamma \vdash v_i : \sigma_i[^{\vec{\varphi}}/_{\vec{t}}]}{\Psi; \Delta; \Theta; \Gamma \vdash v[\vec{\varphi}](\vec{v})}$$

$$\frac{\Psi; \Delta; \Theta; \Gamma \vdash e}{\Psi; \Delta; \Theta, \rho \mapsto (\varphi, n); \Gamma \vdash \mathsf{freergn} \ \rho; e} \qquad \frac{\Psi; \Delta; \Theta, \rho \mapsto (\varphi, n); \Gamma \vdash v : \tau \ \mathsf{at} \ \rho.m \quad \Psi; \Delta; \Theta, \rho \mapsto (\varphi, n); \Gamma \vdash v' : \varphi' \ \tau \quad \Psi; \Delta; \Theta, \rho \mapsto (\mathit{upd} \ \varphi \ m \ \varphi', n); \Gamma \vdash e \quad \Delta \vdash^{CiC} \rho : \mathsf{R} \ \kappa \qquad \Delta \vdash^{CiC} \varphi' : \kappa \to \Omega}{\Psi; \Delta; \Theta, \rho \mapsto (\varphi, n); \Gamma \vdash \mathsf{set} \ v \overset{\varphi'}{:=} v'; e} \qquad \frac{\Delta \vdash^{CiC} \varphi : \mathsf{Nat} \to \kappa \to \Omega \quad \Psi; \Delta, r:\mathsf{R} \ \kappa; \Theta, r \mapsto (\varphi, 0); \Gamma \vdash e}{\Psi; \Delta; \Theta; \Gamma \vdash \mathsf{let} \ r = \mathsf{newrgn} \ \varphi \ \mathsf{in} \ e}$$

$$\frac{\Delta \vdash^{CiC} P : \Pi i:0..n-1.\, \Pi t:\kappa.\, (\varphi \ i \ t) = (\varphi' \ i \ t) \quad \Psi; \Delta; \Theta, \rho \mapsto (\varphi', n); \Gamma \vdash e \qquad \Delta \vdash^{CiC} \rho : \mathsf{R} \ \kappa}{\Psi; \Delta; \Theta, \rho \mapsto (\varphi, n); \Gamma \vdash \mathsf{cast}[P] \ \rho \mapsto \varphi'; e}$$

Figure 6: Static semantics of the language.

is used by get and by function calls to make sure that the pointer is not dangling and also to find out the actual type at that location. The rule for put checks that the region is live, with appropriate type, and updates the size. The rule for function calls checks that the arguments have the proper kind and type and also that the current region environment is equivalent to the one expected by the function. The rule for cast checks that the region is live and that P indeed proves that the new type σ' is equivalent to the old type for all the live locations and for all possible intended types. It does not pay attention to which intended types are actually used at those locations because those intended types are, in general, not known yet when type-checking. The rule for freergn checks that the region is live before and that the rest of the code does not use the region any more. The rule for set does not use the auxiliary rule $v \mapsto \sigma$ because it does not care about the actual type before assignment, since it will overwrite the location. Instead it just checks that the pointer is live and that the new value matches the new type.

3.3 Properties of the language

We state here a few important properties of the language. The proofs can be found in [13]. Since our type language is CiC, we know it is strongly normalizing and confluent.

Lemma 3.1 (Type Preservation)
If $\vdash (M;\Theta;e)$ *and* $(M;\Theta;e) \Longrightarrow (M';\Theta';e')$*, then* $\vdash (M';\Theta';e')$.

Lemma 3.2 (Progress)
If $\vdash (M;\Theta;e)$*, either* $e = \mathsf{halt}\ v$ *or* $(M;\Theta;e) \Longrightarrow (M';\Theta';e')$.

Lemma 3.3 (Complete Collection)
If $\vdash (M;\Theta;e)$ *and* $\forall \nu.n \in \mathit{Dom}(M)\ .\ \nu \in \mathit{Dom}(\Theta)$ *and* $(M;\Theta;e) \Longrightarrow (M';\Theta';e')$*, then* $\forall \nu.n \in \mathit{Dom}(M')\ .\ \nu \in \mathit{Dom}(\Theta')$*. In particular, if* $e' = \mathsf{halt}\ v$ *then* $M' = \bullet$.

PROOF. The proof follows trivially from inspection of the reduction rules. The corollary for halt v uses additionally the preservation lemma to verify that $\vdash (M';\Theta';e')$. □

4. Examples

To get an idea of how the language is used, here are some examples of how you can simulate the behavior of other systems in this language.

4.1 Simple regions

Given the differences between the simple region system presented before and the typed region system, here is what we have to do to translate a program written in the simple region system:

- Add region types and region sizes to Θ. Assuming that intended types are always the same as actual types, the regions's types will always be $\lambda i.\lambda t.t$. On the other hand their size will keep changing, so it needs to be passed around as extra type argument of kind Nat.
- Add the kind of intended types to region kinds R. Since the intended types are the same as actual types, the region kind is $\mathsf{R}\,\Omega$.
- Annotate put with the type of the allocated object.
- Tell set that the region's type is left unchanged. At first it seems like using $\lambda t.t$ for the type annotation φ is enough. But then the type of the region after set is:

$$\begin{array}{c}\lambda i.\mathsf{if}\ (i=n)\ (\lambda t.t)\ ((\lambda i.\lambda t.t)\ i)\\ \Downarrow\ \text{which reduces to}\ \Downarrow\\ \lambda i.\mathsf{if}\ (i=n)\ (\lambda t.t)\ (\lambda t.t)\end{array}$$

  where n is the location referenced by the pointer. But this is not equal to $\lambda i.\lambda t.t$, so we need to add a cast to reshape the region's type to what we need:

$$\mathsf{cast}[P]\ \rho \mapsto \lambda i.\lambda t.t; e$$

  And P needs to show equivalence between the two types:

$$\begin{array}{c}P : \Pi i\!:\!0..n\!-\!1.\,\Pi t\!:\!\kappa.\,((\lambda i.\mathsf{if}\ (i=n)\ (\lambda t.t)\ (\lambda t.t))\ i\ t) = ((\lambda i.\lambda t.t)\ i\\ \Downarrow\\ P : \Pi i\!:\!0..n\!-\!1.\,\Pi t\!:\!\kappa.\,((\mathsf{if}\ (i=n)\ (\lambda t.t)\ (\lambda t.t))\ t) = t\\ \Downarrow\\ P = \lambda i.\lambda t.{<}\,\lambda b.((\mathsf{if}\ b\ (\lambda t.t)\ (\lambda t.t))\ t) = t\,{>}\\ \mathsf{if}\ (i=n)\ (\mathit{refl\_equal}\ t)\ (\mathit{refl\_equal}\ t)\end{array}$$

  Where "*refl_equal* φ" is the proof of $\varphi = \varphi$ and where "$<f>$ if $b\ \varphi_1\ \varphi_2$" is a proof of $f\ b$ if φ_1 is a proof of f *true* and φ_2 is a proof of f *false*.
- Pointer types σ at ρ need to be turned into σ at $\rho.n$, but since we do not know n we have to hide it with an existential package: $\exists n.\sigma$ at $\rho.n$.

For convenience, let's define the two type functions $\mathit{id} = \lambda t.t$ and $\mathit{idr} = \lambda i.\mathit{id}$. The example from the introduction becomes:

$$\begin{array}{l}\textbf{\textit{mktree}}[r,n,t]\{r\mapsto(\mathit{idr},n)\}(x\!:\!t,k:\forall[r,n,t]\{r\mapsto(\mathit{idr},n)\}((\mathit{tree}\ t)\ \mathsf{at}\ r.(n\\ =\ \mathsf{let}\ y = \mathsf{put}[r,\mathit{tree}\ t]\ (\mathit{Node}\ x)\ \mathsf{in}\\ \quad \mathsf{set}\ y \overset{id}{:=} \mathit{Branch}\ y\ y; \mathsf{cast}[P]\ r \mapsto \mathit{idr}; k[r,n{+}1,t](y)\end{array}$$

Here we avoided using existential packages by making the actual address of the new tree passed to the continuation k explicit. To make writing such code easier, one would define a new set that does a traditional weak update by coupling the set and the cast above. The burden of keeping track of the allocation limit n is not as bad as it seems since each function is actually a basic block with a fixed number of allocations in it.

4.2 Stacks

We can use a typed region to represent a contiguous stack of objects. More specifically, a stack is a region S of kind R Unit. The locations in a stack have no intended type, so we use the dummy Unit kind which has a single element denoted $()$. A function using such a stack will have the following shape:

$$\begin{array}{l}\lambda[S\!:\!\mathsf{R}\ \mathsf{Unit}, \mathit{st}\!:\!\mathit{ST}, \mathit{sp}, \mathit{ss}\!:\!\mathsf{Nat}, \ldots]\\ \quad\{S\mapsto(\mathit{st},\mathit{ss}),\ldots\}\\ \quad\left(\begin{array}{l}\mathit{sp}\!:\!()\ \mathsf{at}\ S.\mathit{sp}\\ \mathit{k}\!:\!\forall[\mathit{st'}\!:\!\mathit{ST},\ldots]\\ \qquad\{S\mapsto(\lambda n.\mathsf{if}\ (n<\mathit{sp})(\mathit{st}\ n)(\mathit{st'}\ n),\mathit{ss}),\ldots\}\\ \qquad(\ldots)\to 0\\ \ldots\end{array}\right).\\ \quad\ldots\end{array}$$

The kind of the stack type is defined as $\mathit{ST} = \mathsf{Nat} \to \mathsf{Unit} \to \Omega$ where the second argument, of kind Unit, is unused. The type function *st*, maps the locations in the stack to their current type. The top of the stack is kept in *sp*, both as a value and a type. The size of the stack is kept in the type variable *ss*. The type of the continuation k specifies that the stack S when we return to k has

the same type as before for all elements below *sp*, while the rest can be changed at will and described by the type function *st'*.

When pushing a new element on the stack, we somehow need to find the address of the next consecutive element in region S. Let us extend the language with such a facility, called $\mathsf{ifnext}\ v\ (x, t\ .\ e_1)\ (e_2)$ that simply takes a pointer v, checks whether it is the last element in the region, if so continues with e_2, otherwise continues with e_1 with x bound to a pointer to the next consecutive element in the region and t bound to its intended type. Now pushing an object on the stack is done by:

$$\mathsf{set}\ sp \overset{\lambda\_.\sigma}{:=} v; \mathsf{ifnext}\ sp\ (sp, \_\ .\ e)\ (\mathsf{halt}\ 0)$$

where e is the rest of the computation, $\_$ indicates that we do not care about this argument (it will always be $()$ anyway), σ is the type of v, and where we could do something more clever in case of stack overflow.

Popping elements is simply done implicitly by reverting to a previously saved value of *sp*.

Before returning to the continuation k we need to find an appropriate *st'*, and show that the current type of S is indeed the one expected by the continuation. This proof will be passed to a cast operation just before jumping to k. The first part is trivial since the current stack type is exactly what we need to pass as *st'*. The second part requires proving that $\lambda n.\mathsf{if}\ (n < sp)(st\ n)(st'\ n)$ is equivalent to *st'*. The type of S at that point will have the following shape:

$$\begin{array}{l} st' = \lambda n.\mathsf{if}\ (n = sp)\ (\lambda\_.\sigma_1) \\ \qquad\qquad (\mathsf{if}\ (n = sp{+}1)\ (\lambda\_.\sigma_2) \\ \qquad\qquad\quad (\mathsf{if}\ \ldots\ (st\ n))) \end{array}$$

The proof mainly entails showing that *st* and *st'* (i.e. the old and the new stack types) are equal w.r.t. locations below *sp*, which is easy to show since pushing only modifies locations above *sp*.

4.3 Adoption and focus

The Vault language [6] uses a mix of alias types and traditional types to provide a powerful user-level language used to write device drivers, where the type system is used among other things to check the correct ordering of operations such as open, write, close. They do that by allowing some types to be *tracked*, meaning that they behave linearly like alias types. Operations on objects of tracked types can update their type, so the type system can keep track of their state.

In a subsequent paper [7], the authors extended the language with *adoption* which allows to "untrack" a tracked object so it behaves intuitionistically and *focus* which does the reverse. This allows them to use intuitionistic references while at the same time being able to temporarily strengthen them to a linear reference, using the focus construct:

$$\mathsf{let}\ x = \mathsf{focus}\ e_1\ \mathsf{in}\ e_2$$

The expression e_1 has an intuitionistic type, but while inside e_2, the variable x to which it is bound has a linear type and can thus be modified using strong update. To guarantee soundness, they impose the restriction that even though strong update can modify the type of the object referred to by x, its type should be the same at the end of e_2 as it was at its beginning, so the type modification is only temporary. Also any other object in the same region as e_1 is temporarily unavailable.

We can encode something similar. Let's assume we have a region ρ like the ones used in the simple regions example above. Its type is *idr*, i.e. it does not depend on the location of a given object just as is the case in intuitionistic systems. So references will hide their location using an existential wrapper: $\exists n.\sigma\ \mathsf{at}\ \rho.n$. A code equivalent to the focus construct will then look like:

$$\mathsf{let}\ \langle n, x\rangle = \mathsf{open}\ e_1\ \mathsf{in}\ e_2; \mathsf{cast}[P]\ \rho \mapsto \mathit{idr}\,; \ldots$$

Where e_1 has type $\exists n.\sigma\ \mathsf{at}\ \rho.n$ and x has thus type $\sigma\ \mathsf{at}\ \rho.n$. Within e_2, strong update can be used at will on x, but before reaching cast the actual type at location n needs to be reset to σ. Remember that focus had the additional constraint that other references to ρ could temporarily not be used. In our case, we can still use them, although we will not be able to do anything useful with them unless we know their aliasing relationship with x.

4.4 Garbage collection

While traditional type systems can be used to implement a type-preserving GC [14], those GCs suffer from significant restrictions. In order to implement a realistic type-preserving GC, the type system needs to be able to handle operations such as scanning a region of memory, tracking various complex properties, or, as argued earlier, strong update in the presence of unknown aliasing patterns.

4.4.1 Scanning regions

Creating a new reference out of a reference to an adjacent object in order to scan a region of memory can be done safely in region systems since liveness is a property of regions rather than objects. But in traditional region systems, the new reference cannot be used because nothing is known about its type. It would typically have type $\exists t.t\ \mathsf{at}\ \rho$.

In a typed region system, the new reference will have a type apparently just as useless: $\exists t.t\ \mathsf{at}\ \rho.n$. But the difference is that we know that the object at $\rho.n$ will not just have any type t but instead will have a type of the form $\varphi\ n\ t$. It is then possible for the programmer to ensure that φ gives enough information to carry on the scan.

4.4.2 Generational GC

To see how the strong update is used, let us sketch the types of a generational GC. Let us assume that the source language whose heap we want to collect only has integers, immutable pairs, and mutable ref-cells. Let us take a very simple case where we have 3 regions: all the ref-cells go into region R, whereas the pairs are divided between the nursery Y and the old space O. Since all data is immutable except for ref-cells, we can take the R regions as a conservative approximation of the remembered-set.

We will use the source-level types for the intended types:

$$\tau ::= \mathsf{int} \mid \tau \times \tau \mid \mathsf{ref}\ \tau$$

To translate those source-level types into their low-level representation (their actual type), we create a type function M which takes the three regions and the source type as parameters:

$$\begin{array}{ll} M\ r\ o\ y\ (\mathsf{int}) & \Rightarrow \mathsf{int} \\ M\ r\ o\ y\ (\mathsf{ref}\ \tau) & \Rightarrow \exists n{:}\mathsf{Nat}.\tau\ \mathsf{at}\ r.n \\ M\ r\ o\ y\ (\tau_1 \times \tau_2) & \Rightarrow \exists x{\in}\{o, y\}.\exists n{:}\mathsf{Nat}.(\tau_1, \tau_2)\ \mathsf{at}\ x.n \end{array}$$

The translation of a ref-cell is an intuitionistic reference to region R, as sen by the use of an existential package to hide the actual location inside R. For the translation of a pair, we assume that the language is also extended with bounded-existential packages to represent the fact that the reference can be to either region O or region Y. The detailed description of such an extension can be

found in [13]. The regions's types while the mutator is running are:

$$Y \mapsto \lambda n.\lambda(t_1, t_2).(M\ R\ O\ Y\ t_1) \times (M\ R\ O\ Y\ t_2)$$
$$O \mapsto \lambda n.\lambda(t_1, t_2).(M\ R\ O\ O\ t_1) \times (M\ R\ O\ O\ t_2)$$
$$R \mapsto \lambda n.\lambda t.M\ R\ O\ Y\ t$$

Note how the type of O calls M with both parameters o and y set to O such that those objects cannot refer to Y, thus enforcing the generation barrier. When the collection takes place, the GC, starting from the roots, copies objects from Y to O. Once this is done, it needs to go through the remembered-set R and redirect any reference still pointing to Y. To make it possible to free Y, the type of R should end up as:

$$R \mapsto \lambda n.\lambda t.M\ R\ O\ O\ t$$

so as to reflect the fact that no object in Y is reachable from ref-cells in R. The redirection is done by scanning R and updating each ref-cell at a time. The type of R needs to be kept uptodate as this proceeds, of course, recording the progress of the boundary m between the ref-cells already redirected and the ones left to process:

$$R \mapsto \lambda n.\lambda t.\mathsf{let}\ r = \mathsf{if}\ (m > n)\ (O)\ (Y)\ \mathsf{in}\ M\ R\ O\ r\ t$$

Note how the type of R needs to be updated with each redirection step, requiring a strong update, even though no static information about which other data in R or O might point to the same location we are updating.

5. Related work

The calculus of capabilities [5] was the first calculus to provide safe explicit memory deallocation while allowing dangling pointers. The linear handling of our regions was strongly influenced by that work.

In the work on TAL [15], the authors showed a simple way to handle the problem of separating allocation from object initialization, without resorting to any form of linearity.

Alias types [19, 22] was also a major source of inspiration for our system. It's a type system designed specifically to handle low-level code. In that work, strong update is the only form of update available. Separating object initialization from allocation is very easy, as is explicit deallocation and memory reuse.

The Vault language [6] took the work on alias types and both extended it and gave it a surface syntax (so as to enable the programmer to give that needed aliasing information). In the first paper, they mostly showed how to integrate classical intuitionistic references with alias-types-style statically tracked references. They also showed that tracking references to region objects allows their system to subsume a region type system. The main difference compared to our work is that you have to choose once and for all whether a reference should be intuitionistic or linear (i.e. *tracked*).

In a subsequent paper [7] they addressed that limitation by introducing the operators *adoption*, which allows the user to make a linear reference intuitionistic, and *focus*, which does the converse. This significantly increases the expressiveness of their user-level language. Sadly, these constructs are not applicable to our GC problem for two reasons: First, *adoption* is too high-level for us and seems difficult to adapt to a low-level language. Second, although *focus* does allow some of what we need (such as strong update in the presence of arbitrary aliasing), it requires the type change to be transient, limited to the scope of the construct.

Wang and Appel [23] built a tracing garbage collector on top of a region-based calculus, thus providing both type safety and completely automatic memory management.

Monnier et al. [14] extended Wang and Appel's work by using intensional type analysis [9] to provide a generic *copy* function and to use existential packages to encode closures. The also presented a very primitive form of generational collection and a formally sound, though very ad-hoc, treatment of forwarding pointers. We build directly on their work.

Vanderwaart and Crary [21] design a type system that enforces that the programs correctly manipulate the details of the stack layout required by a particular GC which is kept implicit. The system is designed for a sophisticated GC which uses a static table indexed by the return address to describe the activation frames, much like stack-walking implementations of exception handling. The type system checks among other things that the table provided by the program is correct and used consistently.

Shao et al. [18] proposed to use CiC as the type calculus of a programming language. This allows sophisticated type manipulation and enables programs to express arbitrary properties of the values manipulated. We reuse their idea with the same purpose but by virtue of the rest of the type system we can additionally capture arbitrary properties of the state.

6. Conclusion

We have presented a novel type system that offers an unusual flexibility to play with the typing of memory locations. This type system offers the ability to choose any mix of linear or intuitionistic typing of references and to change this choice over time to adapt it to the current needs. It is able to handle strong update of memory locations even in the presence of unknown aliasing patterns. The reliance on CiC allows very sophisticated type manipulations.

We have shown how to encode the features of other systems in this language. We have also developed a prototype implementation of an extension of this language, using Coq, in which we have written a type-preserving generational garbage collector that can handle cycles and that allows the mutator to perform destructive assignment.

References

[1] A. W. Appel. Foundational proof-carrying code. In *Annual Symposium on Logic in Computer Science*, pages 247–258, June 2001.

[2] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science (volume 2)*. Oxford Univ. Press, 1991.

[3] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Symposium on Programming Languages Design and Implementation*, pages 296–310, New York, NY, 1990. ACM Press.

[4] T. Coquand and G. P. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[5] K. Crary, D. Walker, and G. Morrisett. Typed memory management in a calculus of capabilities. In *Symposium on Principles of Programming Languages*, pages 262–275, San Antonio, TX, Jan. 1999.

[6] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Symposium on Programming Languages Design and Implementation*, May 2001.

[7] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Symposium on Programming Languages Design and Implementation*. ACM Press, May 2002.

[8] N. A. Hamid, Z. Shao, V. Trifonov, S. Monnier, and Z. Ni. A syntactic approach to foundational proof-carrying code. In *Annual Symposium on Logic in Computer Science*, pages 89–100, Copenhagen, Denmark, July 2002.

[9] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Symposium on Principles of*

Programming Languages, pages 130–141, Jan. 1995.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, Oct. 1969.

[11] W. A. Howard. The formulae-as-types notion of constructions. In *To H.B.Curry: Essays on Computational Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[12] G. Huet, C. Paulin-Mohring, et al. The Coq proof assistant reference manual. Part of the Coq system version 6.3.1, May 2000.

[13] S. Monnier. *Principled Compilation and Scavenging*. PhD thesis, Yale University, 2003.

[14] S. Monnier, B. Saha, and Z. Shao. Principled scavenging. In *Symposium on Programming Languages Design and Implementation*, pages 81–91, May 2001.

[15] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. In *Symposium on Principles of Programming Languages*, pages 85–97, Jan. 1998.

[16] G. C. Necula. Proof-carrying code. In *Symposium on Principles of Programming Languages*, Jan. 1997.

[17] C. Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. In M. Bezem and J. Groote, editors, *Proc. TLCA*. LNCS 664, Springer-Verlag, 1993.

[18] Z. Shao, B. Saha, V. Trifonov, and N. Papaspyrou. A type system for certified binaries. In *Symposium on Principles of Programming Languages*, pages 217–232, Jan. 2002.

[19] F. Smith, D. Walker, and G. Morrisett. Alias types. In *European Symposium on Programming*, 2000.

[20] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Symposium on Principles of Programming Languages*, pages 188–201, Jan. 1994.

[21] J. C. Vanderwaart and K. Crary. A typed interface for garbage collection. In *Types in Language Design and Implementation*, New Orleans, LA, Jan. 2003.

[22] D. Walker and G. Morrisett. Alias types for recursive data structures. In *International Workshop on Types in Compilation*, Aug. 2000.

[23] D. C. Wang and A. W. Appel. Safe garbage collection = regions + intensional type analysis. Technical Report TR-609-99, Princeton University, 1999.

[24] B. Werner. *Une Théorie des Constructions Inductives*. PhD thesis, A L'Université Paris 7, Paris, France, 1994.