Statically tracking state with Typed Regions

Stefan Monnier
Université de Montréal
monnier@iro.umontreal.ca

1. INTRODUCTION

Static type systems have proved to be tremendously effective formal systems, making specification and verification sufficiently lightweight and intuitive that most programmers use them without even realizing it. Not only that, but they have shown to adapt very well to most modern language features. The downside is that the properties one can express and verify with garden variety type systems is fairly limited, especially concerning properties which depend on side-effects and tracking the state of objects. Such properties usually require the use of other formal methods, such as some kind of Hoare logic [Hoare 1969; Reynolds 2002], which are targeted specifically at reasoning about imperative programs. And it so happens that those alternative formal methods do not always integrate well with modern language features such as higher-order functions, and polymorphism. The work presented here presents a new type system which has an expressive power comparable to that of a Hoare logic, thus hopefully combining the best of both worlds.

As the technology of certifying compilation and proof carrying code [Necula 1997; Appel 2001; Hamid et al. 2002] progresses, the need to ensure the correctness of the runtime system increases: the carefully designed proof system risks breaking down completely if one of the primitives of the runtime system does not behave *exactly* as intended. The best way to approach this problem is to trim down the trusted part of the runtime system, starting with the garbage collector. For this reason, it is important to be able to write a verifiably type-safe GC, which may come bundled with the application code, rather than hard-coded in the trusted runtime system. but the state of the art in this matter is still very much impractical. So one of this paper's main goal is to present a type system that is sufficiently flexible and powerful to type-check low-level code such as stack manipulation, pointer reversal, or the various parts of a generational garbage collector. This type system provides a form of assignment that can change the type of a location (i.e. a *strong update* [Chase et al. 1990]) even if the set of aliases to this location is not statically known, and it 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 [Walker and Morrisett 2000] 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.

Traditional type systems mostly ensure memory safety. But of course, when typing low-level memory management code, proving that the code does not break the memory safety invariants often amounts to the same as proving that the code is correct. For this

reason the distinction between mere type safety and full correctness becomes blurred in such circumstances. 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).

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 combines traditional region calculi with alias-types calculi to simultaneously offer 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.

—We present the first type-preserving generational collector that allows the mutator to perform destructive assignment.

—We show how to use the calculus of inductive constructions (CiC) to track properties of state. This extends the work of Shao et al. [Shao et al. 2002] where they used CiC as their type language to track arbitrary properties of values.

—We show an encoding of the *focus* operator [Fähndrich and DeLine 2002]. Contrary to the original *focus* operator, our encoding does not impose any scoping constraint and it can be applied to multiple objects in the same region at the same time.

By *intuitionistic* references we mean references that can be freely copied, as opposed to *linear* references which either cannot be copied or whose copies are constrained by the requirement that the type system be able to track them, as with alias types.

Section 2 gives a quick preview of the basic idea developed in this paper. Section 3 introduces the problem of cyclic data-structures as well as two type systems on which our work is built. Section 4 describes the new language. Section 5 shows some examples of what the language can do. We then discuss related work and conclude.

2. OVERVIEW

The system of typed regions presented in this paper is 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 a pointer does not carry any information about the type of the object to which it points. Instead, the type of the pointer indicates just the location to which it points, and a separate environment is used to look up the type of that location. The system of 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.

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 a pointer is currently reversed, or that the object has been replaced by a forwarding pointer.

Let's say we have a pointer of type τ *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, such that the type-level application $\varphi\ n\ \tau$ returns the actual type σ of the object.

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, the actual type of a location is mutable since we can change it 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 the equivalent of 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 main contribution of this type system is to allow mixing those two modes by using for φ a function that mixes those two modes, typically by giving a few specific types to some memory locations, à la alias types, while using traditional types for the remaining locations. Additionally, the mix between those two modes can be changed dynamically; for example, in a pointer-reversal traversal algorithm, we may start with a φ that ignores n, and as we go down the structure, we change φ to adjust the type of those locations whose pointers are currently reversed, and as we go back up the structure, we adjust φ again to note that the object has a "normal" type again, so that in the end we recover a φ that ignores n.

The system also allows us to dynamically recover aliasing information that could not be tracked statically. E.g. in the pointer-reversal case, given an arbitrary pointer into the region, we do not know at first whether this pointer points to an object whose pointers have been reversed or not, but we can look at the object's tag to find out. Once the tag has been checked, the type of the pointer can be refined to indicate whether it points to one of the pointer-reversed objects or not, thus lifting this dynamic aliasing information back into the 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.

Typed regions can be used to attack problems traditionally solved using Hoare logic: regions's types are somewhat equivalent to assertions in Hoare logic, and intended types are basically used as invariants on specific memory locations.

3. BACKGROUND

Before formally presenting the language of typed regions, we present here some of the anecdotal motivation for this work, as well as a quick refresher course on regions and alias types which intends to give a better and more insightful understanding of our language by highlighting the similarities and differences between those systems.

[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.

```
(* Copy heap depth-first from region T to region F. *)
gc_copy : ∀α.α^F → α^T
gc_copy x =
  if is_immediate_value x then return x
  else if x.visited then return x.fwd
  else let x' = alloc[T] x.size
       (* Set forwarding pointer before recursing, to break cycles. *)
       set x.fwd = x'; set x.visited = true
       for i = 0 ... x.size do set x'[i] = gc_copy x[i]
       return x'
```

Fig. 1. The core of GC's copy function

3.1 Cyclic structures

In the course of writing the *gc_copy* routine of a type-preserving 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 can prove 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* [Hinze 1999], or *intensionally polymorphic* [Harper and Morrisett 1995]. The most obvious examples are *gc_copy*, shown below, and *unpickle*, also known as *unmarshall*.

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

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

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. There are several ways to work around this problem:

—Work in a weakly typed setting where we can easily separate allocation from initialization. If we want type safety, this is clearly unacceptable.

—Add an otherwise unused variant to the data type, to forcefully provide a base case. This incurs major costs because every piece of code that manipulate this data structure must now handle the additional variant, which in the present case adds conditional jumps to every access.

—OCaml provides special support to build cyclic data-structures, such as *dlist* above, with $\text{val rec } n = \mathit{ref}\,(\mathit{Node}(0,\ n,\ n))$. This can then be used as a base case, without adding a dummy variant to the type. Compared to the weakly typed solution, this is not bad, but it still incurs an extra cost because we need to pre-initialize every field with those dummy variants until the "real initial" value can be assigned to it. [Syme 2005] proposes to extend this even further, but at even higher runtime costs.

The situation becomes even more problematic is we add *genericity* to the cycles: Figure 1 shows the core of a depth-first *gc_copy* function. Its tentative type should look like $\forall\alpha.\alpha^{\mathrm{F}} \to \alpha^{\mathrm{T}}$, to indicate that it receives an argument in region F and returns an object of the same type but allocated in region T. Since traditional type systems do not know how to type an uninitialized object, the code cannot be typed without first adjusting it so that

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

Fig. 2. Syntax of a region-based language.

appropriate initial values for each field are passed to alloc. This implies a performance penalty since we have to pre-initialize the object in alloc. But also, we do not know statically the type of the fields of the object, so the code has to be able to build such initial values for any type α.

If we want to keep the same performance and genericity that we get in the weakly typed setting, yet show that *gc_copy* correctly preserves types, we need a type system that can decouple allocation from initialization, but none of the systems developed so far [Walker and Morrisett 2000; Morrisett et al. 1998] are sufficiently flexible to handle the case of a function such as *gc_copy*. The tricky part comes when trying to decide which type to give to the field $x.\mathit{fwd}$:

—when it gets assigned, the value that is assigned to it comes straight out of alloc so its type should indicate that the object is not yet initialized.

—on the other hand, when it is used (because $x.\mathit{visited}$ is set), it should be of type α^{T} which is expected to be initialized. And as it happens, whether $x.\mathit{fwd}$ points to a fully initialized object or not depends on whether we are in the middle of copying a cycle, which is not statically known.

Fundamentally, the problem is that none of the existing type systems know how to handle the case where a pointer to an object *escapes* (i.e. is passed around and stored at arbitrary locations such as $x.\mathit{fwd}$) before the object is initialized.

In order to type-check the above code, we need a new type system that is able to update the type of x' (e.g. from uninitialized to initialized) even though we do not statically know all its aliases.

3.2 Regions

Region-based type systems [Tofte and Talpin 1994; Crary et al. 1999] 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 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 2 shows an example of such a language, in continuation passing style, inspired by the calculus of capabilities [Crary et al. 1999]. Values include:

c	an integer constant of type int;
(v_1, v_2)	a pair of type $\sigma_1 \times \sigma_2$;
$\Lambda t{:}\kappa.v$	a type abstraction of type $\forall t{:}\kappa.\sigma$;
$v[\varphi]$	an instantiation of type abstraction v with type φ;
$\nu.n$	a reference of type σ *at* ν to the n^{th} object in region ν;
$\lambda\{\Theta\}(\Gamma).e$	a function of type $\{\Theta\}(\vec{\sigma}) \rightarrow 0$, where Γ is the list of parameters and Θ is the list of regions that need to be live at the time of the call;

Operations are the following:

$v(\vec{v})$	call function v with arguments $\vec{v}$;
halt v	terminate with value v;
$\pi_i v$	select the i^{th} word of pair v;
newrgn	create a new region and return it;
freergn ρ	deallocate region ρ;
put$[\rho]$ v	place object v in region ρ and return a reference to it;
get v	dereference v and return the object it points to;
set $v_1 := v_2$;	assign value v_2 to the location to which v_1 points;

We distinguish here between region variables r and region values ν, while ρ can be either of the two. This means that every execution of newrgn generates a fresh new region value. For simplicity we do not distinguish between term-level region descriptors which exist at run time, and type-level regions which only exist at compile time. Regions are manipulated as types: ρ is a type of kind *R* and σ is a type of kind Ω, while φ is used for types that can be of any kind. Heap types Ψ do not appear in the terms but are used in the typing rules (not shown here) where they keep track of the type of each memory location, such that the type of a reference value $\nu.n$ is $\Psi(\nu.n)$ *at* ν.

The typing rules enforce that all operations like get and put only access regions mentioned in the list Θ of live regions. So all we need for freergn to be safe is that it removes the freed region from Θ thus preventing future operations from accessing it.

Region aliasing (where two or more region variables refer to the same region) needs to be kept under very tight control so as to prevent things like: deallocation of the region via one of the two variables, and then use of the dead region via the other variable. This is usually enforced with some sort of linearity constraint. In the region system presented here, this manifests itself in the use of $\oplus$ in the region environment, which indicates that this environment is not treated like a set. Since the region environment can only be manipulated indirectly by the program, the system can enforce by construction that no region should ever appear more once in Θ. We will expand on this subtle point in Sec. 5.2.

Here is a sample function that creates a cyclic node of the *tree* datatype presented previ-

$$
\begin{array}{lll}
(\textit{kinds}) & \kappa & ::= \Omega \mid \mathsf{Heap} \mid \mathsf{Loc} \\
(\textit{type env}) & \Gamma & ::= \bullet \mid \Gamma, x{:}\sigma \\
(\textit{mem env}) & \Theta & ::= \bullet \mid \epsilon \mid \Theta \oplus \rho \mapsto (\sigma, ..., \sigma) \\
 & & \\
(\textit{locations}) & \varphi, \rho & ::= r \mid \nu \\
(\textit{types}) & \varphi, \sigma & ::= t \mid \mathsf{int} \mid \forall t{:}\kappa.\sigma \mid \rho \mid \{\Theta\}(\vec{\sigma}) \to 0 \\
 & & \\
(\textit{values}) & v & ::= x \mid c \mid \Lambda t{:}\kappa.v \mid \nu \mid \lambda\{\Theta\}(\Gamma).e \\
(\textit{terms}) & e & ::= v(\vec{v}) \mid \mathsf{halt}\ v \mid \mathsf{let}\ x = \pi_i v\ \mathsf{in}\ e \mid \mathsf{let}\ x = v[\varphi]\ \mathsf{in}\ e \\
 & & \quad \mid \mathsf{let}\ (r, x) = \mathsf{new}\ n\ \mathsf{in}\ e \mid \mathsf{set}\ \pi_i v := v; e \mid \mathsf{free}\ \rho; e
\end{array}
$$

Fig. 3. Syntax of an alias-types language.

ously, assuming the language has been extended with support for datatypes.[2].

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

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 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 expects region r to still 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 y into a dangling pointer, which is allowed because liveness of the region is only needed and checked when dereferencing with get.

3.3 Alias types

The alias-types system [Smith et al. 2000; Walker and Morrisett 2000] 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 ρ it is pointing to, so it does not need to change when the location's type or liveness changes.

Figure 3 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 to ν now just have type ν. The environment Ψ which mapped locations to their types is 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; a pointer of type ρ points to an object of type $\Theta(\rho)$. $\mathsf{let}\ (r, x) = \mathsf{new}\ n\ \mathsf{in}\ e$ allocates a new object of size n and returns the location

[2]We use syntactic sugar such that $\textit{name}[t_0{:}\kappa_0, ..., t_n{:}\kappa_n]\{\Theta\}(\Gamma) = e$ stands for $\textit{name} = \Lambda t_0{:}\kappa_0 .... \Lambda t_n{:}\kappa_n.\lambda\{\Theta\}(\Gamma).e$

as both a value x and a type r. In a region calculus, newrgn could also similarly return separate term and type variables so as to distinguish between the region type and the region descriptor passed to put at runtime.

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):

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

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 a type argument r holding the location of the allocated object; 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 and precisely describe the shape of the heap. Witness the fact in the above example that instead of being just *List* t, the type of the circular list explicitly describes a 1-element cycle and thus disallows any other shape. The type language of [Walker and Morrisett 2000] is of course much richer than what we show here, providing a lot more flexibility in the kind of heap shapes you can describe.

While they provides a lot of power when dealing with low-level code, alias types rely on an amount of static information which is not generally available, e.g. when reasoning about a 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.

3.4 Calculus of inductive constructions

In most languages, type expressions are very simple, basically limited to a few constants such as int or void plus some constructors like array, struct, union, or maybe $\forall$ to combine them. In contrast our type expressions are written in a full fledged language complete with functions, datatypes, and even primitive recursion: the calculus of inductive constructions (CiC) [Paulin-Mohring 1993].

CiC is an extension of the calculus of constructions (CC) [Coquand and Huet 1988], which is a higher-order dependently typed λ-calculus. Additionally to being a powerful programming language, CC can encode Church's higher-order predicate logic via the Curry-Howard isomorphism [Howard 1980]: propositions are encoded as types, and proofs of those propositions are terms of the corresponding type. It also has a very useful property for us: proof checking (i.e. type checking) is decidable. Understanding the details of this language is not necessary for this paper; the only relevant aspects are that we can write and manipulate logical propositions and proofs, and that the language includes higher order functions and a kind of datatype (called *inductive definitions*) which makes it feel familiar to SML programmers. We can easily define natural numbers, tuples, lists, and other data

```
(* The basic logical connectives. *)
```
Inductive *False* : Kind := `(* No constructor. *)`.
Inductive *True* : Kind := () : *True*.
type *Unit* = *True*.
type $\neg k = k \to$ *False*.
Inductive " $\vee$ " $(k_1, k_2 :$ Kind$)$: Kind := *left* : $k_1 \to (k_1 \vee k_2)$ | *right* : $k_1 \to (k_1 \vee k_2)$.
Inductive " $\wedge$ " $(k_1, k_2 :$ Kind$)$: Kind := *conj* : $k_1 \to k_2 \to (k_1 \wedge k_2)$.

```
(* Reification of the builtin intensional equality. *)
```
Inductive " $=$ " $(k :$ Kind$)$ $(x : k)$: $k \to$ Kind := *eq_refl* : $x = x$.

```
(* Predicate for list membership. *)
```
Inductive " $\in$ " $(k :$ Kind$)$ $(x : k)$: *List* $k \to$ Kind
:= *inbase* : $\Pi l.x \in (x :: l)$
| *inskip* : $\Pi l, y.x \in l \to x \in (y :: l)$.

```
(* Predicate for list element extraction. *)
```
Inductive *Pick* $(k :$ Kind$)$ $(x : k)$: *List* $k \to$ *List* $k \to$ Kind
:= *pbase* : $\Pi l.$*pick* $x\ l\ (x :: l)$
| *pskip* : $\Pi l_1, l_2, y.$*pick* $x\ l_1\ l_2 \to$ *pick* $x\ (y :: l_1)\ (y :: l_2)$.

```
(* Predicate for order insensitive list equality. *)
```
Inductive " $\sim$ " $(k :$ Kind$)$: *List* $k \to$ *List* $k \to$ Kind
:= *uleq_refl* : $\Pi l.l \sim l$
| *uleq_cons* : $\Pi l_1, l_2, x, l_1', l_2'.$
pick $x\ l_1'\ l_1 \to$ *pick* $x\ l_2'\ l_2 \to l_1' \sim l_2' \to l_1 \sim l_2$.

```
(* Update a function φ at point n to return φ'. *)
```
upd $\varphi\ n\ \varphi' = \lambda i.$if $(i == n)\ (\varphi')\ (\varphi\ i)$.

Fig. 4. Auxiliary CiC declarations for typing rules.

structures such as representations of type expressions:

Inductive *Nat* : Kind := *zero* : *Nat* | *succ* : *Nat* $\to$ *Nat*.
Inductive *List* t : Kind := *nil* : *List* t | *cons* : $t \to$ *List* $t \to$ *List* t.
Notation "$x :: l$" := *cons* $x\ l$.
Inductive Ω^τ : Kind := int : Ω^τ | *ref* : $\Omega^\tau \to \Omega^\tau$ | *pair* : $\Omega^\tau \to \Omega^\tau \to \Omega^\tau$.

The above uses a surface syntax with added sugar. The abstract syntax we use for CiC is (given as a pure type system [Barendregt 1991]):

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

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 analyze inductive definitions. Without syntactic sweetener, the definition of *Nat* above becomes:

$$\begin{array}{l} \mathit{Nat} = \mathsf{Ind}(\mathit{Nat}{:}\mathsf{Kind})\{\mathit{Nat}, \mathit{Nat} \to \mathit{Nat}\} \\ \mathit{zero} = \mathsf{Ctor}\,(0, \mathit{Nat}) \\ \mathit{succ} = \mathsf{Ctor}\,(1, \mathit{Nat}) \end{array}$$

The Ind form defines an inductive type by listing the type of each of its n constructors, much like algebraic data types (except for a strict positivity requirement to make sure the definition is truly inductive). $\mathsf{Ctor}\,(i, \varphi)$ is the i^{th} constructor of the inductive type φ. $\mathsf{Elim}[\varphi_1](\varphi_2)\{\vec{\varphi}\}$ is the elimination constructs which analyzes its φ_2 argument to decide which of its $\vec{\varphi}$ branches to evaluate; φ_1 describes the return type, which may depend on φ_2. The three builtin constants are the following: Kind is the kind of all inductively defined data, Kscm is the type of Kind, and Ext is the type of Kscm. You can safely ignore Ext, and even Kscm is only used infrequently.

In the remainder of this paper, we will generally use the more familiar Coq-style notations or we will even 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.

We can also define propositions such as ordering on numbers:

$$\begin{array}{l} \mathsf{Inductive}\ \ \leq : \mathit{Nat} \to \mathit{Nat} \to \mathsf{Kind} \\ \qquad := \mathit{le\_n} : \Pi n : \mathit{Nat}.\, n \leq n \\ \qquad\quad \mid \mathit{le\_s} : \Pi n, m : \mathit{Nat}.\, n \leq m \to n \leq \mathit{succ}\, m \\ n_1 < n_2 \equiv \mathit{succ}\, n_1 \leq n_2 \end{array}$$

Figure 4 shows some definitions to manipulate region types and unordered lists which we use later on in the typing rules. Note that those definitions often drop some arguments, such as the k argument to $=$ or $\in$ because Coq can infer them and provide them implicitly. *upd* 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. The predicates $x \in l$ and $l_1 \sim l_2$ are used to lookup and reorder order-insensitive lists. We can easily prove that $\sim$ is commutative, that $\mathit{Pick}\, x\, l_1\, l_2 \to l_2 \sim x :: l_1$, and that $x \in l_2 \to \exists l_1.\mathit{Pick}\, x\, l_1\, l_2$.

CiC has been shown to be strongly normalizing [Werner 1994], hence the corresponding logic is consistent. It is supported by the Coq proof assistant [Huet et al. 2000], which we used to experiment with a prototype of the system presented in this paper.

3.5 λ_H

Shao et al. [2002] presented a programming language that can represent and manipulate arbitrarily complex propositions and proofs, by using CiC as their type language. More specifically, λ_H types are CiC terms, and hence CiC types are λ_H kinds. Figure 5 shows part of the syntax of the language in BNF notation. But this is deceptive: the types shown

$$
\begin{array}{lll}
(\textit{types}) & \sigma ::= t \mid \textit{snat}\ n \mid \sigma \to \sigma \mid \textit{tup}\ n\ \varphi \mid \forall t\!:\!\kappa.\sigma \mid \exists t\!:\!\kappa.\sigma \\
\\
(\textit{functions}) & f ::= \Lambda t\!:\!\kappa.f \mid \lambda x\!:\!\sigma.e \\
(\textit{values}) & v ::= x \mid f \mid c \mid (v,...,v) \mid \langle \varphi, v \rangle \\
(\textit{terms}) & e ::= v(v) \mid \mathsf{halt}\ v \mid \mathsf{let}\ x = \mathsf{sel}[P]\ v.v\ \mathsf{in}\ e \mid \mathsf{let}\ x = v[\varphi]\ \mathsf{in}\ e \\
& \quad \mid \mathsf{let}\ \langle t, x \rangle = \mathsf{open}\ v\ \mathsf{in}\ e \mid \mathsf{let}\ x = \mathsf{cast}[P]\ v\ \mathsf{in}\ e
\end{array}
$$

Fig. 5. Syntax of the λ_H language.

are only those of kind Ω, and in reality these are defined as a CiC inductive type:

$$
\begin{array}{lll}
\mathsf{Inductive}\ \Omega\ :\ \mathsf{Kind} := \textit{snat} & : \textit{Nat} \to \Omega \\
\quad \mid \textit{tup} & : \textit{Nat} \to (\textit{Nat} \to \Omega) \to \Omega \\
\quad \mid \to & : \Omega \to \Omega \to \Omega \\
\quad \mid \forall & : \Pi k\!:\!\mathsf{Kind}.(k \to \Omega) \to \Omega \\
\quad \mid \exists & : \Pi k\!:\!\mathsf{Kind}.(k \to \Omega) \to \Omega
\end{array}
$$

The λ_H language is not dependently typed: it keeps the usual phase distinction between types and terms, and hence type checking is decidable even in the presence of side-effects and non termination. But it gets a reasoning power comparable to dependently typed languages by using singleton types: the type *Nat* 3 has only one member, which is the natural number 3. This amounts to lifting values to the level of types. The power of such a language is illustrated in [League and Monnier 2006] where it is used to encode sophisticated OO programming features. A variant of λ_H is now available as an extension of OCaml [Fogarty et al. 2007].

4. TYPED REGIONS

Our new system of typed regions can be thought of as a hybrid between alias-types [Walker and Morrisett 2000] and the calculus of capabilities [Crary et al. 1999] supplemented with the calculus of inductive constructions (CiC), similarly to λ_H [Shao et al. 2002]. 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.

Just like in λ_H, we use CiC as our type language. So in our language, CiC terms are types and CiC types are kinds. Since in CiC propositions and predicates are encoded as types and proofs as terms, that means that in our typed regions system, we can encode predicates as kinds and proofs as types. While CiC is a dependently typed calculus, typed regions intend to preserve a clear phase distinction between types and values, so it is clear that all proof manipulation is done at compile time only.

$$
\begin{array}{lrl}
(\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 \\
 & & \quad \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 \mapsto \{n_0 \mapsto v_0, ..., n_i \mapsto v_i\} \\
(\textit{heap type}) & \Psi & ::= \bullet \mid \Psi, \nu \mapsto \{n_0 \mapsto \tau_0, ..., n_i \mapsto \tau_i\} \\
 & & \\
(\textit{kind env}) & \Delta & ::= \bullet \mid \Delta, t{:}\kappa \\
(\textit{type env}) & \Gamma & ::= \bullet \mid \Gamma, x{:}\sigma \\
(\textit{region env}) & \Theta & ::= \bullet \mid \Theta \oplus \rho{\mapsto}(\varphi, n) \\
 & & \\
(\textit{regions}) & \rho & ::= r \mid \nu \\
(\textit{types}) & \sigma & ::= t \mid \textit{snat}\ n \mid \textit{tup}\ n\ \varphi \mid \forall t{:}\kappa.\sigma \mid \exists t{:}\kappa.\sigma \mid \tau\ \textit{at}\ \rho.n \mid \{\Theta\}(\vec{\sigma}) \to 0 \\
 & & \\
(\textit{functions}) & f & ::= \Lambda t{:}\kappa.f \mid \lambda\{\Theta\}(\Gamma).e \\
(\textit{values}) & v & ::= x \mid f \mid c \mid (v, ..., v)^{\varphi} \mid \langle \varphi, v \rangle \mid \nu.n \\
(\textit{terms}) & e & ::= v(\vec{v}) \mid \mathsf{halt}\ v \mid \mathsf{let}\ x = \mathsf{sel}[P]\ v.v\ \mathsf{in}\ e \mid \mathsf{let}\ x = v[\varphi]\ \mathsf{in}\ e \\
 & & \quad \mid \mathsf{let}\ \langle t, x \rangle = \mathsf{open}\ v\ \mathsf{in}\ e \mid \mathsf{let}\ x = \mathsf{cast}[P]\ v\ \mathsf{in}\ e \\
 & & \quad \mid \mathsf{let}\ r = \mathsf{newrgn}\ \varphi\ \mathsf{in}\ e \mid \mathsf{freergn}\ \rho; e \\
 & & \quad \mid \mathsf{let}\ x = \mathsf{put}[\rho, \tau]\ v\ \mathsf{in}\ e \mid \mathsf{let}\ x = \mathsf{get}[P]\ v\ \mathsf{in}\ e \\
 & & \quad \mid \mathsf{set}[\varphi]\ v := v; e \mid \mathsf{cast}[P]\ \Theta \sim \Theta'; e \mid \mathsf{cast}[P]\ \rho \mapsto \varphi; e \\
(\textit{prog}) & p & ::= \mathsf{fix}\ x_0 = f_0\ \cdots\ x_n = f_n\ \mathsf{in}\ e
\end{array}
$$

Fig. 6. Syntax of the language.

4.1 The language

The syntax of our language is shown in Fig. 6. The language uses continuation passing style. $M, \Psi, \Gamma, \Delta, \Theta$ are environments used in the typing rules and operational semantics. We also use the following syntactic sugar: we sometimes write $f[\varphi_0, ..., \varphi_n](\vec{v})$ instead of $\mathsf{let}\ f_0 = f[\varphi_0]\ \mathsf{in}\ ...\mathsf{let}\ f_n = f_{n-1}[\varphi_n]\ \mathsf{in}\ f_n(\vec{v})$ and we sometimes directly use $v[\varphi]$ where a value is expected instead of using an explicit $\mathsf{let}\ x = v[\varphi]\ \mathsf{in}\ ...x....$ Values include:

c	the integer constant of singleton type *snat* c;
$(v_0, ..., v_{n-1})^{\varphi}$	a tuple of type *tup* $n\ \varphi$;
$\Lambda t{:}\kappa.v$	a type abstraction of type $\forall t{:}\kappa.\sigma$;
$\langle \varphi, v \rangle$	an existential package of type $\exists t{:}\kappa.\sigma$;
$\nu.n$	a pointer of type τ *at* $\nu.n$ to the n^{th} object of region ν;
$\lambda\{\Theta\}(\Gamma).e$	a function of type $\{\Theta\}(\vec{\sigma}) \to 0$; Γ is the list of arguments and e the body; Θ lists the regions used by the function, along with their type and size;

The terms do the following:

$v(\vec{v})$	make a tail-call to function v;
halt v	halt the machine, returning v as the result;
sel$[P]\ v_1.v_2$	select field v_2 from tuple v_1; P is a proof that this field exists;
$v[\varphi]$	an instantiation of type abstraction v with type φ;
open v	open up an existential package;
cast$[P]\ v$	change the type of v according to P;
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$[P]\ v$	fetch the object pointed to by v; P is a proof that the object exists;
set$[\varphi]\ v_1 := v_2; e$	strong update; assign v_2 into location v_1 ad change its type to φ;
cast$[P]\ \rho \mapsto \varphi; e$	set the type of region ρ to φ; P proves φ is a valid replacement;
cast$[P]\ \Theta \sim \Theta'; e$	reorder regions in Θ; P proves that Θ' is a valid reordering;
fix *defs* **in** e	definition of mutually recursive functions *defs*.

The language is similar to the simple region calculus presented before. The main differences are the following:

—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(see next page).

—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. E.g. we will show examples in Sec. 5 where we use kinds such as $\Omega \times \Omega$ where the intended type is a pair of types, *Unit* where the intended type can only be the constant $()$, and an inductively defined Ω^τ where the intended type can be any expression representing some source-level type.

—The kind of a region now takes the form *R* κ where κ specifies the kind of intended types in this region. Given a region ρ of kind *R* κ and a reference of type τ *at* $\rho.n$ then τ has to have kind κ, and the region's type function φ will have kind *Nat* $\rightarrow \kappa \rightarrow \Omega$. The definition of *R*, *at*, and *RDesc* in Fig. 7 shows how these rules are enforced.

—**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 a strong update that can change the type of the location to φ which has kind $\kappa \rightarrow \Omega$. This type needs to be provided 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.

—We add a **cast** operator to replace the type of a value with another equivalent one: CIC's native notion of equivalence is intentional, so it does not know that $2+x$ is extensionally equal to $x + 2$, even though we can easily write a proof that $2 + x = x + 2$. The **cast** operator allows us to provide such proofs to explain to the type-checker why a piece of

code is correct. We also add 2 cast instructions to manipulate the list of regions: one to simply reorder them, and another to change the type of a particular region with another one that is observationally equal.

—get is annotated with a proof that the location is indeed valid.

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. Our code is assumed to live outside of the heap.

The definitions of ρ, Θ, and σ in Fig. 6 are sloppy. In reality those categories are defined directly in the CiC language as inductive definitions (contrary to the other categories which are either part of the CIC's own definition such as s, φ, and Δ, or are outside of CIC). Programs can create their own such inductive definitions for example to manipulate source-level types, as is done in example 6.1. Inductive definitions are more precise than BNF definitions. In a sense, they include the formation rule along with the syntax. Here are their real definitions in CiC, where with is used for mutually recursive definitions:

$$
\begin{array}{l}
\mathsf{Inductive}\ R\ (k : \mathsf{Kind}) : \mathsf{Kind} := \nu : \mathit{Nat} \to R\ k \\
\mathsf{Inductive}\ \Omega\ :\ \mathsf{Kind} := \mathit{snat}\ : \mathit{Nat} \to \Omega \\
\qquad\qquad\qquad\qquad\quad \mid \mathit{tup}\ \ : \mathit{Nat} \to (\mathit{Nat} \to \Omega) \to \Omega \\
\qquad\qquad\qquad\qquad\quad \mid \mathit{at}\ \ \ : \Pi k : \mathsf{Kind}.k \to R\ k \to \mathit{Nat} \to \Omega \\
\qquad\qquad\qquad\qquad\quad \mid \forall\ \ \ : \Pi k : \mathsf{Kind}.(k \to \Omega) \to \Omega \\
\qquad\qquad\qquad\qquad\quad \mid \exists\ \ \ : \Pi k : \mathsf{Kind}.(k \to \Omega) \to \Omega \\
\qquad\qquad\qquad\qquad\quad \mid \to\ \ \ : \mathit{REnv} \to \mathit{List}\ \Omega \to \Omega \\
\mathsf{with}\ \mathit{RDesc} : \mathsf{Kind} := \mathit{RD}\ :\ \Pi k : \mathsf{Kind}.R\ k \to ((\mathit{Nat} \to k \to \Omega) \wedge \mathit{Nat}) \to \mathit{RDesc} \\
\mathsf{with}\ \mathit{REnv} : \mathsf{Kind} := \mathit{List}\ \mathit{RDesc} \\
\mathsf{Notation}\ \text{“}r \mapsto (\varphi, n)\text{”} := (\mathit{RD}\ r\ (\varphi, n))
\end{array}
$$

Fig. 7. Formal definition of Ω, *R* and *REnv*.

Notice how the case for *at* enforces that the kind of τ in τ *at* $\rho.n$ is compatible with the kind of region ρ. Notice also that we do not write the k argument in $(\mathit{RD}\ r\ \varphi\ n)$ because Coq can trivially infer it from the other arguments. Instead of a normal integer type, we use a singleton type for our natural numbers. We can then define our integer type as:

$$\mathsf{Definition\ int} := \exists\ (\lambda n : \mathit{Nat}.\mathit{snat}\ n)$$

4.2 Semantics

The machine state is defined as the pair $(M; e)$ of a memory state and an expression. Figure 8 shows the operational small step semantics. The rules are straightforward, except maybe for the side condition on the rule for put which basically just forces sequential allocation in the region. A real implementation would probably keep track explicitly in M of the size of each region.

Although M is a finite two-dimensional map that is first indexed by regions and then by locations, we liberally abuse the notation $M(\nu.n)$ which really means $M(\nu)(n)$ and presumes that $\nu \in \mathit{Dom}(M)$ and $n \in \mathit{Dom}(M(\nu))$. The same is true for $M, \nu.n \mapsto v$

$$
\begin{array}{ll}
(M; (\lambda\{\Theta\}(\overrightarrow{x:\sigma}).e)(\vec{v})) & \Longrightarrow (M; e[{}^{\vec{v}}/_{\vec{x}}]) \\
(M; \mathsf{let}\ x = \mathsf{sel}[P]\ (v_1, ..., v_n)^{\varphi}.ie\ \mathsf{in}\ ) & \Longrightarrow (M; e[{}^{v_i}/_{x}]) \\
(M; \mathsf{let}\ x = (\Lambda t{:}\kappa.v)[\varphi]\ \mathsf{in}\ e) & \Longrightarrow (M; e[{}^{v[{}^{\varphi}/_{t}]}/_{x}]) \\
(M; \mathsf{let}\ \langle t, x\rangle = \mathsf{open}\ \langle \varphi, v\rangle\ \mathsf{in}\ e) & \Longrightarrow (M; e[{}^{\varphi,\ v}/_{t,\ x}]) \\
(M; \mathsf{let}\ x = \mathsf{cast}[P]\ v\ \mathsf{in}\ e) & \Longrightarrow (M; e[{}^{v}/_{x}]) \\
(M; \mathsf{let}\ r = \mathsf{newrgn}\ \varphi\ \mathsf{in}\ e) & \Longrightarrow (M, \nu \mapsto \{\}; e[{}^{\nu}/_{r}]) \text{ where } \nu \text{ is fresh} \\
(M, \nu \mapsto \{...\}; \mathsf{freergn}\ \nu; e) & \Longrightarrow (M; e) \\
(M; \mathsf{let}\ x = \mathsf{put}[\nu, \tau]\ v\ \mathsf{in}\ e) & \Longrightarrow \begin{array}{l}(M, \nu.n \mapsto v; e[{}^{\nu.n}/_{x}]) \\ \text{where } \nu.n \notin \mathit{Dom}(M) \wedge \nu.n{-}1 \in \mathit{Dom}(M)\end{array} \\
(M; \mathsf{let}\ x = \mathsf{get}[P]\ \nu.n\ \mathsf{in}\ e) & \Longrightarrow (M; e[{}^{M(\nu.n)}/_{x}]) \\
(M; \mathsf{set}[\varphi]\ \nu.n := v; e) & \Longrightarrow (M, \nu.n \mapsto v; e) \\
(M; \mathsf{cast}[P]\ \_; e) & \Longrightarrow (M; e)
\end{array}
$$

Fig. 8. Operational semantics of the language.

$\vdash \Psi$	memory type Ψ is well-formed
$\vdash \Theta$	region type env Θ is well-formed
$\Psi; \Theta \vdash M$	memory M is well formed in Ψ and Θ
$\vdash (M; e)$	machine state is well-formed

$$
\frac{}{\vdash \bullet} \qquad \frac{\vdash \Psi \qquad \bullet \vdash^{\text{CIC}} \nu : \boldsymbol{R}\,\kappa \qquad \forall\, i.\ \bullet \vdash^{\text{CIC}} \varphi_i : \kappa}{\vdash \Psi, \nu \mapsto \{0 \mapsto \varphi_0, \ldots, n \mapsto \varphi_n\}} \qquad \frac{\vdash \Theta \qquad \nu \notin \mathit{Dom}(\Theta)}{\vdash \Theta \oplus \nu \mapsto (\varphi, n)}
$$

$$
\frac{\Psi; \Theta \vdash M \qquad \forall i.\ \text{if } i > n \text{ then } \nu.i \notin \mathit{Dom}(\Psi) \text{ else } \Psi; \bullet; \bullet \vdash v_i : \varphi\ i\ (\Psi(\nu.i))}{\Psi; \Theta \oplus \nu \mapsto (\varphi, n{+}1) \vdash M, \nu \mapsto \{0 \mapsto v_0, \ldots, n \mapsto v_n\}}
$$

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

Fig. 9. Environment formation rules.

which presumes that $\nu \in \mathit{Dom}(M)$ and adds a binding $n \mapsto v$ to $M(\nu)$. Similar abuse is used in the formation rules below on Ψ.

The formation rules for environments are given in Fig. 9 together with the definition of a well-formed machine state $\vdash (M; e)$. These describe the global invariants on which the system relies for soundness and explains how the various maps are linked together, so it is the key to understanding why the type system is sound. The judgment $\Delta \vdash^{\text{CIC}} \varphi : \kappa$ used in those rules, taken directly from CiC and not shown here, states that φ has kind κ in environment Δ. The judgment $\vdash \Psi$ checks that each intended type has a kind consistent with its region. The judgment $\vdash \Theta$ checks that each region has only one binding in Θ

$\Psi;\Delta;\Gamma \vdash v : \sigma$	value v has type σ
$\Psi;\Delta;\Gamma \vdash \mathit{op} : \sigma$	pure operation *op* returns a value of type σ

$$\frac{\Gamma(x) = \sigma}{\Psi;\Delta;\Gamma \vdash x : \sigma} \qquad \frac{}{\Psi;\Delta;\Gamma \vdash c : \mathit{snat}\ c} \qquad \frac{\Psi;\Delta;\Gamma \vdash v : \sigma_1 \qquad \Delta \models^{\text{CIC}} P : \sigma_1 = \sigma_2}{\Psi;\Delta;\Gamma \vdash \mathsf{cast}[P]\ v : \sigma_2}$$

$$\frac{\forall i \qquad \Psi;\Delta;\Gamma \vdash v_i : \varphi\ i}{\Psi;\Delta;\Gamma \vdash (v_0, ..., v_{n-1})^{\varphi} : \mathit{tup}\ n\ \varphi}$$

$$\frac{\Psi;\Delta;\Gamma \vdash v_1 : \mathit{tup}\ n\ \varphi \qquad \Psi;\Delta;\Gamma \vdash v_2 : \mathit{snat}\ m \qquad \Delta \models^{\text{CIC}} P : m < n}{\Psi;\Delta;\Gamma \vdash \mathsf{sel}[P]\ v_1.v_2 : \varphi\ i}$$

$$\frac{\nu \notin \mathit{Dom}(\Psi) \bigvee \tau = \Psi(\nu.n)}{\Psi;\Delta;\Gamma \vdash \nu.n : \tau\ \mathit{at}\ \nu.n} \qquad \frac{\forall\, i \qquad \Delta \models^{\text{CIC}} \sigma_i : \Omega \qquad \Psi;\Delta;\Theta;\Gamma, \overrightarrow{x:\sigma} \vdash e}{\Psi;\Delta;\Gamma \vdash \lambda\{\Theta\}(\overrightarrow{x:\sigma}).e : \{\Theta\}(\vec{\sigma}) \to 0}$$

$$\frac{\Psi;\Delta, t{:}\kappa;\Gamma \vdash v : \sigma}{\Psi;\Delta;\Gamma \vdash \Lambda t{:}\kappa.v : \forall t{:}\kappa.\sigma} \qquad \frac{\Delta \models^{\text{CIC}} \varphi : \kappa \qquad \Psi;\Delta;\Gamma \vdash v : \sigma[\varphi/t]}{\Psi;\Delta;\Gamma \vdash \langle \varphi, v \rangle : \exists t{:}\kappa.\sigma}$$

$$\frac{\Psi;\Delta;\Gamma \vdash v : \forall t{:}\kappa.\sigma \qquad \Delta \models^{\text{CIC}} \varphi : \kappa}{\Psi;\Delta;\Gamma \vdash v[\varphi] : \sigma[\varphi/t]}$$

Fig. 10. Static semantics of values and pure operations.

(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 we do not refer to the same physical region somewhere else under some other name. Note that the construction rule for *RDesc* already ensures that the type of each region is consistent with its kind. 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 and none of its existing bindings can be modified.

Figure 11 shows the formation rules for terms. We do not show any formation rules for types because well-formedness of types is automatically enforced by CiC's own typing rules. 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. 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 particular difficulty.

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

$$\frac{\Psi;\Delta;\Gamma \vdash v : \{\Theta\}(\vec{\sigma}) \to 0 \qquad \Psi;\Delta;\Gamma \vdash v_i : \sigma_i}{\Psi;\Delta;\Theta;\Gamma \vdash v(\vec{v})} \qquad \frac{\Psi;\Delta;\bullet;\Gamma \vdash v : \mathsf{int}}{\Psi;\Delta;\bullet;\Gamma \vdash \mathsf{halt}\ v}$$

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

$$\frac{\Psi;\Delta;\Gamma \vdash v : \exists t{:}\kappa.\sigma \qquad \Psi;\Delta, t{:}\kappa;\Theta;\Gamma, x{:}\sigma \vdash e}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{let}\ \langle t, x\rangle = \mathsf{open}\ v\ \mathsf{in}\ e}$$

$$\frac{\begin{array}{c}\Delta \models^{\text{CIC}} \varphi : \mathit{Nat} \to \kappa \to \Omega \\ \Psi;\Delta, r{:}\mathit{R}\ \kappa;\Theta \oplus r \mapsto (\varphi, 0);\Gamma \vdash e\end{array}}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{let}\ r = \mathsf{newrgn}\ \varphi\ \mathsf{in}\ e} \qquad \frac{\Psi;\Delta;\Theta;\Gamma \vdash e}{\Psi;\Delta;\Theta \oplus \rho \mapsto (\varphi, n);\Gamma \vdash \mathsf{freergn}\ \rho; e}$$

$$\frac{\begin{array}{c}\Delta \models^{\text{CIC}} \rho : \mathit{R}\ \kappa \qquad \Delta \models^{\text{CIC}} \tau : \kappa \qquad \Psi;\Delta;\Gamma \vdash v : \varphi\ n\ \tau \\ \Psi;\Delta;\Theta \oplus \rho \mapsto (\varphi, n{+}1);\Gamma, x{:}\tau\ \mathit{at}\ \rho.n \vdash e\end{array}}{\Psi;\Delta;\Theta \oplus \rho \mapsto (\varphi, n);\Gamma \vdash \mathsf{let}\ x = \mathsf{put}[\rho, \tau]\ v\ \mathsf{in}\ e}$$

$$\frac{\Psi;\Delta;\Theta;\Gamma, x{:}\varphi\ n\ \tau \vdash e \qquad \Psi;\Delta;\Gamma \vdash v : \tau\ \mathit{at}\ \rho.n \qquad \Delta \models^{\text{CIC}} P : \rho \mapsto (\varphi, m) \in \Theta}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{let}\ x = \mathsf{get}[P]\ v\ \mathsf{in}\ e}$$

$$\frac{\begin{array}{ll}\Psi;\Delta;\Theta \oplus \rho \mapsto (\varphi, n);\Gamma \vdash v : \tau\ \mathit{at}\ \rho.m & \Psi;\Delta;\Theta \oplus \rho \mapsto (\mathit{upd}\ \varphi\ m\ \varphi', n);\Gamma \vdash e \\ \Psi;\Delta;\Theta \oplus \rho \mapsto (\varphi, n);\Gamma \vdash v' : \varphi'\ \tau & \Delta \models^{\text{CIC}} \rho : \mathit{R}\ \kappa \qquad \Delta \models^{\text{CIC}} \varphi' : \kappa \to \Omega\end{array}}{\Psi;\Delta;\Theta \oplus \rho \mapsto (\varphi, n);\Gamma \vdash \mathsf{set}[\varphi']\ v := v'; e}$$

$$\frac{\begin{array}{c}\Delta \models^{\text{CIC}} P : \Pi i{:}\mathit{Nat}.\,\Pi t{:}\kappa.\, i < n \to \varphi\ i\ t = \varphi'\ i\ t \\ \Psi;\Delta;\Theta \oplus \rho \mapsto (\varphi', n);\Gamma \vdash e \qquad \Delta \models^{\text{CIC}} \rho : \mathit{R}\ \kappa\end{array}}{\Psi;\Delta;\Theta \oplus \rho \mapsto (\varphi, n);\Gamma \vdash \mathsf{cast}[P]\ \rho \mapsto \varphi'; e} \qquad \frac{\Psi;\Delta;\Theta';\Gamma \vdash e \qquad \Delta \models^{\text{CIC}} P : \Theta \sim \Theta'}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{cast}[P]\ \Theta \sim \Theta'; e}$$

Fig. 11. Static semantics of the terms.

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 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 the strong set 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.

The rule for get checks that P is a proof of $\rho \mapsto (\varphi, m) \in \Theta$ which means that ρ is

live and has type φ. When ρ appears directly in Θ, the type checker can easily provide P for us, in which case we could skip the annotation altogether, but the use of P makes it possible to handle other cases as well. It can be used for example when part of the heap is abstracted in a variable ϵ (as in the example in Sec. 3.3) and ρ is known to exist somewhere in ϵ. Section 5.7 will show how to use it to handle cases where a region ρ can refer to either ρ_1 or ρ_2 and it is not known statically which.

4.3 Properties of the language

We state here a few important properties of the language. The full proofs for an earlier version of this language can be found in [Monnier 2003]. Since our type language is CiC, we know it is strongly normalizing and confluent.

Lemma 4.1 (Type Preservation)
If $\vdash (M; e)$ *and* $(M; e) \Longrightarrow (M'; e')$, *then* $\vdash (M'; e')$.

Lemma 4.2 (Progress)
If $\vdash (M; e)$, *either* $e = \mathsf{halt}\, v$ *or* $(M; e) \Longrightarrow (M'; e')$.

Lemma 4.3 (Complete Collection)
If $\vdash (M; e)$ *and* $(M; e) \Longrightarrow^* (M'; \mathsf{halt}\, v)$ *then* $M' = \bullet$.

PROOF. Type preservation implies that $\vdash (M'; \mathsf{halt}\, v)$, from which we immediately get that $\Theta' = \bullet$ and thus $M' = \bullet$. □

For soundness reasons, it is necessary that every region in Θ be distinct; yet, this is not enforced anywhere in the typing rule of function definitions. Instead it is enforced by construction: the condition is verified in the initial state (by the premise $\vdash \Theta$ in the definition of $\vdash (M; e)$) and is then preserved by the typing rule of each operation. So while you can write and type check functions of the form $\Lambda r.\lambda\{r, r\}(x).e$, you cannot call them, except of course from another similarly ill-defined function.

5. 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. In the previous section, we have kept our language simple and we will need to use some minor extensions in some of those examples.

5.1 Programming with λ_H

Before getting into details of how to use the region types, we present here how the pure part of the language (comparable to λ_H) is used. First, we can define a boolean *kind*. Rather than just use *true* and *false* constants, we use a disjunction so that each one of the two alternatives carries with it a proof of the property represented by the boolean's value:

$$\mathsf{type}\ \mathit{Bool}\ (k : \mathsf{Kind}) = k \vee \neg k$$

Then we can define the corresponding singleton boolean type represented by an underlying integer of value 0 or 1:

$$\mathsf{type}\ \mathit{sbool}\ (k : \mathsf{Kind}, b : \mathit{Bool}\ k) = \mathit{snat}\ (\mathsf{match}\ b\ \mathsf{with}\ \mathit{left}\ \_ \Rightarrow 1 \mid \mathit{right}\ \_ \Rightarrow 0)$$

The corresponding non-singleton boolean type can be defined as follows:

$$\mathsf{type}\ \mathit{bool}\ (k{:}\mathsf{Kind}) = \exists b{:}\mathit{Bool}\ k.\mathit{sbool}\ b$$

In λ_H we can then try to encode a conditional expression via a jump table. Here is a first attempt to write an *if* that does refinement in the sense that the type of each branch is slightly different, reflecting the fact that when it is taken, we learn something about the value of b:

$$\begin{array}{l}\mathsf{fun}\ \mathit{if}[k{:}\mathsf{Kind}, t:\Omega](b{:}\mathit{bool}\ k, x_1{:}\forall\_{:}k.t, x_2{:}\forall\_{:}\neg k.t) : t = \\ \quad \mathsf{let}\ \langle bt, sb\rangle = \mathsf{open}\ b\ \mathsf{in} \\ \quad \mathsf{sel}[P]\ (x_2, x_1)^{\varphi}.sb\end{array}$$

But this code is idealized and we need to flesh it out to make it correct. At the very least we need to fill in φ and P. The type of (x_2, x_1) can be:

$$(x_2, x_1) : \mathit{tup}\ 2\ (\lambda i.\forall\_{:}(\mathsf{match}\ i\ \mathsf{with}\ 0 \Rightarrow \neg k \mid \_ \Rightarrow k).t)$$

So φ can be $(\lambda i.\forall\_{:}(\mathsf{match}\ i\ \mathsf{with}\ 0 \Rightarrow \neg k \mid \_ \Rightarrow k).t)$. As for P:

$$\begin{array}{rl}P \equiv & \mathsf{match}\ bt\ \mathsf{with}\ \mathit{left}\ \_ \Rightarrow \mathit{le\_n} \mid \mathit{right}\ \_ \Rightarrow \mathit{le\_s}\ \mathit{le\_n} \\ : & (\mathsf{match}\ bt\ \mathsf{with}\ \mathit{left}\ \_ \Rightarrow 1 \mid \mathit{right}\ \_ \Rightarrow 0) < 2\end{array}$$

Now even with these annotations, the above definition of *if* fails to give us the expected return type: the return type of the function is the type of the value extracted by sel, i.e. it's φ applied to $(\mathsf{match}\ bt\ \mathsf{with}\ \mathit{left}\ \_ \Rightarrow 1 \mid \mathit{right}\ \_ \Rightarrow 0)$:

$$\forall\_{:}(\mathsf{match}\ (\mathsf{match}\ bt\ \mathsf{with}\ \mathit{left}\ \_ \Rightarrow 1 \mid \mathit{right}\ \_ \Rightarrow 0)\ \mathsf{with}\ 0 \Rightarrow \neg k \mid \_ \Rightarrow k).t$$

whereas we wanted just t. So we need to add a type application with argument:

$$\begin{array}{l}\mathsf{match}\ bt\ \mathsf{with}\ \mathit{left}\ p \Rightarrow p \mid \mathit{right}\ p \Rightarrow p \\ : \mathsf{match}\ (\mathsf{match}\ bt\ \mathsf{with}\ \mathit{left}\ \_ \Rightarrow 1 \mid \mathit{right}\ \_ \Rightarrow 0)\ \mathsf{with}\ 0 \Rightarrow \neg k \mid \_ \Rightarrow k\end{array}$$

Additionally, the typed region calculus uses continuation passing and requires region annotations on functions, so the full code is:

$$\begin{array}{l}\mathsf{fun}\ \mathit{if}[k{:}\mathsf{Kind}, h{:}\mathit{REnv}]\{h\} \\ \qquad (b{:}\mathit{bool}\ k,\ \ x_1{:}\forall\_{:}k.\{h\}() \to 0,\ \ x_2{:}\forall\_{:}\neg k.\{h\}() \to 0) = \\ \quad \mathsf{let}\ \langle bt, sb\rangle = \mathsf{open}\ b\ \mathsf{in} \\ \quad \mathsf{let\ type}\ P = \mathsf{match}\ bt\ \mathsf{with}\ \mathit{left}\ \_ \Rightarrow \mathit{le\_n} \mid \mathit{right}\ \_ \Rightarrow \mathit{le\_s}\ \mathit{le\_n}\ \mathsf{in} \\ \quad \mathsf{let\ type}\ \varphi = (\lambda i.\forall\_{:}(\mathsf{match}\ i\ \mathsf{with}\ 0 \Rightarrow \neg k \mid \_ \Rightarrow k).t)\ \mathsf{in} \\ \quad \mathsf{let}\ x = \mathsf{sel}[P]\ (x_2, x_1)^{\varphi}.sb\ \mathsf{in} \\ \quad x[\mathsf{match}\ bt\ \mathsf{with}\ \mathit{left}\ p \Rightarrow p \mid \mathit{right}\ p \Rightarrow p]()\end{array}$$

Notice how we use the type parameter h to abstract over an arbitrary set of regions and how the type of *if* expresses the fact that it neither creates nor frees any region and neither does it allocate any new object, since any one of those operations would have required the heap h to be different in the argument to *if* than in the argument to the possible continuations x_1 and x_2.

5.2 Unsound region aliasing

Our system shares with alias types the dubious quality of providing the sometimes confusing ability to *write* apparently unsafe code. In the case of alias types, the typical example is

that the language does not prevent you from creating dangling pointers, widely perceived as dangerous: the trick there is that the safety of the alias types system relies on checking the pointer indirections themselves, making dangling pointers perfectly safe (tho mostly useless).

So to clear things up, we will start by presenting a similarly dangerous but legal piece of code, which should help the reader get a better understanding of how the soundness of the language is guaranteed. The example deals with the problem of region aliasing, where one of the aliases is used to free the region while the other aliases may then be used to access the region after it has been freed:

$$
\begin{array}{l}
\mathsf{let}\ r_1 = \mathsf{newrgn}\ \varphi\ \mathsf{in} \\
\mathsf{let}\ x = \mathsf{put}[r_1, \mathsf{int}]\ 1\ \mathsf{in} \\
\mathsf{let}\ f = \Lambda r_2 : R\ \kappa.\lambda\{r_1 \mapsto (\varphi, 1), r_2 \mapsto (\varphi, 1)\}().\mathsf{freergn}\ r_1; \mathsf{freergn}\ r_2; \mathsf{halt}\ x\ \mathsf{in} \\
\ldots
\end{array}
$$

In this example, we first create a region whose handle is stored in the type variable r_1. Then we define a function which expects another region handle r_2 and then frees both r_1 and r_2. Provided that we correctly choose φ, and κ, the above code is perfectly valid. This may give the impression that we can free r_1 twice if we replace ... with $f[r_1]()$, thus making r_2 an alias for r_1; but such a call would be rejected by our type system:

—The environment Θ starts empty, and after the newrgn it contains just $\{r_1 \mapsto (\varphi, 0)\}$.

—After the put, Θ becomes $\{r_1 \mapsto (\varphi, 1)\}$.

—After the function definition, Θ stays unchanged since such pure variable bindings have no effect on the store.

—So when we get to $f[r_1]()$ the environment Θ is still $\{r_1 \mapsto (\varphi, 1)\}$, but the typing rules require it to be equal to the environment specified for the function, i.e. equal to $\{r_1 \mapsto (\varphi, 1), r_2 \mapsto (\varphi, 1)\}$ where r_1 is substituted for r_2 (i.e. $\{r_1 \mapsto (\varphi, 1), r_1 \mapsto (\varphi, 1)\}$). The constraint is clearly not satisfied.

Notice also that even after freeing r_1 and r_2, both r_1 and r_2 are still valid types which happen to refer to regions which are not live anymore (i.e. they are dangling region handles); similarly, the pointer x that pointed to the first element of the region r_1 and is thus now dangling as well, is still a valid value. All those dangling elements are perfectly harmless, because the typing rule of every operation that accesses a region first looks up Θ to check the liveness of that region.

5.3 Weak update

The set operation as defined is very restrictive: it can only apply to a reference to an object in the region that is at the top of Θ and it always changes the type of that region. Some readers might have expected to also find a *weak update* operation in our language, with a typing rule such as:

$$
\frac{\begin{array}{c}\Delta \models^{\text{CIC}} P : \rho \mapsto (\varphi, m) \in \Theta \\ \Psi; \Delta; \Gamma \vdash v : \tau\ \mathit{at}\ \rho.n \qquad \Psi; \Delta; \Gamma \vdash v' : \varphi\ n\ \tau \qquad \Psi; \Delta; \Theta; \Gamma \vdash e\end{array}}{\Psi; \Delta; \Theta; \Gamma \vdash \mathsf{set}[P]\ v := v'; e}
$$

where P is just a proof that the relevant region is alive and has type φ. But it turns out that this is unnecessary since we can already define the weak update in the language by

```
(* Definition of the existential quantifier. *)
Inductive Ex (k:Kind) (P:k → Kind) : Kind := ex_intro : Πx.P x → Ex P.
Notation "∃x, P" := Ex (λx.P).

(* Proof that x ∈ l implies ∃l'.Pick x l' l. *)
Fixpoint ItoP k x (l:List k) (P: x ∈ l) {struct P} : ∃l', Pick x l' l :=
  match P in _ ∈ l return Ex (λl'.Pick x l' l) with
  | INbase l ⇒ ex_intro (λl'.Pick x l' (x :: l)) l (Pbase x l)
  | INskip l y P' ⇒
    match ItoP P' with
    | ex_intro l' PP' ⇒ ex_intro (λl'.Pick x l' (y :: l)) (y :: l') (Pskip y PP').

(* Proof that Pick x l' l implies l ∼ x :: l'. *)
Definition PickULeq k x (l l':List k) (P:Pick x l' l) : l ∼ x :: l' :=
  uleq_cons P (Pbase x l') (uleq_refl l').

(* Proof that ∼ (also known as ULeq) is commutative. *)
Fixpoint uleq_comm k (l1 l2:List k) (P : l1 ∼ l2) {struct P} : l2 ∼ l1 :=
  match P in l1 ∼ l2 return l2 ∼ l1 with
  | uleq_refl l ⇒ uleq_refl l
  | uleq_cons l1 l2 x l1' l2' P1 P2 P' ⇒ uleq_cons P2 P1 (uleq_comm P').
```

Fig. 12. Auxiliary proofs

combining the strong update with a healthy dose of casts:

$$\begin{array}{lcl} \mathsf{set}[P]\ v := v'; e & \simeq & \begin{array}{l} \mathsf{cast}[P_1]\ \Theta \sim \Theta' \oplus \rho \mapsto (\varphi, m); \\ \mathsf{set}[\varphi\ n]\ v := v'; \\ \mathsf{cast}[P_2]\ \rho \mapsto \varphi; \\ \mathsf{cast}[P_3]\ \Theta' \oplus \rho \mapsto (\varphi, m) \sim \Theta; \\ e \end{array} \end{array}$$

The first cast brings the relevant region ρ to the front, the second resets the region's type that was just changed by set, and the last brings ρ back to its original position. The additional elements ρ, φ, m, Θ, and n can be extracted from the type of P and v while the remaining P_1, P_2, P_3, and Θ' are boilerplate CiC expressions parameterized by P, ρ, φ, n, and m. For example, from the proof P that ρ is in Θ, we can build P_1 and P_3 using some auxiliary proofs shown in Fig. 12. The proof P_2 is a bit longer because it needs to reason about equality and its negation over *Nat*, so it is not shown here:

$$\begin{array}{l} P_1 : \Theta \sim \Theta' \oplus \rho \mapsto (\varphi, m) = \mathit{PickULeq}\ (\mathit{ex2}\ (\mathit{ItoP}\ P)) \\ P_3 : \Theta' \oplus \rho \mapsto (\varphi, m) \sim \Theta = \mathit{uleq\_comm}\ P_1 \end{array}$$

5.4 Simple regions

Typed regions are a superset of traditional regions. Traditional regions are just regions where the actual type is always identical to the intended type, i.e. the regions's type is a kind of identity function. If we define *idr* to be the function $\lambda n.\lambda t.t$, then we can encode simple region kinds and types along the following lines, where $\ulcorner \cdot \urcorner$ stands for the translation

of simple region constructs:

$$\begin{array}{lcl}
\ulcorner R \urcorner & = & R\ \Omega \\
\ulcorner \sigma\ \mathit{at}\ \rho \urcorner & = & \exists n.\ulcorner\sigma\urcorner\ \mathit{at}\ \rho.n \\
\ulcorner \Theta \oplus \rho \urcorner & = & \ulcorner\Theta\urcorner \oplus \rho \mapsto (\mathit{idr}, n_\rho) \\
\ulcorner \{\Theta\}(\vec{\sigma}) \to 0 \urcorner & = & \forall \overrightarrow{n_\rho : \mathit{Nat}}.\{\ulcorner\Theta\urcorner\}(\ulcorner\vec{\sigma}\urcorner) \to 0
\end{array}$$

This only shows the types that are affected. The two main changes are first that references are wrapped in existential packages to hide the precise location to which they point, and second that in order to keep track of the region size information, we have to add one type argument per region to every function.

Correspondingly, at the level of terms, every operation using references needs to (un)pack the existential package, and every function and function call has to handle the additional type arguments:

$$\begin{array}{lcl}
\ulcorner \mathsf{let}\ r = \mathsf{newrgn}\ \mathsf{in}\ e \urcorner & = & \mathsf{let}\ r = \mathsf{newrgn}\ \mathit{idr}\ \mathsf{in}\ \ulcorner e \urcorner \\
\ulcorner \mathsf{freergn}\ \rho; e \urcorner & = & \mathsf{freergn}\ \rho; \ulcorner e \urcorner \\
\ulcorner \mathsf{let}\ x = \mathsf{put}[\rho]\ v\ \mathsf{in}\ e \urcorner & = & \mathsf{let}\ x' = \mathsf{put}[\rho, \sigma]\ \ulcorner v \urcorner\ \mathsf{in}\ \mathsf{let}\ x = \langle n_\rho, x' \rangle\ \mathsf{in}\ \ulcorner e \urcorner \\
\ulcorner \mathsf{let}\ x = \mathsf{get}\ v\ \mathsf{in}\ e \urcorner & = & \mathsf{let}\ \langle n, x' \rangle = \mathsf{open}\ \ulcorner v \urcorner\ \mathsf{in}\ \mathsf{let}\ x = \mathsf{get}[P]\ x'\ \mathsf{in}\ \ulcorner e \urcorner \\
\ulcorner \mathsf{set}\ v := v'; e \urcorner & = & \mathsf{let}\ \langle n, x \rangle = \mathsf{open}\ \ulcorner v \urcorner\ \mathsf{in}\ \mathsf{set}[P]\ x := \ulcorner v' \urcorner; \ulcorner e \urcorner \\
\ulcorner \lambda\{\Theta\}(\Gamma).e \urcorner & = & \Lambda \overrightarrow{n_\rho : \mathit{Nat}}.\lambda\{\ulcorner\Theta\urcorner\}(\ulcorner\Gamma\urcorner).\ulcorner e \urcorner \\
\ulcorner v(\vec{v}) \urcorner & = & \ulcorner v \urcorner[\vec{n_\rho}](\ulcorner\vec{v}\urcorner)
\end{array}$$

Note that the set we use is really the weak update described in example 5.3. The proofs P we need for both get and set are the same: they just give the position of ρ in Θ so as to show it is still alive. Given the typing derivation in a hypothetical simple regions source language, those proofs are trivial to build. Similarly, keeping track of the allocation limit n is easy since each function is actually a basic block with a fixed number of allocations in it.

The example from Sec. 3.2:

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

becomes:

$$\begin{array}{l}
\mathbf{mktree}[n, r : R\ \Omega, t : \Omega] \\
\qquad \{r \mapsto (\mathit{idr}, n)\} \\
\qquad (x : t, k : \forall[n]\{r \mapsto (\mathit{idr}, n)\}(\exists m : \mathit{Nat}.(\mathit{tree}\ t)\ \mathit{at}\ r.m) \to 0) \\
\quad = \mathsf{let}\ y' = \mathsf{put}[r, \mathit{tree}\ t]\ (\mathit{Node}\ x)\ \mathsf{in} \\
\qquad \mathsf{let}\ y = \langle n, y' \rangle\ \mathsf{in} \\
\qquad \mathsf{let}\ \langle n, y' \rangle = \mathsf{open}\ y\ \mathsf{in} \\
\qquad \mathsf{set}[P]\ y' := \mathit{Branch}\ y\ y; k[n+1](y)
\end{array}$$

Where the open can be trivially eliminated.

5.5 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 its type can only be of the form $\exists t.t$ *at* ρ which says that nothing is known about the object at that location.

In our system of typed regions, the reference created out of a reference to an adjacent object may have a type apparently just as useless: $\exists t.t$ *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 $\varphi\ n\ t$. So, as long as the programmer ensures that the region type φ gives enough information, she can carry on the scan. Typically φ will force all objects to describe themselves via some standard header. Section 6.5 shows an example where the region's type φ enforces that all objects in the region start with a tag indicating what the rest of the object looks like.

Concretely, to scan regions, we need to add a pointer arithmetic primitive offset to our language. Also in order to be able to use a sentinel when scanning a region, or in order to detect references crossing regions, we will want to be able to compare references and regions. So we will add the following primitives:

$$\begin{array}{l}\mathsf{let}\ x = \mathsf{offset}[P]\ v_1\ v_2\ \mathsf{in}\ e\\ \mathsf{let}\ x = \mathsf{rcmp}[P]\ v\ \rho\ \mathsf{in}\ e\\ \mathsf{let}\ x = v_1 < v_2\ \mathsf{in}\ e\end{array}$$

where $v_1 < v_2$ does the obvious pointer comparison, rcmp checks if a reference points into a particular region, and offset does pointer arithmetic, where v_1 is a pointer and v_2 is an offset to add to it, so if v_1 has value $\nu.n$ and v_2 has value m, then x will be bound to $\nu.(n+m)$ and t will get bound to the *intended* type of that location. The typing rules are:

$$\frac{\begin{array}{ll}\Psi;\Delta;\Gamma \vdash v_1 : \tau\ \textit{at}\ \rho.m_1 & \Delta \models^{\text{CIC}} P : 0 \le m_1{+}m_2 < n \wedge \rho \mapsto (\varphi, n) \in \Theta\\ \Psi;\Delta;\Gamma \vdash v_2 : \textit{snat}\ m_2 & \Psi;\Delta;\Theta;\Gamma, x{:}\exists t.t\ \textit{at}\ \rho.(m_1{+}m_2) \vdash e\end{array}}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{let}\ x = \mathsf{offset}[P]\ v_1\ v_2\ \mathsf{in}\ e}$$

$$\frac{\begin{array}{ll}\Psi;\Delta;\Gamma \vdash v : \tau\ \textit{at}\ \rho_1.n & \Delta \models^{\text{CIC}} P : \rho_2 \mapsto \_ \in \Theta\\ \Psi;\Delta;\Theta;\Gamma, x{:}\textit{bool}\,((\rho_1 \mapsto \_ \in \Theta) \to \rho_1 = \rho_2) \vdash e & \end{array}}{\Psi;\Delta;\Theta;\Gamma \vdash \mathsf{let}\ x = \mathsf{rcmp}[P]\ v\ \rho_2\ \mathsf{in}\ e}$$

$$\frac{\Psi;\Delta;\Gamma \vdash v_1 : \tau_1\ \textit{at}\ \rho.n \qquad \Psi;\Delta;\Gamma \vdash v_2 : \tau_2\ \textit{at}\ \rho.m}{\Psi;\Delta;\Gamma \vdash v_1 < v_2 : \textit{bool}\,(n < m)}$$

P in offset is a proof that the region is alive and that the resulting location is within its bounds. We will use this operator in some of the following example code, but only to jump forward over a single object (with v_2 being the constant 1): for less idealized realization of our system of typed regions, elements of a region may actually have different sizes, so jumping backward, or jumping in a single step over more than one object may not always be possible. Also the size of the object over which to jump should then probably be provided as an additional argument together with a proof that this argument indeed contains the size of the object.

Of course, often in order to know when to stop scanning, we will want to use a runtime check against the region's upper bound. This check will also incidentally give us the proof we need to pass to offset.

5.6 Stacks

We can use a typed region to represent a contiguous stack of objects. As the stack grows, shrinks, and grows again, we clearly need to reuse locations for objects of different types, and since intended types are immutable, we cannot make much use of intended types. More specifically, a stack is a region S of kind *R Unit* where *Unit* is a kind which has a single element which we will denote $()$. A function using such a stack will have the following shape:

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

The kind of the stack type is defined as $\mathit{ST} = \mathit{Nat} \to \mathit{Unit} \to \Omega$ 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 both in the type variable *Sp* and the term variable *sp*. The size of the stack is kept both in the type variable *Ss* and in the term variable *ss* which will be used as a sentinel to detect stack overflow. 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 *St'* describes the type of the rest, which can be changed at will.

When pushing a new element on the stack, we need to find the address of the next consecutive element in region S, so we use the offset operator presented in Sec. 5.5. Pushing an object on the stack is done by:

$$\begin{array}{l}\mathsf{set}[\lambda\_.\sigma]\ \mathit{sp} := v; \\ \mathsf{let}\ \mathit{sp'} = \mathsf{offset}[P]\ \mathit{sp}\ 1\ \mathsf{in} \\ \mathsf{let}\ \langle \_, \mathit{sp}\rangle = \mathsf{open}\ \mathit{sp'}\ \mathsf{in}\ e\end{array}$$

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 P is a proof that the stack is still alive and that we do not overflow it. The proof that the stack is still alive is just *inbase* since S is the first region in Θ. The proof that we do not overflow it will require more work: it could come from an analysis of worst case stack use, or from runtime checks using *ss* and the pointer comparison defined earlier:

$$\begin{array}{l}\mathit{if}\ (\mathit{sp} < \mathit{ss}) \\ \quad (\lambda[p\!:\!(\mathit{Sp} < \mathit{Ss})]\{h\}().\ \\ \qquad \mathsf{set}[\lambda\_.\sigma]\ \mathit{sp} := v; \\ \qquad \mathsf{let}\ \mathit{sp'} = \mathsf{offset}[(\mathit{conj}\ p\ \mathit{inbase})]\ \mathit{sp}\ 1\ \mathsf{in} \\ \qquad \mathsf{let}\ \langle \_, \mathit{sp}\rangle = \mathsf{open}\ \mathit{sp'}\ \mathsf{in}\ e) \\ \quad (\lambda[\_\!:\!\neg(\mathit{Sp} < \mathit{Ss})]\{h\}().\ \\ \qquad \mathsf{error}\ \text{"}\texttt{Stack overflow}\text{"}) \end{array}$$

To pop elements off the stack we simply revert 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:

$$\begin{aligned} st' = \lambda n.\mathrm{if}\ (n = sp)\ (\lambda\_.\sigma_1) \\ (\mathrm{if}\ (n = sp{+}1)\ (\lambda\_.\sigma_2) \\ (\mathrm{if}\ \ldots\ (st\, n))) \end{aligned}$$

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*.

5.6.1 *STAL.* Morrisett et al. [2002] present a typed assembly language STAL with type safe stack manipulation primitives. Basically, every function takes a special sp argument whose type σ is a list of types describing the type of every element in the stack. We can encode their stack types σ by representing them as objects of type *List* Ω:

$$\begin{array}{lcll} nil & \Rightarrow & nil & \\ \tau{::}\sigma & \Rightarrow & cons\ \tau\ \sigma & Sp = length\ \sigma \\ \sigma_1 \circ \sigma_2 & \Rightarrow & append\ \sigma_1\ \sigma_2 & St\ i\ \_ = nth\ \sigma\ (Sp - (succ\ i)) \\ ptr(\sigma) & \Rightarrow & ()\ at\ S.(length\ \sigma) & \end{array}$$

In exchange for the additional proof manipulation that we have to use to explain what we're doing and which are not neccesary (or even possible) in STAL because it is built into its type system, we get to use as many stakcs as we want, and we get to use normal references to point into the stack.

5.7 Region aliases

Looking at example 5.3, one may wonder why we decided to provide a cast operation to reorder Θ rather than just make the typing rules for set, freergn, put, and cast a bit more flexible such that the affected region does not need to be the first in Θ. Indeed, the reader may get the impression that doing so would save us 2 of the 3 casts in the case of weak updates. But this is only true if it is syntactically obvious *where* the region ρ is located in Θ, whereas the code using casts does not have such a restriction: all it requires is a proof that ρ is somewhere in Θ. This flexibility can be used to encode region aliases.

For example, in a context where $\Theta = \{\rho_1 \mapsto (\varphi_1, n_1), \rho_2 \mapsto (\varphi_2, n_2), \rho_3 \mapsto (\varphi_3, n_3)\}$, we can have a pointer v of type τ *at* (ρ, m) together with a proof P that $\rho \in \{\rho_1, \rho_3\}$. Even though we do not know where is ρ in Θ since we do not know whether ρ is ρ_1 or ρ_3, we can apply get or (weak) set to v since they only require a proof that $\rho \mapsto (\varphi, n) \in \Theta$ for some φ and n. In the general case, we do not know whether to use φ_1 and n_1 or φ_2 and n_2, but we can always use:

$$\begin{array}{l} \varphi = \mathsf{match}\ P\ \mathsf{with}\ inbase\ \_ \Rightarrow \varphi_1 \mid \_ \Rightarrow \varphi_3 \\ n = \mathsf{match}\ P\ \mathsf{with}\ inbase\ \_ \Rightarrow n_1 \mid \_ \Rightarrow n_3 \end{array}$$

In most cases where such region aliases are used, φ_1 and φ_3 are very similar (or even equal) in which case a different φ can be chosen that better reflects that similarity.

If we look at what happens with such region aliases inside the weak set, we see that after the first cast, the region environment Θ is neither $\{\rho_1 \mapsto (\varphi_1, n_1), \rho_2 \mapsto (\varphi_2, n_2), \rho_3 \mapsto (\varphi_3, n_3)\}$ nor $\{\rho_1 \mapsto (\varphi_3, n_3), \rho_2 \mapsto (\varphi_2, n_2), \rho_3 \mapsto (\varphi_1, n_1)\}$. Instead it's just $\Theta' \oplus$

$\rho \mapsto (\varphi, n)$, which illustrates the unusual shape that the Θ environment can take and which the simplistic BNF syntax in Fig. 6 cannot capture.

5.8 Focus

The Vault language [DeLine and Fähndrich 2001] 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 [Fähndrich and DeLine 2002], the authors extended the language with *adoption* which makes it possible 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: $\exists n.\sigma\ \mathit{at}\ \rho.n$. A code equivalent to the focus construct will then look like:

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

Where e_1 has type $\exists n.\sigma\ \mathit{at}\ \rho.n$ and x has thus type $\sigma\ \mathit{at}\ \rho.n$. 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. The cast will usually be needed in order to massage the type of ρ which may look like $\lambda i.\mathsf{if}\ (i = n)\ (\lambda t.t)\ (\lambda t.t)$ so P will simply prove that this is equivalent to *idr*.

Remember that focus had the additional constraint that the operation is scoped and that other references to ρ could temporarily not be used. In our encoding, the focus operation does not have to be scoped: we can place the "unfocus" step performed by cast anywhere. Better yet, we don't have the constraint that other references can't be used either. Instead, our region's type will simply not be uniform any more, so without knowing the aliasing relationship of those other references we will not know precisely what is the actual type of the location they point to. The difference may seem minor, but if the program can recognize focused objects by putting a special runtime tag on them, then it can easily dynamically recover the needed aliasing information. A common example of such an operation is when you do a graph traversal with pointer reversal: the pointer reversal requires a focus operation, and objects whose pointer is reversed are tagged such that the code can tell at runtime if the object is being traversed (and is thus currently focused) or not.

6. GARBAGE COLLECTION

Now that we have shown some basic uses of our system, we will present a more significant application, which is a generational garbage collector, which will demonstrate how strong update can be used even in the absence of static information about the existing aliases.

The basic idea of a type-preserving GC, proposed by Wang and Appel [2001], is to layer a stop&copy collector on top of a region calculus, where the whole heap is placed in a single region and where the *copy* function copies the heap from the *from* region to a new *to* region and then frees the *from* region.

Because the type of the heap contains region annotations which will necessarily be different before copying than after, the *copy* function cannot just be of type $\forall t.t \to t$ but instead has to be of the form $\forall t.M\ F\ t \to M\ T\ t$ where F and T are the source and destination regions, t represents the source level type that should be preserved, and M is a type function that translates it into its lower-level representation, annotated with details that the mutator does not care about. E.g. in order to work correctly, the *copy* routine might require things like tag bits and mark bits or in case of generational GC it will need to make sure the mutator obeyed the generation barriers and provides a correct remembered set. All those added constraints will need to be somehow encoded in M since *copy* has obviously no control over t. Let us sketch the types of a type-preserving generational GC.

6.1 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:

$$\tau ::= \mathit{int} \mid \mathit{pair}\,\tau\,\tau \mid \mathit{ref}\,\tau \mid \mathit{mu}\,\tau \mid t$$

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: $M\ r\ o\ y\ \tau$ is the low-level representation of the source-level type τ, it will include extra fields such as a place for forwarding pointers and will encode the generational constraint to make sure that no object in O points directly into Y. This constraint will be obtained by instantiating the o and y arguments of M with the same region O for objects in the region O.

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

The translation of a ref-cell is an intuitionistic reference to a location in region R, as seen by the use of an existential package to hide the actual location inside R. For the translation of a pair, we have to express the fact that it can live either in the nursery or in the old region and we cannot statically tell which, so we use region aliases. The regions's types while the

$$
\begin{array}{l}
\textbf{\textit{GC}}[t,...]\,\{r,o,y\}\,(x : M\ r\ o\ y\ t,\quad x_0 : \exists t.t\ \textit{at}\ r.0,\quad k : \forall[y']\{r,o,y'\}(M\ r\ o\ y'\ t) \to 0) \\
\quad = \textit{GCcopy}[y,o,...](x : M\ r\ o\ y\ t, \\
\qquad\qquad \lambda[...]\{...\}(x' : M\ r\ o\ o\ t). \\
\qquad\qquad\quad \textit{GCredirect}[r,o,y,...](x_0 : \exists t.t\ \textit{at}\ r.0, \\
\qquad\qquad\qquad\qquad\qquad\qquad \lambda[...]\{...\}(). \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{freergn}\ y; \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{let}\ y' = \mathsf{newrgn}\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textit{GCwiden}[...](x_0, \lambda[...]\{...\}(). \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad k[y'](x'))))
\end{array}
$$

Fig. 13. Sketch of the GC

mutator is running are:

$$
\begin{array}{l}
R \mapsto \lambda n.\lambda t.M\ R\ O\ Y\ t \\
O \mapsto \lambda n.\lambda(t_1,t_2).(M\ R\ O\ O\ t_1) \times (M\ R\ O\ O\ t_2) \\
Y \mapsto \lambda n.\lambda(t_1,t_2).(M\ R\ O\ Y\ t_1) \times (M\ R\ O\ Y\ t_2)
\end{array}
$$

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 of Y 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 up-to-date 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.

6.2 Overview of the GC

Figure 13 shows a sketch of the *GC* function. If you ignore the nesting due to the use of continuation passing style, you can see that the function does the following:

(1) it takes 3 arguments: a pointer x which is the single root of the heap, a pointer x_0 to the beginning of the region R, and a continuation k;
(2) it first calls *GCcopy* which copies recursively x from the region Y to the region O;
(3) it then calls *GCredirect* which traverses the region R and redirects all references that point to Y to point to new copies in O;
(4) then it can free the nursery Y and allocate a new one Y';
(5) finally the last step before returning to the continuation k is a call to *GCwiden* which is a big no-op that simply updates all the types of the form $M\ r\ o\ o$ to $M\ r\ o\ y'$ so as to allow the use of the new region Y'.

The GC code is split into the following parts:

GCcopy: depth-first copy of an object from Y to O.

GCrmap: an auxiliary higher-order function that scans the region R, applying a given function to every element. Used by both *GCredirect* and *GCwiden*.

GCredirect: scan the region R redirecting pointers to Y by copying the object to O and redirecting the reference to use the new copy.

GCwiden: update the type of objects in R to refer to the new region Y'.

GC: the toplevel function.

In order for *GCredirect* and *GCwiden* to be able to do their job, we need both a reference to the beginning and the end of the region R. We decided to place the reference to the end directly inside the region R at its location 0, so that the reference x_0 that point to the location $R.0$ gives us access both to the beginning as well as the end of the region, and it also has the benefit of ensuring that we do not need to cater to the special case where the region is empty.

6.3 Depth-first copy

To be able to copy parts of the heap, we need to be able to get runtime type information, which our language does not provide directly. So we need to refine the definition of M to add tags:

$$\begin{aligned}&\mathsf{Inductive}\ \Omega^\tau : \mathsf{Kind} := \mathit{int} : \Omega^\tau \mid \mathit{pair} : \Omega^\tau \to \Omega^\tau \to \Omega^\tau \mid \mathit{ref} : \Omega^\tau \to \Omega^\tau\\ &\qquad\qquad \mid \mathit{mu} : \Omega^\tau \to \Omega^\tau \mid \mathit{var} : \mathit{Nat} \to \Omega^\tau\end{aligned}$$

$$\begin{aligned}&\mathsf{type}\ \mathit{Mtag}\ (\mathit{int}) &&= 0\\ &\quad \mid (\mathit{pair}\ t_1\ t_2) &&= 1\\ &\quad \mid (\mathit{ref}\ t) &&= 2\\ &\quad \mid (\mathit{mu}\ t) &&= 3\\ &\quad \mid (\mathit{var}\ n) &&= 4\end{aligned}$$

$$\begin{aligned}&\mathsf{type}\ \mathit{Mdata}\ r\ o\ y\ (\mathit{int}) &&= \exists n.\mathit{snat}\ n\\ &\quad \mid r\ o\ y\ (\mathit{pair}\ t_1\ t_2) &&= \exists b.\exists\_ : b \in \{o, y\}.\exists n.(t_1, t_2)\ \mathit{at}\ b.n\\ &\quad \mid r\ o\ y\ (\mathit{ref}\ t) &&= \exists n.t\ \mathit{at}\ r.(\mathit{succ}\ n)\\ &\quad \mid r\ o\ y\ (\mathit{mu}\ t) &&= \forall t.t\\ &\quad \mid r\ o\ y\ (\mathit{var}\ n) &&= \forall t.t\end{aligned}$$

$$\mathsf{type}\ t_1 \times t_2 = \mathit{tup}\ 2\ (\lambda i.\mathsf{match}\ i\ \mathsf{with}\ 0 \Rightarrow t_1 \mid \_ \Rightarrow t_2)$$
$$\mathsf{type}\ M'\ r\ o\ y\ t = \mathit{snat}\ (\mathit{Mtag}\ t) \times \mathit{Mdata}\ r\ o\ y\ t$$

Notice how we map *ref* types to references at *succ* n so as to ensure that those references do not point to the special location 0 of the region R. Both *Mtag* and *Mdata* pay attention to the *var* case, but this case should never occur, since it corresponds to an occurrence out of scope. The definition of M' is not very useful for recursive types either since it maps them arbitrarily to the void type $\forall t.t$. So we define M as M' where recursive types have been unfolded once:

$$\begin{aligned}&\mathsf{type}\ \mathit{unfold}\ (\mathit{mu}\ t) &&= \mathit{DIsubstitute}\ t\ (\mathit{mu}\ t)\\ &\quad \mid t &&= t\end{aligned}$$
$$\mathsf{type}\ M\ r\ o\ y\ t = M'\ r\ o\ y\ (\mathit{unfold}\ t)$$

DIsubstitute $\tau_1\ \tau_2$ is a function not shown which replaces every occurrence of the topmost variable in τ_1 with τ_2. Having to give tags to types such as *var* and *mu* is not very satisfactory, but with a more precise definition of Ω^τ we can easily rule out free occurrences of

$$
\begin{array}{l}
\textsf{type } \mathit{TauxK}\ \mathit{tag} = \textsf{match } \mathit{tag} \textsf{ with } 0 \Rightarrow \mathit{Unit} \mid 1 \Rightarrow \Omega^\tau \wedge \Omega^\tau \mid 2 \Rightarrow \Omega^\tau \mid 3 \Rightarrow \Omega^\tau \mid 4 \Rightarrow \mathit{Nat} \\
\textsf{type } \mathit{Taux}\ t = \textsf{match } t \textsf{ with } \mathit{int} \Rightarrow () \mid \mathit{pair}\ t_1\ t_2 \Rightarrow (t_1, t_2) \mid \mathit{ref}\ t \Rightarrow t \mid \mathit{mu}\ t \Rightarrow t \mid \mathit{var}\ n \Rightarrow n \\
\textsf{type } \mathit{TfromTag}\ \mathit{tag}\ (\mathit{aux} : \mathit{TauxK}\ \mathit{tag}) = \\
\quad \textsf{match } \mathit{tag} \textsf{ with } 0 \Rightarrow \mathit{int} \mid 1 \Rightarrow \mathit{pair}\ (\mathit{fst}\ \mathit{aux})\ (\mathit{snd}\ \mathit{aux}) \mid 2 \Rightarrow \mathit{ref}\ \mathit{aux} \mid 3 \Rightarrow \mathit{mu}\ \mathit{aux} \mid 4 \Rightarrow \mathit{var}\ \mathit{aux} \\
\textsf{type } \mathit{EqTfromTag}\ t : (t = \mathit{TfromTag}\ (\mathit{Mtag}\ t)\ (\mathit{Taux}\ t)) = \\
\quad \textsf{match } t \textsf{ with } \mathit{int} \Rightarrow \mathit{eq\_refl} \mid \mathit{pair}\ t_1\ t_2 \Rightarrow \mathit{eq\_refl} \mid \mathit{ref}\ t \Rightarrow \mathit{eq\_refl} \mid \mathit{mu}\ t \Rightarrow \mathit{eq\_refl} \mid \mathit{var}\ n \Rightarrow \mathit{eq\_refl} \\
\textsf{type } \mathit{Eq2Eq}\ t_1\ t_2\ f\ (P : t_1 = t_2) : (f\ t_1 = f\ t_2) = \textsf{match } P \textsf{ with } \mathit{eq\_refl} \Rightarrow \mathit{eq\_refl} \\
\\
\textsf{type } \mathit{GCdispatchType}\ r\ o\ y\ h\ \mathit{fk}\ \mathit{tag} \\
\quad = \forall \mathit{aux} : \mathit{TauxK}\ \mathit{tag}.\textsf{let } t' = \mathit{Tfromtag}\ \mathit{tag}\ \mathit{aux} \textsf{ in } \{h\}(M'\ r\ o\ y\ t',\ \ \mathit{fk}\ t') \rightarrow 0 \\
\\
\textbf{\textit{GCdispatchVoid}}[n, h, \mathit{fk}, aux]\{h\}(x : \mathit{snat}\ n \times \forall t.t, k : \mathit{fk}\ \mathit{aux}) = x.1[\{h\}() \rightarrow 0]() \\
\\
\textbf{\textit{GCdispatch}}[r, o, y, h, \mathit{fk}, t]\ \{h\} \\
\qquad \begin{pmatrix}
x & : M'\ r\ o\ y\ t, \\
k_i & : \forall \mathit{aux}.\{h\}(\mathit{snat}\ 0 \times \exists n.\mathit{snat}\ n, \mathit{fk}\ \mathit{int}) \rightarrow 0, \\
k_\times & : \forall \mathit{aux}.\{h\}(\mathit{snat}\ 1 \times \exists b.\exists\_ : b \in \{o, y\}.\exists n.\mathit{aux}\ \mathit{at}\ b.n, \mathit{fk}\ (\mathit{pair}\ (\mathit{fst}\ \mathit{aux})\ (\mathit{snd}\ \mathit{aux}))) \rightarrow 0, \\
k_r & : \forall \mathit{aux}.\{h\}(\mathit{snat}\ 2 \times \exists n.\mathit{aux}\ \mathit{at}\ r.(\mathit{succ}\ n), \mathit{fk}\ (\mathit{ref}\ \mathit{aux})) \rightarrow 0, \\
k & : \mathit{fk}\ t
\end{pmatrix} \\
= \textsf{let type } \varphi = \mathit{GCdispatchType}\ r\ o\ y\ h\ \mathit{fk} \\
\qquad \mathit{tag} = \textsf{sel}[\mathit{le\_s}\ \mathit{le\_n}]\ x.0 \\
\qquad \mathit{table} = (k_i, k_\times, k_r, \mathit{GCdispatchVoid}[3, h, \mathit{fk} \circ \mathit{mu}], \mathit{GCdispatchVoid}[4, h, \mathit{fk} \circ \mathit{var}])^\varphi \\
\qquad \mathit{branch} = \textsf{sel}[\mathit{TagLT5}\ t]\ \mathit{table}.\mathit{tag} \\
\qquad \mathit{branch'} = \mathit{branch}[\mathit{Taux}\ t] \\
\qquad \mathit{branch''} = \textsf{cast}\ [\mathit{Eq2Eq}\ (\lambda t'.\{h\}(x : M'\ r\ o\ \mathit{by}\ t', k : \mathit{fk}\ t') \rightarrow 0)\ (\mathit{EqTfromTag}\ t)] \\
\qquad\qquad\qquad \mathit{branch'} \\
\quad \textsf{in } \mathit{branch''}(x, k)
\end{array}
$$

Fig. 14. The *GCdispatch* code.

var. We cannot on the other hand rule out *mu* because *unfold* only does one step of unfolding and removing all *mu*s may require a variable or even infinite number of unfolding steps, so we cannot code it in CiC. All in all, the only remaining problem is that we have to occasionally worry about tags 3 and 4 even though they are never used, and that we have to be careful to only use recursive types whose unfolding is not itself a *mu* type, which is usually easy to ensure. It is interesting to see how thanks to the extra indirection provided by the regions's types, we can encode the source-level recursive types even though our language does not itself provide recursive types.

Figure 14 shows an auxiliary function *GCdispatch* which implements a switch statement on objects of type $M\ r\ o\ y\ t$. This is similar to the *if* function presented in Sec. 5.1. All it does, is extract the tag of its argument x and use it to look up a branch table. The subtlety in the code is due to the fact that we need the encoded switch statement to refine the type t in each branch although the switch only examines some integer that depends on t. For that we define the functions *TfromTag* and *Taux* such that $t = \mathit{TfromTag}\ (\mathit{Mtag}\ t)\ (\mathit{Taux}\ t)$. This allows us to reflect the info we get about the tag into knowledge about t. The function *TfromTag* is used in *GCdispatchType*, which is the function that describes the commonality between each branch in the branch table. *EqTfromTag* and *Eq2Eq* are auxiliary proofs used to show that those functions indeed correctly split and reconstruct the same t. Note that although we have 5 different tags, *GCdispatch* only has three branches, because it automatically directs the two impossible cases (the *mu* and *var* cases) to *GCdispatchVoid*.

This *GCdispatch* function reproduces within our low-level language the functionality

$$
\begin{array}{l}
\textbf{\textit{GCid}}[tf, hf, hd, \mathit{aux}]\{hf\ hd\}(x : tf\ \mathit{aux}, k : \forall[hd]\{hf\ hd\}(tf\ \mathit{aux}) \to 0) = k[hd]x \\
\textbf{\textit{GCidInt}} = \mathit{GCid}[\lambda \mathit{aux}.\mathit{snat}\ 0 \times \exists n.\mathit{snat}\ n] \\
\textbf{\textit{GCidRef}}[r] = \mathit{GCid}[\lambda \mathit{aux}.\mathit{snat}\ 2 \times \exists n.\mathit{aux}\ \mathit{at}\ r.(\mathit{succ}\ n)]
\end{array}
$$

$$
\begin{array}{l}
\textbf{\textit{GCcopyPair}}[r, o, on, y, yn, h, by, P : by \in \{o, y\}, \mathit{aux}] \\
\qquad \{\mathit{GCcopyRenv}\ r\ o\ on\ y\ yn\ h\} \\
\qquad \begin{pmatrix} x : \mathit{snat}\ 1 \times \exists b.\exists\_ : b \in \{o, by\}.\exists n.\mathit{aux}\ \mathit{at}\ b.n, \\ k : \forall[on]\{\mathit{GCcopyRenv}\ r\ o\ on\ y\ yn\ h\}(\mathit{snat}\ 1 \times \exists b.\exists\_ : b \in \{o, by\}.\exists n.\mathit{aux}\ \mathit{at}\ b.n) \to 0 \end{pmatrix} \\
= \mathsf{let}\ x' = \mathsf{sel}[\mathit{le\_n}]\ x.1 \\
\quad \langle b, \langle P', \langle n, x'' \rangle \rangle \rangle = \mathsf{open}\ x' \\
\quad \mathit{test} = \mathit{rcmp}[\mathit{inskip}\ \mathit{inbase}]\ x''\ y \\
\quad \mathsf{in}\ \mathit{if}\ (\mathit{test}) \\
\qquad (\lambda[P_= : (b \mapsto \_ \in\ \Theta) \to b = y]\{\mathit{GCcopyRenv}\ r\ o\ on\ y\ yn\ h\}(). \\
\qquad\quad \mathsf{let}\ P'_= : (b = y) = P_= (\mathsf{match}\ P', P\ \mathsf{with}\ |\ \mathit{inskip}\ \_, \mathit{inskip}\ \_ \Rightarrow\ (\mathit{inskip}\ \mathit{inbase}) |\ \_, \_ \Rightarrow \mathit{inbase}) \\
\qquad\qquad x_p = \mathsf{get}[\mathit{inskip}\ \mathit{inbase}]\ x'' \\
\qquad\qquad x_0 = \mathsf{sel}[\mathit{le\_s}\ \mathit{le\_n}]\ x.0 \\
\qquad\qquad x_1 = \mathsf{sel}[\mathit{le\_n}]\ x.1 \\
\qquad\quad \mathsf{in}\ \mathit{GCcopy}[r, o, on, y, yn, h, \mathit{fst}\ \mathit{aux}, y, \mathit{inskip}\ \mathit{inbase}] \\
\qquad\qquad\qquad (x_0, \lambda[on]\{\mathit{GCcopyRenv}\ r\ o\ on\ y\ yn\ h\}(x'_0 : M\ r\ o\ y\ (\mathit{fst}\ \mathit{aux})). \\
\qquad\qquad\qquad\quad \mathit{GCcopy}[r, o, on, y, yn, h, \mathit{snd}\ \mathit{aux}, y, \mathit{inskip}\ \mathit{inbase}] \\
\qquad\qquad\qquad\qquad (x_1, \lambda[on]\{\mathit{GCcopyRenv}\ r\ o\ on\ y\ yn\ h\}(x'_1 : M\ r\ o\ y\ (\mathit{snd}\ \mathit{aux})). \\
\qquad\qquad\qquad\qquad\quad \mathsf{let}\ new = \mathsf{put}[o, \mathit{aux}]\ (x_0, x_1) \\
\qquad\qquad\qquad\qquad\quad \mathsf{in}\ k[on + 1]((1, \langle o, \langle \mathit{inbase}, \langle on, new \rangle \rangle \rangle))))))) \\
\qquad (\lambda[P_\neq : \neg((b \mapsto \_ \in\ \Theta) \to b = y)]\{\mathit{GCcopyRenv}\ r\ o\ on\ y\ yn\ h\}(). \\
\qquad\quad \mathsf{let}\ P_= : (b = o) = (\mathsf{match}\ P', P\ \mathsf{with}\ |\ \mathit{inskip}\ \_, \mathit{inskip}\ \_ \Rightarrow\ (P_\neq\ (\lambda\_.\mathit{eq\_refl})) |\ \_, \_ \Rightarrow \mathit{inbase}) \\
\qquad\quad \mathsf{in}\ k[on]((1, \langle o, \langle \mathit{inbase}, \langle n, x'' \rangle \rangle \rangle)))))
\end{array}
$$

$$
\begin{array}{l}
\textbf{\textit{GCcopy}}[r, o, on, y, yn, h, t, by, P : by \in \{o, y\}] \\
\qquad \{\mathit{GCcopyRenv}\ r\ o\ on\ y\ yn\ h\} \\
\qquad (x : M\ r\ o\ by\ t, \quad k : \forall[on]\{\mathit{GCcopyRenv}\ r\ o\ on\ y\ yn\ h\}(M\ r\ o\ o\ t) \to 0) \\
= \mathit{GCdispatch} \begin{bmatrix} r, o, by, \\ \mathit{GCcopyRenv}\ r\ o\ on\ y\ yn\ h, \\ \lambda t'.\forall[on]\{\mathit{GCcopyRenv}\ r\ o\ on\ y\ yn\ h\}(M'\ r\ o\ o\ t') \to 0, \\ \mathit{unfold}\ t \end{bmatrix} \\
\qquad\qquad \begin{pmatrix} x, \\ \mathit{GCidInt}[(\lambda on.\mathit{GCcopyRenv}\ r\ o\ on\ y\ yn\ h), on], \\ \mathit{GCcopyPair}[r, o, on, y, yn, h, by, P], \\ \mathit{GCidRef}[(\lambda on.\mathit{GCcopyRenv}\ r\ o\ on\ y\ yn\ h), on], \\ k \end{pmatrix}
\end{array}
$$

Fig. 15. The *GCcopy* code.

typically provided by refining pattern matching on inductive definitions or GADTs [Xi et al. 2003], including the fact that some of the branches are eliminated because their typing context contains a contradiction, witnessing the fact that the code is unreachable.

The *GCcopy* entry point is shown in Fig. 15. It just uses *GCdispatch* to either do nothing (via *GCid*), if the value to copy is a ref-cell or an integer, or delegate to *GCcopyPair*. A more interesting part of *GCcopy* is the way it describes its region's types:

$$
\begin{array}{l}
\mathsf{type}\ \mathit{TupRgn}\ o\ y\ r = \lambda n.\lambda t.(M\ r\ o\ y\ (\mathit{fst}\ t)) \times (M\ r\ o\ y\ (\mathit{snd}\ t)) \\
\mathsf{type}\ \mathit{GCcopyRenv}\ r\ o\ on\ y\ yn\ h \\
\qquad = (h \oplus y \mapsto (\mathit{TupRgn}\ o\ y\ r, yn) \oplus o \mapsto (\mathit{TupRgn}\ o\ o\ r, on))
\end{array}
$$

where *fst* and *snd* are the usual projection functions for tuples. Since it only accesses

```
GCrmap[i, r, rn, o, y, y', hf, hd]
        {GCrmapRenv i r rn o y y' (hf hd)}
        ( x : ∃t.t at r.i,
          lim : ∃t.t at r.rn,
          f : ∀[i, t, hd]{GCrmapRenv i r rn y' y' (hf hd)}
                (M r o y t, ∀[hd]{GCrmapRenv i r rn y' y' (hf hd)}(M r o y' t) → 0) → 0
          k : ∀[hd]{GCrmapRenv 0 r rn y' y' (hf hd)}() → 0 )
  = let ⟨t, x'⟩ = open x
        ⟨_, lim'⟩ = open lim
        notdone = x' < lim'
    in if notdone
        (λ[P : i < rn]{GCrmapRenv i r rn o y y' (hf hd)}().
          let p = offset[(P, inbase)] x' 1
              x = get[inbase] p
          in f[hd](x, λ[hd]{GCrmapRenv i r rn y' y' (hf hd)}(x' : M r o y' t).
                        set[M r o y'] p := x';
                        cast[...] r ↦ RefRgn ((succ i) r rn o y' y');
                        GCrmap[succ i, r, rn, o, y, y', hf, hd](p, lim, f, k)))
        (λ[P : ¬i < rn]{GCrmapRenv i r rn o y y' (hf hd)}().
          cast[λi'.λt.λP' : i' < succ rn....];
               r ↦ RefRgn 0 r rn o y' y'
          k[h]())
```

Fig. 16. The *GCrmap* code.

values in region y and only modifies region o, the rest of the memory is abstracted in h. *GCcopyPair* checks with *rcmp* if the pair is in y and if so, recurses on both fields. In case it does not point to y, we would really want to return just x, but instead we rebuild a new pair of the tag and the value, because the existential package in the *Mdata* part needs to be changed to reflect the fact that we know the content does not point to y. We could easily improve this code to avoid this reconstruction and return x directly if we add coercions such as the ones presented in [Monnier 2007].

6.4 Mapping over R

One of the unusual functionalities offered by typed regions is the ability to scan a region, which we need for our GC. Figure 16 shows the *GCrmap* function that abstracts such a scan.

It take two references x and lim that point in a region r, along with a function f to apply to all elements in r from right after x upto and including lim. Of course it also takes a continutation k. Since this function only cares about the region R its region's types only describe R and abstract the rest in h:

```
type RefRgn i r rn o y y' =
    (λn.match n with zero ⇒ λ_.∃t.t at r.rn | succ n' ⇒ M r o (if (i < succ n') y y'),
     succ rn)
type GCrmapRenv i r rn o y y' h = h ⊕ r ↦ RefRgn i r rn o y y'
```

In the *GCrmap* code, h is actually decomposed into *hf* and *hd* where $h = hf\ hd$ because the function f may modify h, so *hf* describes the invariant part of h and *hd* is the variable part.

$$
\begin{array}{l}
\textbf{\textit{GCredirectAux}}[r, rn, o, y, h, i, t, on] \\
\qquad \{\textit{GCrmapRenv}\ i\ r\ rn\ o\ y\ o\ (\textit{GCcopyRenv}\ r\ o\ on\ y\ yn\ h)\} \\
\qquad \begin{pmatrix} x : M\ r\ o\ y\ t, \\ k : \forall[on]\{\textit{GCrmapRenv}\ i\ r\ rn\ o\ y\ o\ (\textit{GCcopyRenv}\ r\ o\ on\ y\ yn\ h)\}(M\ r\ o\ o\ t) \to 0 \end{pmatrix} \\
= \mathsf{cast}[...](\textit{GCrmapRenv}\ i\ r\ rn\ o\ y\ o\ (\textit{GCcopyRenv}\ r\ o\ on\ y\ yn\ h)) \\
\qquad \sim (\textit{GCcopyRenv}\ r\ o\ on\ y\ yn\ (\textit{GCrmapRenv}\ i\ r\ rn\ o\ y\ o\ h)) \\
\quad \textit{GCcopy}[r, o, on, y, yn, (\textit{GCrmapRenv}\ i\ r\ rn\ o\ y\ o\ h), t, y, \textit{inskip inbase}] \\
\qquad (x, \lambda[on]\{\textit{GCcopyRenv}\ r\ o\ on\ y\ yn\ (\textit{GCrmapRenv}\ i\ r\ rn\ o\ y\ o\ h)\}(x' : M\ r\ o\ o\ t). \\
\qquad\quad \mathsf{cast}[...](\textit{GCcopyRenv}\ r\ o\ on\ y\ yn\ (\textit{GCrmapRenv}\ i\ r\ rn\ o\ y\ o\ h)) \\
\qquad\qquad \sim (\textit{GCrmapRenv}\ i\ r\ rn\ o\ y\ o\ (\textit{GCcopyRenv}\ r\ o\ on\ y\ yn\ h)) \\
\qquad\quad k[on](x'))
\end{array}
$$

$$
\begin{array}{l}
\textbf{\textit{GCredirect}}[r, rn, o, on, y, yn, h] \\
\qquad \{\textit{GCrmapRenv}\ 0\ r\ rn\ o\ y\ o\ (\textit{GCcopyRenv}\ r\ o\ on\ y\ yn\ h)\} \\
\qquad \begin{pmatrix} x_0 : \exists t.t\ \textit{at}\ r.0, \\ k : \forall[on]\{\textit{GCrmapRenv}\ 0\ r\ rn\ o\ o\ o\ (\textit{GCcopyRenv}\ r\ o\ on\ y\ yn\ h)\}() \to 0 \end{pmatrix} \\
= \mathsf{let}\ \langle \_, x_0' \rangle = \mathsf{open}\ x_0 \\
\qquad \mathit{lim} = \mathsf{get}[\mathit{inbase}]\ x_0' \\
\quad \mathsf{in}\ \textit{GCrmap}[0, r, rn, o, y, o, (\lambda on.\textit{GCcopyRenv}\ r\ o\ on\ y\ yn\ h), on](x_0, \mathit{lim}, \textit{GCredirectAux}, k)
\end{array}
$$

Fig. 17. The *GCredirect* code.

The type *RefRgn* used to describe R gives a special type to its element 0, which is hence forced to hold a pointer to the last element of R; also the type is parameterized by i which is the index of the scan: all elements before i (other than the element 0) have a type of the form $M\ r\ o\ y\ t$, whereas all the ones after i have a type of the form $M\ r\ o\ y'\ t$, so the scan start with a region with all non-zero elements of type $M\ r\ o\ y\ t$ and at the end, they have all been updated to $M\ r\ o\ y'\ t$. This is a prime example of a code that uses a strong update, yet does not have any particular aliasing information available about the locations it updates: there can be any number of references pointing from anywhere into any part of region r.

6.5 Redirecting pointers

Once the copy from the heap root is performed, we need to check every reference cell in region R and if it points to an object in Y redirect it by copying it into region O. The code is shown in Fig. 17. We also need to update the type of region R at each step, so that at the end of the redirection loop, the type of R does not refer to Y any more. This redirection loop basically uses *GCrmap* to apply *GCcopy* to every element of R. But of course, *GCcopy* cannot be passed directly to *GCrmap*: the main problem is that the two functions expects their regions in a different order, so we introduce a wrapper *GCredirectAux* which uses cast to reorder the regions back and forth between the order expected by the two functions.

6.6 Widening R to allow references to the new Y'

After all the objects have been copied and the references redirected, the nursery Y is not needed any more and can be freed, and a new one allocated in its place. But we cannot yet return to the main continuation: all our objects now have types of the form $M\ r\ o\ o\ t$, whereas the continuation expects objects of type $M\ r\ o\ y\ t$. This is no problem since $M\ r\ o\ o\ t$ is a sort of subtype of $M\ r\ o\ y\ t$, but since our type system does not provide subsumption, we have to explicitly cast those objects by unpacking and repacking them.

$$
\begin{array}{l}
\textbf{\textit{GCwidenPair}}[i, r, rn, o, y, h, \mathit{aux}] \\
\qquad\qquad \{\mathit{GCrmapRenv}\ i\ r\ rn\ o\ o\ y\ h\} \\
\qquad\qquad \begin{pmatrix} x : \mathit{snat}\ 1 \times \exists b.\exists\_ : b \in \{o, o\}.\exists n.\mathit{aux}\ \mathit{at}\ b.n, \\ k : \forall[\_]\{\mathit{GCrmapRenv}\ i\ r\ rn\ o\ o\ y\ h\}(\mathit{snat}\ 1 \times \exists b.\exists\_ : b \in \{o, y\}.\exists n.\mathit{aux}\ \mathit{at}\ b.n) \to 0 \end{pmatrix} \\
= \mathsf{let}\ x_1 = \mathsf{sel}[\mathit{le\_n}]\ x.1 \\
\qquad \langle b, \langle \_, x_2 \rangle \rangle = \mathsf{open}\ x_1 \\
\quad \mathsf{in}\ k((1, \langle o, \langle \mathit{inbase}, x_2 \rangle \rangle))
\end{array}
$$

$$
\begin{array}{l}
\textbf{\textit{GCwidenAux}}[r, rn, o, y, h, i, t, hd] \\
\qquad\qquad \{\mathit{GCrmapRenv}\ i\ r\ rn\ o\ o\ y\ h\} \\
\qquad\qquad \begin{pmatrix} x : M\ r\ o\ o\ t, \\ k : \forall[\_]\{\mathit{GCrmapRenv}\ i\ r\ rn\ o\ o\ y\ h\}(M\ r\ o\ y\ t) \to 0 \end{pmatrix} \\
= \mathit{GCdispatch} \begin{bmatrix} r, o, o, \\ \mathit{GCrmapRenv}\ i\ r\ rn\ o\ o\ y\ h, \\ \lambda m.\forall[\_]\{\mathit{GCrmapRenv}\ i\ r\ rn\ o\ o\ y\ h\}(m\ r\ o\ y) \to 0 \end{bmatrix} \\
\qquad\qquad \begin{pmatrix} x, \\ \mathit{GCidInt}[\lambda\_.\mathit{GCrmapRenv}\ i\ r\ rn\ o\ o\ y\ h, ()], \\ \mathit{GCwidenPair}[\ldots], \\ \mathit{GCidRef}[\lambda\_.\mathit{GCrmapRenv}\ i\ r\ rn\ o\ o\ y\ h, ()], \\ k \end{pmatrix}
\end{array}
$$

$$
\begin{array}{l}
\textbf{\textit{GCwiden}}[r, rn, o, y, h] \\
\qquad\qquad \{\mathit{GCrmapRenv}\ 0\ r\ rn\ o\ o\ y\ h\} \\
\qquad\qquad \begin{pmatrix} x_0 : \exists t.t\ \mathit{at}\ r.0, \\ k : \forall[\_]\{\mathit{GCrmapRenv}\ 0\ r\ rn\ o\ y\ y\ h\}() \to 0 \end{pmatrix} \\
= \mathsf{let}\ \langle \_, x_0' \rangle = \mathsf{open}\ x_0 \\
\qquad \mathit{lim} = \mathsf{get}[\mathit{inbase}]\ x_0' \\
\quad \mathsf{in}\ \mathit{GCrmap}[0, r, rn, o, o, y, (\lambda\_.h), ()](x_0, \mathit{lim}, \mathit{GCwidenAux}[r, rn, o, y, h], k)
\end{array}
$$

Fig. 18. The *GCwiden* code.

Luckily it only needs to be done at the roots, i.e. on x and on all elements of R. The *GCwiden* takes care of updating the type of R accordingly, by scanning it again, using *GCrmap*.

GCwidenAux is the function that update a single element of R. Because of how we structured our objects, we only have to unpack&repack an existential when the pointed object is a pair. That turns out to be unfortunate here, since we now have to use *GCdispatch* to as to call *GCwidenPair* when applicable.

Notice that this whole loop is nothing more than a complex no-op in the sense that it only modifies existentials. And indeed, this loop is equivalent to the no-op that was called widen in [Monnier et al. 2001]. This code was one of the main motivation for the development of the coercion calculus presented in [Monnier 2007], and indeed such a coercion calculus could be used here to replace *GCwiden* by a coercion. Notice that this coercion would still have to do the same loop, examine the same data, unpack and repack the same existentials, except that it would be completely done at the level of types.

6.7 Putting it all together

Figure 19 shows the *GC* function itself which binds all the previous functions together. Compared to the sketch shown at the beginning, the main difference is the addition of a few cast operations. They do nothing more than re-order regions in the region environment to match the order expected by the next function, except for the cast performed right after

```
type GCrenv r rn o on y yn h = GCcopyRenv r o on y yn (GCrmapRenv 0 r rn o y y h)

GC[r, rn, o, on, y, yn, t, h]
   {GCrenv r rn o on y yn h}
   (x : M r o y t,   x0 : ∃t.t at r.0,   k : ∀[y', on'']{GCrenv r rn o on'' y' 0 h}(M r o y' t) → 0)
 = GCcopy[r, o, on, y, yn, (GCrmapRenv 0 r rn o y y h), t, y, inskip inbase]
       (x : M r o y t,
        λ[on']{GCrenv r rn o on' y yn h}(x' : M r o o t).
          cast[...](GCcopyRenv r o on' y yn (GCrmapRenv 0 r rn o y o h))
                ∼ (GCrmapRenv 0 r rn o y o (GCcopyRenv r o on' y yn h))
          GCredirect[r, rn, o, on, y, yn, (GCcopyRenv r o on' y yn h)]
                (x0 : ∃t.t at r.0,
                 λ[on'']{GCrmapRenv 0 r rn o o o (GCcopyRenv r o on'' y yn h)}().
                   cast[...](GCrmapRenv 0 r rn o o y (GCcopyRenv r o on'' y yn h))
                         ∼ (h ⊕ o ↦ ... ⊕ r ↦ ... ⊕ y ↦ ...)
                   freergn y;
                   let y' = newrgn λn.λt.snat 0 in
                   cast[...]y' ↦ TupRgn o y' r
                   cast[...](h ⊕ o ↦ ... ⊕ r ↦ ... ⊕ y' ↦ (TupRgn o y' r, 0))
                         ∼ (GCrmapRenv 0 r rn o o y (GCcopyRenv r o on'' y' 0 h))
                   GCwiden[r, rn, o, y, h]
                         (x0, λ[_]{(GCrmapRenv 0 r rn o y y (GCcopyRenv r o on'' y' 0 h))}().
                            cast[...](GCrmapRenv 0 r rn o y y (GCcopyRenv r o on'' y' 0 h))
                                  ∼ (GCcopyRenv r o on'' y' 0 (GCrmapRenv 0 r rn o y y h))
                            k[y', on'']((x'))))
```

Fig. 19. Toplevel of the GC

allocating the new nursery. This one addresses the problem of self-reference: the type of the new region y' refers to y', so when we allocate that region, the initial type that we pass to newrgn cannot be the proper type since we do not know y' yet. For this reason, newrgn is provided with a dummy type $\lambda n.\lambda t.\mathit{snat}\ 0$, which is promptly replaced by the proper $\mathit{TupRgn}\ o\ y'\ r$ via a cast.

7. RELATED WORK

7.1 Languages

The calculus of capabilities [Crary et al. 1999] 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. Monadic regions [Fluet and Morrisett 2006] show a simple and clean way to design a region calculus, based on the idea of encoding region types in a polymorphic λ-calculus. The prototype Coq encoding of our work uses such a monadic structure, for no other reason than its elegance. Hicks et al. [2004] presents actual experience with the use of such region-based memory management.

In the work on TAL [Morrisett et al. 1998], 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 [Walker and Morrisett 2000] was the first type system that accommodated a strong update primitive, even in the presence of aliasing, so long as the aliasing pattern can be described statically. Our handling of both regions and pointers is strongly influenced by this work.

The Vault language [DeLine and Fähndrich 2001] 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. One 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 [Fähndrich and DeLine 2002] 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.

Shao et al. [2002] proposed to use CiC as the type language of a programming language. This allows sophisticated type manipulation and enables programs to express arbitrary properties of the values manipulated. By keeping a clear separation between the computational language and the type language, they get to have complete freedom in the choice of computational language, as well as being able to reuse CiC wholesale as a black box, rather than reinvent a logic and its consistency proof. 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.

Monnier [2007] shows how a language similar to λ_H can be extended very simply with a coercion calculus. Adding these coercions to our language is easy and makes it possible to improve the code of our generational GC significantly.

The Deputy type system [Condit et al. 2007] extends C with a notion of dependent typing which can handle side effects while staying decidable, by postponing some of the checks to the runtime. Their goal is fairly different from ours and for our use case, their main limitation is that they do not attempt to deal with global invariants such as the ones preserved by a garbage collector.

7.2 Type-safe GC

Wang and Appel [1999] 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. [2001] extended Wang and Appel's work by using intensional type analysis [Harper and Morrisett 1995] to provide a generic *copy* function and to use existential packages to encode closures. They 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 that work.

Vanderwaart and Crary [2003] design a type system to check correct handling of details of the stack layout required by a particular GC which is kept implicit. The system is designed to handle even very sophisticated GCs which use 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.

Hawblitzel et al. [2004] introduced a very low-level language along the lines of alias types but using linear logic, with a very expressive type system. Their language allows types to be stored in run time data structures, which gives them a lot of flexibility. They show how to write a type-safe garbage collector and use the language to write low-level operating systems code like device drivers, but it is not clear how to extend this work to our case where we garbage collect a typed language and hence need type-*preservation* rather

than only type-*safety*.

Bierman et al. [2003] show how to do type-safe marshalling and unmarshalling, although their type-safety concern is very different from ours.

7.3 Hoare logic

An alternative approach to ours is to go straight to Hoare logic and prove correctness of the code. There is a lot of work in that area which tackles similar problems to ours. The main difference is first that we limit ourselves to type-safety, in order to keep the proof burden to its minimum. And second that we aim to stay within a language which can not only be used for such low-level work but can also be used, with an appropriate dose of syntactic sugar, as a classic high-level source language.

Separation Logic [Reynolds 2002] is a form of Hoare logic to reason about mutable data structures in local and modular ways. Yang [2001] shows how to use this kind of Hoare logic to reason about data structures with arbitrary sharing and cycles, more specifically traversal of a homogeneous graph using pointer reversal. This approach has been extended in [Birkedal et al. 2004] to specify and prove the correctness of an implementation of Cheney's copying GC. Because the devil is in the details, that work shares surprisingly little with ours, except that we also divide the heap into disjoint subheaps, either statically in the form of regions, or dynamically as in Sec. 6.5 where the index i of *RefRgn* is used to divide the region R into the part that has already been redirected and the part that has not.

Support for stack allocation in a typed language was already proposed in STAL [Morrisett et al. 2002]. Ahmed and Walker [2003] present a much more ambitious solution to that problem.

Filliâtre [2003] formalizes Hoare-logic style rules in CiC, thus making it possible to formally prove more or less arbitrary properties of side effecting programs. The computational language is completely external to the logic, so the type system is not integrated into the logic either. This approach could probably be extended to make a completely formal system that supports Separation Logic.

The Hoare Type Theory [Nanevski et al. 2006; Nanevski et al. 2007] is the closest in spirit and expressiveness to ours. They define a language that uses dependent typing together with side-effects, preserving decidability by confining all side effecting and non-terminating primitives inside a monad. Their language is also a variant of the Calculus of Constructions, so their logic is comparable to ours. To reason about the state of the heap as well as deal with aliasing, they use the Separation Logic, which is lower-level than our typed regions. Contrary to us, they do not stratify their type language on top of their computational language. This makes it more difficult to formalize and prove that types and proofs can be completely erased at runtime and imposes extra constraints on the design of the computation language. The flip side is of course is that their computational language enjoys true dependent types, whereas we have to simulate them with singleton types which sometimes involves a lot of code duplication between the computational and type levels.

8. 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 toy 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.

8.1 Acknowledgments

Thank you to Christopher League, Gregory Morrisett, Zhong Shao, and Valery Trifonov for their comments and criticisms, and to the anonymous reviewers for the numerous and helpful comments.

REFERENCES

AHMED, A. AND WALKER, D. 2003. The logical approach to stack typing. See TLDI'03 [2003], 74–85.

APPEL, A. W. 2001. Foundational proof-carrying code. In *Annual Symposium on Logic in Computer Science*. 247–258.

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

BIERMAN, G., HICKS, M., SEWELL, P., STOYLE, G., AND WANSBROUGH, K. 2003. Dynamic rebinding for marshalling and update, with destruct-time? In *International Conference on Functional Programming*. 99–110.

BIRKEDAL, L., TORP-SMITH, N., AND REYNOLDS, J. C. 2004. Local reasoning about a copying garbage collector. In *Symposium on Principles of Programming Languages*. Venice, Italy, 220–231.

CHASE, D. R., WEGMAN, M., AND ZADECK, F. K. 1990. Analysis of pointers and structures. In *Symposium on Programming Languages Design and Implementation*. ACM Press, 296–310.

CONDIT, J., HARREN, M., ANDERSON, Z., GAY, D., AND NECULA, G. C. 2007. Dependent types for low-level programming. See ESOP'07 [2007].

COQUAND, T. AND HUET, G. P. 1988. The calculus of constructions. *Information and Computation 76*, 95–120.

CRARY, K., WALKER, D., AND MORRISETT, G. 1999. Typed memory management in a calculus of capabilities. In *Symposium on Principles of Programming Languages*. San Antonio, TX, 262–275.

DELINE, R. AND FÄHNDRICH, M. 2001. Enforcing high-level protocols in low-level software. See PLDI'01 [2001].

ESOP'07 2007. *European Symposium on Programming*.

FÄHNDRICH, M. AND DELINE, R. 2002. Adoption and focus: Practical linear types for imperative programming. In *Symposium on Programming Languages Design and Implementation*. ACM Press.

FILLIÂTRE, J.-C. 2003. Verification of non-functional programs using interpretations in a type theory. *Journal of Functional Programming 13,* 4 (July), 709–745.

FLUET, M. AND MORRISETT, G. 2006. Monadic regions. *Journal of Functional Programming 16(4-5)*, 485–545.

FOGARTY, S., PAŠALIĆ, E., SIEK, J., AND TAHA, W. 2007. Concoqtion: Indexed types now! In *Workshop on Partial Evaluation and Semantics-Based Program Manipulation*.

HAMID, N. A., SHAO, Z., TRIFONOV, V., MONNIER, S., AND NI, Z. 2002. A syntactic approach to foundational proof-carrying code. See LICS'02 [2002], 89–100.

HARPER, R. AND MORRISETT, G. 1995. Compiling polymorphism using intensional type analysis. In *Symposium on Principles of Programming Languages*. 130–141.

HAWBLITZEL, C., WEI, E., HUANG, H., KRUPSKI, E., AND WITTIE, L. 2004. Low-level linear memory management. In *Informal proceedings of the SPACE Workshop*. Venice, Italy.

HICKS, M., MORRISETT, G., GROSSMAN, D., AND JIM, T. 2004. Experience with safe manual memory-management in cyclone. In *International Symposium on Memory Management*. 73–84.

HINZE, R. 1999. Polytypic programming with ease. In *International Symposium on Functional and Logic Programming*. Lecture Notes in Computer Science, vol. 1722. 21–36.

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

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

HUET, G., PAULIN-MOHRING, C., ET AL. 2000. The Coq proof assistant reference manual. Part of the Coq system version 6.3.1.

LEAGUE, C. AND MONNIER, S. 2006. Typed compilation against non-manifest base classes. *Lecture Notes in Computer Science 3956*, 77–98.

LICS'02 2002. *Annual Symposium on Logic in Computer Science*. Copenhagen, Denmark.

MONNIER, S. 2003. Principled compilation and scavenging. Ph.D. thesis, Yale University.

MONNIER, S. 2007. The Swiss coercion. In *Programming Languages meets Program Verification*. ACM Press, Freiburg, Germany.

MONNIER, S., SAHA, B., AND SHAO, Z. 2001. Principled scavenging. See PLDI'01 [2001], 81–91.

MORRISETT, G., CRARY, K., GLEW, N., AND WALKER, D. 2002. Stack-based typed assembly language. *Journal of Functional Programming 12,* 1 (Jan.), 43–88.

MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. 1998. From system F to typed assembly language. In *Symposium on Principles of Programming Languages*. 85–97.

NANEVSKI, A., AHMED, A., MORRISETT, G., AND BIRKEDAL, L. 2007. Abstract predicates and mutable ADTs in Hoare Type Theory. See ESOP'07 [2007].

NANEVSKI, A., MORRISETT, G., AND BIRKEDAL, L. 2006. Polymorphism and separation in Hoare Type Theory. In *International Conference on Functional Programming*. Portland, Oregon, 62–73.

NECULA, G. C. 1997. Proof-carrying code. In *Symposium on Principles of Programming Languages*.

PAULIN-MOHRING, C. 1993. Inductive definitions in the system Coq—rules and properties. In *Proc. TLCA*, M. Bezem and J. Groote, Eds. LNCS 664, Springer-Verlag.

PLDI'01 2001. *Symposium on Programming Languages Design and Implementation*.

REYNOLDS, J. C. 2002. Separation logic: a logic for shared mutable data structures. See LICS'02 [2002].

RÉMY, D. 1992. Typing record concatenation for free. In *Symposium on Principles of Programming Languages*. 166–176.

SHAO, Z., SAHA, B., TRIFONOV, V., AND PAPASPYROU, N. 2002. A type system for certified binaries. In *Symposium on Principles of Programming Languages*. 217–232.

SMITH, F., WALKER, D., AND MORRISETT, G. 2000. Alias types. In *European Symposium on Programming*.

SYME, D. 2005. Initializing mutually referential abstract objects. In *Workshop on ML*. Electronic Notes in Theoretical Computer Science, vol. 148(2). 3–25.

TLDI'03 2003. *Types in Language Design and Implementation*. New Orleans, LA.

TOFTE, M. AND TALPIN, J.-P. 1994. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Symposium on Principles of Programming Languages*. 188–201.

VANDERWAART, J. C. AND CRARY, K. 2003. A typed interface for garbage collection. See TLDI'03 [2003].

WALKER, D. AND MORRISETT, G. 2000. Alias types for recursive data structures. In *International Workshop on Types in Compilation*.

WANG, D. C. AND APPEL, A. W. 1999. Safe garbage collection = regions + intensional type analysis. Tech. Rep. TR-609-99, Princeton University.

WANG, D. C. AND APPEL, A. W. 2001. Type-preserving garbage collectors. In *Symposium on Principles of Programming Languages*.

WERNER, B. 1994. Une théorie des constructions inductives. Ph.D. thesis, A L'Université Paris 7, Paris, France.

XI, H., CHEN, C., AND CHEN, G. 2003. Guarded recursive datatype constructors. In *Symposium on Principles of Programming Languages*. New Orleans, LA, 224–235.

YANG, H. 2001. An example of local reasoning in BI pointer logic: the Schorr-Waite graph marking algorithm. In *Informal proceedings of the SPACE Workshop*. London, England.